CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Accelerating Embedded Code of Simulink with Pipeline-Friendly Code Synthesis and Parallel Computing

Master's thesis in Computer Science and Networks

Yeben Zhang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Accelerating Embedded Code of Simulink with Pipeline-Friendly Code Synthesis and Parallel Computing

Yeben Zhang

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Accelerating Embedded Code of Simulink
with Pipeline-Friendly Code Synthesis and Parallel Computing
Yeben Zhang

Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering
Advisor: Henrik Esmaili, Zeekr Technology Europe
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Accelerating Embedded Code of Simulink
with Pipeline-Friendly Code Synthesis and Parallel Computing
Yeben Zhang
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The model-based design (MBD) software development method is becoming more common in the automotive engineering industry due to the growing importance of software in vehicle development. With this approach, automotive engineers create control models visually, and then a code generator in the MBD tool automatically produces the code based on the visual model.

However, code generators often need to consider code compatibility on different platforms. Therefore, the performance of these codes cannot be guaranteed to be optimal. In order to optimize performance, this thesis first examines existing performance optimization schemes. Moreover, two performance optimization schemes are proposed based on the existing methods.

Modern processors are structured around pipelines, which consist of multiple stages. Every instruction must go through each stage. Ideally, each instruction at every stage can be processed in just one CPU cycle. When a task takes longer than one cycle, it causes a pipeline stall, typically due to data hazards. These hazards happen when the execution of the current instruction relies on the data value produced by the previous instruction.

The first method analyzes the pipeline state to identify the cause of a pipeline stall when there is only one pipeline on the CPU. In a pipeline, if the next instruction does not depend on the result of the previous instruction, it does not have to wait for the previous instruction to execute. When such an instruction is executed first, it is referred to as out-of-order execution. A solution was found to reduce the pipeline stall by implementing out-of-order execution.

Based on this approach, Mercury is being researched in academic studies. It proposes an algorithm to minimize data hazards by reordering code statements at the code level. However, the flaws of this method are evident in analysis of the evaluation section. This method adversely impacts the performance of complex models that heavily utilize cache for storing intermediate variables. The thesis investigates the causes of negative optimization. When the data size of these variables exceeds the cache, the performance becomes slower compared to the original Simulink-generated code. This thesis improves the Mercury algorithm. The enhanced thesis method can also optimize the performance of code running on embedded devices with limited cache resources.

The second approach is based on the multi-core architecture of modern processors. It decouples a task system into several small tasks and assigns these tasks to multiple cores at a fine granularity based on the data dependencies relationship and the

execution time. This method minimizes barrier wait times and improves overall execution time.

In the experiment, thesis conducted tests on hardware to compare a new method with an existing one. The hardware used for the tests is the Infineon development board, which is commonly used in the automotive industry to control specific functions in vehicles. This board is equipped with three Tricore processors, each with a 6-stage pipeline to process software instructions. The new method aims to improve the performance of a single task running on a single processor by addressing pipeline stalls. The task in thesis is the control program of two proportional integral derivative controller. This control program is widely used in industrial systems. The program has two inputs. The first input is the expected temperature of the controlled object, and the second is the temperature detected by the sensor. The output of the controller is the voltage of the air conditioning compressor. According to the control parameters, the control program will output different voltages to stabilize the temperature of the controlled object at the expected temperature. Compared to Mercury, the thesis method achieves the same optimization when the cache has sufficient capacity. When the cache resources are insufficient, the thesis method achieves about 20% improvement compared to Mercury. The second approach increases the parallelism rate of multiple tasks and reduces the processing time when multiple tasks are scheduled to the three processors. Compared to the existing coarse-grain scheduling method, the thesis fine-grain method achieves about 4% improvement.

Keywords: Computer architecture, pipeline, data hazard, out of order, multi-core, parallelization.

