MFRODO: Efficient and Memory-Sensitive Simulink Code Generation via Redundancy Elimination

Zehong Yu*, Yixiao Yang✉†, Zhuo Su✉§, Haowei Qiu*, Rui Wang†, Aiguo Cui‡, Zhan Shu¶ and Yu Jiang*
*KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China
†Information Engineering College, Capital Normal University, Beijing 100089, China
§School of Software, Beihang University, Beijing 100191, China
‡HUAWEI Technologies, Co. LTD. Shanghai 200120, China
¶NSFOCUS Technologies Group Co., Ltd., Beijing 100102, China

***Abstract*—Simulink has emerged as the fundamental infrastructure that supports modeling, simulation, verification, and code generation for embedded software development. 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 overlook the presence of redundant calculations within data-intensive models widely used to perform substantial data processing in embedded scenarios, which can significantly degrade the performance and introduce additional memory usage.**

This paper proposes MFRODO, an efficient and memory-sensitive code generator for data-intensive Simulink models through redundancy elimination. MFRODO begins by conducting model analysis to construct the dataflow graph and derive the I/O mapping of each block. Then, for each block within the dataflow graph, MFRODO recursively determines its calculation range by leveraging the I/O mapping of its subsequent blocks and marks optimization blocks whose calculation range is eliminated. For optimizable blocks, MFRODO eliminates the redundant calculations and reduces the memory space associated with these calculations. Finally, MFRODO rebuilds the I/O mappings of these optimizable blocks to ensure code correctness and synthesizes the embedded code for deployment. We implemented and evaluated MFRODO on benchmark Simulink models, in terms of execution duration, memory usage, and code generation overhead across different compilers and architectures. The results show that, compared with the Simulink Embedded Coder, DFSynth, and HCG, MFRODO achieves performance improvements ranging from 1.17× - 8.55×, while reducing BSS segment usage by 12.00% - 52.71%. Besides, MFRODO reduces compile time by 91.8% - 98.7% and code synthesis time by 94.3% - 99.6% compared with Simulink Embedded Coder, while incurring comparable overhead to DFSynth and HCG.

***Index Terms*—Simulink Models, Data-Intensive, Efficient, Memory-Sensitive, Code Generation**

I. INTRODUCTION

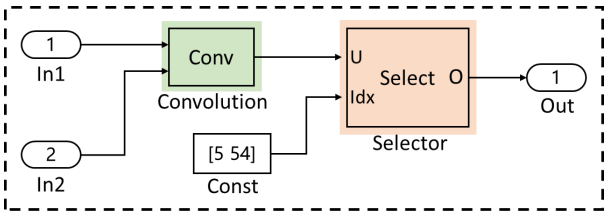
Simulink [1] has become the fundamental infrastructure in embedded scenarios [2], [3], [4], [5]. Developers can use Simulink's modeling elements, aka blocks, to build the model that represents the target system. Besides, Simulink supports model-based simulation, verification, and code generation to accelerate the development of embedded software. Code generation is a crucial part of model-driven design, which can automatically generate deployable code for target models, thereby saving massive labor efforts and being widely utilized in embedded software development. However, due to the tight performance and resource constraints of embedded applications, it is essential to ensure code efficiency and minimize memory usage.

Many commercial and research code generators have contributed to generating high-quality embedded code for deployment. Simulink Embedded Coder [6], the built-in tool of Simulink, is one of the most widely used code generators for embedded software development. Based on the user-constructed model, it can generate embedded code that can be deployed directly and supports various optimization options to ensure code efficiency, such as expression folding. In addition, it implements memory usage optimizations, such as data buffer reuse. Academic research has primarily focused on optimizations aimed at improving the performance of generated code. DFSynth [7] specializes in optimizing complex branch blocks inside Simulink models. It decomposes the target model into blocks wrapped by control statements and designs well-structured templates for code synthesis. HCG [8] tries to accelerate the execution of time-consuming blocks with the model. It first identifies parallel computation opportunities in the model and then synthesizes SIMD instructions for enhancing performance.

Despite the promising results have been achieved by these works in various scenarios, they still exhibit limitations when generating code for data-intensive Simulink models. Data-intensive models, which process arrays as inputs and perform extensive calculations on them, are common in real-world embedded systems, including but not limited to real-time DSP systems, electric drive systems [9], [10], [11], [12]. These models are related to loop constructs in the generated code, which are time-consuming and make up the majority of the computational overhead. While existing code generators are adept at accelerating individual statements within the generated code, they overlook the presence of redundant statements related to those models. This oversight can significantly undermine the efficiency of the resulting embedded software and waste the limited memory resources of the embedded devices.

Specifically, within the data-intensive models, data-truncation blocks are often utilized to select specific data segments for further calculations, such as `Selector` block and `Pad` block. For example, consider a `Convolution` model which performs the same convolution [13] on the input data, as shown in Figure 1. The same convolution is a type of

✉ Zhuo Su and Yixiao Yang are the corresponding authors.



(a) Convolution Model

```
// The range of 'In1' can be reduced to 59
void convolution(double In1[68], double In2[10], double Out[50]) {
    ...
    // The calculation range should be 5 to 54
    for (i = 0; i < 59; i++) {
        acc = 0;
        for (j = 0; j<= 9; j++)
            acc += In1[i+j] * In2[9-j];
        // The range of 'Convolution' can be reduced to 50
        Convolution[i] = acc;
    }
    // Assign the result to 'Out'
    memcpy(&Out[0], &Convolution[5], 50U * sizeof(double));
}
```

(b) Generated Code

Fig. 1: A sample model to illustrate the motivation of our work. The green part represents the code generated from the Convolution block. The orange part represents the code generated from the Selector block. The variables highlighted in red indicate those whose memory usage can be reduced.

convolution where the output data is of the same dimension as the input data; however, the implementation of Simulink for `Convolution` block is full padding which increases the output size, as shown in the code segment highlighted in green. Therefore, a `Selector` block is needed to select the part of `Convolution` block's output that represents the same convolution as shown in the code segment highlighted in orange, resulting in unnecessary calculations. Similarly, the memory usage of variables 'In1' and 'Convolution' related to these unnecessary calculations can be reduced. Unfortunately, all state-of-the-art code generators neglect this impact on code generation. In other words, they first translate the `Convolution` block as full padding and then translate the `Selector` block to select the same convolution part as output, compromising the quality of the embedded software. Moreover, considering the constrained performance and resource capabilities of embedded devices, such inefficiency is particularly problematic and deemed unacceptable [14].

To generate efficient and memory-sensitive code for data-intensive models by eliminating inherently redundant calculations and memory usage, we must address the following three challenges: ① The first challenge is accurately identifying the optimizable blocks influenced by data-truncation blocks. Simulink blocks within the model are interconnected, and indirectly connected blocks can also influence each other. Therefore, when identifying optimizable blocks, it is essential to consider not only the ones directly connected to the data-truncation blocks but also those indirectly connected. ② The second challenge is accurately eliminating the redundant calculations within the optimizable blocks. For effective optimization, both efficiency and correctness should be ensured. A loose elimination retains numerous time-consuming calculations for execution, leading to under-optimization. Conversely, an excessive elimination may yield pronounced performance improvement but at the cost of omitting crucial calculations, resulting in incorrect code. Therefore, an in-depth analysis of data interaction among blocks should be conducted to figure out the precise elimination range for optimization. ③ The third challenge is accurately reducing the memory usage of optimizable blocks. The deletion can cause the memory space of affected blocks to be shifted, which in turn results in the mapping between inputs and outputs being misaligned. This misalignment may result in the execution results being assigned to the wrong address, thus generating incorrect code. However, confirming the input/output mapping after the deletion is difficult, as it requires to thoroughly analyze the complex semantics of the target blocks.

To address the above challenges, we propose MFRODO, an efficient and memory-sensitive code generator for data-intensive Simulink models via redundancy elimination. First, MFRODO parses the target Simulink model to extract essential details, including blocks, connections, and functionalities. According to the collected information, MFRODO constructs the dataflow graph and derives the I/O (Input/Output) mapping of each block. For each block within the dataflow graph, MFRODO recursively determines its calculation range by leveraging the obtained I/O mapping of its subsequent blocks and labels those whose calculation range is eliminated as optimizable. MFRODO reduces the memory space of inputs and outputs associated with the unnecessary calculations. It then utilizes block functionalities to rebuild the I/O mapping of the optimizable blocks. Finally, MFRODO generates streamlined code for optimizable blocks, considering the precise calculation range and reduced memory space. This code is then integrated with the code generated from other basic blocks to produce the embedded code for deployment.

We implemented and evaluated the effectiveness of MFRODO on benchmark Simulink models [15], [7], [16], across different compilers and architectures. The results illustrate that MFRODO gains pronounced performance improvement. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by MFRODO is 1.26× - 8.55×, 1.32× - 5.75×, and 1.17× - 3.75× faster in terms of execution duration. Additionally, we measured the memory usage of the generated codes. The statistics show that, compared with other code generators, MFRODO reduces BSS (Block Started By Symbol) segment usage of program memory by 12.00% - 52.71%, further validating the effectiveness of our approach. We also measured the code generation overhead. MFRODO reduces compile time by 91.8% - 98.7% and code synthesis time by 94.3% - 99.6% compared with Simulink Embedded Coder, while consuming comparable overhead to DFSynth and HCG.

