PHCG: Optimizing Simulink Code Generation for Embedded System with SIMD Instructions

Zhuo Su, Dongyan Wang, Zehong Yu, Yixiao Yang, Yu Jiang, Rui Wang, Wanli Chang, Wen Li, Aiguo Cui and Jiaguang Sun

Abstract—Simulink is widely used for the model-driven design of embedded systems. It is able to generate optimized embedded control software code through expression folding, variable reuse, etc. However, for some commonly used computing-sensitive models, such as the models for signal processing applications, the efficiency of the generated code is still limited.

In this paper, we propose PHCG, an optimized code generator for the Simulink model with SIMD instruction synthesis. It will select the optimal implementations for intensive computing actors based on adaptively pre-calculation of the input scales, and synthesize the appropriate SIMD instructions for batch computing actors based on the iterative dataflow graph mapping. In addition, actors of the same type that can be executed in parallel can be combined into batch computing actors as much as possible by merging isomorphic subgraphs. We implemented and evaluated its performance on benchmark Simulink models. Compared to the built-in Simulink Coder and the most recent DFSynth, the code generated by PHCG achieves an improvement of 38.9%-92.9% and 41.2%-76.8% in terms of execution time across different architectures and compilers, respectively.

Index Terms—Code generation, model-driven design, SIMD instruction, Simulink

I. INTRODUCTION

Simulink is one of the most widely used model-driven design tools and is increasingly used in embedded scenarios such as smart transportation, avionics and vehicles [2], [3], [4], [5]. It supports the behavior modeling, simulation, and code generation of embedded control software [6]. The automatic code generation releases the developers from hard-work coding, but the efficiency of the generated code is hard to ensure and may affect the performance and the throughput of the whole system [7], [5].

For optimization, expression folding and variable reuse are mainly used in Simulink Coder [8] to generate more compact code. Recently, DFSynth [9] optimizes the code generation of Simulink models with complex branching logic.

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

D. Wang is with the Information Technology Center, Renmin University of China, Beijing 100872, China (e-mail: wd@ruc.edu.cn).

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

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

Yu Jiang and Yixiao Yang are the corresponding authors.

This paper is an extended version of a conference paper [1].

It transforms the branch logic to control flow code logic based on semantics analysis. Although they perform well in many cases, the efficiency is still limited for models that contain intensive computing actors (e.g. fast Fourier transform), batch computing actors (e.g. batch Add) and parallelizable actors.

For the intensive computing actors, which usually take batch data as input to perform complex calculations, the tools such as Simulink Coder and DFsynth will generate a generic function for computation. But in fact, for an intensive computing actor, there are many different implementations, and their efficiency varies at the different input scales [10], [11]. Take FFT (Fast Fourier Transform) as an example. As shown in Figure 1, we can see that no one implementation can always perform better than the others for all input data lengths. For example, Mix-FFT performs best on large input-scales, but performs worse on small input-scales. When generating code, the input and output scales of the actors in different models are uncertain. So we should dynamically select the more appropriate implementation codes based on the model information to achieve optimal efficiency.

For the batch computing actors, which take an array as input and output, and each element of the output array is calculated from its corresponding input element with the same array index, existing tools will generate repeated code segments or function loops to accomplish the task. For example, Simulink Coder uses the method shown in Figure 2 to generate code. But if the SIMD (Single Instruction Multiple Data) instructions are used, only two operations are required, which are vmlaq_f32

1Mix FFT is obtained from the website: http://www.corix.dk/Mix-FFT/mix-fft.html, Rad-2 FFT is a Radix-2 division FFT implementation, and Galois FFT is obtained from the website: https://hackage.haskell.org/package/galois-fft-0.1.0
(vector multiplication and addition) and $vrecpsq_f32$ (vector reciprocal) [12], [13]. Making full use of the compound SIMD instructions of the processor can effectively improve the running speed of the generated code [14]. For example, the $vhadd$ instruction in ARM architecture adds two vector integers and then right shifts the addition result by one bit. When the composition of batch actors is complex in the model, we should select the appropriate compositions of SIMD instructions for vector acceleration.

In this paper, we propose PHCG to optimize the code generation of the Simulink models with SIMD instruction synthesis. First, the basic arithmetic actors which can be executed in parallel need to be combined as batch computing actors. Then, the intensive computing actors and batch computing actors will be classified for code generation. After that, for the intensive computing actors, PHCG will choose the optimal implementation to generate code. For batch computing actors, PHCG will generate a group of SIMD instructions.

We implemented and evaluated PHCG on benchmark Simulink models, which also contain intensive computing actors, batch computing actors. The results show that PHCG achieves excellent performance. Compared with the built-in Simulink Coder and the most recent DFsynth [9], the code generated by PHCG achieves an improvement of 38.9%-92.9% and 41.2%-76.8% in terms of execution time across different architectures and compilers, respectively. Not only that, but comparative experiments in terms of lines of code, memory usage of program and the time of code generation also illustrate the effectiveness of PHCG.

