Efficient Code Generation for Data-Intensive Simulink Models via Redundancy Elimination

Zehong Yu[†], Zhuo Su[✉ †], Yu Jiang[✉ †], Aiguo Cui[‡], Rui Wang[§]
[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[‡] HUAWEI Technologies Co. LTD., Shanghai, China
[§] Information Engineering College, Capital Normal University, Beijing, 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 undermine the efficiency and performance of the generated code.

This paper proposes FRODO, an efficient code generator for data-intensive Simulink models via redundancy elimination. FRODO first conducts model analysis to construct the dataflow graph and derive the I/O mapping of each block. Then, for each block within the dataflow graph, FRODO recursively determines its calculation range by leveraging the I/O mapping of its subsequent blocks. After that, FRODO generates concise code for optimizable blocks in accordance with the precise calculation range. We implemented and evaluated FRODO on benchmark Simulink models. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by FRODO is 1.17× - 8.55× faster in terms of execution duration across different compilers and architectures, without incurring additional overhead of memory usage.

KEYWORDS

Simulink Models, Data-Intensive, Code Generation

1 INTRODUCTION

Simulink [14] has emerged as the fundamental infrastructure in embedded scenarios [2, 3, 6], which provides blocks to build the model that represents the target system. Based on the constructed model, Simulink supports model-based simulation, verification, and code generation to accelerate the development of embedded software. Code generation, as a crucial step in model-driven design, can generate the deployable source code for the target Simulink model, thus saving significant labor efforts and receiving major adoption in embedded software development.

Many research and commercial code generators have contributed to generating high-efficiency embedded code. One of the most commonly used commercial tools is Simulink Embedded Coder [13]. Based on the user-constructed model, it can generate deployable code and supports various optimization options, such as expression folding and variable reuse. Academic works also have made efforts to improve code efficiency. DFSynth [15] 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 [17] tries to accelerate the execution of time-consuming blocks with the model. It first discovers the parallel computation in the model and then synthesizes SIMD instructions for enhancing performance.

Although the aforementioned tools have shown promising results in various scenarios, they still exhibit limitations in generating code for data-intensive Simulink models. Data-intensive models, which accept arrays as inputs and perform extensive calculations on them, are prevalent in realistic embedded scenarios, including but not limited to real-time DSP systems and electric drives systems [4, 11, 12]. These models are related to loop statements in the deployed code, which are time-consuming and make up the majority of the computational overhead but are overlooked by existing generators that are mainly adept at accelerating individual statements within the generated code.

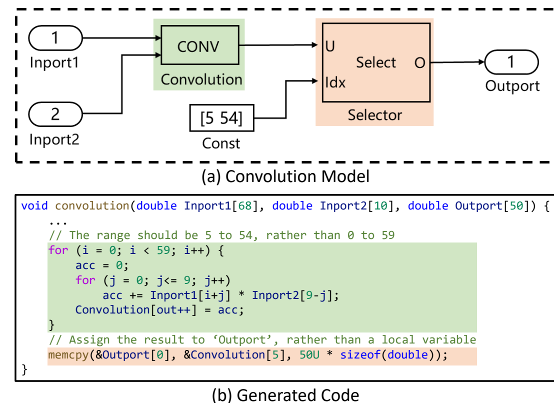

Figure 1: A sample model to illustrate the motivation of our work. The green part represents the code generated from the Convolution block; The orange part represents the code generated from the Selector block.

Specifically, within the data-intensive Simulink 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 [5] on the input data, as shown in Figure 1. The same convolution is a type of 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 highlighted in green. Therefore, a `Selector` block is needed to select the part of `Convolution`

Yu Jiang and Zhuo Su are the corresponding authors.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3656217

block's output that represents the same convolution as shown in the code highlighted in orange, resulting in unnecessary calculations. 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 efficiency of the embedded software. Moreover, considering the constrained performance capabilities of embedded devices, such inefficiency is particularly problematic and deemed unacceptable [10].

To generate efficient code for data-intensive models by eliminating inherently redundant calculations, we have to deal with the following two challenges: ① The first challenge involves 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 involves accurately eliminating the redundant calculations within the optimizable blocks. For optimization, it is of vital importance to ensure both efficiency and correctness. 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.

