Synthesizing Hardware-Specific Instructions for Efficient Code Generation of Simulink

Zehong Yu
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

Zhuo Su*
School of Software,
Beihang University
Beijing, China

Rui Wang
Information Engineering College,
Capital Normal University
Beijing, China

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

Abstract

Simulink has become a pivotal tool in embedded scenarios, offering a model-driven approach for embedded software development. Given the tight performance and resource constraints in embedded applications, it is crucial to ensure the efficiency of the code generated from Simulink models. Code generators implement various optimizations to enhance performance. However, they neglect the potential of hardware-specific instructions available in modern processors, such as saturation-type instructions, which accomplish complex operations in fewer cycles. Moreover, relying on state-of-the-art compilers to use these instructions is also not as effective as expectation, due to their complex semantics.

This paper proposes Amica, an efficient code generator for Simulink models with hardware-specific instruction synthesis. The key insight of Amica is to leverage model semantics to effectively synthesize the appropriate instructions. Amica first converts the model into the dataflow graph and crafts a series of optimization rules represented as dataflow subgraph with constraints related to block parameters, data types, and other critical properties. Then, Amica iteratively matches these rules with dataflow graph to obtain the optimizable candidates. The candidate that maximizes latency reduction is chosen to update the dataflow graph. Finally, Amica synthesizes the appropriate instructions for optimizable blocks in accordance with instruction syntax and block properties. We implemented and evaluated Amica on benchmark Simulink models. Compared with the state-of-the-art code generators Simulink Embedded Coder, Mercury, and Frodo, the code generated by Amica is 1.29× - 8.36× faster in terms of execution time across different platforms. Besides, Amica reduces 6% - 53% assembly code size of the compiled programs, while performing similarly in terms of data segment size and BSS segment size.

*Zhuo Su and Yu Jiang are the corresponding authors.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764536

Keywords

Simulink, Code Generation, Hardware-specific Instruction

ACM Reference Format:
Zehong Yu, Zhuo Su, Rui Wang, and Yu Jiang. 2026. Synthesizing Hardware-Specific Instructions for Efficient Code Generation of Simulink. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3764536

1 Introduction

Simulink [22] has become a pivotal tool in embedded scenarios, offering a model-driven approach to design embedded systems, such as aerospace design, automotive system, and healthcare system [6, 8, 17, 29]. Code generation plays a crucial role in model-driven design, which automatically transforms high-level models into embedded code, thereby eliminating manual efforts and reducing coding errors. Embedded devices inherently have low-power and high real-time features, making it essential for applications to meet specific energy and performance goals [28]. Thus, generating efficient code for embedded scenarios is both critical and challenging.

To meet these requirements, recent works have developed several effective approaches to ensure code efficiency. Simulink Embedded Coder [7], the built-in toolkit, offers various options for optimization. For instance, it implements expression folding and variable reuse to reduce redundant assignments, and merges loop constructs to minimize conditional evaluations for improved performance. DFSynth [23] aims to optimize complex branch blocks in Simulink models by decomposing the target model into blocks embedded within control statements and generating tailored code templates for each block. Other works have leveraged hardware features to accelerate code execution. For example, Mercury [33] adjusts the code translation order to prevent data hazards, thereby improving instruction pipeline utilization and overall code performance.

Despite their advancements, these state-of-the-art code generators overlook the importance of hardware-specific instructions, resulting in suboptimal code efficiency. In other words, they fail to identify optimizable blocks within the target model and synthesize appropriate hardware instructions for these blocks. Modern processors include specialized instructions in their instruction sets, that perform complex operations requiring multiple regular instructions. For example, Tricore [9], developed by Infineon [1] and widely deployed in automotive applications, provides a range of specialized

instructions aimed at meeting stringent real-time requirements, e.g., `QSEED.F` for calculating the reciprocal square root. Such instructions are implemented in hardware and can minimize execution latency compared to software-based implementations.

Moreover, relying on the compilers to use these hardware-specific instructions is not as effective as expectation. State-of-the-art compilers, e.g., GCC and Clang, use pattern-matching for instruction selection, and have difficulty in exploiting complex hardware-specific instructions, e.g., saturation-type instructions. The intricate natures of these instructions, e.g., complex control flows and various variants code sequences, make it difficult to develop universal matching strategies [18]. Our primitive experiments also show that using hardware-specific instructions through inline assembly achieves significant performance improvement under high-level compiler optimization (see §3).

In fact, the Simulink model contains rich semantics, e.g., parameters and connections, and they can bring benefits when synthesizing hardware-specific instructions. For instance, saturation is an inherent parameter of Simulink block and can be used to determine whether saturation-type instructions should be synthesized. However, to effectively utilize model semantics for hardware-specific instruction synthesis, we must address the following two challenges: (1) **The first challenge** is to develop precise optimization rules and effectively use them for instruction synthesis. Given the rich model semantics, optimization rules must account for not only the properties of blocks, but also for their connections. However, it is necessary to exact critical properties from a large set of less relevant ones, and optimization rules often require analyzing connections among multiple blocks, leading to a vast search space. Additionally, since the same block may relate to multiple optimization rules, selecting the appropriate rules is essential for maximizing performance improvement. (2) **The second challenge** is to design a coarse-grained framework for optimization. Different hardware platforms offer functionally equivalent but semantically different instructions. Crafting distinct optimization rules for each instruction can be both tedious and error-prone. Moreover, synthesizing appropriate instructions requires careful consideration of their specific semantics. Therefore, developing a unified optimization scheme that effectively handles the diversity of instruction variants and model semantics is challenging.

To address the aforementioned challenges, we introduce Amica, an efficient code generator for Simulink models via hardware-specific instruction synthesis. First, Amica parses the model to obtain essential contents for constructing dataflow graph of the target model, such as block properties and connections. Meanwhile, Amica crafts a series of optimization rules, represented as dataflow subgraph with constraints related to block parameters, data types, and other critical properties. This process filters out irrelevant content for subsequent optimization. For the functionally equivalent but semantically different instructions, Amica merges them into the same optimization rule, while developing distinct optimization rules for platform-specific instructions. After that, Amica selects applicable optimization rules in accordance with the target platform, and iteratively matches them with dataflow graph to identify optimization candidates. The candidate that maximizes latency reduction is selected to update the dataflow graph, with the updated blocks labeled as optimizable. Finally, Amica synthesizes the appropriate instructions for these optimizable blocks, taking into account instruction syntax and block properties. This synthesized code is then integrated with code generated from other basic blocks in accordance with the translation sequence, yielding high-efficiency code for deployment.

We implemented and evaluated the effectiveness of Amica on benchmark Simulink models [23, 33], across different platforms. The results demonstrate that Amica gains pronounced performance improvement. Compared with the state-of-the-art code generators Simuink Embedded Coder, Mercury, and Frodo, the code generated by Amica is 1.29× - 5.12×, 1.69× - 8.33×, and 1.68× - 8.36×, in terms of execution time. We also collected the assembly code size, data segment size, and BSS segment size of the compiled programs. The statistics show that compared with Simulink Embedded Coder, Mercury, and Frodo, Amica reduces the assembly code size by 6% - 53%, 6% - 50%, and 9% - 49%, respectively, while performing similarly in terms of data segment size and BSS segment size.

In summary, this paper makes the following contributions:

- We proposed and implemented Amica, an efficient code generator through hardware-specific instruction synthesis. Based on model semantics, Amica crafts optimization rules and iteratively selects the appropriate rule for optimization to minimize execution latency.
- We evaluated Amica on benchmark models. The results show that Amica achieves significant performance improvement and effectively reduces the assembly code size.

2 Background

2.1 Model-Driven Design and Simulink

Model-driven design is a software development approach that prioritizes the creation, refinement, and manipulation of models as the primary artifacts during the design and implementation process. By employing model as the high-level abstraction, it allows developers to concentrate on system functionality and architecture, rather than low-level code and implementation. This leads to the widespread adoption of model-driven design in the embedded systems domain, particularly for the design of complex control systems [4, 10–12].

Simulink [22], a part of the MATLAB software suite, is the most widely used model-driven design tool, which facilitates the development of various embedded and real-time scenarios, including but not limited to aerospace design, automotive system, and healthcare system [6, 8, 15, 17, 29, 31]. It provides a graphical programming environment for users to develop and analyze the target dynamic systems [25, 27], and implements corresponding toolsets for four critical model-driven design stages mentioned above.

