HCG: Optimizing Embedded Code Generation of Simulink with SIMD Instruction Synthesis

Zhuo Su[†], Zehong Yu[†], Dongyan Wang[‡], Yixiao Yang✉ [†], Yu Jiang✉ [†], Rui Wang[§], Wanli Chang[¶], Jiaguang Sun[†]

[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[‡] Information Technology Center, Renmin University of China, Beijing, China
[§] Information Engineering College, Capital Normal University, Beijing, China
[¶] Department of Computer Science, University of York, York, United Kingdom

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 HCG, 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. 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 HCG achieves an improvement of 38.9%-92.9% and 41.2%-76.8% in terms of execution time across different architectures and compilers, respectively.

KEYWORDS

Code generation, model-driven design, Simulink, SIMD instruction

1 INTRODUCTION

Simulink is one of the most widely used model-driven design tools and is increasingly used in embedded scenarios [10, 13, 19]. It supports the behavior modeling, simulation, and code generation of embedded control software. 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 [17].

For optimization, expression folding and variable reuse are mainly used in Simulink Coder [16] to generate more compact code. Recently, DFSynth [18] optimizes the code generation of Simulink models with complex branching logic. 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*) and batch computing actors (e.g. *batch Add*).

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 [2, 6]. Take *FFT*(*Fast Fourier Transform*)[1] 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, and we should dynamically select the more appropriate implementation codes based on the model information.

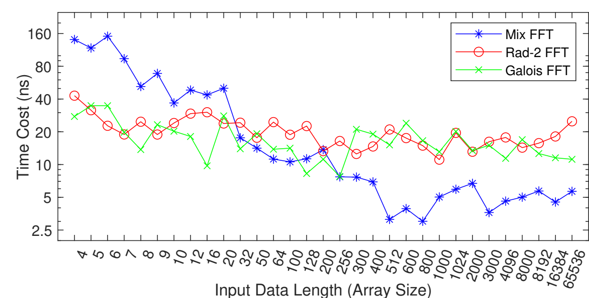


Figure 1: The time cost of different implementations of FFT intensive computing actor on different input data lengths.

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 $vmlaq_f32$ (vector multiplication and addition) and $vrecpsq_f32$ (vector reciprocal) [4, 5]. 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 right shifts the addition result by one bit. When the composition of batch actors is complex, we should select the appropriate compositions of SIMD instructions for vector acceleration.

In this paper, we propose HCG to optimize the code generation of the Simulink models with SIMD instruction synthesis. First, the intensive computing actors and batch computing actors will be identified according to their type and input scale information. Then, for the intensive computing actors, HCG will choose the

[1]Mix 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

Yixiao Yang and Yu Jiang are the corresponding authors.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530582

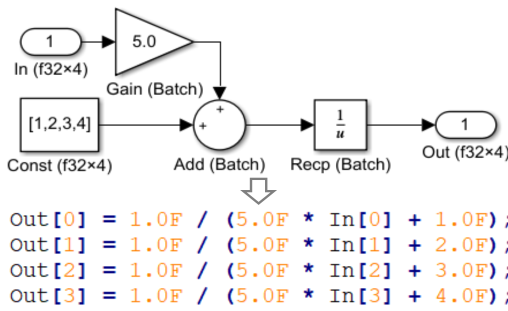


Figure 2: A sample model with batch computing actors and the corresponding code generated by Simulink Coder. It contains four multiplications, four additions and four reciprocal.

optimal implementation based on the pre-calculation according to the actor type and its input scale. For batch computing actors, HCG will convert them into a directed dataflow graph and iteratively generate the optimal SIMD instructions with graph mapping. It will start with some subgraphs that already match some initial SIMD instructions, and then iteratively select the topmost and leftmost node of the dataflow graph to extend those subgraphs. When the graph is completely mapped, more appropriate SIMD instructions will be selected to replace the original initial instruction sets and generate a more efficient and compact code.

We implemented and evaluated HCG on benchmark Simulink models, which also contain intensive computing actors and batch computing actors. The results show that HCG demonstrates excellent performance. Compared with the built-in Simulink Coder and the most recent DFsynth [18], the code generated by HCG achieves an improvement of 38.9%-92.9% and 41.2%-76.8% in terms of execution time across different architectures and compilers, respectively.