To address the above challenges, we propose FRODO, an efficient code generator for data-intensive Simulink models via redundancy elimination. Firstly, FRODO extracts essential details from the target Simulink model, including blocks and connections. Based on the collected information, FRODO constructs the dataflow graph and derives the I/O (input-output) mapping of each block. Secondly, for each block within the dataflow graph, FRODO recursively determines its calculation range by leveraging the I/O mapping of its subsequent blocks. After that, FRODO generates concise code for optimizable blocks in accordance with the accurate calculation range. This code is then cooperated with the code generated from other basic blocks, yielding high-efficiency code for deployment.

We implement and evaluate the effectiveness of FRODO on benchmark Simulink models [15, 18], across different compilers and architectures. The results illustrate that FRODO gains pronounced performance improvement. Compared with the state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, the code generated by FRODO is 1.26× - 8.55×, 1.32× - 5.75×, and 1.17× - 3.75× faster in terms of execution duration, without incurring additional overhead of memory usage.

2 BACKGROUND AND RELATED WORK

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 [1, 7–9]. This approach usually encompasses four critical stages: model construction, model simulation, model verification, and code generation. Each stage serves a unique purpose and contributes to the overall efficiency and effectiveness of the model-driven design process.

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 [16]. ① Model parse, as a preparation stage, analyzes the target model to collect the critical information for further usages, such as the model structure and blocks. ② Dataflow analysis derives the sequential relationship and connectivity between blocks. ③ Scheduling infers the translation sequence of model blocks, based on the sequential relationship. ④ Code synthesis generates corresponding code for each block, and then assembles them into deployable code according to the translation sequence.

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 [13] specializes in generating production-quality code by employing various high-level optimization techniques, including expression folding, variable reuse, etc. DFSynth [15] 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. HGC [17] accelerates the generated code by selecting optimal implementations for intensive computing blocks and synthesizing appropriate SIMD instructions for batch computing blocks. Mercury [18] prioritizes adjusting the code translation order and avoiding instruction pipeline stalls.

Main Difference. FRODO differs from these works by utilizing model semantics to avoid time-consuming redundant calculations. FRODO proactively conducts model analysis to construct the dataflow graph and derive the I/O mapping of each block. Then, based on the gathered information, FRODO recursively determines the precise calculation range of each block. For optimizable blocks, FRODO generates streamlined code following the eliminated calculation range, thereby eliminating the redundant calculations.

3 DESIGN

FRODO mainly contains two key components: Model Analysis and Redundancy Elimination, as shown in Figure 2. ① Firstly, FRODO parses the target model to collect critical information, including blocks, connections, etc. Then, based on the connections and block property library, FRODO constructs the dataflow graph and derives the I/O mapping of each block. ② Secondly, for each block within the dataflow graph, FRODO recursively determines its calculation range by leveraging the I/O mapping of its subsequent blocks. After that, FRODO utilizes the element-level code library to generate concise code for optimizable blocks. This code is then synthesized with code generated from other blocks, yielding high-efficiency code.

3.1 Model Analysis

Model Parse. FRODO implements a customized parser to extract critical model information from the target Simulink 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. FRODO interprets these files to parse the dataflow information, such as blocks and connections. Besides, for `Subsystem` blocks within the model, FRODO flattens them, and maps their inports and outports to the corresponding external blocks for further analysis.

Dataflow Graph Construction. FRODO first constructs the dataflow graph. For each block, FRODO defines the appropriate runtime data structure to preserve critical contents, such as inputs and outputs. For connections, FRODO 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.

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, FRODO 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 actors 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` 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 index port, the contained mapping should undergo a change. Therefore, FRODO 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. FRODO 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, FRODO constructs a comprehensive I/O mapping for the `Selector` block. Leveraging this derived I/O mapping, redundant calculations can be effectively identified and eliminated.

3.2 Redundancy Elimination

Simulink supports data-truncation blocks for modeling purposes, including but not limited to `Selector`, `Pad`, and `Submatrix`. 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.