2.2 Code Generation

Code generation is the centerpiece of model-driven design, which automatically converts the target model into embedded code for deployment. On the one hand, it releases the developers from tedious and error-prone coding tasks, thereby improving software development efficiency. On the other hand, the quality and correctness of the generated code should be ensured, as it directly impacts the effectiveness of the target software and incorrect code can lead to undefined behaviors and unexpected outputs.

In general, code generation consists of three key steps: model parse, schedule convert, and code synthesis [24]. ① Model parse, as the preparation step, interprets the target model to collect the critical information for subsequent usage, including blocks, parameters, and connections. Such information describes the hierarchical architecture, dataflow, and input-output relationships within the model, as well as constraints or dependencies essential for scheduling and code synthesis. ② Schedule convert first derives the sequential relationship and connectivity between blocks in accordance with the connections, and then conducts topology-based analysis to obtain the translation sequence of model blocks. Note that, as each iteration of topology-based analysis may have multiple candidate blocks, this results in the presence of multiple equivalent translation sequences. ③ Code synthesis is responsible for generating the corresponding code for each block in accordance with its functionality and parameters. It then assembles the code for all blocks into deployable code based on the derived translation sequence. As long as the code generator preserves correctness and adheres to the original model semantics, it can customize each block's implementation to enhance the performance of the generated code.

3 Motivation

Modern processors integrate specialized instructions alongside general-purpose ones to enhance performance-critical applications, such as real-time vehicle system, digital signal processing, and artificial intelligence [13, 19, 20]. These specialized instructions can perform complex operations that typically require multiple regular instructions, thereby reducing execution latency and improving system efficiency. For example, consider calculating the square root of a floating-point number. Software-based implementation of this operation typically uses iterative algorithms such as Newton's Method [3]. These algorithms often involve multiple arithmetic operations, including additions, multiplications, and divisions, and take several processor cycles to converge to a precise result. In contrast, square root can be calculated in fewer cycles when utilizing dedicated hardware instructions, i.e., `QSEED.F` in Tricore, `FSQRT` in ARM, and `FSQRT.S` in RISC-V.

Motivation Example. We introduce a sample model shown in Figure 1 to quantitatively illustrate the benefit of hardware-specific instructions in code generation. The left code snippets show the code generated by Simulink Embedded Coder for calculating reciprocal square root (highlighted in blue), calculating absolute value difference (highlighted in green), and performing saturation addition (highlighted in orange). The right counterparts are the corresponding instructions written in inline assembly on the Tricore platform. We compiled all the code versions using GCC with the `-O3` flag enabled. The experimental results show that incorporating hardware-specific instructions yields significant performance improvement. Moreover, this suggests that even state-of-the-art compiler with the highest optimization fail to synthesize these instructions for speedup. Although handwriting inline assembly can satisfy performance requirements in some cases, its steep requirements for expert knowledge of the target instruction set architecture, along with its tedious and error-prone nature, makes it less practical in most cases. Therefore, it is urgent to develop an effective method to introduce hardware-specific instructions in code generation for performance improvement.

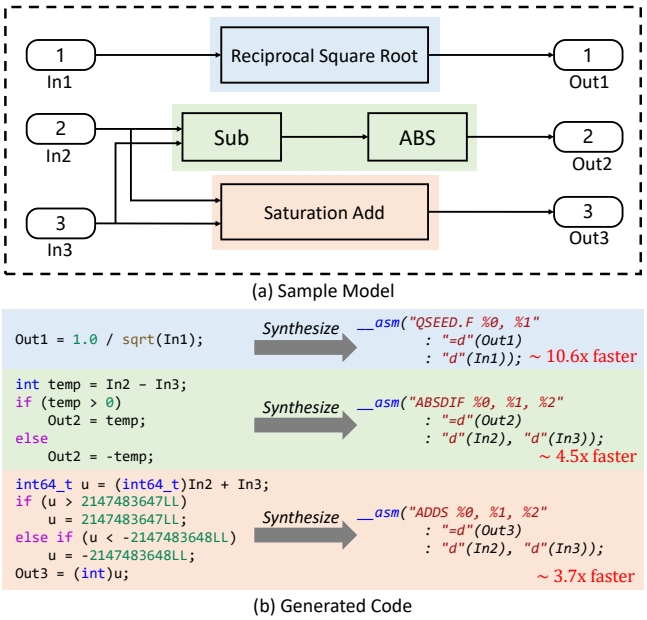


Figure 1: A sample model to illustrate the benefits of hardware-specific instructions. The left code snippets are generated by Simulink Embedded Coder, while the right code snippets are the corresponding hardware-specific instructions on the Tricore platform.

Existing Approaches. Code generators [7, 23, 32, 33] have made notable efforts to generate high-efficiency code. For example, Simulink Embedded Coder [7] design various optimizations, such as expression folding and loop fusion, to avoid redundant assignments and judgments, while Mercury [33] adjusts the translation sequence to decrease instruction pipeline stalls. However, they overlook the potential of hardware-specific instructions for performance improvement, and fail to synthesize these instructions in performance-critical scenarios. Furthermore, state-of-the-art compilers, such as GCC, which typically rely on pattern-matching for instruction selection, also have difficulty in exploiting intricate hardware-specific instructions, e.g., saturation-type instructions and rounding-type instructions, due to their complex semantics: complex control flows and various code sequence variants [18]. In summary, existing approaches fail to synthesize hardware-specific instructions for the target model, limiting the performance of the generated code.

Observation. We observed that considering hardware-specific instruction synthesis from a model-centric perspective can effectively address the above issues. In other words, model contains rich high-level semantics, e.g., block parameters and connectivity, and these semantics provide benefits when synthesizing hardware-specific instructions during code generation. For instance, saturation is an inherent parameter of Simulink block which can be utilized to determine whether saturation-type instructions should be synthesized, and analyzing connections between blocks allows us to synthesize instructions like `ABSDIF` in Tricore without the need to account for complex control flows.

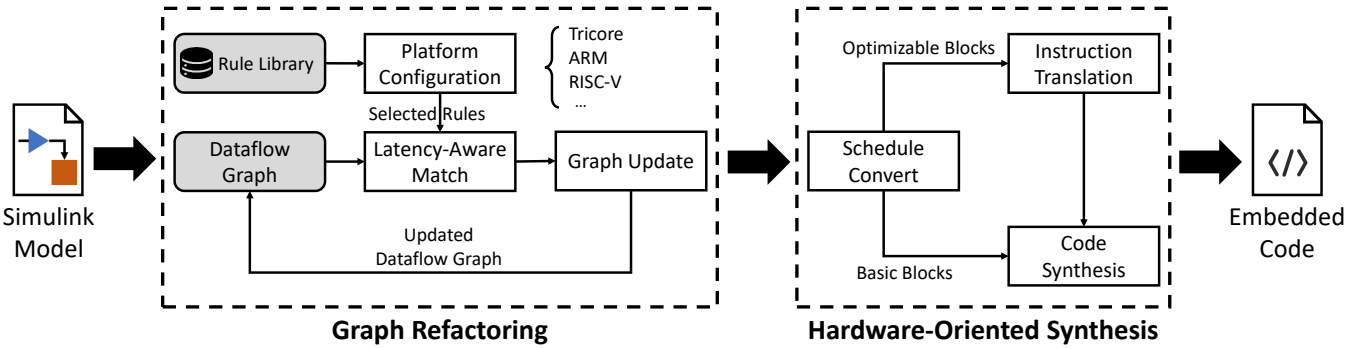


Figure 2: The overview of Amica. It mainly contains two key components. (1) Graph Refactoring focuses on iteratively using appropriate optimization rules to update the dataflow graph. (2) Hardware-Oriented Synthesis is utilized to synthesize corresponding instructions for optimizable blocks and generate embedded code.

This motivates us to design Amica, which aims to leverage model semantics to effectively synthesize appropriate hardware-specific instructions to maximize performance improvement. Amica proactively crafts optimization rules based on the critical model semantics, including block parameters, connections, data types, etc., and then matches these rules with the model dataflow graph. To maximum performance improvement, Amica iteratively selects the rule which minimizes execution latency for instruction synthesis, during the optimization process.