**II. RELATED WORK**

**A. Model-driven Design**

Model-driven design is a widely used software development method for embedded scenarios. It mainly consists of three components: behavior modeling, simulation and code generation [5], [21], [3], [2], [22], [23]. Behavior modeling is used to construct the formal model with text or graphics according to the user requirement; Simulation is used for debugging and functional correctness verification of the model. Code generation is the key step to translate the model into code for deployment on embedded devices. There are many design tools, such as Ptolemy-II, Tsmart, Polychrony in academic [15], [24], [16], [25], and Simulink, SCADF, DaVinci Developer in industry [6], [17], [26]. Among them, Simulink developed by MathWorks is the most popular for its powerful model simulation and code generation capabilities. It provides a rich library of components to support the design of systems in multiple areas of industry. Moreover, the combination of data flow semantics and state flow semantics gives it a strong model representation capability.

**B. Data flow model**

Data flow model is a kind of computation graph model which is composed of actors, ports, and data connections [27], [19]. Where the actor represents a minimal computation unit and its computation rules are usually determined by the type of itself. For example, the Add actor is used to perform addition operations. Ports are divided into inputs and outputs for receiving data and sending data. Ports can be attached to actors to describe their own inputs and outputs, or they can exist individually in the model to describe the external inputs and outputs of the entire computation graph. The data connection is a channel for data transfer to represent the flow of data. The source of the data connection can be either an import of the model or an output of an actor. The destination of the data connection can be either an output of the model or an import.
of an actor. The models in Figure 2 and Figure 3 are data flow models of Simulink. In these models, the directed lines are the data connections that indicating the flow of data. In particular, the batch computing actors that is focused of this paper are marked with “Batch” in the figure.

C. Code Generation

Code generation plays an important role because it will convert the constructed model into code deployed in real embedded devices [6], [15], [28], [29]. Most code generators perform the following steps to generate code [30]: ① Model parse transforms model file into structured actor information; ② Schedule analysis obtains the scheduling relationship among model actors; ③ Code synthesis generates fire code for each actor; ④ Code composition integrates the fire code of each actor into the output code according to the schedule. For Simulink, the built-in Simulink Coder [8] works very well and it supports efficient code generation for different architectures and compilers with optimizations such as expression folding and output variable reuse. There are also a lot of academic works focusing on code generation [9], [18], [19], [31], [32]. DFSynth [9] is the most-recent research work for code generation of Simulink models. Based on schedule analysis and branch information marking, it supports well-structured code generation for complex branch logic.

D. Main Difference

The main difference between PHCG and those existing generators is that PHCG is able to generate more optimal implementation with SIMD instruction synthesis. For the basic arithmetic actors that can be executed in parallel, it will combine them into batch computing actors to generate SIMD instructions; For those intensive computing actors, it will determine the choice of implementation based on the pre-calculation of the input scales adaptively; For those batch computing actors, it will determine the proper SIMD instructions set according to the iterative dataflow graph mapping.

III. PHCG DESIGN

PHCG takes the Simulink model as input and generates efficient and deployable code for embedded devices as output. It mainly consists of three components: Actor Parallelization, Actor Classification and SIMD Instruction Synthesis, as demonstrated in Figure 4. First, the Simulink model file needs to be analyzed by the model parser as a directed calculation graph. Then the calculation graph will be parallelized at the actor level through isomorphic subgraph search and subgraph merging, respectively. After that, the intensive computing actors, batch computing actors and remainder basic actors will be classified and dispatched for instruction synthesis. Next, those actors are synthesized in different ways accordingly. For intensive computing actors, PHCG considers the actor type and the input scale to select the suitable and optimal implementation code. For example, the FFT (Fast Fourier Transform) actor in Figure 1 with 1024 floating point data as input will be translated into the Radix-4 butterfly FFT implementation code to adapt the input data scale. For batch computing actors, PHCG converts them into a dataflow graph and iteratively generates the optimal SIMD instructions with graph mapping. For instance, the composition of a 4-batch Add actor and a 4-batch Multiply actor in Figure 2 will be translated into a vmadd instruction (batch multiply and add instruction) instead of four add instructions and four mul instructions. For remainder basic actors and the code snippets composition, the conventional translation method of the built-in Simulink Coder will be used.

A. Actor Parallelization

For a given Simulink model, the first step is to parse the model into structured actors, connections and other model elements in memory. Then the actors and the connections will be represented as a directed dataflow graph for further analysis. For generating more SIMD instructions at the SIMD instruction synthesis step, the basic arithmetic actors need to be combined into batch computing actors as more as possible. An example of actor parallelization is shown in Figure 3. In this example model, four Add actors, four Sub actors, two Mul actors and two Shl actors are merged into batch computing actors corresponding to their types, respectively. Table I demonstrates the most frequently used basic arithmetic actors in Simulink model libraries [6]. The Actor Parallelization process is mainly implemented by two algorithms, the isomorphic subgraph search algorithm which is used to find
computing subgraphs with the same topology in the directed dataflow graph and the subgraph merging algorithm which is used to merge subgraphs with the same topology. This process is iterative, it finds the largest isomorphic subgraph to merge each time until no isomorphic subgraph is found. Choosing the largest isomorphic subgraph for merging each time can increase the parallelism of the model as much as possible. If there are multiple largest isomorphic subgraphs of the same size, just select one at random and the rest isomorphic subgraphs will be output in later iterations.