2 RELATED WORK

Model-driven design is a widely used software development method, especially in safety-critical embedded scenarios. It mainly consists of three components: behavior modeling, simulation and code generation [8, 9, 13, 15]. There are many supporting tools, such as Ptolemy-II, Tsmart, Polychrony in academic [3, 11, 12], and Simulink, SCADE, DaVinci Developer in industry [1, 7, 16]. Among them, Simulink developed by MathWorks is the most popular for its powerful model simulation and code generation capabilities.

Code generation plays an important role because it will convert the constructed model into code deployed in real embedded devices [3, 16]. Most code generators perform the following steps to generate code: 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 [16] works very well and it supports efficient code generation for different architectures and compilers with optimizations such as expression folding and output variable reuse. DFSynth [18] 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.

The main difference between HCG and those existing generators is that HCG is able to generate more optimal implementation with SIMD instruction synthesis. For those intensive computing actors, it will determine the choice of implementation based on the pre-calculation of the input scales adaptively; and for those batch computing actors, it will determine the proper SIMD instructions set according to the iterative dataflow graph mapping.

3 HCG DESIGN

HCG takes the Simulink model as input and generates efficient and deployable code for embedded devices as output. It mainly consists of two components: *Actor Dispatch* and *SIMD Instruction Synthesis*, as demonstrated in Figure 3. First, the Simulink model file needs to be analyzed by the model parser, and 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, HCG 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, HCG 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.

3.1 Actor Dispatch

For a given Simulink model, the first step is to parse the model into structured actors, ports and other model elements in memory. Then, 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 DFSynth, 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 HCG to synthesize more efficient code with SIM 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 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 actor will be identified as a batch computing actor. Table 1 demonstrates the most frequently used intensive computing actors and batch computing actors in Simulink model libraries [16].

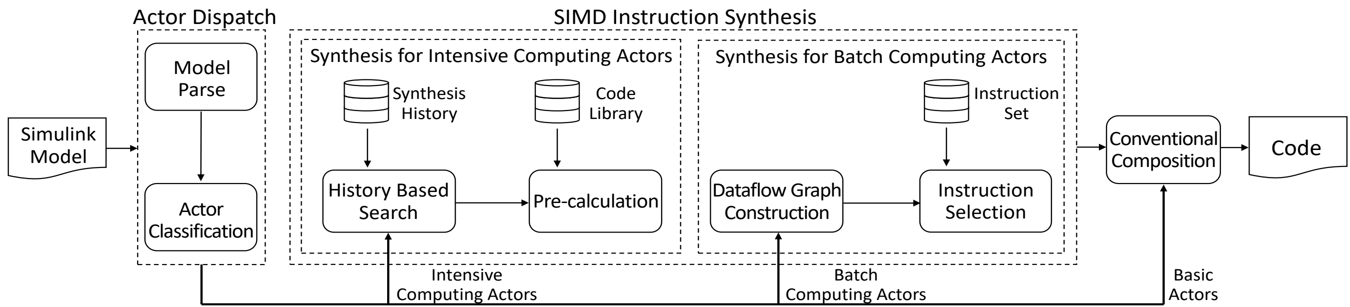


Figure 3: Overview of HCG. The intensive computing actors and batch computing actors in the model are classified for code synthesis after parsing the model. Then each intensive computing actor is translated into an optimal implementation which is suitable for the actor. Batch computing actors are integrated into a dataflow graph and synthesised into SIMD instructions.

Table 1: Most frequently used intensive computing actors and batch computing actors in Simulink model libraries.

(a) Intensive computing actors

Type	Description
MatMul	2x2, 3x3, 4x4 Matrix multiplication
MatInv	2x2, 3x3, 4x4 Matrix inversion
MatDet	2x2, 3x3, 4x4 Matrix determinant calculation
FFT/IFFT	1, 2-D (Inverse) Fast Fourier transform
DCT/IDCT	1, 2-D (Inverse) Discrete cosine transform
Conv	1, 2-D Convolution

(b) Batch computing actors

Type	Description
Add/Sub/Mul/Div	Add, Subtract, Multiply, Divide
Shr/Shl	Right shift, Left shift
BitNot/And/Or/Xor	Bit-wise Not/And/Or/Xor
Min/Max	Minimum, Maximum
Abs/Abd	Absolute, Absolute difference
Recp/Sqrt	Reciprocal, Square Root

3.2 SIMD Instruction Synthesis

The identified intensive computing actors and batch computing actors are passed to the *SIMD Instruction Synthesis* module for optimal implementation generation.

3.2.1 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. HCG 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 1.

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.