II. Background

A. Model-Driven Design

Model-driven design is a software development approach that emphasizes the creation, refinement, and manipulation of models as the primary artifacts throughout the design and implementation process. By utilizing high-level abstractions,

it enables developers to focus more on system functionality and architecture, rather than low-level code implementation. The goal of model-driven design is to shift the focus from manual coding to constructing detailed abstract models that can be automatically transformed into executable code, thereby enhancing development efficiency. These models serve as a blueprint for the system, enabling deeper analysis to verify not only their correctness but also their alignment with requirements and specifications. These advantages lead to the widespread adoption of model-driven design in the embedded systems domain, particularly for the design of complex control systems [17], [18], [19], [20].

In general, model-driven design typically involves four critical processes: behavior modeling, simulation, verification, and code generation. Each process serves a specific purpose and contributes to the overall efficiency and effectiveness of model-driven design. Behavior modeling describes the dynamic behaviors of the target system through models, defining how the system should behave in response to various inputs, states, and interactions. This behavior is often represented graphically using dataflow models with computational blocks, as well as automata models containing states and transitions. Simulation executes the constructed model to observe its behavior under specific conditions and scenarios. It allows developers to test the system's behavior in a virtual environment before implementation, offering insights into its functionality and helping to identify potential issues. Verification ensures that the model accurately represents the system and behaves according to the specified requirements. This process often employs formal methods to mathematically prove the correctness of the model, such as verifying adherence to certain properties and identifying design errors, such as divide-by-zero issues. Code generation automatically converts the constructed model into executable code. This is a key benefit of model-driven design, as it automates the transformation of high-level models into functioning software or systems.

B. Simulink and Code Generation

Simulink is the most widely used model-driven design tool and especially valuable in industries, such as aerospace, automotive, robotics, and telecommunications [21], [22], [23], [24], where complex systems need to be modeled, simulated, and validated before physical implementation. It provides a block diagram interface that allows users to visually create models by dragging and connecting blocks that represent different components, including mathematical operations, signal sources, etc. Simulink also offers a comprehensive set of predefined blocksets for diverse applications. Once the model is created, Simulink enables users to simulate the system's behavior over time and under varying conditions, verify the correctness of the system design against specifications, and supports code generation to automatically convert the model into embedded code for deployment [25].

Code generation is a crucial part of model-driven design, which releases the developers from error-prone coding tasks and accelerates software development. It primarily consists of four essential steps to generate code [26]: model parse, dataflow analysis, scheduling, and code synthesis. These steps work together seamlessly to convert the user-constructed Simulink models into efficient and deployable code for target hardware platforms. ① Model parse, as a preparation stage, analyzes the target model to collect the critical information for further usages, such as model structure, blocks, and connections. ② Dataflow analysis derives the sequential relationship and connectivity between blocks. This step can be customized to obtain specific information on demand. By analyzing how data moves through the model, we can obtain the execution order and the interaction of various blocks. ③ Scheduling infers the translation sequence of model blocks, based on the sequential relationship. It adopts a topological-based method to iteratively select candidate blocks for translation. Note that, as each iteration may have multiple candidate blocks, this results in the presence of multiple equivalent translation sequences. ④ Code synthesis generates corresponding code for each block, 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 the performance of the generated code, such as by utilizing SIMD instructions.

III. Design

MFrodo mainly contains two key components: Range Determination and Redundancy Elimination, as shown in Figure 2. ① Firstly, MFrodo parses the target model to collect critical information, including blocks, connections, etc. Then, based on the connections and block properties, MFrodo constructs the dataflow graph and derives the I/O mapping of each block. For each block within the dataflow graph, MFrodo recursively determines its input and output ranges as well as its subsequent blocks within the dataflow graph. The blocks whose ranges are eliminated are labeled as optimizable. ② Secondly, for optimizable blocks, MFrodo reduces the memory space related to unnecessary calculations, and records the new indices of inputs and outputs. Using these indices and block functionalities, MFrodo rebuilds I/O mappings of optimizable blocks to ensure code correctness. After that, MFrodo utilizes the element-level code library to generate concise code for optimizable blocks by the precise ranges. This code is then synthesized with code generated from other blocks, yielding high-efficiency code for deployment.

A. Range Determination

Simulink supports data-truncation blocks for modeling purposes, including but not limited to `Selector` block, `Pad` block, and `Submatrix` block. These blocks are utilized to select specific segments of input data for subsequent calculations. However, if the unselected data segments do not factor into subsequent calculations, any previous calculations involving those segments can be considered redundant and the related memory space can be reduced. Before optimization, MFrodo should first determine the precise calculation range of each block within the target model.

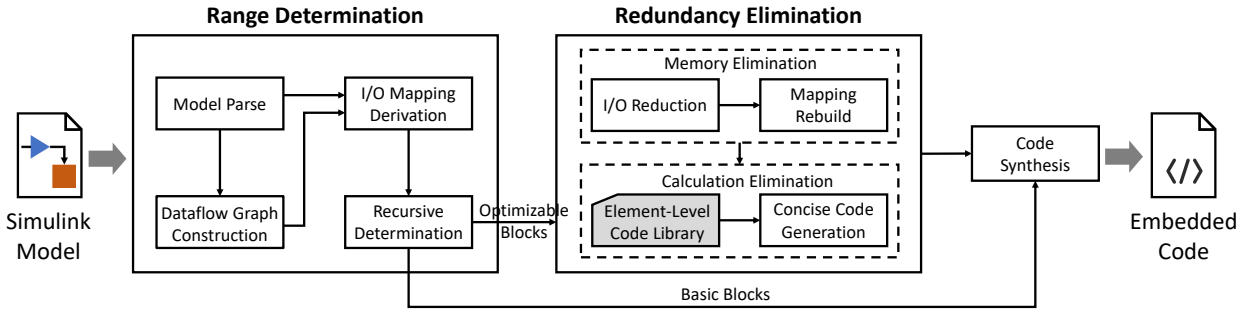


Fig. 2: Overall framework of MFRODO. (1) Range Determination: MFRODO parses the target model to collect critical information. Then, MFRODO constructs the dataflow graph and derives the I/O mapping of each block. For each block, MFRODO recursively determines its precise input and output ranges as well as its subsequent blocks. (2) Redundancy Elimination: For optimizable blocks, MFRODO reduces the memory space related to unnecessary calculations and records the new indices of inputs and outputs. Based on these indices, MFRODO rebuilds the I/O mappings to ensure code correctness.

Model Parse. Since Simulink does not support an open-source framework for developers to manipulate and modify the target model, MFRODO implements a customized parser to extract critical model information from the target model. Specifically, the Simulink model is wrapped by a `ZIP` file that contains different components, including model structure, parameters, and other properties. These components are recorded in the `XML` files. MFRODO interprets these files to parse the dataflow information, such as blocks and connections. Besides, for `Subsystem` blocks within the model, MFRODO flattens them and maps their inports and outports to the corresponding external blocks for further analysis.

Dataflow Graph Construction. Before determining the calculation range for each block, MFRODO first constructs the dataflow graph. For each block, MFRODO defines the appropriate runtime data structure to preserve critical contents, such as inputs, outputs, and parameters. For connections, MFRODO records both the source block and the destination block. Notably, it is vital to identify the output of the source block and the input of the destination block, as different ports can have distinct functionalities and mismatched ports can result in incorrect code.

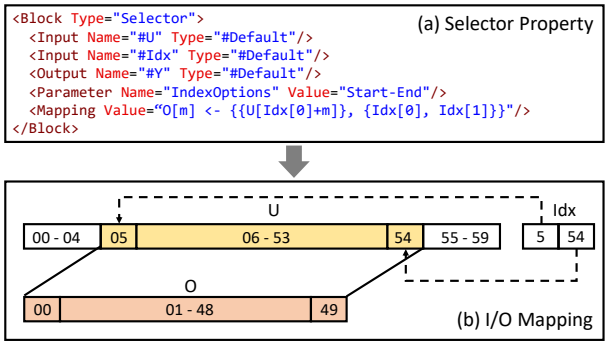


Fig. 3: An example of I/O mapping derivation. The subfigure (a) is `Selector` block's property, while the subfigure (b) shows the mapping between inputs and output.