### Table 1: Most Frequently Used Basic Arithmetic Actors in Simulink Model Libraries

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub/Mul/Div</td>
<td>Add, Subtract, Multiply, Divide</td>
</tr>
<tr>
<td>Shr/Shl</td>
<td>Right shift, Left shift</td>
</tr>
<tr>
<td>BitNot/And/Or/Xor</td>
<td>Bit-wise Not/And/Or/Xor</td>
</tr>
<tr>
<td>Min/Max</td>
<td>Minimum, Maximum</td>
</tr>
<tr>
<td>Abs/Absd</td>
<td>Absolute, Absolute difference</td>
</tr>
<tr>
<td>Recp/Sqrt</td>
<td>Reciprocal, Square Root</td>
</tr>
</tbody>
</table>

1) **Isomorphic subgraph search**: A prerequisite for some subgraphs to be merged is that they must be isomorphic. An isomorphic subgraph is composed of two or more subgraphs whose actor types and connectivity relationships between actors are the same. To explore more parallelism between actors, we need to find as large isomorphic subgraphs as possible. The algorithm of the largest isomorphic subgraph search is shown bellow in Algorithm 1. An example of isomorphic subgraph search with four Sub actors as the initial isomorphic subgraph (also called seed) is shown in Figure 5.

![Fig. 5](image_url) An example data flow model (a) An example data flow graph which corresponds to the model in Figure 3. Subfigure (b) shows all searched isomorphic subgraph based on Sub node. The largest isomorphic subgraph is Sub-Mul graph, and it contains eight nodes. Subfigure (c) is an illustration of the extension process.

In algorithm 1, the actors which are supported to merge are classified by their type, as shown in lines 1-4. The actors with the same type will be used as the seed and will be iteratively extended later. In lines 5-15, all isomorphic subgraph extended from the seeds will be found and stored into a list variable named `isoGraphList`. The process of obtaining all isomorphic subgraphs is carried out separately according to the actor types of seeds. It means that all supported actors will be used as initial seeds for subgraph extension. For example, all actors with Add type will be considered as a seed for subgraph extension. In line 7, each actor with same type needs to be converted form a single actor node into a directed graph format. Then, a queue of extensible isomorphic subgraphs named `extIsoGraphQ` is used to extend subgraphs iteratively until it becomes empty, as shown in lines 8-11. For each isomorphic subgraph in `extIsoGraphQ` is a candidate largest isomorphic subgraph and it will be added into `isoGraphList` in line 12. Line 13 attempts to extend a actor node on an isomorphic subgraph in the queue. The `extendOnce` function will output all possible subgraphs that extend one node, and these output subgraphs are not contained in each other. This means that all isomorphic subgraphs will be explored without omission. The extend result will be add into the queue for further extension, as shown in lines 14-15. Finally, in lines 16-24, the largest isomorphic subgraph which has the maximum number of actors will be returned. Especially, there are no data dependencies between all subgraphs contained in the returned isomorphic subgraph. This is because only then these subgraphs can be executed in parallel. The dependencies of these subgraphs are analyzed from the data flow of the model and are represented as an undirected graph, as shown in line 19. In this undirected graph, each node represents a subgraph in the isomorphic subgraph, and each edge indicates that the two subgraphs do not have data dependencies on each other. To obtain the maximum number of subgraphs that can be executed in parallel, we use the maximum clique algorithm [33], [34] to obtain the largest isomorphic subgraphs without dependencies, as shown in lines 20-21. In our design, the Bron-Kerbosch algorithm is used to solve the maximum clique problem [35]. As shown in Figure 5.(b), the largest isomorphic subgraph is Sub-Mul graph, it contain four isomorphic subgraphs with a total of eight nodes.

It is important to note that the function named `extendOnce` in line 13. Figure 5.(c) shows the execution of `extendOnce` function twice. For one extension of the isomorphic subgraph, the function extends each subgraph in the isomorphic subgraph with a node of the same position and the same type, or as many subgraphs as possible if they cannot all be extended with an identical node. That is, the subgraph extension uses a maximum priority strategy. As shown in Figure 5.(c), each Sub node (graph) can be extension with an Mul node, which is located on the lower left side of the Sub node. As for the nodes on the upper right of the Sub node, it is not possible to extend all of them, because they are of different types, two are Add nodes and two are Shl nodes. Once they cannot all be extended with an identical node, we need to use fewer subgraphs for the extension. This may result in multiple extensions, and the resulting isomorphic subgraphs cannot be further extended from each other. This ensures the completeness of the extended results. Performing a subgraph search based on one type of actor may lead to duplication. For example, subgraph extension based on the four Mul nodes in Figure 5.(a) also gives the same results as shown in Figure 5.(b). For these duplicate searches we can check the already obtained isomorphic subgraphs to reduce the search space.
Algorithm 1: The largest isomorphic subgraph search

```
Input: Graph: The directed dataflow graph of a given model
Input: TypeSet: All supported types of batch arithmetic actors
Output: LargestIsoGraph: The largest isomorphic subgraph
1    actorMap = {} // A map structure, key: actor type, value: actor list
     // eg: {Add:{A1, A2}, Sub:{A3, A4, A5}}
2    for actor in Graph do 3    if actor.Type in TypeSet then
     4        actorMap[actor.Type].add(actor)
5    isoGraphList = {} // A list of found isomorphic subgraph
6    for key, value in actorMap do
7        isoGraphSeed = isoGraph(value)
8        extIsoGraphQ = {} // A queue of extensible isograph
9        extIsoGraphQ.add(isoGraphSeed)
10       while not extIsoGraphQ.empty() do
11           curIsoGraph = extIsoGraphQ.pop()
12           isoGraphList.add(curIsoGraph)
13           isoGraphExtedList = extendOnce(curIsoGraph)
14           if not isoGraphExtedList.empty() then
15              extIsoGraphQ.add(isoGraphExtedList)
16    LargestIsoGraph = NULL
17    maxActorCount = 0
18    for isoGraph in isoGraphList do
19        depGraph = getDependencyGraph(isoGraph, Graph) // Construct dependency graph between subgraphs
20        maxClque = getMaximumClque(depGraph)
21        // Solve the Maximum Clique Problem
22        indGraph = getIndependentGraph(isoGraph, maxClque) // Obtain the maximal independent isomorphic subgraph
23        if indGraph.ActorCount < maxActorCount then
24            LargestIsoGraph = indGraph
25    return LargestIsoGraph
```