4 Design

In this section, we detail the design of Amica, an efficient code generator by synthesizing hardware-specific instructions to enhance performance. Figure 2 shows the overall framework of Amica, which consists of two key components. (1) **Graph Refactoring:** Amica first parses the Simulink model to obtain essential contents including block properties, connections, inports/outports, etc, and utilizes them to construct dataflow graph as the preparation step. Meanwhile, Amica crafts a series of optimization rules represented as dataflow subgraphs with constraints related to block parameters, data types and other block properties. For the functionally equivalent but semantically different instructions, Amica merges them into the same optimization rule, while developing distinct optimization rules for platform-specific instructions. Based on the target platform, Amica selects applicable rules from rule library and matches them with dataflow graph to obtain optimization candidates. The candidate that maximizes latency reduction is chosen to update the dataflow graph, which will be used in next optimization iteration. The updated blocks are labeled as optimization blocks. (2) **Hardware-Oriented Synthesis:** For the refactored dataflow graph, Amica conducts topology-based analysis on the connections between blocks to derive translation sequence. For optimizable blocks, Amica synthesizes the corresponding instructions for translation in accordance with instruction syntax and block properties such as data type and parameters. For basic blocks, Amica directly generates code based on their functionalities. After that, the generated code snippets are integrated together according to the translation sequence, yielding high-efficiency embedded code for deployment.

4.1 Graph Refactoring

Preparation. The Simulink model is encapsulated as a `Zip` file containing various components, including graphical structures and model parameters. Amica first unzips the target model, and then parses the obtained files to extract critical dataflow information for constructing the dataflow graph, such as blocks, inports/outports, and connections, and incorporates block properties into the corresponding nodes of the dataflow graph, including data type, data length, and parameters.

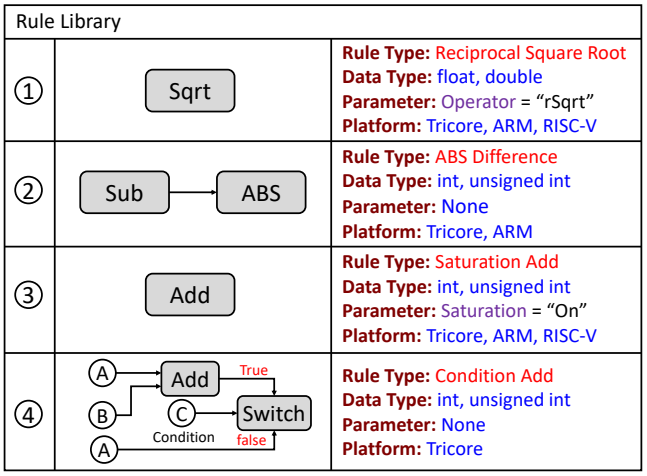


Figure 3: Typical optimization rules in rule library. The left parts are dataflow subgraphs of the corresponding rules, while the right parts are the required constraints.

To filter irrelevant contents for optimization, Amica crafts all optimization rules based on the dataflow graph and associated block properties, and encapsulates them into a rule library for subsequent usage. Specifically, each optimization rule is essentially a dataflow subgraph with constraints on block parameters, data types, and other properties relevant to instruction synthesis. Figure 3 shows some typical optimization rules in the rule library. In these rules, the left parts represent the dataflow subgraphs, while the right parts are the required constraints. Using the 'Saturation Add' rule as an

illustrative example. The subgraph for this rule is a Add block with constraints requiring that the block's 'saturation' parameter is enabled and the data type is either 'int' or 'unsigned int'. Besides, for the functionally equivalent but semantically different instructions across different platforms, Amica merges them into the same optimization rule. For instance, the 'platform' constraint of 'Saturation Add' rule has three values: Tricore, ARM, and RISC-V. For platform-specific instructions, Amica designs dedicated optimization rules, such as 'Condition Add' rule in Figure 3.

Platform Configuration. Before performing specific optimizations, Amica configures itself according to the deployment platform of the generated code. It first selects applicable optimization rules from the rule library based on the 'platform' constraint. Then, using the corresponding instruction set documentation, Amica estimates the potential reduction in execution latency achieved by utilizing the relevant hardware instructions. This estimation is used in the subsequent optimization process.

Latency-Aware Match. For the selected optimization rules, Amica employs a latency-aware method to obtain the optimal rules which can maximize execution latency reduction for updating the dataflow graph. The process of latency-aware match is detailed in Algorithm 1. First, for each selected rule, Amica invokes `GraphMatch` function to match it with the dataflow graph to determine if the rule is applicable (line 6). We present the procedure of `GraphMatch` in the following content. The matched rules, i.e., `GraphMatch(`G`, `r`) ==` $true$, are optimization candidates for further utilization (line 7). Since the same block may correspond to multiple candidate optimization rules, Amica selects the rule that maximizes the reduction in execution latency for optimization. In this way, Amica not only enhances the performance improvement, but also resolves potential conflicts. For each candidate rule, Amica first obtains its execution latency reduction calculated in platform configuration (line 13). Amica then compares this value with the current maximum latency reduction (line 14). If the new value is greater, Amica uses it to update the maximum value and designates the current rule as the optimal one (line 15-16).

The procedure of `GraphMatch` is as follows. The subgraph of the optimization rule generally consists of multiple blocks, and the results are determined after executing the last block. We called the last block as feature block. For example, in Figure 3, ABS block and `Switch` block are feature blocks of 'ABS Difference' rule and 'Condition Add' rule, respectively. Based on this observation, `GraphMatch` is a back-to-front matching process, where the feature block is matched first, followed by its parent blocks. Specifically, for a target optimization rule r, Amica begins by matching its feature block b_f with candidate blocks b_c of dataflow graph G. Note that, Amica not only matches the block type, but also data type, parameter, and other constraints. If any of these constraints are violated, the matching process fails. Conversely, if the initial match is successful, Amica recursively applies the matching process with parent blocks of b_f and b_c. If all the blocks of the selected optimization rule r are successfully matched, Amica records the corresponding matched subgraphs of dataflow graph G as optimization candidates.

Graph Update. Amica updates the dataflow graph using the obtained optimal rule by encapsulating the matched subgraph into a new block. The inports and outports of the subgraph become the inports and outports of this new block. Besides, Amica assigns

Algorithm 1: Latency-Aware Match

```
Input: G: Dataflow graph of target model
       R : Optimization rules
Output: r_o: The optimal rule for graph update
1  Function LatencyAwareMatch(G, R):
2      // Candidate Rules for optimization
3      Candidates ← ∅
4      for r in R do
5          // Match subgraphs and constraints
6          if GraphMatch(G, r) == true then
7              Candidates.append(r)
8          end
9      end
10     // The maximum value of latency reduction
11     max ← 0
12     for c in Candidates do
13         reduce ← c.latencyReduction()
14         if max < reduce then
15             max ← reduce
16             r_o ← c
17         end
18     end
19     return r_o
20 End Function
```

corresponding functionality to this block based on the rule type and labels it as optimizable. Figure 4 shows an example that illustrates the process of latency-aware match and graph update. Suppose both of ABS blocks enable the 'saturation' parameter, and the target platform is Tricore. Therefore, this dataflow graph has three candidate optimization rules: Rule 1 for 'Saturation ABS Difference', Rule 2 for 'Saturation ABS', and Rule 3 for 'Saturation ABS'. Since Rule 1 reduces the maximum execution latency, Amica selects it for updating dataflow graph. Subsequently, Amica replaces the corresponding subgraph with a new block called `Saturation ABSDIF`, preserving the original inports and outports. Note that, this is an iterative process. Amica continuously performs latency-aware match to obtain optimization candidates, and then selects the optimal one for updating the dataflow graph until there are no candidates.

4.2 Hardware-Oriented Synthesis

Schedule Convert. For the dataflow graph, Amica employs a topology-based method to analyze connections between blocks and derive the translation sequence in accordance with blocks dependencies. The blocks in the sequence are categorized into two types: optimizable blocks and basic blocks. For optimizable blocks, Amica synthesizes corresponding hardware-specific instructions to enhance performance. In contrast, Amica directly translates basic blocks into code based on their inherent functionalities.