Calculation Range Determination. Leveraging the obtained dataflow graph and I/O mappings, FRODO employs a recursive method to identify and eliminate the redundant calculations mentioned above. Algorithm 1 presents the overall procedure of calculation range determination. The main idea of this algorithm is to initially determine the calculation range of the child blocks, which are then employed to determine the calculation range of their parent blocks. First, FRODO traverses the dataflow graph to identify the root blocks and records them for further usage (lines 2-7). The root block is defined as the 0-in-degree block in the dataflow graph. Since these blocks provide the source data for all calculations, determining their precise calculation range is crucial for eliminating redundant calculations. For each block, FRODO defines a map called *range* to record the calculation range, which is initially set to be an empty set (line 8). Then, FRODO invokes the `recursive` function to determine the precise calculation range of root blocks, as well as their subsequent blocks (lines 9-11). After that, FRODO identifies the blocks whose calculation ranges have been modified as optimizable blocks and generates streamlined code for them.

Algorithm 1: Calculation Range Determination

```
Input: graph: Dataflow graph of the target model
       mapping : I/O mapping of each block
Output: range: Calculation range of each block
1  Function rangeDetermine(graph, mapping):
2      roots ← ∅ // root blocks
3      for block in graph do
4          if block is root then
5              roots.append(block)
6          end
7      end
8      range ← ∅
9      for block in roots do
10         recursive(graph, mapping, range, block)
11     end
12     return range
13 End Function
14 Function recursive(graph, mapping, range, block):
15     b_c ← graph[block].child // child blocks
16     if b_c = ∅ then
17         range[block] ← mapping[block.output]
18     end
19     else
20         r_o ← ∅ // output range
21         for child in b_c do
22             recursive(graph, mapping, range, child)
23             r_o ← r_o ∪ range[child]
24         end
25         range[block] ← mapping[r_0]
26     end
27 End Function
```

The procedure of `recursive` function is represented in lines 14-27 in Algorithm 1. First, FRODO searches the dataflow graph to obtain child blocks of *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*, FRODO derives the input ranges and assigns them to the respective entry in *ranges* (lines 16-18). Conversely, if $b_c \neq \varnothing$, FRODO must consider the influence of the child blocks. To do this, FRODO first defines a variable called r_o to store the output range of *block* (line 20). Then, for each child block, FRODO invokes the `recursive` function to drive the ranges of their inputs (line 22). Note that, 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, FRODO can merge the corresponding input range of the child blocks to derive the accurate output range of *block* (line 23). Subsequently, FRODO determines the precise calculation range (line 25).

Concise Code Generation. For optimizable blocks, FRODO first obtains a suitable code snippet for replacement from the element-level code library, according to the calculation range. Then, FRODO replaces the placeholders in the selected code snippet with the actual values according to the block parameters. 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 elements, while the snippet ② is utilized to generate code for consecutive elements. Depending on the calculation

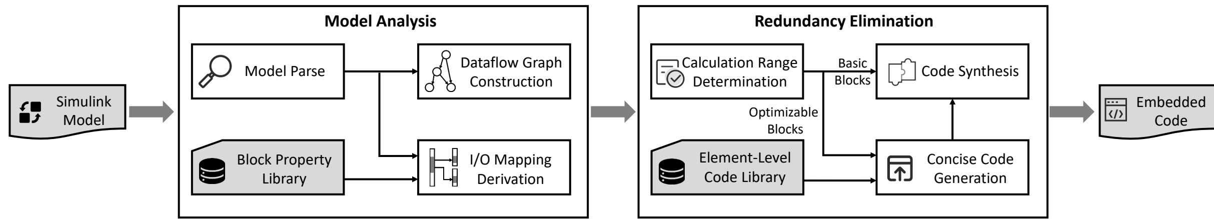

Figure 2: An overall framework of Frodo. Model Analysis is mainly used as a pre-processing step to extract critical information from the target model for subsequent optimization. Based on the gathered information, Redundancy Elimination is utilized to eliminate redundant calculations to improve performance.

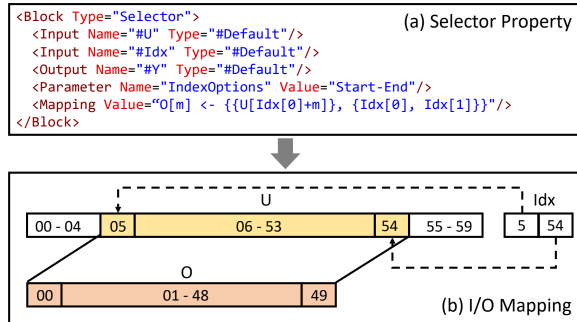

Figure 3: An example of I/O mapping derivation. The subfigure (a) is Selector block's property, while the subfigure (b) shows the mapping between inputs and output.

range, Frodo selects an appropriate code snippet and replaces the placeholders with the corresponding actual values. For instance, \$Input2_size\$ is replaced with the size of the second input of the Convolution block.