2) Subgraph merging: For the largest isomorphic subgraph from Algorithm 1, we need to merge all the subgraphs in it. The process of merging is performed on the model according to the isomorphic subgraph. The essence of merging subgraphs is to transform those actors at corresponding positions in the isomorphic subgraph into a batch computing actor. But for the integrity of the model, the data merging actors and the data splitting actors need to be added to the model separately, as shown in Figure 3. The model on the right in Figure 3 is constructed after three times of subgraph merging algorithm. The first time, the Add and Sub actors are merged. The second time, the Mul actors are merged. The third time, the Shl actors are merged.

Algorithm 2 shows the detail of subgraph merging process. First, we need two list variables to store the external imports and outputs of the isomorphic subgraph, in lines 2-3. The data for these imports come from actors outside the subgraph, and the data from these outputs are output to actors outside the subgraph. Then, in lines 4-8, the external imports are collected by traversing the actors corresponding to the isomorphic subgraph and determining whether the data source is in the subgraph. We just need to traverse the actors corresponding to the first subgraph in isomorphic subgraph. It is because that other actors can be found through the isomorphic relations. The collecting of external outputs is similar with imports, omitted in line 9. After that, actors for data merging and data splitting need to be created in the model.

For each import in extInportList, a data merging actor will be created to combine individual data into an array, in line 11. It will connect to the original source of the imports with the same position in isomorphic subgraph and break the original connection between these imports and their source. In its place, a batch data connection is made, in line 15. Similar processing is used to create data splitting actors and related connections based on extOutportList, omitted in line 16. Finally, in lines 17-21, the actors corresponding to the first subgraph will be converted to batch computing actors and others will be deleted.

Algorithm 2: Subgraph merging

```
Input: Model: The Simulink model
Input: IsoGraph: Isomorphic Subgraph
Output: OptModel: The optimized Simulink model
1    OptModel = Model
2    extInportList = {} // Store the external imports of IsoGraph
3    extOutportList = {} // Store the external outputs of IsoGraph
4    for node in IsoGraph[0] do
5        // "[0]" indicates the first subgraph in IsoGraph
6        for import in node.actor do
7            if not import.srcNode in IsoGraph[0] then
8                extInportList.add(import)
9        end for
10       end for
11       for port in IsoGraph.getSamePosPort(import) do
12           connect(port.src, actor)
13           disconnect(port.src, port)
14       end for
15       connect(import, actor)
16    end for
17    for node in IsoGraph do
18        if node in IsoGraph[0] then
19            OptModel.convertBatchActor(node.actor)
20        else
21            OptModel.delete(node.actor)
22        end if
23    end for
24    return OptModel
```

B. Actor Classification

Each actor will be translated into a snippet of code representing the execution logic of the actor semantic. In the conventional code generation method of Simulink Coder or DSYnth, actors are translated using actor templates that contain the fire code of each actor. In our work, the intensive computing actors and batch computing actors are separated by PHCG to synthesize more efficient code with SIMD instructions. The above two types of actors are identified and dispatched with the actor type and the input scale.

The intensive computing actor is the actor that takes an array as input, and the output of the actor is calculated from at least one pair of array elements. The input and output elements do not correspond one-to-one. For example, an actor whose type is FFT will be identified as an intensive computing actor, and the Fast Fourier Transform is a complex calculation process with large-scale input. The batch computing actor is the actor that also takes an array as input and output, but different from the intensive computing actor, each element of the output array is calculated from its corresponding input element with the same array index. For example, if the type of an actor is Multiply and at least one of its input ports is an array, the
C. SIMD Instruction Synthesis

The identified intensive computing actors and batch computing actors (Contains the actors resulting from the actor parallelization process.) are passed to the SIMD Instruction Synthesis module for optimal implementation generation.

1) Code synthesis for intensive computing actors: There are many efficient implementations with built-in SIMD instructions for an intensive computing actor, for example, the three implementations of FFT actor presented in Figure 1. But the performance of different implementations varies at different input scales. Hence, to generate more efficient code for deployment, it is necessary to consider the input scale of the actor adaptively. PHCG will perform pre-calculation to decide which implementation is the best for the corresponding input scale. For acceleration, it will also store the history implementation information for a quick search. The overall procedure is presented in Algorithm 3.