I/O Mapping Derivation. To determine the accurate calculation range for generating concise code, it is essential to figure out the I/O mapping of each block, i.e., to ascertain the range of input data required to produce the desired output. To accomplish this, MFRODO begins by crafting a specialized block property library tailored to the block type and parameters. This library encapsulates critical details such as type, parameters, and mapping. Notably, even for blocks of the same type, the contained mapping can vary depending on the specific parameters. For instance, consider the `Selector` block property shown in Figure 3(a). The value, `Start-End`

Algorithm 1: Recursive Determination

Input: G: Dataflow graph of the target model
M: I/O mapping of each block
Output: r_i: Input range of each block
r_o: Output range of each block

Function `determine`(G, M)**:**
  // root blocks
  $roots \leftarrow \varnothing$
  // traverse blocks in dataflow graph
  for b *in* G **do**
    if *block is root* **then**
      $roots$`.append(`$block$`)`
    end
  end
  $r_i, r_o \leftarrow \varnothing$
  for b *in* $roots$ **do**
    `recursive`(G, M, b, r_i, r_o)
  end
  return r_i, r_o
End Function
Function `recursive`(G, M, b, r_i, r_o)**:**
  // child blocks
  $b_c \leftarrow graph[b].child$
  if $b_c = \varnothing$ **then**
    // use I/O mapping to get input range
    $r_i[b] \leftarrow M[b.output]$
    $r_o[b] \leftarrow b.output$
  end
  else
    for *child* *in* b_c **do**
      `recursive`$(G, M, child, r_i, r_o)$
      $r_o[b] \leftarrow r_o[b] \cup r_i[child]$
    end
    $r_i[b] \leftarrow M[r_o]$
  end
End Function

means that the `Selector` block selects the data from the start index to the end index. However, if the value `Start-End` is changed to `IndexPort` which means that the `Selector` block selects the data based on the value of the index port, the contained mapping should change. Therefore, MFRODO must extract the corresponding I/O mapping from the block property library based on the type and parameter of the target block. The extracted I/O mapping signifies the relationship between a single output datum and the input data. MFRODO should extend this relationship to include each output element. For example, consider the I/O mapping of the `Selector` block shown in Figure 3(b). In the `Selector` block, the value of 'U' indicates the data source, and the value of 'Idx' indicates the start index and end index of the selected data. Therefore, the start position value of 'O' corresponds to the 5th datum of the 'U', and the end position value of 'O' corresponds to the 54th datum of 'U', i.e., $O[0] = U[5], O[49] = U[54]$. Extending this relationship to the remaining data, MFRODO constructs a comprehensive I/O mapping for the `Selector` block. Leveraging this derived I/O mapping, redundant calculations can be effectively identified and eliminated.

Recursive Determination. To eliminate the redundant calculations, MFRODO designs a recursive method to determine the precise input range and output range of each block within the dataflow graph. After that, the precise ranges are utilized to generate streamlined code without redundant memory space and calculations. Algorithm 1 presents the overall procedure of recursive determination. The main idea of this algorithm is to initially determine the ranges of the child blocks, which are then employed to determine the ranges of their parent blocks. First, MFRODO traverses the dataflow graph to identify the root blocks and records them for further usage (lines 2-9). The root block is defined as the 0-in-degree block in the dataflow graph. For instance, in the `Convolution` model shown in Figure 1, the `Inport1` block and `Inport2` block are the root blocks. Since these blocks provide the source data for all calculations, determining their precise range is crucial for eliminating redundant calculations and memory space. For each block, MFRODO defines two maps called r_i and r_o to record the input range and output range of each block respectively, which are initially set to be an empty set (line 10). Then, MFRODO invokes the `recursive` function to determine the precise range of root blocks, as well as their subsequent blocks (lines 9-11). After that, MFRODO identifies the blocks whose ranges have been eliminated as optimizable blocks and generates streamlined code for them.

The procedure of `recursive` function is represented in lines 16-31 in Algorithm 1. First, MFRODO searches the dataflow graph to obtain child blocks of block b, i.e., the blocks that are connected to block b, and stores them into b_c. For example, the `Convolution` block is the child block of `Input1` block and `Input2` block. If $b_c = \varnothing$, it represents that $block$ has no child blocks. Based on the I/O mapping and the output range of block b, MFRODO derives the precise input range and output range, and assigns them to the respective entry (lines 19-23). Conversely, if $b_c \neq \varnothing$, MFRODO must consider the influence of the child blocks. To do this, MFRODO should first obtain the precise output range of block b. For each child block, MFRODO invokes the `recursive` function to drive the ranges of their inputs (line 26). Since, for a connection between the source block and the destination block, the data of the source block's output is equal to the data of the destination block's input. Therefore, MFRODO derives the accurate output range of block b by merging the corresponding input range of the child blocks (line 27). Subsequently, MFRODO uses I/O mapping obtained before to determine the precise input range of block b and update the value of $r_i[b]$ (line 29).

B. Redundancy Elimination

Based on the precise ranges of optimizable blocks, MFRODO adopts a two-step approach to eliminate redundant memory space and calculations. First, MFRODO removes the memory space allocated for inputs and outputs associated with unnecessary calculations within the optimizable blocks. Then, MFRODO leverages an element-level code library to generate streamlined code that aligns with the reduced memory space and the precise calculation range.

Algorithm 2: Memory Elimination

```
   Input: B_o: Optimizable blocks
          M: I/O mapping of each block
          r_i: Input range of each block
          r_o: Output range of each block
 1 Function eliminate(B_o, M, r_i, r_o):
 2     // traverse optimizable blocks
 3     for b in B_o do
 4         // record the new input index
 5         index_i ← ∅
 6         pointer ← 0
 7         // traverse each datum of input
 8         for i in b.input do
 9             if i is in r_i[b] then
10                 index_i[i] ← pointer
11                 pointer++
12             end
13         end
14         // record the new output index
15         index_o ← ∅
16         pointer ← 0
17         // traverse each datum of output
18         for o in b.output do
19             if o is in r_o[b] then
20                 index_o[o] ← pointer
21                 pointer++
22             end
23         end
24         // rebuild mapping
25         for o in r_o[b] do
26             // obtain new output index
27             index ← index_o[o]
28             // Align with the new input index
29             M[index] ← index_i[M[o]]
30         end
31     end
32 End Function
```

Memory Elimination. Algorithm 1 outlines the overall procedure of memory elimination. The core concept of this algorithm is to reorganize the memory space of inputs and outputs in a compact form and rebuild the I/O mapping in

accordance with the new indices of inputs and outputs. First, MFRODO defines a map called $index_i$ to record the new index of each input datum after elimination and defines a variable called $pointer$ to point to the next available input memory location (line 5-6). If the input datum i is available in $r_i[b]$, it represents that the input datum i is related to the necessary calculation and should be preserved (line 9). In such cases, MFRODO assigns $pointer$ to $index_i[b]$ as the new index of this input datum and increments $pointer$ to point to the next available memory location (line 10-11). Similarly, MFRODO defines a map called $index_o$ to record the new index of each necessary output datum, initializes $pointer$ to point to the next available output memory location, and repeats the above procedure to update their new indices (line 15-23). In this way, MFRODO reassigns memory location to each valid datum of inputs and outputs, reducing redundant memory usage and preserving new indexes for rebuilding I/O mapping. As the memory location of valid input and output datum are shifted, MFRODO must rebind the new output indices with new input indices to ensure the correctness of the generated code. For each valid output datum, MFRODO first obtains its new index which is preserved in $index_o$ (line 27). Then, MFRODO uses I/O mapping to identify the input data currently associated with the output datum, i.e., $M[o]$, and uses $index_i$ to get the new indices of this input data after elimination (line 28). After that, these new indices of input data are assigned to the corresponding entry of M for rebuilding mapping. Note that one output datum may correspond to multiple input data, it depends on the target block functionalities. For example, an output datum of `Convolution` block requires multiple input data for calculation. Therefore, rebuilding mapping must account for block functionalities.

Block Type: Convolution

①
```
int size = $Input2_size$;
float Conv_acc = 0;
for (int i = 0; i <= size; i++)
    Conv_acc += $Input1$[$Index$ + i] * $Input2$[size-i];
$Name_Output$[$Index$] = Conv_acc;
```

②
```
int size = $Input2_size$;
for (int i = $Start$; i < $End$; i += $Interval$){
    float Conv_acc = 0;
    for (int j = 0; j <= size; j++)
        Conv_acc += $Input1$[i + j] * $Input2$[size-j];
    $Name_Output$[i] = Conv_acc;
}
```

Fig. 4: An element-level code library of the `Convolution` block. The variables highlighted in red need to be substituted with the corresponding parameters of the target block.

Calculation Elimination. Based on the precise ranges and the rebuilt I/O mapping of optimizable blocks, MFRODO utilizes code snippets from the pre-constructed element code library to generate streamlined code without redundant calculations. Specifically, MFRODO first retrieves the corresponding code snippets for the target blocks, then replaces the placeholders in these snippets with actual values derived from the block parameters. This personalized code is then synthesized to create embedded code ready for deployment. Take Figure 4 as an example, which displays an element-level code library of the `Convolution` block. The snippet ① is utilized to generate code for individual output datum, while the snippet ② is utilized to generate code for consecutive data. The variables highlighted in red need to be substituted with the corresponding parameters of `Convolution` block. For instance, $Input2_size$ is replaced with the size of the second input of the `Convolution` block, and $Name_Output$ is replaced with the output name of the `Convolution` block.

Code Synthesis. First, MFRODO determines the translation sequence of the blocks by employing a topological-based method. Then, for basic blocks, MFRODO supports customized DLL (Dynamic Link Library) files to generate the corresponding code. Notably, the same type of blocks may have different detailed information, resulting in differences in the generated code. For instance, the code generated for `Convolution` blocks varies depending on the input type (e.g., float versus double). Consequently, MFRODO needs to configure such pivotal information as parameters within the DLL files to obtain the required code. Subsequently, the code generated for basic blocks and optimizable blocks is synthesized, forming the function code of the target model in accordance with the translation sequence above. Additional relevant information is encapsulated in certain header files for later usage. Finally, all of the above code is bundled together for deployment.