Instruction Translation. Amica begins by selecting the appropriate hardware-specific instructions based on the target platform, block functionalities, and data type. These factors can lead to different instructions for synthesis. For example, consider the 'Saturation Add' rule, the choice between 'int' and 'unsigned int' data type requires distinct instructions for synthesis. Then, Amica generates inline assembly code to represent the corresponding instructions,

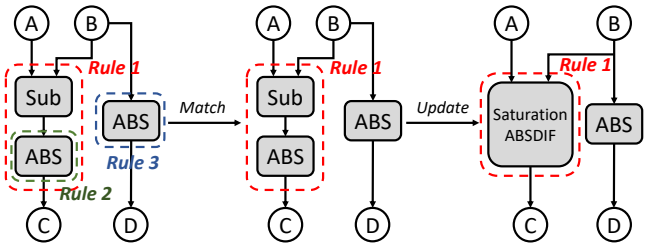


Figure 4: An example illustrates the process of latency-aware match and graph update. The subgraphs framed by the dashed line represent the parts that can be optimized.

and maps the input and output data to the inports and outports of the target block, respectively. Figure 5 presents the synthesized inline assembly code for the SQRT block with float data type on the Tricore, ARM, and RISC-V platforms. Notably, since the Tricore platform only supports the QSEED.F instruction for calculating the reciprocal square root, Amica compensates by multiplying the result of QSEED.F by the value of In to obtain the square root.

```
// Tricore
__asm ( "QSEED.F %0, %1" : "=d"(Out) : "d"(In));
__asm ( "MUL %0, %1"     : "=d"(Out) : "d"(In));
// ARM
__asm ( "FSQRT %s0, %s1" : "=w"(Out) : "w"(In));
// RISC-V
__asm ( "FSQRT.S %0, %1" : "=f"(Out) : "f"(In));
```

Figure 5: An example shows the synthesized instructions for SQRT block with float data type on the Tricre, ARM, and RISC-V platforms.

Code Synthesis. For basic blocks, Amica supports customized DLL (Dynamic Link Library) files to generate the corresponding code based on their functionalities. Note that, the same type of blocks may have differences in block properties, such as data type, leading to the different generated code. Therefore, Amica configures these vital properties as parameters of the corresponding DLL file for generating precise 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.

5 Implementation

We have developed Amica [1] in C++ with 22,156 lines of code. For optimization, Amica implements various optimization rules, encapsulated in the rule library for Tricore, ARM, and RISC-V platforms. These include but not limited to saturation-type rules, rounding-type rules, type-conversion rules, and complex-operation rules. Additionally, Amica supports a wide range of Simulink blocksets for code generation, such as math operation blocksets and DSP system blocksets, which are frequently used in embedded scenarios. For each supported block, Amica designs the corresponding code library in accordance with the block type and parameters.

[1]The implementation and the results are available at the repository. https://anonymous.4open.science/r/AMICA-58C7

6 Evaluation

In this section, we evaluate the effectiveness of Amica by conducting a series of experiments. Our evaluation addresses the following research questions:

- **RQ1:** How well does Amica perform compared to other state-of-the-art code generators? (§6.1)
- **RQ2:** Whether or not Amica is still effective under high-level compiler optimization? (§6.2)
- **RQ3:** How well does Amica perform in terms of assembly code size and memory usage? (§6.3)
- **RQ4:** Is the performance achieved by Amica statistically significant? (§6.4)

Evaluation Setup: To assess the effectiveness of our approach, we compared Amica with two state-of-the-art code generators, Simulink Embedded Coder [7], Mercury [33] and Frodo [32]. The comparison experiments were conducted on three embedded platforms using GCC for compilation, Tricore (Infineon TC387), ARM (Cortex A72), and RISC-V (XuanTie C906). We evaluated the performance of Amica on benchmark Simulink models collected from both academia and industry [21, 26, 33]. Table 1 shows the details of benchmark models, including their functionality, and number of blocks. The models range in size from 57 to 495 blocks, covering a spectrum of realistic model complexities. Besides, selected models include a diverse set of Simulink block types, such as commonly-used blockset, discrete blockset, math-operation blockset, DSP blockset, and subsystems. Because Simulink Embedded Coder is an integrated tool for Simulink, we use Simulink as an abbreviation in the following experimental content.

Table 1: Details of benchmark models.

Model	Functionlity	#Block
ABS	Safety anti-lock braking system	130
HighPass	HighPass filter model	57
Temperature	Automotive temperature control system	70
HybridPass	HybridPass filter model	158
RTVibrate	Equipment preservation model	91
RAC	Robot arm control system	495
BLDC	Brushless DC motors	198
Quat	UAV attitude control system	288
Spray	Startup and pressure regulation control system	111

6.1 Comparison Experiments on -O2 Flag

Developers tend to use -O2 flag for embedded scenarios, because it delivers a balanced optimization that enhances performance without sacrificing stability, assembly code size, or predictability. Therefore, to evaluate the performance of Amica compared with other code generators, we conducted experiments on the benchmark models using this compilation setting. To minimize statistical errors, each generated code was executed 100,000 times to obtain

average execution times. Table 2 presents the experimental results under the -O2 flag. On the Tricore, ARM, and RISC-V platforms, the execution time of the code generated by AMICA is 1.35× - 4.62×, 1.46× - 4.58×, and 1.35× - 2.19× faster, respectively, compared to the code generated by Simulink. Compared to the code generated by Mercury, the execution time is 2.43× - 5.29×, 2.06× - 8.33×, and 1.84× - 3.72× faster on the respective platforms. Besides, compared to the code generated by Frodo, the execution time is 2.80× - 5.42×, 1.88× - 8.36×, and 1.84× - 3.72× faster on the respective platforms. These results indicate that AMICA achieves significant performance improvement over other code generators.

Table 2: Comparison of the code execution time using -O2 flag on Tricore, ARM, and RISC-V platforms (Unit: Second).

Model	Platform	Simulink	Mercury	Frodo	AMICA	AMICA Improvement		
						Simulink	Mercury	Frodo
ABS	Tricore	21.05	38.73	33.77	7.32	2.87×	5.29×	4.61×
	ARM	1.83	3.33	3.34	0.40	4.58×	8.33×	8.36×
	RISC-V	5.22	7.56	7.66	2.39	2.19×	3.17×	3.21×
HighPass	Tricore	22.08	27.87	34.47	6.36	3.47×	4.38×	5.42×
	ARM	1.30	1.90	1.98	0.52	2.50×	3.64×	3.79×
	RISC-V	2.83	4.55	4.55	1.96	1.44×	2.32×	2.32×
Temperature	Tricore	29.08	34.29	33.88	11.76	2.47×	2.92×	2.88×
	ARM	1.64	2.34	2.17	1.12	1.46×	2.08×	1.90×
	RISC-V	3.78	6.17	6.17	2.39	1.58×	2.59×	2.59×
HybridPass	Tricore	41.16	72.92	73.80	18.91	2.18×	3.86×	3.90×
	ARM	2.85	3.27	2.99	1.59	1.79×	2.06×	1.88×
	RISC-V	7.02	8.03	8.01	4.36	1.61×	1.84×	1.84×
RTVibrate	Tricore	12.65	26.03	26.36	8.13	1.56×	3.20×	3.24×
	ARM	1.12	2.07	1.97	0.67	1.67×	3.10×	2.95×
	RISC-V	1.82	3.99	4.29	1.28	1.43×	3.12×	3.36×
RAC	Tricore	120.22	154.47	176.74	38.90	3.09×	3.97×	4.54×
	ARM	5.65	10.48	10.47	1.89	2.99×	5.55×	5.54×
	RISC-V	17.93	29.22	24.28	10.28	1.74×	2.84×	2.36×
BLDC	Tricore	122.44	111.98	112.48	26.49	4.62×	4.23×	4.25×
	ARM	4.24	5.64	5.51	1.81	2.34×	3.12×	3.05×
	RISC-V	11.41	16.57	17.08	7.09	1.61×	2.34×	2.41×
Quat	Tricore	32.26	30.32	34.90	12.47	2.59×	2.43×	2.80×
	ARM	1.14	2.18	2.25	0.63	1.80×	3.44×	3.55×
	RISC-V	3.67	8.42	8.48	2.72	1.35×	3.09×	3.12×
Spray	Tricore	27.88	72.85	69.02	20.63	1.35×	3.53×	3.35×
	ARM	0.99	1.57	1.57	0.67	1.48×	2.34×	2.35×
	RISC-V	3.85	10.63	10.64	2.86	1.35×	3.72×	3.72×

Simulink outperforms Mercury on all benchmark models but does not perform as well as AMICA. Simulink employs various optimization techniques for code generation, such as expression folding and variable reuse. These methods reduce the number of intermediate variables and assignment operations by reusing the output ports of previously executed blocks for subsequent data transfers. By analyzing the assembly code, we observed that when manipulating array data, the redundant assignment operations eliminated by Simulink are not optimized away by GCC under the -O2 flag. This limitation arises because, in the C language, an array name is essentially a pointer, and GCC must consider the possibility that different arrays might point to overlapping memory regions, which restricts its ability to apply aggressive optimization strategies. However, Simulink can accurately determine this, as Simulink model semantics ensure that there is no data space overlap between different blocks.