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;
}
```

Figure 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.

Code Synthesis. First, Frodo determines the translation sequence of the blocks by employing a topological-based method. Then, for basic blocks, Frodo supports customized DLL (Dynamic Link Library) files to generate the corresponding 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, Frodo identifies the root blocks for optimization, i.e., block ①, block ②, and block ③. For each root block, Frodo recursively determines its calculation range as well as subsequent blocks. For instance, consider the Step 1. Frodo first determines the calculation range of actor ⑥, as it has no child actors. After that, Frodo determines the calculation range of actor ⑤, given that actor ⑥ is connected to actor ⑤. Then, Frodo determines the calculation range of actor ④ from [0, 59] to [5, 54], as it connects to a data-truncation actor (actor ⑤). Finally, Frodo repeats the aforementioned step until the calculation range of actor ① is determined. Based on the calculation ranges, Frodo classifies blocks into basic blocks and optimizable blocks. Frodo generates concise code for optimizable blocks in accordance with the eliminated calculation range. This code is then synthesized with code generated from basic blocks, yielding high-efficiency embedded code.

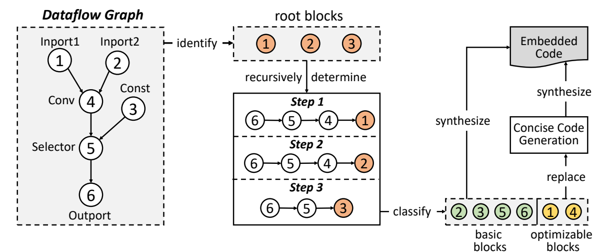

Figure 5: Illustration of redundancy elimination.

4 EVALUATION

Implementation: We have developed Frodo [1] in C++ with 24,156 lines of code. The tool supports numerous blocks, including math operation blocks, matrix operation blocks, complex blocks, etc. For each supported block, we manually developed the corresponding block property library and element-level code library for optimization, according to the block type and parameters. These libraries are recorded as external files to support cross-architectures.

Evaluation Setup: To investigate the effectiveness of our approach, we compared Frodo with three state-of-the-art code generators, Simulink Embedded Coder [13], DFSynth [15], and HCG [17]. The comparison experiments were conducted across different architectures with two C-Compilers, GCC and Clang. We evaluated the performance of Frodo on a benchmark of 10 commonly used data-intensive Simulink models collected from industry [17, 18], as shown in Table 1. We also generated a large number of random test

[1]The implementation and the results are represented at the repository.https://anonymous.4open.science/r/Frodo-2E05/

cases for the code generated by Frodo and compared the results with those from model simulations. The consistency between them underscores the correctness of Frodo.

Table 1: 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

4.1 Effectiveness on Benchmark Models

To demonstrate the performance of the code generated by Frodo, we conducted the experiment on the benchmark models. The experiment environment was an experimental machine (Win11, AMD Ryzen 7 5800X, 32GB memory). The generated code was compiled by the state-of-the-art compilers, i.e., GCC (v11.3.0) and Clang (v14.0.6), employing the `-O3` optimization flag. In addition, to eliminate statistical errors, the generated code was repeatedly executed 10,000 times to obtain the average results for comparison.

Table 2: Comparison of the code execution duration on x86 using GCC and Clang compilers, both compiled with `-O3` flag.

Model	GCC				Clang			
	Simulink	DFSynth	HCG	Frodo	Simulink	DFSynth	HCG	Frodo
AudioProcess	1.583s	0.492s	0.517s	**0.333s**	1.574s	0.583s	0.419s	**0.202s**
Decryption	0.370s	0.303s	0.261s	**0.213s**	0.370s	0.211s	0.184s	**0.119s**
HighPass	0.865s	0.291s	0.326s	**0.160s**	0.558s	0.323s	0.307s	**0.182s**
HT	0.651s	0.715s	0.650s	**0.311s**	0.711s	0.753s	0.743s	**0.317s**
Kalman	0.370s	0.266s	0.260s	**0.201s**	0.400s	0.333s	0.311s	**0.223s**
Back	0.304s	0.451s	0.699s	**0.241s**	0.789s	0.536s	0.759s	**0.250s**
Maintenance	0.931s	0.295s	0.386s	**0.223s**	0.859s	0.343s	0.271s	**0.189s**