Before the pre-calculation, we will perform a preliminary and lightweight search based on the synthesis history information. It will traverse the implementation synthesis history and decide whether there is an existing index that matches the type and input size of the intensive computing actor, as presented in Lines 3-6. If there is a matched index, the corresponding implementation will be returned as the synthesized code for the current actor. If not, the code library will be loaded according to the computing actor type. The code library is a one-to-many implementation list and contains all different implementations for each specific actor.

Then, we will perform pre-calculation on these implementations contained in the library and compare their efficiency on the corresponding input scales. In line 9, a variable is defined to record the minimum cost of the best implementation. To measure the cost of each implementation, a piece of test input data is generated randomly according to the input size of the computing actor, as shown in line 10. In lines 11-14, each implementation in the list needs to be filtered by the input data type and size, because some special implementations only serve special data types and sizes. For example, the Radix-2 FFT implementation aims to speed up the FFT with the input size of $2^n$. In line 14, the implementations that passed the filtering run with the piece of test data and return a cost value. If the cost is lower than the recorded cost, the best implementation will be replaced by the current one with minimum cost also being refreshed, as shown in lines 15-17. Finally, the best implementation for the specific actor with the current input type and size will be stored and returned.

2) Code synthesis for batch computing actors: The code synthesis for batch computing actors is based on the iterative dataflow graph mapping and mainly consists of two steps. The first step of dataflow graph construction is to collect the interconnected actors which have the same I/O scales and bit-width of data element, according to the connections among the identified batch computing actors. The second step of instruction selection is to generate the optimal SIMD instructions based on the iterative mapping on dataflow graphs. Figure 6.(a) and (b) illustrate a sample model and the corresponding directed dataflow graph. Some examples of SIMD instructions shown in Figure 6.(c) will be selected to map to the directed dataflow graph based on their own computing graph. To obtain higher efficiency, PHCG tries to give preference to map more complex SIMD instructions. The algorithm of SIMD instruction selection is shown in Algorithm 4.

The following describes the details of SIMD instruction selection. To find the largest instructions to map the largest subgraphs of the directed dataflow graph from top to down. The larger the instruction graph mapped, the higher the computation efficiency. First, we need to calculate the batch size and the batch count according to the size of the input data and the bit-width of the vector register. The batch size indicates how much data can be stored by the vector register and the batch count indicates how many batches of input data there are. If the batch count is less than 1, it means that the input data is not enough to completely fill the vector register.
and the conventional synthesis method of Simulink will be called to translate the dataflow graph instead of the SIMD instruction selection, as shown in lines 1-4. A snippet of loop code is generated to perform batch calculation cyclically when the batch count is greater than or equal to 2, as shown in lines 5-8. Note that the loop starts with an index of offset, indicating that the length of the remaining data cannot fill the entire vector register. In line 9, the data preparation variable with SIMD data type is generated according to the external input of the dataflow graph. For example, one of the data preparation variable code of the dataflow graph in Figure 6.(b) is $\text{int32x4_t a\_batch = vld1q\_s32(a)}$.

Then the dataflow graph will be mapped part by part until it is completed mapped, as shown in lines 10-22. For a non-empty graph, the topmost and leftmost node will be extended to some subgraphs within the limits of the max graph depth and the max graph node count of the candidate SIMD instructions’ computing graph, as shown in lines 12-13. For example, three subgraphs will be extended from the Sub node (subgraph) in Figure 6.(b), which are Sub – Mul, Sub – Add and Sub, respectively. To obtain higher efficiency, subgraphs with more computation cost will be tried to be matched first. For a subgraph, it must be a convex graph (The nodes of the graph do not indirectly depend on the results of its own nodes.) and its independence must be ensured (It does not depend on any variables that have not been generated), or the subgraph will be discarded, as shown in lines 15-16. In lines 17-19, the matching SIMD instruction will be searched among all candidate SIMD instructions according to the subgraph. If the search fails, the subgraph will be discarded too. Once the matching SIMD instruction is found, in line 20, the calculation code with SIMD will be added into the loop code. For example, the calculation code of the Sub subgraph is int32x4_t Sub\_batch = vsubq\_s32(b\_batch, c\_batch). When the data source type does not match the input data type of the subgraph, a type conversion code with SIMD will be generated. Then the subgraph will be removed from the total dataflow graph to continue the algorithm. Finally, the remaining computation code has the same computation logic as the code inside the loop, and it will be added to the front of the loop code as needed. Listing 1 shows the

```
int32x4_t a\_batch = vld1q\_s32(a); //Load data to vector register
int32x4_t b\_batch = vld1q\_s32(b);
int32x4_t c\_batch = vld1q\_s32(c);
int32x4_t d\_batch = vld1q\_s32(d);
int32x4_t Sub\_batch = vsubq\_s32(b\_batch, c\_batch); //Batch Sub
int32x4_t Shr\_batch = vhaddq\_s32(a\_batch, Sub\_batch);
int32x4_t Add\_batch = vsubq\_s32(Sub\_batch, Sub\_batch);
vst1q\_s32(Sub\_out, Shr\_batch); //Store data to memory
vst1q\_s32(Sub\_out, Add\_batch);
```

Listing 1. The SIMD instructions of the sample model in Figure 6 generated according to Algorithm 4