Figure 5 illustrates how we perform redundancy elimination on a target Simulink model (`Convolution` model as shown in Figure 1). After converting the target model into the customized dataflow graph, MFRODO identifies the root blocks for optimization, i.e., block ①, block ②, and block ③. For each root block, MFRODO recursively determines its input and output ranges as well as subsequent blocks. For instance, consider the Step 1. MFRODO first determines the input and output ranges of block ⑥, as it has no child blocks. After that, MFRODO determines the ranges of block ⑤, given that block ⑥ is connected to block ⑤. Then, MFRODO adjusts the output range of block ④ from [00 - 59] to [05 - 54], as it connects to a data-truncation block (block ⑤). MFRODO repeats the aforementioned steps until the ranges of block ① is determined. MFRODO classifies blocks into basic blocks and optimizable blocks. For the optimizable blocks, i.e., block ① and block ④, MFRODO first reduces their redundant memory space. According to the determined ranges, the memory space of block ① is reduced to [00 - 58] and the memory space of block ④ is reduced to [00 - 49]. Note that, as blocks are inter-connected, the memory space of a specific block corresponds to its output data, while its input data originates from the output data of other blocks. After that, MFRODO rebuilds the I/O mappings of block ① and block ④. Since block ① is an `Inport` block, its input and output data have a one-to-one relationship, making the mapping straightforward to rebuild. For block ④, MFRODO should rebuild the mapping between its output data and output data of block ①, as block ① connects to the first input of block ④ and the memory space of block ① has been reduced. Based on the functionalities of block ④, the i^{th} output datum is related to the input data from i^{th} to $(i+9)^{th}$. Finally, MFRODO generates concise code for optimizable blocks in accordance with the eliminated ranges and rebuilt mappings. This code is then synthesized with code generated from basic blocks, yielding embedded code.

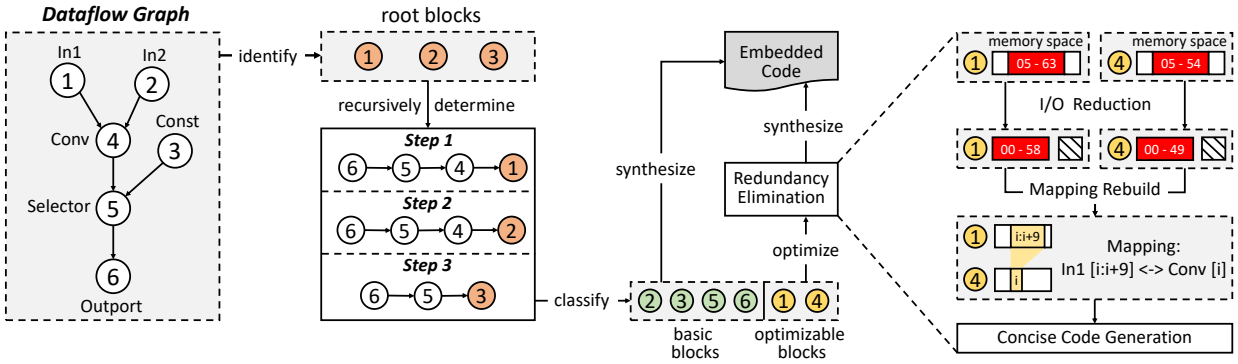


Fig. 5: Example workflow of MFRODO. The table shows the process of recursive determination. In each step, MFRODO determines the precise ranges of the root block and its subsequent blocks. After that, MFRODO reduces the redundant memory space and calculations of optimizable blocks, and generates concise code. The memory space highlighted in red is necessary, while the memory space highlighted in white is redundant.

IV. IMPLEMENTATION

We have implemented MFRODO [1] in C++, consisting of 26,342 lines of code. To parse Simulink models, we use `Unzip` library to unpack the Simulink `SLX` files and use `TinyXML` to extract model information for `XML` files. Currently, MFRODO supports code generation for over 54 types of Simulink blocks, including math operation blockset (e.g., `Add` block, `Multiply` block), logic blockset (e.g., `Relational Operator` block, `Switch` block), matrix-operation blockset (e.g., `Matrix Multiply` block, `Transpose` block), data-truncation blocks (e.g., `Selector` block, `Pad` block), complex blocks (e.g., `Sqrt` block, `Convolution` block), and subsystems (e.g., `IF` subsystem, `For-Iterator` subsystem). These blocks are commonly used in real-world embedded scenarios. We have also developed property libraries and element-level code libraries for these blocks to enable optimization. However, MFRODO does not yet support domain-specific blocks, such as those from the Lidar toolbox. We plan to extend support to these blocks in future work to further enhance the scalability and applicability of MFRODO.

V. EVALUATION

Evaluation Setup: To investigate the effectiveness of our approach, we compared MFRODO with four state-of-the-art code generators, Simulink Embedded Coder [6], DFSynth [7], HCG [8], and FRODO [27]. The comparison experiments were conducted across different architectures with two C-Compilers, i.e., GCC and Clang, and enabled `-O1` flag and `-O3` flag. Additionally, We generated a large number of random test cases for the code generated by MFRODO and compared the results with those from model simulations. The consistency between them underscores the correctness of MFRODO. Besides, we collected memory usage statistics for the code generated by all the code generators to measure the effectiveness of memory elimination. Since Simulink Embedded Coder is a built-in tool for Simulink, we use Simulink as the abbreviation in the following content.

[1]The implementation and the results are represented at the repository.https://anonymous.4open.science/r/MFrodo-5D6F/

TABLE I: Description of benchmark models.

Model	Functionality	#Block
AudioProcess	Vehicle audio analysis	51
Decryption	Decryption protocol	39
HighPass	HighPass filter model	49
HT	Hermitian transpose matrix calculation	26
Kalman	Automotive temperature control module	46
Back	Backpropagation in the CNN model	24
Maintenance	Industry equipment preservation model	165
Maunfacture	Product quality assessment model	29
RunningDiff	Differential amplifier	106
Simpson	Numerical integration model	30
RAC	Industrial robot control model	471
Quat	UAV attitude estimation and control model	265
ABS	Safety anti-lock braking system	419
Edge	CT/MRI Edge Enhancement	621

Benchmark Models: We evaluated MFRODO on a benchmark of 14 commonly used data-intensive Simulink models collected from industry product lines and academic works [8], [28], [15]. Table I shows the details of these benchmark models, including model functionalities and block counts. Benchmark models cover well-established domains, including automotive system, DSP system , industrial automation, healthcare system, and aerospace system. The models range in size from 24 to 621 blocks, covering a spectrum of realistic model complexities. Besides, these models include a diverse set of Simulink block types, such as discrete blockset, math-operation blockset, and diverse subsystems.

A. Effectiveness on Benchmark Models

To demonstrate the performance of the code generated by MFRODO, we conducted the experiment on the benchmark models. The experiment environment was an experimental machine (Win11, AMD Ryzen 7 5800X, 32GB RAM). The generated code was compiled by GCC (v11.3.0) and Clang (v14.0.6), with `-O1` flag and `-O3` flag enabled. To ensure statistical accuracy, the generated code was executed 10,000 times, and the average results were used for comparison.

Comparison Experiments on `-O3` flag. To assess the performance of MFRODO under high-level compiler optimization, we first conducted experiments on `-O3` flag. Table II shows the experiment results. For comparison experiments compiled with GCC, the execution duration of MFRODO is 1.26× - 5.64× faster than Simulink, 1.32× - 5.75× faster than DFSynth, and 1.22× - 2.89× faster than HCG. As for compiling with Clang, the execution duration of MFRODO is 1.77× - 7.78× faster than Simulink, 1.49× - 4.99× faster than DFSynth, and 1.39× - 3.03× faster than HCG. The data reveals that under the highest optimization level that can be applied to the code, MFRODO achieves a significant performance improvement. It also shows that the redundant calculations eliminated by MFRODO cannot be effectively addressed by the state-of-the-art code generators, even when utilizing the sophisticated optimization techniques within `-O3` flag implemented by compilers.

TABLE II: Comparison of the code execution duration on x86 using GCC and Clang with `-O3` flag.