Mercury focuses on enhancing performance by generating instruction pipeline-friendly codes, and it interleaves the translation of independent blocks to avoid pipeline stalls. However, on benchmark models, its overall performance is relatively limited. This limitation stems from the fact that GCC, when using -O2 flag for compilation, already implements similar techniques. For instance, the `-fschedule-insns` flag schedules independent instructions to fully utilize the processor pipeline during execution. As a result, the optimizations implemented by Mercury offer minimal additional benefits over the inherent capabilities of GCC, and the code generated by Mercury does not perform as well as that produced by Simulink and AMICA. As for Frodo, it focuses on eliminating redundant calculations caused by data-truncation blocks. These blocks select specific data for further usage, and the calculations related to unselected data become redundant. However, benchmark models do not involve such operations, and the code generated by Frodo is unsatisfactory in this context.

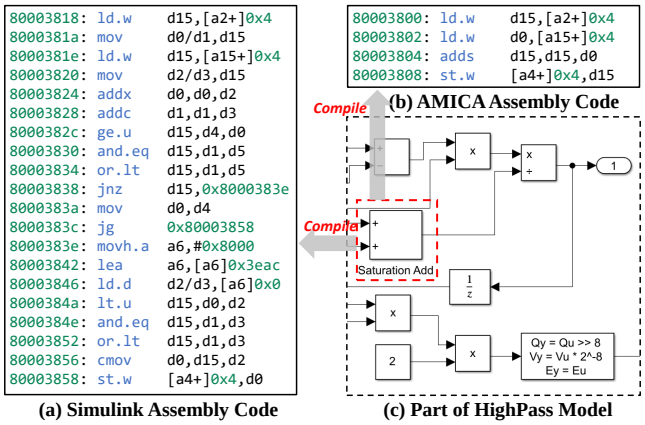


Figure 6: An example illustrates the reason of the performance gap between Simulink and AMICA. The subfigure (a) and subfigure (b) show the assembly code of Simulink and AMICA, for `Saturation Add` block on the Tricore platform. The subfigure (c) shows a portion of the HighPass model, and the `Saturation Add` block is framed by red dashed line.

Compared to other code generators, AMICA converts the Simulink model into dataflow graph, and uses applicable optimization rules to modify the dataflow graph in accordance with the target platform and execution latency. In this way, AMICA can synthesize appropriate hardware-specific instructions for optimizable blocks, thereby improving overall performance of the generated code. The experimental results illustrate other code generators, even state-of-the-art compiler, is unable to effectively utilize hardware-specific instructions. To further demonstrate this point, we also analyzed the corresponding assembly codes. For example, consider the HighPass model shown in Figure 6(c). Figure 6(a) and Figure 6(b) shows the compiled assembly code of Simulink and AMICA, respectively, for `Saturation Add` block framed by the red dashed line on the Tricore platform. Since `Saturation Add` block should determine whether the result exceeds the upper or lower bounds of 'int', the corresponding assembly code of Simulink contains numerous comparison and conditional instructions, e.g., `ge.u`, `jnz`, and `and.eq`.

The assembly code of Mercury exhibits a similar pattern. In contrast, as shown in Figure 6(b), Amica accomplishes this using a single instruction. Consequently, Amica achieves significant performance improvement over Simulink and Mercury. Moreover, we observed that the performance improvement of Amica varies across different platforms. In particular, Amica achieves the greatest performance improvement on the ARM platform. This is because Amica synthesizes a set of hardware-specific instructions on the ARM platform, and a single hardware-specific instruction within the ARM platform reduces more execution latency compared to other platforms.

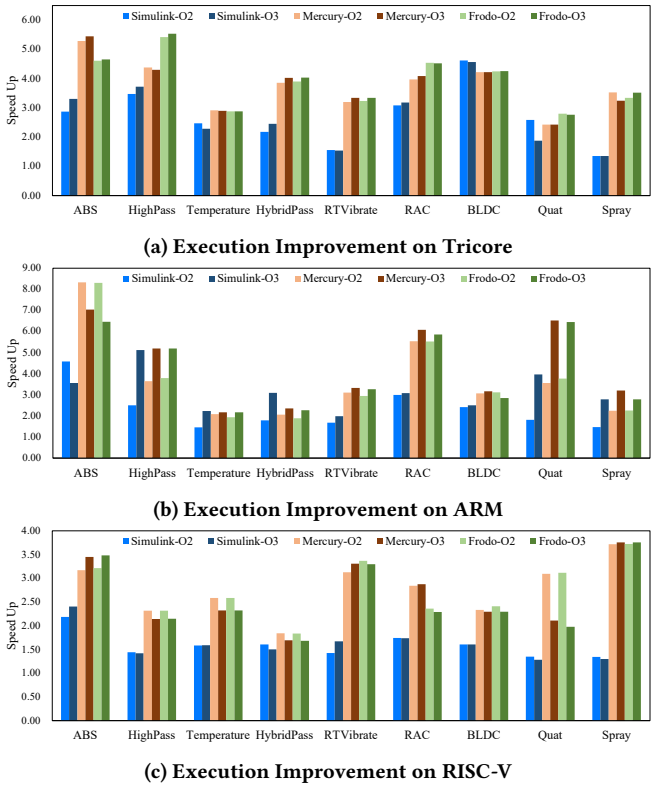
6.2 Comparison Experiments on -O3 Flag

To assess whether Amica remains effective under high-level compiler optimizations, we conducted additional comparison experiments using GCC with `-O3` flag enabled. As in our previous experiments, the generated code was repeatedly executed 100,000 times to obtain reliable average execution times and minimize statistical variability. Figure 7 illustrates the experiment results. On the Tricore, ARM, and RISC-V platforms, the execution times of the code generated by Amica is 1.35× - 4.57×, 1.98× - 5.12×, and 1.29× - 2.41× faster, respectively, compared to the code generated by Simulink. When compared to the code generated by Mercury, the execution time is 2.90× - 5.45×, 2.17× - 7.04×, and 1.69× - 3.75× faster on the respective platforms. Besides, compared to the code generated by Frodo, the execution time is 2.77× - 5.54×, 2.16× - 6.46×, and 1.68× - 3.76× faster on the respective platforms. These results clearly demonstrate that Amica continues to deliver significant performance improvement. Moreover, even with the highest optimization level, GCC fails to effectively leverage hardware-specific instructions, underscoring the importance of synthesizing such instructions in embedded scenarios where performance and resource constraints are critical.

We observed that, in general, the performance improvement of Amica is higher under `-O3` flag, compared with the experiments under `-O2` flag. By analyzing the corresponding assembly code, we found that under `-O3` flag, GCC employs various effective optimization techniques and Amica benefits from these optimizations. For example, GCC enables loop-related optimizations such as `-floop-unroll-and-jam` and `-fpredictive-commoning`. These optimizations help to further eliminate redundant assignments, conditional checks, and calculations that are not specifically addressed by Amica. Besides, on the `ABS` model with the ARM platform shown in Figure 7(b), the performance improvement offered by Amica is less pronounced. A detailed analysis of the assembly code indicates that the code generated by Amica exhibits only minor differences between the `-O2` and `-O3` flag, whereas the assembly code of Simulink and Mercury undergoes significant changes. This suggests that in some cases, hardware-specific instruction synthesis may limit the effectiveness of the optimization techniques employed by the compiler. Nonetheless, given the substantial overall performance gains achieved by Amica, it is still worthy.