Acknowledgements

I want to start by thanking Professor Vincenzo Massimiliano Gulisan. Thanks to him for his help throughout the entire project (from planning, language, structuring, and other necessary processes) and for his valuable comments and suggestions, which resulted in this report. Then, I would like to thank advisor Henrik Esmaili. He gave me freedom in the research direction of this master thesis and gave me help. Secondly, I would like to express my sincere thanks to our colleagues at Zeekr Tech EU. They have provided me with a perfect working space and made me feel rich in diversity, multiculturalism, and inclusiveness. In addition, I would like to thank Chalmers for teaching me knowledge in the entire master's program. Especially Professor Vincenzo Massimiliano Gulisan's Operating Systems course and Computer Architecture by Professor Per Stenström. I couldn't completed this project without the knowledge I learned in these courses. Finally, I would like to express my heartfelt thanks to my parents and friends. During the years of study, they have given me endless sprite support and encouragement. Thank you.

Yeben Zhang, Gothenburg, 2024-07-05

Contents

List of Figures

List of Tables

1

Introduction

This thesis is based on optimizing automotive software operations in the automotive industry. In simple terms, there are a large amount of embedded devices in modern cars, and each device runs different software. Some software is written by hand, and a toolchain automatically generates others. This master thesis studies the automatically generated code's optimization running time and memory occupation.

This chapter overviews the entire car's software development process and background. Section 1.1.1 introduces the modern car software development procedure and the complexity that comes with this progress. Section 1.1.2 details the Model-Based Design (MBD) development process and the advantages and drawbacks of this approach. Section 1.2 introduces the significance and aim of this thesis project. Sections 1.3 and 1.4 detail the limitations and ethical factors considered in the thesis project.

1.1 Background

1.1.1 Automotive Industry

Over the past decades, automotive embedded systems have evolved from an electronic and mechanical engineering discipline to a combination of software [1]. The advancements in automotive intelligence and information technology have yielded significant outcomes, notably marked by the escalating adoption of electronic control unit (ECU) chips.

The first and most important reason is that the number of sensors and signal control components in modern vehicles has increased significantly. These include traditional engine control systems, anti-lock braking systems, power steering, and electronic stability systems. There is also electric drive control on electric vehicles, battery management systems, onboard charging systems, entertainment systems, and autonomous driving systems. The increased number of electronic devices also leads to the need for complex software systems to support data exchange and communication between parts and vehicles. For example, advanced driving assistance systems often use camera data, which has a large amount of data in the transmission process. This data is much larger than the traditional sensor data.

Besides, the shift from distributed to centralized electrical and electronic architecture

must be realized to meet the market demand for electrification. Each ECU is only responsible for a single program in a distributed architecture. In a centralized architecture, each ECU runs multiple programs. The complexity of these systems requires more efficient software to ensure real-time operation. This evolution has established software development as one of the most critical technologies in automotive design.

The software employed in various domains of contemporary vehicles exhibits distinct characteristics, including the imperative to ensure complete real-time functionality in the driver assistance systems domain. This different ECU software makes up an overly complex system. Engineers need to develop complex and reliable electronic control and entertainment systems to meet the needs of automotive electronics in vehicle safety, auxiliary driving, ride comfort, and other aspects.

As a result, this type of increasing software complexity leads to longer development times and costs than before. As software complexity increases, the goal is to shift to automating code generation and reducing production and testing time without sacrificing safety. Modern cars may have c microprocessors and codes. Thus, a large amount of automotive companies are trying to solve this challenge with automation chain tools rather than hand-writing code. This type of toolchain requires new development methods, which can save time and ensure safety. For this reason, the automotive industry focuses on a new trend in MBD rather than the traditional approach of hand-writing code. MBD helps bring design prototypes into production, using visual models that can help engineers create complex systems while shortening product development cycles.

1.1.2 Coder Generator Development

Model-to-code generation involves mapping one model language to another using a specific language to describe the mapping process. This approach allows the mapping rules to be modified and reused like ordinary code.

The current well-developed function code generators can analyze models and generate code. In academia, includes Ptolemy-II, as well as Matlab Simulink, Ansys SCADE, dSpace TargetLink, and Vector DaVinci Configurator Pro in the industry [2]–[6].

Most model-based code generators perform the following steps to generate code[7]:

1) model parsing converts the model file into structured Actor information.
2) Scheduling analysis to obtain scheduling relationships among model participants.
3) Code synthesis generates code for each control block.
4) Code combination.

Based on the above, MBD software is widely used for software development in embedded systems, especially in the automotive software development field. This type of software has obvious advantages compared to traditional hand-written code.

1) Through the use of powerful component libraries (essentially written program modules), compared with the C language, it can simplify the complexity of code and programming.