Algorithm 1: Synthesis for intensive computing actors

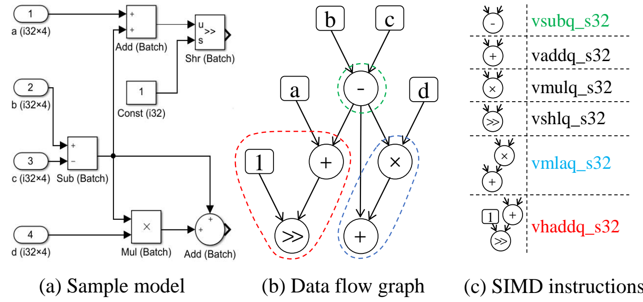
Input: $ActorType$: Type of the intensive computing actor
Input: $DataType$: Data type of the actor's input
Input: $DataSize$: Data size of the input port
Output: $ImplBest$: The selected optimal implementation for the actor

```
1  SelectionHistory = loadSelectionHistory(ActorType)
2  ImplBest = NULL
3  for Selection in SelectionHistory do
4      if Selection.DataType == DataType and
        Selection.DataSize == DataSize then
5          ImplBest = Selection.Algorithm
6          return ImplBest
7  ImplList = loadCodeLibrary(ActorType)
8  ImplBest = ImplList.getGeneralImplementation()
9  MinCost = MAX
10 TestInput = generateTestInput(DataSize)
11 for ImplTest in ImplList do
12     if not ImplTest.canHandleDataType(DataType) or
        not ImplTest.canHandleDataSize(DataSize) then
13         continue
14     Cost = runImplementation(ImplTest, TestInput)
15     if Cost < MinCost then
16         ImplBest = ImplTest
17         MinCost = Cost
18 storeSelection(ActorType, DataType, DataSize, ImplBest)
19 return ImplBest
```

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.

3.2.2 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 4.(a) and Figure 4.(b) illustrate a sample Simulink model and the corresponding directed dataflow graph. Some examples of SIMD instructions shown in Figure 4.(c) will be selected to map to the directed dataflow graph based on their own computing graph. To obtain higher efficiency, HCG tries to give preference to map more complex SIMD instructions. The algorithm of SIMD instruction selection is shown in Algorithm 2.



(a) Sample model (b) Data flow graph (c) SIMD instructions

Figure 4: SIMD instruction selection. (a) is a Simulink model with batch computing actors. (b) is a directed dataflow graph constructed from the Simulink model on the left. (c) shows some candidate SIMD instructions and their corresponding computing graphs. The SIMD instructions named *vsubq_s32*, *vmlaq_s32* **and** *vhaddq_s32* **are selected for potential implementations of subgraphs in (b) with different color.**

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 computational 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,

Algorithm 2: 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.DataLength / BatchSize
3  if BatchCount < 1 then
4      return conventionalTranslate(Graph)
5  LoopCode = ∅ // Main loop code for SIMD calculation
6  Offset = Graph.DataLength % BatchSize
7  if BatchCount ≥ 2 then
8      LoopCode.addBatchLoop(Offset, Graph.DataLength, BatchSize)
         // e.g. for (i = offset; i < dataLength; i += batchSize) {...}
9  LoopCode.addDataLoadSIMDCodeAndVar(Graph)
     // e.g. int32x4_t a_batch = vld1q_s32(&a[i])
10 LastGraph = Graph
11 while LastGraph ≠ ∅ do
12     Node = LastGraph.getTopLeftNode()
13     SubgraphSet = Node.extendGraphs() //Sort by the cost of subgraph
14     for Subgraph in SubgraphList do
15         if isNotConvexGraph(Subgraph) or
            isNotIndependent(Subgraph) then
16             continue
17         Ins = InsSet.getMatchInstruction(Subgraph)
18         if Ins == NULL then
19             continue
20         LoopCode.addCalculationSIMDCode(Subgraph, Ins)
             // e.g. int32x4_t c_batch = vsubq_s32(a_batch, b_batch)
21         LastGraph.removeNodes(Subgraph)
22         break
23 LoopCode.addDataStoreSIMDCode(Graph) // e.g. vst1q_s32(&a[i], a_batch)
24 RemainCode = ∅ // Process the remaining data
25 if Offset ≠ 0 then
26     RemainCode = getRemainCalulationCode(LoopCode)
27 return RemainCode + LoopCode
```

one of the data preparation variable code of the dataflow graph in Figure 4.(b) is $int32x4_t\ a_batch = vld1q_s32(a)$.

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 4.(b), which are $Sub - Mul$, $Sub - Add$ and *Sub*, respectively. To obtain higher efficiency, subgraphs with more computational 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 $int32x4_t\ Sub_batch = vsubq_s32(b_batch, c_batch)$. And 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 SIMD instructions of the sample model in Figure 4 generated according to Algorithm 2.