6.3 Comparison Experiments on Other Metrics

Embedded devices often have limited memory resources, and the available memory for both the program code and data is typically much smaller than general-purpose devices. Therefore, to further



(a) Execution Improvement on Tricore

(b) Execution Improvement on ARM

(c) Execution Improvement on RISC-V

Figure 7: The execution improvement of the code generated by Amica versus other code generators on the Tricore, ARM, and RISC-V platforms under -O2 flag and -O3 flag.

evaluate the quality of the code generated by Amica, we collected the assembly code size, data segment size, and BSS (Block Started by Symbol) segment size of the compiled programs.

Evaluation on Assembly Code Size. Table 3 presents the code size of the compiled program. On the Tricore platform, the code generated by Amica is reduced by 11% - 53%, 26% - 42%, and 24% - 41% compared to Simulink, Mercury, and Frodo, respectively. On the ARM platform, Amica reduces the code size by 8% - 30%, 11% - 50%, and 15% - 49%, relative to Simulink, Mercury, and Frodo, respectively. On the RISC-V platform, Amica results in a code size reduction of 6% - 26%, 6% - 39%, and 9% - 40% compared to Simulink, Mercury, and Frodo, respectively.

In general, the size of the code generated by Amica is the smallest, compared to Simulink, Mercury, and Frodo. This reduction is due to that Amica synthesizes hardware-specific instructions to execute complex operations that would typically require multiple instructions. For example, consider the HighPass model shown in Figure 6. To perform saturation addition, the assembly code of Simulink takes 20 instructions to check if the results exceeds the upper or lower bounds of 'int', while Amica only takes 4 instructions to accomplish the same task. The assembly code of Mercury follows a similar pattern to Simulink. These statistics illustrate that Amica not only improves the performance of the generated code but also significantly reduces its size. This indicates that Amica is

Table 3: Comparison of the assembly code size on Tricore, ARM, and RISC-V platforms (Unit: Byte).

Model	Platform	Simulink	Mercury	Frodo	Amica	Amica Reduction		
						Simulink	Mercury	Frodo
ABS	Tricore	2100	1784	1728	1320	37%	26%	24%
	ARM	3897	3972	3916	2879	26%	28%	26%
	RISC-V	2627	2501	2701	2024	23%	19%	25%
HighPass	Tricore	1984	1544	1584	928	53%	40%	41%
	ARM	3244	3391	3395	2871	11%	15%	15%
	RISC-V	2147	2136	2213	2012	6%	6%	9%
Temperature	Tricore	1752	1952	1924	1132	35%	42%	41%
	ARM	3253	3969	3881	2997	8%	24%	23%
	RISC-V	2388	2466	2619	2160	10%	12%	18%
HybridPass	Tricore	2848	2512	2572	1536	46%	39%	40%
	ARM	3680	4503	4599	3331	9%	26%	28%
	RISC-V	2652	2892	2983	2446	8%	15%	18%
RTVibrate	Tricore	1576	1588	1608	980	38%	38%	39%
	ARM	3437	3999	3815	3035	12%	24%	20%
	RISC-V	2471	2329	2603	1833	26%	21%	30%
RAC	Tricore	3224	3720	3972	2644	18%	29%	30%
	ARM	4982	7297	7257	3672	26%	50%	49%
	RISC-V	3880	5323	5365	3242	16%	39%	38%
BLDC	Tricore	3372	3032	3108	1852	45%	39%	40%
	ARM	4697	6133	6153	3403	28%	45%	45%
	RISC-V	3609	4339	4505	2725	24%	37%	40%
Quat	Tricore	3664	3774	4020	2764	25%	26%	31%
	ARM	4651	5332	5484	3863	17%	28%	30%
	RISC-V	3925	5341	5305	3172	19%	41%	40%
Spray	Tricore	1844	2400	2508	1648	11%	31%	34%
	ARM	3967	5067	5067	3535	11%	30%	30%
	RISC-V	3047	3593	3569	2801	8%	22%	22%

Table 4: Comparison of the data segment usage and BSS segment usage (Unit: Byte).

Model	Platform	Data Segment				BSS Segment			
		Simulink	Mercury	Frodo	Amica	Simulink	Mercury	Frodo	Amica
ABS	Tricore	640	640	640	640	1360	1344	1344	1344
	ARM	712	680	680	680	1320	1296	1296	1296
	RISC-V	648	600	600	600	1304	1288	1288	1288
HighPass	Tricore	612	608	608	608	960	960	960	960
	ARM	656	656	656	656	912	912	912	912
	RISC-V	600	592	592	592	904	904	904	904
Temperature	Tricore	620	620	620	620	1104	1088	1088	1088
	ARM	656	664	664	664	1072	1040	1040	1040
	RISC-V	616	592	592	592	1048	1032	1032	1032
HybridPass	Tricore	580	584	584	584	1700	1684	1684	1684
	ARM	640	656	656	656	1712	1680	1680	1680
	RISC-V	592	592	592	592	1688	1672	1672	1672
RTVibrate	Tricore	600	604	604	600	800	784	784	784
	ARM	672	680	680	656	816	784	784	784
	RISC-V	608	592	592	584	792	776	776	776
RAC	Tricore	620	592	592	588	3124	3116	3116	3116
	ARM	704	680	680	648	3112	3088	3088	3088
	RISC-V	640	608	608	592	3096	3080	3080	3080
BLDC	Tricore	620	592	592	588	3124	3116	3116	3116
	ARM	704	680	680	648	3112	3088	3088	3088
	RISC-V	640	608	608	592	3096	3080	3080	3080
Quat	Tricore	620	592	592	588	3124	3116	3116	3116
	ARM	704	680	680	648	3112	3088	3088	3088
	RISC-V	640	608	608	592	3096	3080	3080	3080
Spray	Tricore	620	592	592	588	3124	3116	3116	3116
	ARM	704	680	680	648	3112	3088	3088	3088
	RISC-V	640	608	608	592	3096	3080	3080	3080

more suitable for embedded scenarios, as minimizing memory usage allows embedded devices to handle more complex tasks within limited hardware constraints simultaneously.

We found that the reduction of assembly code size varies across platforms. The code size reduction on the Tricore platform is more significant than on the ARM and RISC-V platforms. This is due to the varying numbers of synthesized hardware-specific instructions. For the benchmark models, Amica synthesizes more hardware-specific instructions on the Tricore platform, which leads to a greater reduction in assembly code size. We also observed that the assembly code reduction for the HighPass model on the RISC-V platform is relatively small (6% - 9% reduction). By analyzing the generated code, we found that on the RISC-V platform, Amica matches 2 rules with the HighPass model and synthesizes 7 hardware-specific instructions, which are fewer than that on other platforms and for other models. Although the number of synthesized instructions is relatively small, Amica still achieves 1.44× and 2.32× performance improvement, compared to Simulink and Mercury with `-O2` flag. This highlights the importance of synthesizing hardware-specific instructions to bypass time-consuming regular instructions.

Furthermore, we observed that the assembly code sizes for the same model vary across different platforms. Specifically, given the same C code, the assembly code on the ARM platform is larger than that on the RISC-V and Tricore platforms. By analyzing the corresponding assembly code, we found that ARM requires more instructions for the ABI (Application Binary Interface), which results in a higher number of instructions for function calls, parameter passing, and stack management. The differences in the instruction set architectures of these platforms also lead to variations in the size and complexity of the generated assembly code. Tricore, as an automotive-grade platform, demands stricter resource constraints. Therefore, the instruction set architecture of Tricore places greater emphasis on generating compact assembly code, which leads to the smallest code size compared to ARM and RISC-V.

Evaluation on Data Segment and BSS Segment. We also collected the data segment size and BSS segment size of the compiled program to further evaluate Amica, and the detailed statistics are shown in Table 4. Our finding reveals that the sizes of both the data segment and BSS segment remain consistent across different code generators and platforms. The data segment is responsible for storing initialized global and static variables. In the context of the model code, the primary contributors to data segment usage are the referenced header files and model configuration parameters, such as `stdlib.h`. As a result, the data segment size is similar across code generated by different code generators. The BSS segment, on the other hand, is used to store uninitialized global and static variables. In the case of the model code, the main contributors to BSS segment usage are input and output data. Consequently, the BSS segment size is nearly identical across various code generators.

6.4 Hypotheses Testing

To assess the statistical significance of Amica's performance improvements, we repeated the comparison experiments 10 times across different platforms and compiler optimization flags. The following hypotheses are tested:

Table 5: Hypotheses testing results on performance improvement.