Model	Compiler	Execution Duration (Unit: Second)				MFRODO Improvement		
		Simulink	DFSynth	HCG	MFRODO	Simulink	DFSynth	HCG
AudioProcess	GCC	1.58	0.49	0.52	**0.33**	**4.75**×	1.48×	1.55×
	Clang	1.57	0.58	0.42	**0.20**	**7.78**×	2.89×	2.07×
Decryption	GCC	0.37	0.30	0.26	**0.21**	**1.74**×	1.42×	1.22×
	Clang	0.37	0.21	0.18	**0.12**	**3.09**×	1.76×	1.54×
HighPass	GCC	0.87	0.29	0.33	**0.16**	**5.38**×	1.81×	2.03×
	Clang	0.56	0.32	0.31	**0.18**	**3.07**×	1.77×	1.69×
HT	GCC	0.65	0.72	0.65	**0.31**	2.09×	**2.29**×	2.09×
	Clang	0.71	0.75	0.74	**0.32**	2.24×	**2.37**×	2.34×
Kalman	GCC	0.37	0.27	0.26	**0.20**	**1.84**×	1.32×	1.29×
	Clang	0.40	0.33	0.31	**0.22**	**1.79**×	1.49×	1.39×
Back	GCC	0.30	0.45	0.70	**0.24**	1.26×	**1.87**×	2.89×
	Clang	0.79	0.54	0.76	**0.25**	3.15×	**2.14**×	3.03×
Maintenance	GCC	0.93	0.30	0.39	**0.22**	**4.17**×	1.32×	1.73×
	Clang	0.86	0.34	0.27	**0.19**	**4.52**×	1.81×	1.43×
Manufacture	GCC	2.25	0.97	0.66	**0.49**	**4.62**×	2.00×	1.35×
	Clang	3.45	1.11	0.88	**0.53**	**6.55**×	2.12×	1.68×
RunningDiff	GCC	0.71	0.72	0.19	**0.13**	5.64×	**5.75**×	1.54×
	Clang	0.58	0.59	0.20	**0.12**	**4.88**×	4.99×	1.65×
Simpson	GCC	0.95	0.43	0.43	**0.27**	3.57×	**1.61**×	1.63×
	Clang	1.39	0.55	0.41	**0.25**	5.58×	**2.22**×	1.65×
RAC	GCC	1.40	2.01	1.82	**0.98**	1.42×	**2.05**×	1.85×
	Clang	1.25	1.54	1.53	**0.95**	1.32×	**1.63**×	1.62×
Quat	GCC	0.48	0.85	0.84	**0.26**	1.87×	**3.30**×	3.27×
	Clang	0.48	0.68	0.67	**0.27**	1.81×	**2.56**×	2.49×
ABS	GCC	0.90	1.22	1.13	**0.43**	2.08×	**2.81**×	2.61×
	Clang	0.93	0.96	0.94	**0.39**	2.38×	**2.47**×	2.41×
Edge	GCC	1.27	1.38	1.33	**0.83**	**1.54**×	1.67×	1.61×
	Clang	1.18	1.09	1.08	**0.67**	**1.77**×	1.62×	1.61×
Average	GCC	0.93	0.74	0.68	**0.36**	**3.00**×	2.19×	1.90×
	Clang	1.04	0.68	0.62	**0.33**	**3.57**×	2.27×	1.90×

The performance of the code generated by Simulink is relatively limited, mainly because it lacks effective optimization techniques for data-intensive models. Simulink indeed employs some optimization techniques, including SIMD instruction utilization and expression folding. However, it usually fails to effectively identify the target blocks to apply SIMD instructions, resulting in limited performance. As for expression folding, compilers employ a similar and effective implementation in the compilation process. Besides, we found that the code generated by Simulink significantly underperforms other code generators on `AudioProcess` model and `Manufacture` model. This disparity is due to the fact that these models contain `Convolution` blocks and Simulink generates numerous boundary judgments to ascertain whether values should undergo convolution calculations. These judgments hinder the optimization techniques supported by compilers, such as loop unrolling. In contrast, other code generators, including DFSynth, HCG, and MFRODO, avoid such statements by padding the data length of inputs with zeros, thereby facilitating compiler optimization.

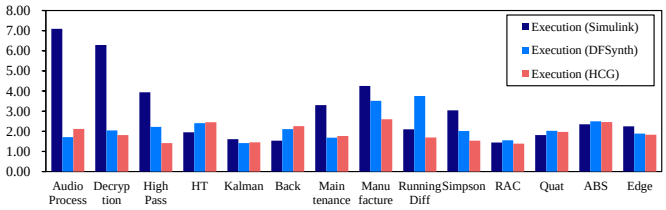
DFSynth mainly focuses on generating concise code for complex branch blocks within the model, thus lacking optimization techniques for data-intensive models. HCG synthesizes appropriate SIMD instructions for compute-intensive blocks to improve the efficiency of the generated code. However, compilers, including GCC and Clang, utilize similar optimization techniques to speed up the execution. As a result, at high levels of optimization, the optimization methods provided by HCG become less effective and may even have a negative impact. For example, consider the generated code of the `Back` model. By analyzing the assembly code, we found that HCG's employment of SIMD instructions, such as `_mm256_fmadd_pd`, prompts the compiler to generate assembly code mirroring HCG's methodology. This, in turn, hinders other potential optimization techniques supported by compilers from effectively manifesting. Consequently, the compiled assembly code is both verbose and lengthy.

Compared to other code generators, MFRODO strategically exploits critical information, specifically the dataflow graph and I/O mapping. This enables MFRODO to accurately determine the input range and output range of each block, thereby identifying a significant amount of time-consuming and redundant calculations. In fact, compilers have similar implementations, e.g., `-fmove-loop-invariants`, which strive to eliminate or move code that does not change across iterations outside of the loop. However, the effectiveness of these techniques depends on the compiler's capability to determine that the code is indeed invariant. Due to the invisibility of high-level information contained in the model, such as inputs/outputs, and block functionality, compilers are unable to classify variables as invariants definitively. Moreover, the intricate data types, such as pointers, pose substantial analytical challenges for compilers. Consequently, compilers often fail to employ aggressive optimization techniques, thereby limiting the potential for performance improvement.

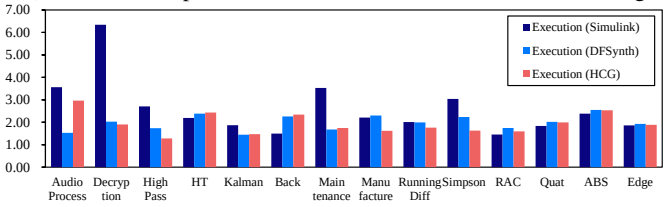
Comparison Experiments on `-O1` flag. We conducted the comparison experiments using `-O1` flag to further validate the effectiveness of MFRODO and explore how MFRODO interacts with compiler heuristics. Table III shows the experiment results. For comparison experiments compiled with GCC, the execution duration of MFRODO is 1.32× - 6.04× faster than Simulink, 1.50× - 5.01× faster than DFSynth, and 1.34× - 2.81× faster than HCG, respectively. For comparison experiments compiled with Clang, the execution duration of MFRODO is 1.28× - 7.66× faster than Simulink, 1.51× - 4.71× faster than DFSynth, and 1.21× - 2.38× faster than HCG, respectively. These statistics indicate that MFRODO still achieves pronounced performance improvement.

TABLE III: Comparison of the code execution duration on x86 using GCC and Clang with `-O1` flag.

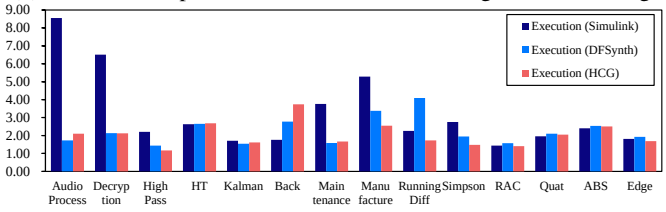
Model	Compiler	Execution Duration (Unit: Second)				MFRODO Improvement		
		Simulink	DFSynth	HCG	MFRODO	Simulink	DFSynth	HCG
AudioProcess	GCC	2.13	0.55	0.58	**0.35**	**6.04**×	1.57×	1.64×
	Clang	1.70	0.53	0.52	**0.22**	**7.66**×	2.41×	2.37×
Decryption	GCC	0.74	0.60	0.50	**0.37**	**1.97**×	1.60×	1.34×
	Clang	0.71	0.54	0.32	**0.27**	**2.63**×	2.02×	1.21×
HighPass	GCC	2.71	1.32	1.42	**0.51**	**5.34**×	2.60×	2.81×
	Clang	2.60	0.93	0.91	**0.49**	**5.33**×	1.91×	1.86×
HT	GCC	0.64	0.88	0.88	**0.40**	1.59×	**2.18**×	**2.18**×
	Clang	0.72	0.88	0.89	**0.40**	1.83×	2.23×	**2.24**×
Kalman	GCC	1.33	1.16	1.16	**0.77**	**1.72**×	1.50×	1.50×
	Clang	1.28	1.24	1.23	**0.82**	**1.55**×	1.51×	1.49×
Back	GCC	0.85	1.15	1.11	**0.48**	1.76×	**2.39**×	2.30×
	Clang	1.01	1.09	0.83	**0.51**	1.97×	**2.12**×	1.62×
Maintenance	GCC	1.17	0.66	0.73	**0.37**	**3.14**×	1.75×	1.95×
	Clang	1.04	0.56	0.69	**0.35**	**3.01**×	1.62×	2.00×
Manufacture	GCC	2.68	1.61	1.37	**1.01**	**2.65**×	1.59×	1.36×
	Clang	2.97	2.24	1.34	**0.99**	**2.98**×	2.25×	1.35×
RunningDiff	GCC	1.46	1.51	0.42	**0.30**	4.84×	**5.01**×	1.41×
	Clang	1.55	1.41	0.42	**0.30**	**5.17**×	4.71×	1.40×
Simpson	GCC	1.85	2.03	0.87	**0.55**	3.40×	**3.73**×	1.60×
	Clang	1.84	2.03	0.83	**0.54**	3.38×	**3.73**×	1.53×
RAC	GCC	1.45	2.31	2.00	**1.10**	1.32×	**2.11**×	1.82×
	Clang	1.33	1.75	1.67	**1.04**	1.28×	**1.67**×	1.60×
Quat	GCC	0.59	0.97	0.84	**0.31**	1.94×	**3.19**×	2.76×
	Clang	0.59	0.69	0.68	**0.29**	2.05×	**2.39**×	2.38×
ABS	GCC	1.79	2.00	1.97	**0.89**	2.01×	**2.25**×	2.21×
	Clang	1.61	1.75	1.70	**0.68**	2.37×	**2.57**×	2.50×
Edge	GCC	2.70	2.25	2.21	**1.46**	**1.85**×	1.53×	1.51×
	Clang	2.70	2.00	1.97	**1.17**	**2.31**×	1.71×	1.69×
Average	GCC	1.53	1.42	1.23	**0.68**	**2.67**×	2.10×	1.90×
	Clang	1.49	1.35	1.17	**0.65**	**2.87**×	2.12×	1.92×



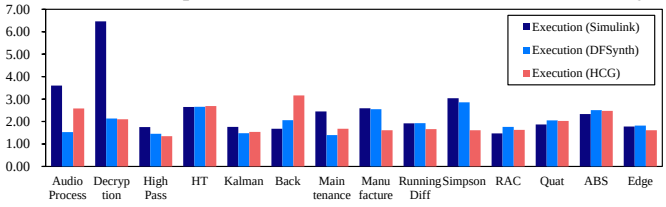
(a) Execution Improvement on ARM with GCC under `-O1` flag.