**Algorithm 4: Synthesis for batch computing actors**

```
Input: Graph: The directed dataflow graph of batch computing actors with same I/O scales and data bit width
Input: InsSet: All candidate SIMD instructions
Input: VectorWidth: The bit width of each vector register
Output: RetCode: The output code with SIMD instruction
1 BatchSize = VectorWidth / Graph.DataBitWidth
2 BatchCount = Graph.DataLen / BatchSize
3 if BatchCount ≥ 2 then
4 return conventionalTranslate(Graph)
5 LoopCode = ∅ // Main loop code for SIMD calculation
6 Offset = Graph.DataLen % BatchSize
7 if BatchCount ≥ 2 then
8 LoopCode.addLoop(Offset, Graph.DataLen, BatchSize)
9 // e.g. for (i = offset; i < dataLen; i += batchSize) {...}
10 LastGraph = Graph
11 while LastGraph ≠ ∅ do
12 Node = LastGraph.getTopLeftNode()
13 SubgraphSet = Node.extendGraphs()
14 //Sort by the cost of subgraph
15 for Subgraph in SubgraphList do
16 SubgraphIns = Subgraph.getMatchInstruction(Subgraph)
17 if InsSet.get() == NULL then
18 continue
19 if InsSet.getMatchInstruction(Subgraph) then
20 InsSet.addIns(Subgraph, SubgraphIns)
21 if Offset ≠ 0 then
22 RemainCode = getRemainCalculationCode(LoopCode)
23 return RemainCode + LoopCode
```

D. Implementation

PHCG 

Model 2 is implemented in C++, with 28,386 lines of code. Unzip and Tinyxml libraries are used to parse the Simulink model. A model optimizer is implemented to parallelize the actors. A synthesis engine is implemented to translate intensive computing actors and batch computing actors to optimal implementations, respectively. Then conventional composition codes are implemented to synthesize the final deployable code.

The implementation and the benchmark Simulink models are uploaded on the GitHub to facilitate the review: https://github.com/CodeGenHCG/HCG.
For the support of cross-architecture, the code library for intensive computing actors and the instruction set information for batch computing actors are extracted as external files. Especially for the instruction set information, the calculation graph and the code format of each SIMD instruction is defined as the following form: \( \text{Graph} : \text{Add}, i32, 4, I_1, I_2, O_1; \text{Code} : O_1 = \text{vaddq} \_s32(I_1, I_2) \). In this way, the SIMD instruction synthesizer just needs to replace the I/O variable for code generation on different architectures.

IV. Evaluation

We evaluate the effectiveness of code generated by PHCG in terms of execution time against DFSynth and Simulink Coder. Besides, we also evaluate the effectiveness of PHCG on different processor architectures with the two most widely used C-Compilers, GCC and Clang. We conducted comparative experiments on the benchmark models of Simulink and DFSynth. FFT, DCT and Conv are models containing intensive computing actors, which are used for Fast Fourier transform, discrete cosine transform and convolution for one-dimensional computing actors, which are used for fast Fourier transform, DFSynth. FFT, DCT and Conv are models containing intensive computation functions and loop calculation codes. We also conducted the experiment on the two most widely used C-Compilers (GCC 11.1.0 and Clang 12.0.1).

Each subfigure in Figure 7 shows the execution time of code generated by Simulink Coder, DFSynth and PHCG running on an ARM processor and Intel processor compiled with GCC and Clang. We can see that code generated by PHCG always performs better than that of Simulink Coder and DFSynth. For example, compared with Simulink Coder and DFSynth on Intel processor with GCC, PHCG decreases execution time by 76.5% and 67.6% on average respectively. The results in Figure 7.(b) are quite different from the others, especially for the batch computing models. This is because the code generated by Simulink Coder contains scattered Intel SIMD instructions, but no SIMD instruction is generated by Simulink Coder to accelerate the computing. Simulink Coder also generates generic functions for intensive computing actors. Same as DFSynth, Simulink Coder has no actor parallelization capability.

B. Effectiveness on Different Architectures

To verify the ability of cross-architecture support, we repeated the experiment mentioned in Section IV-A on Intel architecture (ArchLinux 5.14.16 x64, Intel i7-8700). Since the Intel processor and ARM embedded device we used exist a performance gap, the number of executions on Intel is 10x than ARM. To eliminate the impact of different compilers, we also conducted the experiment on the two most widely used C-Compilers (GCC 11.1.0 and Clang 12.0.1).

Table III shows the average result of the execution time. In general, compared with the code generated by Simulink Coder and DFSynth, the code generated by PHCG decreases the execution time by 41.3%-71.9% and 41.2%-75.4% respectively. These statistics above illustrate that PHCG can generate correct code that achieves higher performance.

The reason for less execution time of PHCG compared to DFSynth is that DFSynth cannot generate batch computation code for intensive and batch computing actors with SIMD instructions, much less for parallelizable actors. It is difficult to obtain better efficiency with DFSynth based on generic intensive computation functions and loop calculation codes. As for Simulink Coder, it supports some SIMD instructions but usually fails to identify some batch computing actors in models. For example, the model named FIR contains two connected batch computing actors, batch Mul (i32*1024) and batch Add (i32*1024), but no SIMD instruction is generated by Simulink Coder to accelerate the computing. Simulink Coder also generates generic functions for intensive computing actors. Same as DFSynth, Simulink Coder has no actor parallelization capability.