```
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 = vmlaq_s32(Sub_batch, Sub_batch, d_batch);
vst1q_s32(Shr_out, Shr_batch); //Store data to memory
vst1q_s32(Add_out, Add_batch);
```

Listing 1: The SIMD instructions of the sample model in Figure 4 generated according to Algorithm 2

3.3 Implementation

HCG [2] is implemented in C++, with 26,315 lines of code. Unzip and Tinyxml libraries are used to parse the Simulink model. The synthesis engine is implemented to translate intensive computing actors and batch computing actors to optimal implementations, respectively. And conventional composition codes are implemented to synthesize the final deployable code.

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: $Graph : Add, i32, 4, I_1, I_2, O_1 ; Code : O_1 = vaddq_s32(I_1, I_2);$. In this way, the SIMD instruction synthesizer just needs to replace the I/O variable for code generation on different architectures.

4 EVALUATION

We evaluate the effectiveness of code generated by HCG in terms of execution time against DFSynth and Simulink Coder. Besides, we also evaluate the effectiveness of HCG 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 signal, respectively. HighPass, LowPass and FIR are models containing batch computing actors such as *batch Add*, *batch Sub* and *batch Mul*, which are used for high pass filtering, low pass filtering and finite impulse response filtering, respectively.

4.1 Effectiveness on Benchmark Models

The generated code of HCG, Simulink and DFSynth are all presented in the GitHub repository. For the time used to accomplish the code generation, these three tools performed almost the same, that 2 seconds for Simulink Coder, and 1 second for both DFSynth and HCG. For the time efficiency of the generated code, they executed with the same number of 10,000 times in the same environment (Debian 10 x64, ARM Cortex A72, GCC).

Table 2: Comparison on execution time

Model	Simulink	DFSynth	HCG	HCG Improvement	
				Simulink	DFSynth
FFT	0.459s	0.503s	0.183s	60.2%	63.7%
DCT	0.430s	0.451s	0.121s	71.9%	73.2%
Conv	0.591s	0.722s	0.178s	69.9%	75.4%
HighPass	0.447s	0.446s	0.262s	41.3%	41.2%
LowPass	0.369s	0.305s	0.164s	55.5%	46.1%
FIR	0.415s	0.551s	0.205s	50.6%	62.8%

Table 2 shows the average result of the execution time. In general, compared with the code generated by Simulink Coder and DFSynth, the code generated by HCG decreases the execution time by 41.3%-71.9% and 41.2%-75.4% respectively. Furthermore, we found that the memory usage of the code generated by HCG is almost the same compared to Simulink Coder and DFSynth, with only ± 1% difference, and their computation results of each execution are consistent. This is because the computing resources of the codes generated by these tools are almost the same. These statistics above illustrate that HCG can generate correct code that achieves higher performance while using almost the same amount of memory.

The reason for less execution time compared to DFSynth is that DFSynth cannot generate batch computation code for intensive and batch computing actors with SIMD instructions. It is difficult to obtain better efficiency with DFSynth based on generic intensive computational functions and cyclic computational codes. As for Simulink Coder, it supports some SIMD instructions but usually fails to identify 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 speed. And Simulink Coder also generates generic functions for intensive computing actors.

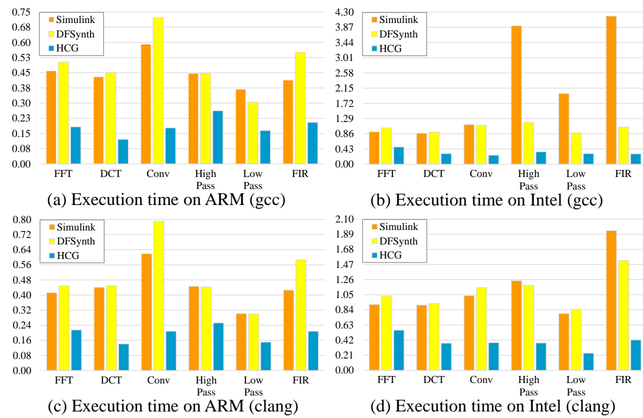
4.2 Effectiveness on Different Architectures

To verify the ability of cross-architecture support, we repeated the experiment mentioned in Section 4.1 on Intel architecture (Arch-Linux 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).

Each subfigure in Figure 5 shows the execution time of code generated by Simulink Coder, DFSynth and HCG running on an ARM processor and Intel processor compiled with GCC and Clang. We can see that code generated by HCG always performs better than that of Simulink Coder and DFSynth. For example, compared with Simulink Coder and DFSynth on Intel processor with GCC, HCG decreases execution time by 76.5% and 67.6% on average respectively. The results in Figure 5.(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 (Some actors are not translated into composite SIMD

[2]The implementation and the benchmark Simulink model is uploaded on the anonymous website to facilitate the review: https://github.com/CodeGenHCG/HCG.