(b) Execution Improvement on ARM with Clang under `-O1` flag.



(c) Execution Improvement on ARM with GCC under `-O3` flag.



(d) Execution Improvement on ARM with Clang under `-O3` flag.

Fig. 6: The execution improvement of the code generated by MFRODO versus other code generators on ARM.

Compared with the results under `-O3` flag, the absolute execution duration under `-O1` is longer across all code generators. This is expected, as compilers apply fewer heuristic and aggressive optimizations at lower-level flags. For example, `-O3` flag enables advanced techniques, such as loop unrolling, loop fusion, and vectorization, which are not present or limited under `-O1` flag. Besides, we observed that relative performance improvement of MFRODO over Simulink is smaller under `-O1` flag than under `-O3` flag. This is because Simulink implements various optimizations, such as expression folding and loop fusion, and these optimizations remain effective under `-O1` flag. In contrast, MFRODO focuses on eliminating redundant calculations, which synergizes more strongly with aggressive compiler optimizations enabled under `-O3` flag.

B. Effectiveness on Different Architectures

To further assess the effectiveness of MFRODO across different architectures, we conducted repetitive experiments on an industrial machine with an ARM processor (Linux v6.1.21, ARM Cortex A72, 8GB RAM). The generated code was compiled by GCC (v11.3.0) and Clang (v14.0.6) with `-O1` flag and `-O3` flag enabled, respectively.

Figure 6 shows the execution improvement of the code generated by MFRODO compared to other code generators on the ARM architecture under `-O1` flag and `-O3` flag. Each bar represents the performance improvement achieved by MFRODO compared to the corresponding code generators. Compared to previous experiments on the x86 architecture, MFRODO achieves better performance improvement on the ARM architecture. Specifically, under GCC with `-O1` flag and `-O3` flag, the execution duration of MFRODO is 1.44× - 8.55× faster than Simulink, 1.42× - 4.10× faster than DFSynth, and 1.17× - 3.75× faster than HCG. When compiled with Clang, the execution duration of MFRODO is 1.45× - 6.46× faster than Simulink, 1.40× - 2.85× faster than DFSynth, and 1.28× - 3.17× faster than HCG. Due to the constrained performance capabilities of embedded devices, the hardware optimization techniques employed by them are limited. For example, consider the support for SIMD instructions. The AMD processor utilized above supports 512-bit SIMD instructions, while the ARM processor utilized in the embedded device only supports 128-bit SIMD instructions. As a result, for embedded devices with limited resources, the primary performance bottlenecks arise from the logic of the generated code. MFRODO takes advantage of this and demonstrates a greater performance improvement. Given that real-world applications often involve embedded processors, MFRODO is better aligned with practical use cases.

C. Effectiveness on Memory Usage

We conducted set of experiments to analyze memory usage across the code generated by different code generators, aiming to assess the effectiveness of the memory elimination technique employed by MFRODO. This metric is crucial, as embedded devices typically operate under stringent resource constraints while being expected to support a diverse range of applications. Consequently, optimizing memory utilization and execution efficiency are both essential for enhancing overall performance. By minimizing resource consumption, we can enable embedded devices to handle more complex tasks under the limited hardware capabilities.

TABLE IV: Comparison experiments on Data segment usage.

Model	Data Segment (Unit: Byte)				
	Simulink	DFSynth	HCG	FRODO	MFRODO
AudioProcess	**616**	640	640	624	624
Back	**624**	632	632	**624**	**624**
Decryption	**616**	640	640	640	640
HighPass	632	632	632	632	**624**
HT	**632**	**632**	**632**	**632**	**632**
Kalman	632	632	**624**	**624**	**624**
Maintenance	632	640	640	**624**	**624**
Maunfacture	**624**	640	640	**624**	**624**
RunningDiff	**632**	**632**	**632**	**632**	**632**
Simpson	**624**	632	632	**624**	**624**
RAC	**648**	**648**	**648**	**648**	**648**
Quat	**648**	**648**	**648**	**648**	**648**
ABS	**624**	**624**	**624**	**624**	**624**
Edge	**624**	**624**	**624**	**624**	**624**
Average	**628**	635	635	630	630

We utilized the `Size` command in Linux to collect the memory usage of the generated code. We collect two parts of memory usage: Data segment and BSS (Block Started by Symbol) segment. Data segment is used to store initialized global and static variables. For the model code, the main sources of memory usage in this segment are referenced header files and some model configuration parameters, for example `stdio.h`. Therefore, the data segment of the code generated by different code generators is almost the same, as shown in Table IV. Besides, compared to BSS segment, the memory usage of Data segment is relatively small. This observation indicates that the BSS segment is the primary source of memory usage for the generated code.

BSS segment is used to store uninitialized global and static variables. For the model code, the input and output data for the entire model are stored in the BSS segment. Additionally, some blocks may have internal states to store the current execution results for use in subsequent computations, which are also allocated in the BSS segment. As a result, BSS segment accounts for the majority of the memory usage in the model code. As shown in Table V, BSS segment usage of MFRODO is reduced by 12.00% - 52.71%, 13.64% - 47.76%, 13.64% - 47.76%, and 13.64% - 47.76%, compared with Simulink, DFSynth, HCG, and FRODO respectively. These statistics indicate that MFRODO's memory elimination approach effectively removes memory overhead associated with redundant calculations.

In typical embedded deployments (e.g., ARM Cortex-M and TriCore platforms), such a reduction alleviates static memory pressure, enables concurrent task execution, and allows the use of lower-cost MCUs with smaller SRAM footprints. For instance, the Infineon Tricore TC397, a representative automotive control MCU, provides only 64 KB of Data RAM available. In the Edge model, MFRODO saves approximately 4.4 KB of BSS usage compared to Simulink, which corresponds to nearly 7% of the total memory on such devices. This demonstrates that MFRODO can facilitate the deployment of more complex models or additional safety functions within the same memory budget.

TABLE V: Comparison experiments on BSS segment usage.

Model	BSS Segment (Unit: B)					MFRODO Reduction			
	Simulink	DFSynth	HCG	FRODO	MFRODO	Simulink	DFSynth	HCG	FRODO
AudioProcess	9896	9152	9152	9152	**7904**	**20.13%**	13.64%	13.64%	13.64%
Decryption	8528	7328	7328	7328	**5280**	**38.09%**	27.95%	27.95%	27.95%
HighPass	2688	2656	2656	2656	**2208**	**17.86%**	16.87%	16.87%	16.87%
HT	3528	3488	3488	3488	**2720**	**22.90%**	22.02%	22.02%	22.02%
Kalman	7376	5280	5280	5280	**3488**	**52.71%**	33.94%	33.94%	33.94%
Back	2976	2976	2976	2976	**2336**	**21.51%**	**21.51%**	**21.51%**	**21.51%**
Maintenance	6400	5840	5840	5840	**5008**	**21.75%**	14.25%	14.25%	14.25%
Maunfacture	16400	17304	17304	17304	**14432**	12.00%	**16.60%**	**16.60%**	**16.60%**
RunningDiff	11392	12864	12864	12864	**6720**	41.01%	**47.76%**	**47.76%**	**47.76%**
Simpson	10896	10848	10848	10848	**6368**	**41.56%**	41.30%	41.30%	41.30%
RAC	7240	7208	7208	7208	**5800**	**19.89%**	19.53%	19.53%	19.53%
Quat	1816	1848	1848	1848	**1272**	29.96%	**31.17%**	**31.17%**	**31.17%**
ABS	3016	3000	3000	3000	**1400**	**53.58%**	53.33%	53.33%	53.33%
Edge	11840	11464	11464	11464	**6984**	**41.01%**	39.08%	39.08%	39.08%
Average	7428	7232	7232	7232	**5137**	**31.00%**	28.50%	28.50%	28.50%

We observed that the BSS segment usage remains identical among code generated by DFSynth, HCG, and FRODO, primarily because these generators employ the same allocation strategy for input/output data and internal states. Conversely, Simulink has its own allocation strategy for scheduling input/output data and internal states. As a result, Simulink outperforms code generators, except MFRODO, in terms of BSS segment usage for some models, such as the `Maunfacture` model, while underperforming in others, such as the `Kalman` model. On the one hand, Simulink implements memory-related optimizations, which shrinks memory usage. For example, Simulink reuses previously defined variables for further usage and folds expressions to decrease intermediate variables. On the other hand, for each model, Simulink defines specific variables to preserve real-time features required by embedded systems, such as task scheduling. However, these variables become redundant when the target models do not incorporate real-time features, leading to increased BSS segment usage.