Model	Platform	Amica Improvement under -O2 flag						Amica Improvement under O3 flag					
		Simulink		Mercury		Frodo		Simulink		Mercury		Frodo	
		P Value	Cohen's d	P Value	Cohen's d	P Value	Cohen's d	P Value	Cohen's d	P Value	Cohen's d	P Value	Cohen's d
ABS	Tricore	5.49e-55	8.08e+05	1.44e-56	1.21e+06	2.10e-42	3.23e+04	4.18e-55	8.33e+05	1.33e-55	9.46e+05	2.23e-55	8.93e+05
	ARM	7.73e-33	2.80e+03	3.03e-22	1.86e+02	8.25e-17	4.63e+01	5.49e-32	2.25e+03	5.38e-36	6.27e+03	9.66e-37	7.59e+03
	RISC-V	2.15e-24	3.22e+02	2.71e-25	4.06e+02	1.54e-25	4.32e+02	6.70e-26	4.74e+02	1.19e-26	5.74e+02	9.44e-27	5.89e+02
HighPass	Tricore	4.57e-54	6.38e+05	6.18e-55	7.97e+05	5.03e-55	8.16e+05	2.05e-54	6.98e+05	9.71e-56	9.79e+05	2.36e-55	8.87e+05
	ARM	6.26e-29	1.03e+03	5.50e-30	1.35e+03	9.43e-30	1.27e+03	1.34e-30	1.58e+03	1.52e-29	1.20e+03	7.34e-30	1.30e+03
	RISC-V	1.54e-18	7.21e+01	7.79e-21	1.30e+02	1.59e-21	1.55e+02	1.58e-22	2.00e+02	1.29e-24	3.41e+02	9.29e-25	3.54e+02
Temperature	Tricore	7.68e-57	1.30e+06	6.53e-56	1.02e+06	2.40e-46	8.86e+04	3.12e-56	1.11e+06	3.24e-56	1.11e+06	1.06e-45	7.51e+04
	ARM	4.69e-11	1.06e+01	2.74e-13	1.88e+01	2.23e-13	1.92e+01	1.40e-18	7.28e+01	5.58e-19	8.06e+01	5.29e-29	1.05e+03
	RISC-V	1.56e-30	1.55e+03	1.01e-33	3.50e+03	3.96e-33	3.01e+03	2.28e-31	1.92e+03	2.79e-35	5.22e+03	1.65e-33	3.32e+03
HybridPass	Tricore	1.75e-58	1.98e+06	2.51e-59	2.45e+06	3.22e-37	8.57e+03	1.18e-58	2.06e+06	3.89e-60	3.02e+06	2.15e-38	1.16e+04
	ARM	6.19e-34	3.70e+03	1.36e-27	7.30e+02	2.78e-21	1.45e+02	3.53e-23	2.36e+02	2.20e-21	1.49e+02	2.36e-23	2.47e+02
	RISC-V	3.96e-27	6.49e+02	1.24e-29	1.23e+03	1.52e-27	7.21e+02	2.55e-16	4.08e+01	5.05e-17	4.89e+01	3.40e-19	8.52e+01
RTVibrate	Tricore	1.08e-51	3.48e+05	2.97e-54	6.70e+05	2.71e-54	6.77e+05	5.32e-20	1.05e+02	3.75e-21	1.41e+02	4.43e-18	6.41e+01
	ARM	9.28e-22	1.64e+02	1.81e-24	3.28e+02	1.32e-19	9.47e+01	5.32e-20	1.05e+02	3.75e-21	1.41e+02	4.43e-18	6.41e+01
	RISC-V	9.43e-24	2.73e+02	4.57e-19	8.24e+01	1.04e-26	5.83e+02	1.42e-23	2.61e+02	1.65e-25	4.29e+02	1.66e-25	4.28e+02
RAC	Tricore	8.06e-61	3.59e+06	5.30e-62	4.86e+06	2.05e-62	5.40e+06	1.26e-62	5.70e+06	1.24e-62	5.71e+06	1.99e-62	5.42e+06
	ARM	9.70e-23	2.11e+02	2.32e-26	5.33e+02	9.74e-26	4.54e+02	1.09e-18	7.48e+01	1.14e-20	1.24e+02	4.10e-20	1.08e+02
	RISC-V	1.29e-32	2.64e+03	3.75e-35	5.05e+03	1.73e-32	2.56e+03	8.10e-34	3.59e+03	1.71e-36	7.12e+03	1.39e-32	2.62e+03
BLDC	Tricore	6.53e-62	4.75e+06	2.90e-61	4.03e+06	4.35e-61	3.85e+06	7.72e-60	2.80e+06	8.33e-60	2.77e+06	5.39e-60	2.91e+06
	ARM	5.90e-13	1.72e+01	1.10e-13	2.08e+01	1.24e-14	2.65e+01	5.82e-15	2.88e+01	5.75e-15	2.89e+01	3.70e-15	3.03e+01
	RISC-V	1.07e-25	4.50e+02	2.32e-29	1.15e+03	1.08e-28	9.68e+02	1.80e-24	3.29e+02	1.32e-28	9.46e+02	3.22e-29	1.11e+03
Quat	Tricore	8.83e-57	1.28e+06	1.89e-55	9.10e+05	5.13e-57	1.36e+06	3.56e-54	6.56e+05	9.91e-56	9.77e+05	1.09e-56	1.25e+06
	ARM	2.69e-35	5.24e+03	1.07e-15	3.48e+01	6.00e-16	3.71e+01	1.02e-15	3.50e+01	3.94e-15	3.01e+01	1.64e-14	2.57e+01
	RISC-V	3.99e-23	2.33e+02	3.82e-30	1.40e+03	1.02e-28	9.74e+02	2.37e-31	1.91e+03	1.07e-26	5.81e+02	3.72e-26	5.06e+02
Spray	Tricore	1.17e-53	5.75e+05	5.71e-59	2.24e+06	1.08e-59	2.69e+06	2.34e-55	8.88e+05	2.88e-59	2.41e+06	4.90e-59	2.28e+06
	ARM	1.25e-13	2.05e+01	1.99e-12	1.51e+01	1.42e-15	3.37e+01	3.61e-15	3.04e+01	2.44e-26	5.30e+02	5.06e-27	6.31e+02
	RISC-V	3.89e-13	1.81e+01	2.62e-17	5.26e+01	1.75e-05	2.39e+00	1.96e-14	2.52e+01	2.51e-28	8.81e+02	2.71e-29	1.13e+03

- H_0**:** There are no statistically significant improvements in execution time between Amica and other code generators.
- H_1**:** Amica achieves statistically significant improvements in execution time compared to other code generators.

Table 5 presents the results. All p-values (≤ 0.05) fall below the standard significance threshold, confirming that the observed performance differences are statistically significant. Moreover, the consistently large Cohen's d values indicate strong effect sizes, which can be attributed to the highly stable execution times observed across the 10 repeated runs (variation $\leq 0.001\%$). These findings further reinforce the reliability and robustness of Amica's performance improvements.

7 Discussion

Scalability of Amica. Currently, Amica supports code generation from Simulink models for Tricore, ARM, and RISC-V platforms. However, Amica can be extended to support code generation for other platforms, such as the x86 platform. This extension would require crafting optimization rules based on the corresponding instruction set architecture and designing inline assembly code for instruction translation in accordance with instruction syntax. Besides, since for functionally equivalent but semantically different instructions, Amica merges them into the same optimization rule. Therefore, for these instructions on the x86 platform, Amica can simply incorporate them into the existing rules for optimization, without repetitive implementations. In our future work, we plan to explore a wider array of hardware-specific instructions for synthesis and extend support to more platforms.

Code Quality & Correctness. To ensure the quality of the generated code, we follow the MISRA [14] C security standard. Amica generates code in strict compliance with this standard for elements such as functions, variables, and statements. To ensure the correctness of the generated code, we have employed several effective approaches. For each benchmark model, we generated a variety of test cases as input data for the code generated by Amica and other code generators. The execution results across all these code generators were found to be consistent. Additionally, we verified the consistency between the execution of the generated code and the simulation of the corresponding model, ensuring that both yielded identical results. For each hardware-specific instruction for synthesis, we also generated comprehensive test cases for observing and analyzing their execution behavior. Only those instructions that strictly adhere to the specification and conform to the model semantics are used by Amica in code generation.