Maunfacture	2.251s	0.973s	0.658s	**0.486s**	3.449s	1.114s	0.883s	**0.526s**
RunningDiff	0.708s	0.722s	0.193s	**0.125s**	0.576s	0.589s	0.195s	**0.118s**
Simpson	0.949s	0.428s	0.433s	**0.266s**	1.385s	0.551s	0.409s	**0.248s**

Table 2 shows the experiment results and performance improvement. For comparison experiments compiled with GCC, the execution duration of Frodo 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 Frodo is 1.79× - 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, Frodo achieves a significant performance improvement. This demonstrates the effectiveness and practicability of our approach.

The performance of the code generated by Simulink is relatively limited. Simulink indeed employs some optimization techniques, including SIMD instruction utilization and expression collapse. However, it usually fails to effectively identify the target blocks to apply SIMD instructions, resulting in limited performance. As for expression collapse, 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.

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, Frodo strategically exploits critical information, specifically the dataflow graph and I/O mapping. This enables Frodo to accurately determine the calculation 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 within the generated code, 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.

4.2 Effectiveness on Different Architectures

To demonstrate the effectiveness of Frodo on different architectures, we conducted repetitive experiments on an industrial 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.

Figure 6 shows the execution improvement of the code generated by Frodo compared to other code generators on the ARM architecture. Each bar represents the performance improvement achieved by Frodo compared to the corresponding code generator. Compared to previous experiments on the x86 architecture, Frodo achieves better performance improvement on the ARM architecture. For comparison experiments compiled with GCC, the execution duration of Frodo is 1.71× - 8.55× faster than Simulink, 1.44× - 4.10× faster than DFSynth, and 1.17× - 3.75× faster than HCG. As for compiling with Clang, the execution duration of Frodo is 1.68× - 6.46× faster than Simulink, 1.40× - 2.85× faster than DFSynth, and 1.34× - 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 performance, the major performance bottlenecks come from the logic of the generated code. Frodo benefits from this and achieves a better performance improvement.

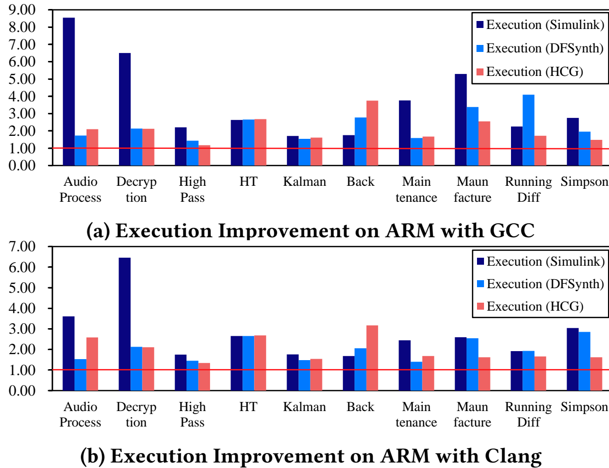

Figure 6: The execution improvement of the code generated by Frodo versus other code generators on ARM. The red line shows the execution duration of Frodo as the baseline.

5 DISCUSSION

To fully evaluate our approach, we conducted experiments in terms of memory. Our findings reveal that the code generated by the different code generators consumes a comparable amount of memory during execution. Given that they use the same quantity of variables and abstain from memory allocation functions such as `malloc`, the heap size among the compiled files remains consistent. Since all codes are compiled with the same compiler, their stack memory is consistent. Other memory metrics have little impact due to the relatively small size of the target files. This shows that Frodo enhances code performance without incurring memory overhead.

Besides the above discussions, there are some threats to validity. For complex blocks, such as the `Convolution` block, Frodo should generate multiple instances of code for the same block type in accordance with their distinct calculation range. This results in longer code relative to other code generators. Frodo can avoid such code duplication by generating a generic function interface and configuring the derived calculation range as parameters. For blocks with discontinuous calculation ranges, Frodo generates appropriate code and distinct variables for each discrete calculation range. This may hinder the effective utilization of advanced optimizations, such as SIMD instructions. To address this, Frodo can allocate a continuous memory space for these distinct variables.