D. Code Generation Overhead

We also conducted experiments to measure the code generation overhead of MFRODO and other code generators. Specifically, the total overhead of code generation can be divided into two parts: (1) **Compile Time:** This includes parsing the target Simulink model to extract structural and semantic information, constructing the dataflow graph, and performing optimizations such as recursive I/O range analysis and memory reduction.

(2) **Code Synthesis Time:** This involves generating code for each block and assembling them according to the translation sequence. The detailed statistics are summarized in Table VI.

TABLE VI: Comparison experiments on code generation time.

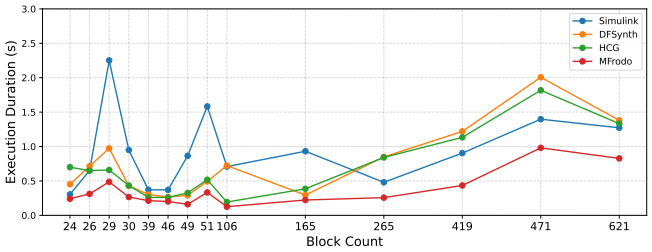
Model	Compile Time (Unit: Second)				Code Synthesis Time (Unit: Second)			
	Simulink	DFSynth	HCG	MFRODO	Simulink	DFSynth	HCG	MFRODO
AudioProcess	7.566	**0.090**	0.097	0.096	1.569	**0.006**	**0.006**	**0.006**
Decryption	2.911	**0.085**	0.089	0.088	0.998	**0.007**	**0.007**	**0.007**
HighPass	2.913	**0.067**	0.071	0.068	0.989	**0.006**	**0.006**	**0.006**
HT	3.408	**0.051**	0.058	0.054	1.132	**0.006**	**0.006**	**0.006**
Kalman	5.808	**0.076**	0.079	0.079	0.969	**0.007**	**0.007**	**0.007**
Back	2.593	**0.043**	0.047	0.047	0.954	**0.006**	**0.006**	**0.006**
Maintenance	3.204	**0.184**	0.201	0.191	0.934	**0.014**	**0.014**	**0.014**
Maunfacture	2.427	**0.069**	0.074	0.070	0.926	**0.005**	**0.005**	**0.005**
RunningDiff	3.058	**0.051**	0.056	0.057	0.998	**0.007**	**0.007**	**0.007**
Simpson	3.282	**0.060**	0.069	0.062	1.058	**0.060**	**0.060**	**0.060**
RAC	3.990	**0.313**	0.348	0.327	1.229	**0.049**	**0.049**	**0.049**
Quat	3.475	**0.178**	0.189	0.175	1.140	**0.020**	**0.020**	**0.020**
ABS	2.772	**0.199**	0.213	0.219	1.229	**0.025**	**0.025**	**0.025**
Edge	2.352	**0.344**	0.369	0.352	1.140	**0.059**	**0.059**	**0.059**
Average	3.560	**0.134**	0.151	0.147	1.080	**0.022**	**0.022**	**0.022**

As shown in Table VI, compared to Simulink, MFRODO reduces compile time by 91.8% - 98.7% and code synthesis time by 94.3% - 99.6%. The higher overhead in Simulink is due to its extensive internal and external processing. Simulink invokes auxiliary toolchain components during code generation, such as Microsoft Visual C++ toolchain and NMake, which introduces substantial I/O and process launch overhead. Moreover, since Simulink is a graphical editor and integrated within Matlab, Simulink Embedded Coder frequently interacts with Simulink's GUI backend to retrieve configuration directives and block metadata, which further increases overhead. In contrast, MFRODO is a lightweight and standalone code generator that performs direct model parsing and optimizations without heavyweight toolchain invocations and runtime GUI interactions, resulting in faster code generation. DFSynth and HCG follow a similar lightweight architecture and thus show comparable code generation time to MFRODO.

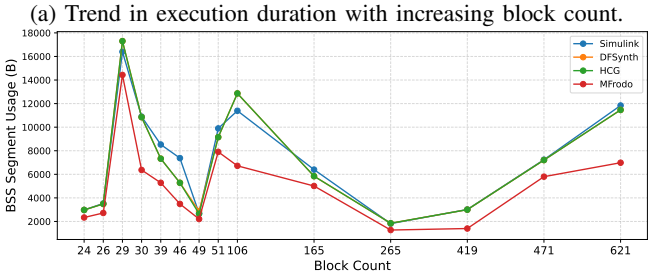
E. Trends in execution duration and BSS segment usage

We conducted experiments to explore the trends in execution duration time and BSS segment usage as the block count increased. We selected experiment results on the x86 architecture using GCC with the `-O3` flag as representative data, while similar trends were also observed under other configurations, e.g., Clang and ARM. The trends are presented in Figure 7.

Specifically, for execution duration, we observed that, the execution duration of the generated code among all code generators generally increases as the block count grows. This is because that, a higher number of blocks typically results in longer generated code, thereby increasing the overall execution duration. Besides, we conducted a correlation analysis between execution duration and block count, and the correlation coefficients are 0.196, 0.807, 0.834, and 0.783 for Simulink, DFSynth, HCG, and MFRODO, respectively. These results indicate a strong positive correlation for DFSynth, HCG, and MFRODO. In contrast, Simulink shows a weak correlation, primarily due to two outliers, i.e., Maunfacture model (29 blocks) and AudioProcess model (51 blocks). Despite relatively small block counts, these models contain `Convolution` blocks. Simulink generates extensive boundary checks for these blocks to determine whether values should undergo convolution calculations, introducing significant execution overhead.



(a) Trend in execution duration with increasing block count.



(b) Trend in BSS segment usage with increasing block count.

Fig. 7: The trends in execution duration and BSS segment usage as the block count increased.

As for the trend between BSS memory usage and block count, we observed no clear correlation. The correlation coefficients are -0.031, -0.022, -0.019, and -0.099 for Simulink, DFSynth, HCG, and MFRODO, respectively, indicating a weak or negative correlation in all cases. BSS segment is used to store uninitialized global and static variables. For the model code, the input data and output data are stored in the BSS segment. Consequently, BSS memory usage depends more on the size and data types of input/output arrays than on the block count in the model. For example, a small number of blocks that operate on large arrays may consume more BSS memory than a large model composed of simple scalar operations. As mentioned before, Data segment is used to store initialized global and static variables. For the model code, the main sources of data segment are referenced header files and some model configuration parameters, for example, `stdio.h`. As a result, data segment usage remains nearly identical across different models, and we did not conduct separate experiments.

VI. DISCUSSION

Scalability of MFRODO. At present, we conducted experiments on x86 and ARM architectures, and MFRODO achieves significant performance improvement and reduces memory usage. However, MFRODO are not only suitable for them, but also for other architectures, such as RISC-V and Tricore. The improvements come from eliminating redundant calculations and memory usage related to data-truncation blocks. Therefore, the code employed on other architectures also benefits from our approach. Besides, MFRODO focuses on optimizing the code generation for Simulink models. In fact, there exist other model-driven design tools, and some of them are even

widely used in various scenarios, for example, Ptolemy-II [29] and SCADE [30]. To support code optimization for these tools, MFRODO first requires to parse the corresponding model files to obtain relevant information, including block properties, and connections. Once this information is obtained, MFRODO can apply redundancy elimination on these models for improved performance and reduced memory usage.

Code Correctness. To ensure code correctness, we have deployed several effective approaches. For each benchmark model, we generated numerous test cases as input data for the code generated by MFRODO and other code generators, and the execution results among them are consistent. Besides, we also confirmed the consistency between the code execution and model simulation. During optimization, MFRODO recursively determines the precise calculation range for each block. Specifically, for a given target block, MFRODO first determines the range of its child blocks and then uses the derived ranges to determine the range of the target block. This process ensures that the relevant calculations for the final results are preserved. For memory elimination, MFRODO rebuilds the I/O mapping of each block in accordance with block properties, ensuring that valid output data are correctly mapped to the corresponding input data.

Incremental Optimization. Currently, MFRODO processes the entire model from scratch in each code generation cycle. As shown in Table VI, the overhead of this approach is relatively small, i.e., approximately 1.256s on average across all models, and is acceptable. However, we acknowledge that, for iterative development where model changes occur frequently, supporting incremental optimization would further reduce the code generation overhead. To support this, MFRODO can be extended to first detect changed blocks by comparing dataflow graphs before and after the modification. Then, MFRODO would recursively determine the calculation range of modified blocks and their parent blocks, and apply redundancy elimination and memory reduction accordingly. Finally, code would be regenerated only for the changed regions and integrated with the previously generated code according to the translation sequence. We plan to incorporate incremental optimization into MFRODO as part of our future work.

Threats to Validity. For complex blocks, such as the `Convolution` block, MFRODO generates multiple instances of code for the same block type in accordance with their distinct calculation range. As a result, MFRODO may generate longer code compared to other code generators for models containing multiple complex blocks. MFRODO can avoid such code duplication by generating a generic function interface and configuring the derived calculation range as parameters. For code generation, Simulink generates a unified function by default, which combines the execution code for each subsystem in the target model. In contrast, MFRODO generates a separate function for each subsystem. While this approach improves the readability and maintainability of the generated code, it introduces additional overhead due to the increased function calls, potentially reducing performance. To address this, MFRODO will support the same strategy as Simulink for handling subsystems in performance-critical scenarios.