instructions.), and GCC cannot organize these SIMD instructions together, which results in frequent data exchange between memory and vector registers. At this point, memory latency becomes the main performance bottleneck. In contrast, the SIMD instructions generated by HCG are continuous, and the results of SIMD calculation are directly used by the next SIMD calculation without being written to the memory, which effectively avoids memory latency. Note that HCG is not only useful on Intel and ARM, we can simply expand it to other architectures by replacing the corresponding SIMD instruction set in Algorithm 2.



(a) Execution time on ARM (gcc) (b) Execution time on Intel (gcc)

(c) Execution time on ARM (clang) (d) Execution time on Intel (clang)

Figure 5: The execution time of the six benchmark models on ARM and Intel with two different C-Compilers, GCC and Clang. X axis is the models and Y axis is the execution time(s).

4.3 Discussion

Currently, HCG mainly focuses on the Simulink model, but its optimizations can be customized to other models and actors easily, because HCG 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[3], only one more constraint is needed to be satisfied for dataflow graph construction in Algorithm 2, 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. Furthermore, the optimizations of HCG can work together with other code generators for more complex scenarios. For example, we can integrate the branch scheduling of [18] into HCG.

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, HCG 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.

5 CONCLUSION

In this paper, HCG 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 and batch computing actors. More specifically, 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 HCG can perform well on benchmark Simulink models. The code generated by HCG 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.

ACKNOWLEDGEMENTS

This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No.2019YFB1706200).

REFERENCES

[1] Gérard Berry. 2007. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 19–33.

[2] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 253–260.

[3] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. 2002. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *Readings in Hardware/Software Co-Design*, Giovanni De Micheli, Rolf Ernst, and Wayne Wolf (Eds.). Morgan Kaufmann, San Francisco, 527–543.

[4] ARM Developer. [n.d.]. Arm Neon technology. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon

[5] Intel Developer. 2021. Intel® Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[6] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 3. IEEE, 1381–1384.

[7] Vector Informatik GmbH. [n.d.]. *DaVinci Developer.* https://www.vector.com/us/en-us/products/solutions/autosar-classic/

[8] Karim Jahed and Juergen Dingel. 2019. Enabling model-driven software development tools for the internet of things. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*. IEEE Press, 93–99.

[9] Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun, and Lui Sha. 2018. Dependable model-driven development of cps: From stateflow simulation to verified implementation. *ACM Transactions on Cyber-Physical Systems* 3, 1 (2018), 12.

[10] Yu Jiang, Mingzhe Wang, Zhuo Su, Yixiao Yang, and Huihui Wang. 2021. Formal Design of Multi-Function Vehicle Bus Controller. *IEEE Transactions on Intelligent Transportation Systems* 22, 6 (2021), 3880–3889.

[11] Yu Jiang, Hehua Zhang, Zonghui Li, Yangdong Deng, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2015. Design and optimization of multiclocked embedded systems using formal techniques. *IEEE Transactions on Industrial Electronics* 62, 2 (2015), 1270–1278.

[12] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 2003. Polychrony for system design. *Journal of Circuits, Systems, and Computers* 12, 03 (2003), 261–303.

[13] Faruk Pasic. 2018. Model-driven development of condition monitoring software. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, 162–167.

[14] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[15] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. 2019. A model-driven workflow for distributed microservice development. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 1260–1262.

[16] Simulink and Matlab. [n.d.]. *Simulink Documentation.* https://www.mathworks.com/help/simulink/index.html

[17] Claudia M Sosa-Reyna, Edgar Tello-Leal, and David Lara-Alabazares. 2018. Methodology for the model-driven development of service oriented IoT applications. *Journal of Systems Architecture* 90 (2018), 15–22.

[18] 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* (2021).

[19] Jiaguang Sun and Lui Sha. 2016. Safety-Assured Formal Model-Driven Design of the Multifunction Vehicle Bus Controller. *Formal Methods LNCS 9995* (2016), 757.