Threats to Validity. First, for models that lack semantics for hardware-specific instruction synthesis, Amica may not deliver performance improvements. In fact, complex mathematical operations and saturation logic are prevalent in many real-world scenarios, such as automotive systems and DSP systems, where Amica has shown pronounced performance benefits. Besides, code optimizations targeting other aspects, such as expression folding, can be combined with Amica to generate more efficient code. Second, we used GCC in the experiments for compiling the generated code, as the compilation toolchains for Tricore and RISC-V are built on GCC. This may limit the evaluation of Amica's effectiveness on other compilers. However, Amica is also compatible with other

compilers, e.g., Clang, and can yield performance improvements. These gains stem from synthesizing hardware-specific instructions that accelerate operations that are difficult for compilers to synthesize automatically. Besides, we conducted comparative experiments on the ARM platform using Clang, and the results consistently demonstrate the effectiveness of Amica.

8 Related Work

Code Generators. Recently, many research and commercial tools have made remarkable efforts to improve the performance of the generated code. Specifically, Simulink Embedded Coder [7], the built-in tool, specializes in generating production-quality code by employing various high-level optimization techniques, including expression folding, variable reuse, etc. These powerful optimizations make it widely utilized in embedded scenarios for code generation. In academics, there are some noteworthy works to generate efficient code. DFSynth [23] 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. Besides, DFSynth designs tailored code templates for each block for code generation. Mercury [33] leverages processor instruction pipeline specificity to improve code performance. Mercury approximately estimates the execution latency of each block, and then uses the least penalty priority to select the suitable blocks for translation, thereby reducing instruction pipeline stalls. Frodo [32], the most recent work, focuses on eliminating redundant calculations within the code. When the target model contains data-truncation blocks, Frodo recursively determines the precise calculation range of each block to avoid redundant calculations.

Different from these works, Amica proactively leverages model semantics for synthesizing hardware-specific instructions during code generation to enhance performance. Based on both model semantics and instruction set architectures, Amica crafts optimization rules for refactoring the dataflow graph of the target model, and employs a latency-aware approach which considers block constraints and execution latency to select appropriate instructions.

Compiler Optimizations. For instruction selection, state-of-the-art compilers, e.g., GCC and Clang, typically rely on pattern-matching to identify and select the most suitable instructions during translation, which has yielded significant improvement in synthesizing arithmetic and vector instructions. In addition to these techniques, some notable works [2, 5, 18, 30] propose alternative approaches for instruction selection. Diospyros [30] uses equality saturation for synthesizing vector instructions. However, it does not efficiently support constraints and rule predicates, e.g., saturation constraint and type-conversion enforcement, and is not efficient enough for usage. Pitchfork [18] uses Halide [16] IR for fast instruction selection, but its design is not ideally suited for general-purpose scenarios, where toolchains are predominantly based on GCC or Clang.

Different from previous works, Amica integrates pattern-matching with model semantics to synthesize hardware-specific instructions by considering both block parameters and connectivity. This approach enables a more precise and context-aware instruction selection process. Moreover, synthesizing hardware-specific instructions during code generation is inherently more extensible, as it does not depend on compilation toolchains tailored to specific scenarios, thereby facilitating adaptability across a broader range of platforms.

9 Conclusion

This paper introduces Amica, an efficient code generator for Simulink models with hardware-specific instruction synthesis. Amica leverages model semantics to craft optimization rules, and iteratively selects the rule that minimizes execution latency for synthesizing hardware-specific instructions. We evaluated Amica on benchmark Simulink models across different platforms. The results that compared with state-of-the-art code generators, Amica is 1.29× - 8.36× faster in terms of execution time across, and reduces 6% - 53% assembly code size.

Acknowledgments

This research is sponsored in part by the National Key Research , Development Project (No. 2022YFB3104000) and NSFC Program (No. U2441238, 62021002, 62372263), and the Fundamental Research Funds for the Central Universities.

References

[1] Infineon Technologies AG. 2025. *Infineon.* https://www.infineon.com.
[2] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 2022. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1004–1016. doi:10.1145/3503222.3507714
[3] Saba Akram and Quarrat Ul Ann. 2015. Newton raphson method. *International Journal of Scientific & Engineering Research* 6, 7 (2015), 1748–1752.
[4] 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.
[5] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 300–313. doi:10.1145/3168821
[6] 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.
[7] Simulink Embedded Coder. 2025. *Simulink Embedded Coder Documentation.* MathWorks. https://www.mathworks.com/solutions/embedded-code-generation.html.
[8] Jon Friedman. 2006. MATLAB/Simulink for automotive systems design. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 1. IEEE, 1–2.
[9] Infineon. 2025. *32-bit AURIX™ TriCore™ Microcontroller.* https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/.
[10] Jean-Marc Jézéquel. 2008. Model driven design and aspect weaving. *Software & Systems Modeling* 7 (2008), 209–218.
[11] 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.
[12] 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.
[13] Kimon Karras, Evangelos Pallis, George Mastorakis, Yannis Nikoloudakis, Jordi Mongay Batalla, Constandinos X Mavromoustakis, and Evangelos Markakis. 2020. A hardware acceleration platform for AI-based inference at the edge. *Circuits, Systems, and Signal Processing* 39, 2 (2020), 1059–1070.
[14] The MISRA Consortium Limited. 2025. *MISRA C Documentation.* https://misra.org.uk/.
[15] Srihari Palli, Azad Duppala, Rakesh Chandmal Sharma, and LV Rao. 2022. Dynamic simulation of automotive vehicle suspension using MATLAB Simulink. *Int J Veh Struct Syst* 14 (2022).

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. doi:10.1145/2499370.2462176

[17] Ramzy Rammouz, Lioua Labrak, Nacer Abouchi, J Constantin, Y Zaatar, and D Zaouk. 2015. A generic Simulink based model of a wireless sensor node: Application to a medical healthcare system. In *2015 International Conference on Advances in Biomedical Engineering (ICABME)*. IEEE, 154–157.

[18] Alexander J Root, Maaz Bin Safeer Ahmad, Dillon Sharlet, Andrew Adams, Shoaib Kamil, and Jonathan Ragan-Kelley. 2024. Fast Instruction Selection for Fast Digital Signal Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 125–137. doi:10.1145/3623278.3624768

[19] Sergio Saponara. 2019. Hardware accelerator IP cores for real time Radar and camera-based ADAS. *Journal of Real-Time Image Processing* 16, 5 (2019), 1493–1510.

[20] Gregor Schewior, Holger Flatt, Carsten Dolar, Christian Banz, and Holger Blume. 2011. A hardware accelerated configurable ASIP architecture for embedded real-time video-based driver assistance applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 209–216.

[21] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2022. SLNET: a redistributable corpus of 3rd-party simulink models. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 237–241. doi:10.1145/3524842.3528001

[22] Simulink and Matlab. 2025. *Simulink Documentation*. https://www.mathworks.com/products/simulink.html.

[23] 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.

[24] 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* 41, 10 (2021), 3252–3265.

[25] Zhuo Su, Zehong Yu, Dongyan Wang, Wanli Chang, Bin Gu, and Yu Jiang. 2024. Test Case Generation for Simulink Models using Model Fuzzing and State Solving. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/3691620.3694991

[26] 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.

[27] Zhuo Su, Zehong Yu, Dongyan Wang, Yixiao Yang, Rui Wang, Wanli Chang, Aiguo Cui, and Yu Jiang. 2023. STCG: State-Aware Test Case Generation for Simulink Models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC56929.2023.10247787

[28] Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 19–34.

[29] Eugene Titov, Jason Lustbader, Daniel Leighton, and Tibor Kiss. 2016. *Matlab/Simulink framework for modeling complex coolant flow configurations of advanced automotive thermal management systems*. Technical Report. National Renewable Energy Lab.(NREL), Golden, CO (United States).

[30] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. doi:10.1145/3445814.3446707

[31] Liuping Wang. 2020. *PID control system design and automatic tuning using MATLAB/Simulink*. John Wiley & Sons.

[32] Zehong Yu, Zhuo Su, Yu Jiang, Aiguo Cui, and Rui Wang. 2024. Efficient Code Generation for Data-Intensive Simulink Models via Redundancy Elimination. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 20, 6 pages. doi:10.1145/3649329.3656217

[33] 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.