VII. Related Work

Commercial and Academic Code Generators. Recently, both commercial and research tools have made remarkable efforts to improve the performance of the generated code. Specifically, Simulink Embedded Coder [6] is the most widely-used commercial tool, which specializes in generating production-quality code by employing various high-level optimization techniques. For instance, for execution performance, Simulink Embedded Code synthesizes SIMD instructions for complex blocks and folds expressions to avoid redundant assignments. For memory efficiency, it reuses previously defined variables to reduce memory usage. These optimizations have made Simulink Embedded Coder a popular choice for embedded system development, with its generated code often used directly for deployment.

In the academic realm, several noteworthy efforts have been made as well. DFSynth [7] focuses on optimizing model branch logic, such as `Switch` block. It 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. For code optimization, it designs tailored templates for each block and uses them to synthesize high-performance embedded code. HGC [8] accelerates the execution of intensive computing blocks and batch computing blocks. For intensive computing blocks, it selects optimal implementations for the code library based on the input scale and execution estimation. For batch computing blocks, it combines blocks within the dataflow graph to synthesize appropriate SIMD instructions for performance improvement. Mercury [16] prioritizes adjusting the code translation order and avoiding instruction pipeline stalls. Initially, Mercury collects data dependencies through model dataflow traversal and records the properties of each block. It then approximately estimates the execution latency of required instructions fetched from corresponding blocks and employs a topology-based method to identify candidate blocks for code synthesis. Lastly, Mercury uses the least penalty priority approach to iteratively select the most suitable block for code synthesis, consequently releasing data dependencies with its subsequent blocks.

MFRODO differs from these works by utilizing model semantics to identify and eliminate time-consuming redundant calculations and their associated memory usage. MFRODO first proactively conducts model analysis to construct the dataflow graph and derive the I/O mapping of each block. Based on the gathered information, MFRODO recursively determines the precise calculation range of each block. For optimizable blocks, MFRODO eliminates the redundant calculations and reduces memory usage related to these calculations. Finally, MFRODO rebuilds the I/O mappings of the optimizable blocks to ensure the correctness of the generated code.

LLM-based Code Generation. While LLM-based code generation has shown promise in general-purpose programming tasks, they are not yet adopted for model-driven development and safety-critical scenarios, such as vehicle control system and aerospace system. Most LLM-based tools, e.g., Cursor [31] and Copilot [32], are trained on generic code

corpora, and designed to assist in manual coding. Thus, they lack native support for semantic interpretation of Simulink models. For example, Simulink models are stored as `.SLX` files, which encapsulate extensive semantics, such as model structures, blocks, parameter configurations, connections, etc. Unfortunately, LLM-based tools do not effectively understand these semantics, let alone generate deployable code. Moreover, LLMs are known to exhibit hallucination, i.e., producing plausible-looking but semantically incorrect code, which raises concerns in safety-critical scenarios where correctness guarantees are essential. In contrast, MFRODO implements a customized parser to extract critical model information from the target model and performs redundancy elimination based on I/O mappings and dataflow analysis. While LLM-based code generation is not yet suitable for model-to-code tasks, they could be valuable in future extensions of MFRODO, for example, code template development.

MFRODO vs FRODO. This paper is an extended and revised version of a preliminary conference paper [27] (FRODO). In terms of paper contributions, this paper introduced a memory elimination method to reduce the memory usage of the generated code. Based on the derived ranges of optimizable blocks, memory elimination first reduces the memory space related to the redundant calculations and records the new indices of inputs and outputs. Then, using these indices and block properties, memory elimination rebuilds the input/output mapping of the optimizable blocks. This ensures that valid output data are correctly mapped to the corresponding input data, thereby maintaining the correctness of the generated code. We collected the memory usage of the code generated by different code generators, to further validate the effectiveness of memory elimination. The results show that, BSS segment usage, which constitutes the majority of the memory usage in model code, is reduced by 12.00% - 52.71%, 13.64% - 47.76%, 13.64% - 47.76%, and 13.64% - 47.76%, compared with Simulink, DFSynth, HCG, and Frodo respectively.

VIII. CONCLUSION

In this paper, we propose MFRODO, an efficient and memory-sensitive code generator for data-intensive Simulink models via redundancy elimination. MFRODO first conducts model analysis to construct the dataflow graph and derive the I/O mapping of each block. Then, based on the collected information, MFRODO determines the precise calculation range of each block. After that, MFRODO generates streamlined code for optimization blocks by eliminating the redundant calculations and the corresponding memory usage, thereby improving the quality of the generated code. We evaluate the effectiveness of MFRODO on benchmark Simulink models across different compilers and architectures. The results show that compared with state-of-the-art code generators, MFRODO achieves 1.17× - 8.55× performance improvements and reduces BSS segment usage by 12.00% - 52.71%. Besides, MFRODO reduces compile time by 91.8% - 98.7% and code synthesis time by 94.3% - 99.6% compared with Simulink Embedded Coder, while consuming comparable overhead to DFSynth and HCG.

IX. ACKNOWLEDGMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2024YFF1401300).

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

[7] 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.

[8] 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.

[9] A. A. Giordano and A. H. Levesque, *Modeling of digital communication systems using SIMULINK*. John Wiley & Sons, 2015.

[10] H. Abu-Rub, A. Iqbal, and J. Guzinski, *High performance control of AC drives with Matlab/Simulink*. John Wiley & Sons, 2021.

[11] A. Martyanov, E. Solomin, and D. Korobatov, "Development of control algorithms in matlab/simulink," *Procedia engineering*, vol. 129, pp. 922–926, 2015.

[12] H. Le-Huy, "Modeling and simulation of electrical drives using matlab/simulink and power system blockset," in *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No. 37243)*, vol. 3. IEEE, 2001, pp. 1603–1611.

[13] M. L. Glossary, *Machine Learning*, 2023, https://machinelearning.wtf/terms/same-convolution/.

[14] S. Kotel, F. Sbiaa, M. Zeghid, M. Machhout, A. Baganne, and R. Tourki, "Performance evaluation and design considerations of lightweight block cipher for low-cost embedded devices," in *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*. IEEE, 2016, pp. 1–7.

[15] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Slnet: A redistributable corpus of 3rd-party simulink models," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 01–05.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] N. Horri and M. Pietraszko, "A tutorial and review on flight control co-simulation using matlab/simulink and flight simulators," *Automation*, vol. 3, no. 3, pp. 486–510, 2022.

[22] J. Friedman, "Matlab/simulink for automotive systems design," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–2.
[23] M. Gouasmi, M. Ouali, B. Fernini, and M. Meghatria, "Kinematic modelling and simulation of a 2-r robot using solidworks and verification by matlab/simulink," *International Journal of Advanced Robotic Systems*, vol. 9, no. 6, p. 245, 2012.
[24] R. Róka, "The environment of fixed transmission media and their negative influences in the simulation," *International Journal of Mathematics and Computers in Simulation*, vol. 9, pp. 190–205, 2015.
[25] Y. Cheng, Z. Yu, Z. Su, T. Chen, X. Zhang, and Y. Jiang, "Accmos: Accelerating model simulation for simulink via code generation," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3656218
[26] 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.
[27] Z. Yu, Z. Su, Y. Jiang, A. Cui, and R. Wang, "Efficient code generation for data-intensive simulink models via redundancy elimination," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3656217
[28] Z. Yu, Y. Yang, Z. Su, R. Wang, Y. Tao, and Y. Jiang, "Knight: Optimizing code generation for simulink models with loop reshaping," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 44, no. 2, p. 444–457, Aug. 2024. [Online]. Available: https://doi.org/10.1109/TCAD.2024.3438691
[29] J. Eker, J. Janneck, E. 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.
[30] 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.
[31] Anysphere, *Cursor*, 2025, https://www.cursor.com/.
[32] Github, *Copilot*, 2025, https://github.com/features/copilot.

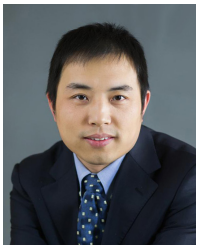
Zehong Yu received the B.S. degree in software engineering from Southeast University in 2021, and M.S.E. degree in software engineering from Tsinghua University in 2024. Currently, he is pursuing a Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests are 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.

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 currently works as an associate professor at Beihang University, Beijing, China. His research interests are in the areas of model driven development and software security.

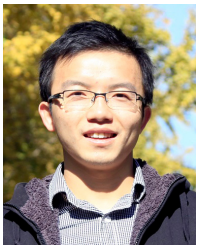
Haowei Qiu received the B.S. degree in software engineering from Tsinghua University, Beijing, China, in 2025. He is currently pursuing the M.S.E. degree in software engineering at the School of Software, Tsinghua University, Beijing, China. His research interests are in the area of model driven development.

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.



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.

Zhan Shu received the M.S. degree in computer science in 2018 and the Ph.D. degree in computer science in 2023, both from Binghamton University, State University of New York, USA. He is currently a postdoctoral researcher with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, and also a security researcher at NSFOCUS Technologies Group Co., Ltd. His research interests include AI-empowered security operations, automated penetration testing, and intelligent threat 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.