6 CONCLUSION

In this paper, we propose Frodo, an efficient code generator for data-intensive Simulink models via redundancy elimination. Frodo first conducts model analysis to construct the dataflow graph and derive the I/O mapping of each block. Then, based on the collected information, Frodo determines the precise calculation range of each block and generates concise code for optimizable blocks. We evaluate the effectiveness of Frodo on benchmark Simulink models across different compilers and architectures. The results show that compared with state-of-the-art code generators Simulink Embedded Coder, DFSynth, and HCG, Frodo achieves significant performance improvement without additional memory usage.

7 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] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. 2006. Developing applications using model-driven design environments. *Computer* 39, 2 (2006), 33–40.
[2] David Christhilf and Barton Bacon. 2006. Simulink-Based Simulation Architecture for Evaluating Controls for Aerospace Vehicles (SAREC-ASV). In *AIAA Modeling and Simulation Technologies Conference and Exhibit*. 6726.
[3] RH Ali Faris, AA Ibrahim, MM Abdulwahid, MF Mosleh, et al. 2021. Optimization and Enhancement of Charging Control System of Electric Vehicle Using MATLAB SIMULINK. In *IOP Conference Series: Materials Science and Engineering*, Vol. 1105. IOP Publishing, 012004.
[4] Arthur A Giordano and Allen H Levesque. 2015. *Modeling of digital communication systems using SIMULINK*. John Wiley & Sons.
[5] Machine Learning Glossary. 2023. *Machine Learning*. https://machinelearning.wtf/terms/same-convolution/.
[6] Darko Hercog and Karel Jezernik. 2005. Rapid control prototyping using MATLAB/Simulink and a DSP-based motor controller. *International Journal of Engineering Education* 21, 4 (2005), 596.
[7] Jean-Marc Jézéquel. 2008. Model driven design and aspect weaving. *Software & Systems Modeling* 7 (2008), 209–218.
[8] Yu Jiang, Han Liu, Houbing Song, Hui Kong, Rui Wang, Yong Guan, and Lui Sha. 2018. Safety-assured model-driven design of the multifunction vehicle bus controller. *IEEE Transactions on Intelligent Transportation Systems* 19, 10 (2018), 3320–3333.
[9] Yu Jiang, Hehua Zhang, Huafeng Zhang, Xinyan Zhao, Han Liu, Chengnian Sun, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2014. 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*. 711–714.
[10] Sonia Kotel, Fatma Sbiaa, Medien Zeghid, Mohsen Machhout, Adel Baganne, and Rached Tourki. 2016. 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, 1–7.
[11] Hoang Le-Huy. 2001. 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, 1603–1611.
[12] AS Martyanov, EV Solomin, and DV Korobatov. 2015. Development of control algorithms in Matlab/Simulink. *Procedia engineering* 129 (2015), 922–926.
[13] Simulink. 2023. *Simulink Embedded Coder Documentation*. https://www.mathworks.com/solutions/embedded-code-generation.html.
[14] Simulink and Matlab. 2023. *Simulink Documentation*. https://www.mathworks.com/help/simulink/index.html.
[15] Zhuo Su, Dongyan Wang, Yixiao Yang, Yu Jiang, Wanli Chang, Liming Fang, Wen Li, and Jiaguang Sun. 2021. Code Synthesis for Dataflow-Based Embedded Software Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 1 (2021), 49–61.
[16] Zhuo Su, Dongyan Wang, Yixiao Yang, Zehong Yu, Wanli Chang, Wen Li, Aiguo Cui, Yu Jiang, and Jiaguang Sun. 2021. MDD: A Unified Model-driven Design Framework for Embedded Control Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).
[17] Zhuo Su, Zehong Yu, Dongyan Wang, Yixiao Yang, Yu Jiang, Rui Wang, Wanli Chang, and Jiaguang Sun. 2022. HCG: optimizing embedded code generation of simulink with SIMD instruction synthesis. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1033–1038.
[18] Zehong Yu, Zhuo Su, Yixiao Yang, Jie Liang, Yu Jiang, Aiguo Cui, Wanli Chang, and Rui Wang. 2022. Mercury: Instruction Pipeline Aware Code Generation for Simulink Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4504–4515.