C. Comparison on Other Important Aspects

To more fully evaluate our work, we also measured some other important indicators of the codes generated by different works. 1) LoC (Lines of Code): It can reflect the simplicity of the code. Fewer lines of code means simpler code.
Evaluation on ToCG (Time of Code Generation): It represents the ability of the code generator to handle more or more complex models. Table IV below shows the comparison on the above three indicators.

Evaluation on LoC (Lines of Code): For a fair evaluation, we counted the library part of each generated code separately. Because highly efficient code is usually more complex in terms of coding. For example, the FFT library generated by PHCG contains 858 lines of code. But in fact users are more concerned with the code that corresponds to the model logic rather than how the library is implemented. From Table IV, PHCG generates the shortest main logic code on most of the models. This means that users can more easily read, understand or reuse the generated code. It is particularly effective in reducing code when there are parallelizable actors in the model. This is because the actor parallelization process performed by PHCG essentially reduces the number of actors in the model. As for Simulink, it generates code with a lot of redundant data structure definitions and run-time environment configuration. Also in the Conv code generated by Simulink, the convolution algorithm is embedded in the model logic function, which makes the code logic difficult to understand. As for DFSynth, since it generates code with a similar structure as PHCG, it has about the same number of LoC as PHCG on models other than those with parallelizable actors.

Evaluation on MUoP (Memory Usage of Program): As shown in Table IV, all the codes require about the same amount of memory for execution, since they actually use almost the same number of variables and do not use any memory allocation functions such as malloc. Even though PHCG generates more complex and efficient library code for some models with intensive computing actors, it does not bring additional memory overhead. Besides, the gap in the total number of lines of code does not have a significant impact on the memory usage metrics.

Evaluation on ToCG (Time of Code Generation): Since both DFSynth and PHCG are lightweight code generators implemented in C++, they are fast from parsing the model to generating code in just a few dozen milliseconds. While Simulink takes a longer time to accomplish the code generation. So we

---

TABLE IV: COMPARISON ON OTHER IMPORTANT ASPECTS

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>LoC</th>
<th>MUoP (kB)</th>
<th>ToCG (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>DFSynth</td>
<td>45 + L:355</td>
<td>2632</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>47 + L:858</td>
<td>2644</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>DCT</td>
<td>DFSynth</td>
<td>47 + L:448</td>
<td>2640</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>47 + L:888</td>
<td>2652</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>Conv</td>
<td>DFSynth</td>
<td>42 + L:27</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>42 + L:36</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>High</td>
<td>DFSynth</td>
<td>70</td>
<td>2624</td>
<td>1.41s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>66</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>Low</td>
<td>DFSynth</td>
<td>60</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>58</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>FIR</td>
<td>DFSynth</td>
<td>51</td>
<td>2624</td>
<td>1.39s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>53</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>HP(P)</td>
<td>DFSynth</td>
<td>833</td>
<td>2632</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>66</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>LP(P)</td>
<td>DFSynth</td>
<td>520</td>
<td>2628</td>
<td>1.32s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>58</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>FIR(P)</td>
<td>DFSynth</td>
<td>617</td>
<td>2628</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td></td>
<td>PHCG</td>
<td>54</td>
<td>2624</td>
<td>&lt; 0.01s</td>
</tr>
</tbody>
</table>
probed the behavior of Simulink during code generation using a program monitoring approach. The monitoring results show that Simulink uses a large number of temporary files to store code generation such as models and configurations needed for code generation, resulting in a lot of time spent reading and writing to the hard disk. It is worth mentioning that although PHCG uses pre-calculation to select the optimal implementation from the code library, it is able to generate code quickly for models with intensive computing actors because PHCG maintains a synthesis history to obtain the optimal solution faster.

D. Comparison with the Vectorization of Compiler

Because of the SIMD mechanism introduced by the processor, modern compilers also try to compile parallelizable code into SIMD instructions as much as possible [36], [37], [38], [39]. However, current compilers still have a great limitation when it comes to SIMD instruction selection for source code due to the complexity of code analysis [40], [41]. To explore whether the compiler can also achieve the same effect as PHCG, we conducted a comparison experiment on Intel with GCC compiler. We enabled the highest level optimization flag of the compiler (-O3) and also tried three cost models used for vectorization (-fvec-cost-model=unlimited, dynamic, cheap). The experiment results are shown in Table V.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>unlimited</th>
<th>dynamic</th>
<th>cheap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Simulink</td>
<td>41.6%</td>
<td>42.3%</td>
<td>42.3%</td>
</tr>
<tr>
<td>DCT</td>
<td>Simulink</td>
<td>64.2%</td>
<td>64.7%</td>
<td>64.7%</td>
</tr>
<tr>
<td>Conv</td>
<td>Simulink</td>
<td>80.8%</td>
<td>80.8%</td>
<td>80.8%</td>
</tr>
<tr>
<td>HighPass</td>
<td>Simulink</td>
<td>53.8%</td>
<td>52.6%</td>
<td>53.8%</td>
</tr>
<tr>
<td>LowPass</td>
<td>Simulink</td>
<td>59.4%</td>
<td>59.4%</td>
<td>59.4%</td>
</tr>
<tr>
<td>FIR</td>
<td>Simulink</td>
<td>72.5%</td>
<td>72.8%</td>
<td>72.8%</td>
</tr>
<tr>
<td>HP(P)</td>
<td>Simulink</td>
<td>85.3%</td>
<td>83.6%</td>
<td>83.6%</td>
</tr>
<tr>
<td>LP(P)</td>
<td>Simulink</td>
<td>18.8%</td>
<td>18.8%</td>
<td>13.3%</td>
</tr>
<tr>
<td>FIR(P)</td>
<td>Simulink</td>
<td>48.0%</td>
<td>48.0%</td>
<td>50.0%</td>
</tr>
</tbody>
</table>

We can see that the compilers still have a lot of room for improvement in automatic vectorization, especially for the code that we artificially thought would be easy to automatically vectorize, that is, the code generated by Simulink and DFSynth for the models named HP(P), LP(P) and FIR(P). Simulink supports expression folding, so it generates code that looks similar to the code in Figure 2. While DFSynth does not support expression folding, its code looks like performing four Mul operations, then four Add operations, and finally four Div operations. However, only the code for LP(P) generated by Simulink can be vectorized better by compiler. Experiment results show that although the compiler enable the highest level optimization flag the code generated by PHCG can perform better result.

V. Discussion

The extensibility of our work: Currently, PHCG mainly focuses on the Simulink model, but its optimizations can be customized to other models and actors easily, because PHCG only aims to optimize the implementation part of actors and does not affect other actions (e.g., composition part) in code generation. For example, to extend to the model of Ptolemy [15], only one more constraint is needed to be satisfied for dataflow graph construction in Algorithm 4, that is, the batch computing actors must have the same branch information. So, the actors on each branch can be ensured to be translated into code in the correct place. In addition, the optimizations of PHCG can work together with other code generators for more complex scenarios. For example, we can integrate the branch scheduling of DFSynth [9] into PHCG. Furthermore, the parallelization process of PHCG is mainly based on data flow graph, and these methods we proposed would be very suitable for program or compiler optimization if existing code could be represented as such data flow graphs. However, the conversion from code to the high-level data flow graph we need may be complex. The code is more flexible compared to the model because it has elements such as pointer, class object, loop statement, etc. So the extension of our work to program or compiler optimization will be our future work.

The complexity of isomorphic subgraph search: Searching for isomorphic subgraphs in directed graphs is indeed an NP-hard problem. However, the algorithm we proposed is more targeted to data flow graphs with computational node information, which makes the problem we faced much simplified. The main body of the largest isomorphic subgraph search is a BFS algorithm and the time complexity of extending one node at a time is $O(2^N)$ (N is the number of neighboring nodes of the current isomorphic subgraph). Although it is still an NP-hard problem, it hardly reaches the worst case in practical scenarios. First, getting the initial subgraph by node type already decomposes the isomorphic subgraph search problem largely. Second, we have used the already found subgraphs to de-duplicate the subsequent search process. Third, for the subgraph extension process, we use the maximum extension first strategy, which generally takes only linear time to achieve the result of $O(2^N)$ problem in practical scenarios.

The capability of the parallelization strategy: The selection and merging of the largest isomorphic subgraph each time during actor parallelization is essentially a greedy algorithm. It is also difficult to know how to merge actors to obtain the highest efficiency of the generated code for the model, because it is affected by many aspects such as SIMD instruction synthesis, compiler optimization and processor execution. Therefore, this paper only presents a possible approach to parallelize actors. It finds the approximate optimal solution only at the model level using this greedy algorithm. For two largest isomorphic subgraphs or when the second largest graph is close to the largest one, it can be discussed in two cases: (1) If there is no any overlap between the two graphs, then
either one can be merged and the remaining one can be merged at the next iteration. This has no effect on the result of actor parallelization. (2) If there is an overlap between the two graphs, after selecting one of the largest subgraphs for merging, there may be some actors in the remaining part that cannot be parallelized. This may eventually affect the execution efficiency of the generated code, but since we have used SIMD instructions to significantly improve the performance, this effect is much smaller in comparison.

The limitation of SIMD instruction synthesis: Results demonstrate that for the Simulink models with more intensive and batch computing actors, we can achieve higher improvements. Nevertheless, when the model contains one or two batch computing actors, PHCG will still translate them into SIMD instructions. In these cases, the efficiency of the SIMD instructions may be less than the code generated by the conventional method because of the cost of data transmission between memory and vector registers. We can solve this problem by a preliminary check and setting a threshold to trigger the SIMD instruction synthesis.

VI. CONCLUSION

In this paper, PHCG is proposed to optimize the code generation of Simulink models with SIMD instruction synthesis, especially for the increasingly widely-used computing-sensitive models that contain intensive computing actors, batch computing actors and parallelizable actors. More specifically, isomorphic subgraph merging is used to parallelize actors as much as possible; adaptive pre-calculation on input scales is used to mitigate the performance variance of intensive computing actors on different scenarios; and the largest graph mapping based SIMD instruction selection is used to generate the optimal implementations of batch computing actors. Experiments show that PHCG can perform well on benchmark Simulink models. The code generated by PHCG will reduce the execution time by 38.9%-92.9% and 41.2%-76.8% in terms of different compilers and architectures, compared to the built-in Simulink Coder and DFSynth, respectively.

VII. ACKNOWLEDGMENT

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

REFERENCES


12