2) The convenience of programming, code maintenance, and debugging is brought by standard graphics. It's like comparing DOS with Windows, one based entirely on a user-friendly graphical interface and the other on plain text. Automotive engineers who have no programming experience can get started with modeling design software quickly.

3) Integrate the work of system engineers and programmers, simplifying the development process. With MBD and automated code production, the boundaries between the work of automotive engineers and programmers become less clear. This property enables the early rapid prototyping and faster validation of products. In the traditional design process, automotive engineers use design information to transmit and process text and picture-based documents, which are difficult to understand and subject to interpretation bias. After that, the programmer manually creates embedded code from text and image-based documents. Finally, the programmer and the automotive engineer work together to verify the design correctly. Programmers and automotive engineers usually don't have a deep understanding of each other's work. These complex steps make the process error-prone and add a large amount of communication costs and time. These tools allow automotive engineers to do most of the work independently with a bit of code knowledge.

4) Automatic code generation and manual coding can be easily combined by placing indicative information in the model to guide the code generation process. Object code no longer requires manual maintenance. In this way, it can eliminate the redundancy characteristic of traditional development. The object code, analysis, and design can be kept in sync manually.

Despite the benefits, this code generation tool has a clear disadvantage. For example, the MBD development approach always focuses on models that represent the functionality of natural systems, which allows developers to do their work at an appropriate level of abstraction, defining the model of the system in advance without considering any details about the implementation of the code. The automated code generator plays an important role. The link of automatic code generation has dramatically weakened the importance of coding in the traditional sense, and the generated code is only one means to achieve the goal. Even the testing is mainly carried out at the model level to test that the output of the models is the same as the design target so that the performance optimization is based on the model, not the code.

Besides, since the generated code follows fixed templates, it causes redundancy and other drawbacks. The advantage of graphics is logical control design rather than pure data operation and management.

Figure 1.1 is a synthetic example of Simulink model with the code generated by Simulink Coder to demonstrate how the model relates to the code generated by the generation tool. The model has seven inputs and one output. Therefore, the code

generated based on this model also has these inputs and output. The code uses multiplication and addition operations, combined with a conditional judgment, to determine the output value of Out1. The automotive engineer is only required to design the input calculation using a visual arithmetic model, as shown in Figure 1.1. The engineer can obtain the code below through the automatic code generator without considering how to create variables, etc.

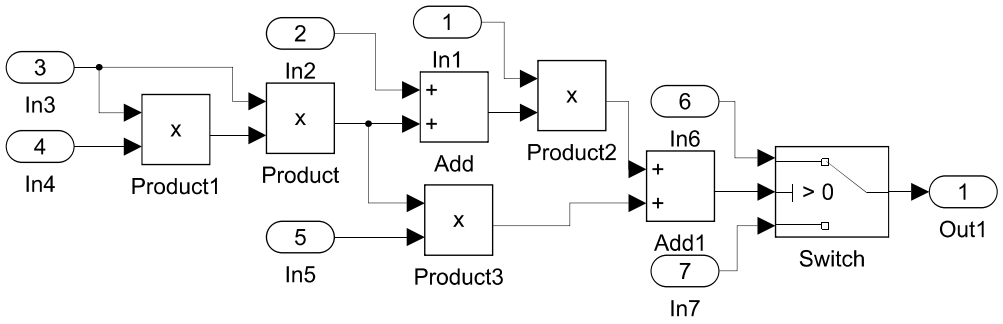

Figure 1.1: Visual Model of MBD.

```
/* Model step function */
void mbd_step(void)
{
  /* Product: '<Root>/Product1' incorporates:
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   */
  mbd_B.Product1 = mbd_U.In3 * mbd_U.In4;
  /* Product: '<Root>/Product' incorporates:
   *  Inport: '<Root>/In3'
   */
  mbd_B.Product = mbd_U.In3 * mbd_B.Product1;
  /* Sum: '<Root>/Add' incorporates:
   *  Inport: '<Root>/In2'
   */
  mbd_B.Add = mbd_U.In2 + mbd_B.Product;
  /* Product: '<Root>/Product2' incorporates:
   *  Inport: '<Root>/In1'
   */
  mbd_B.Product2 = mbd_U.In1 * mbd_B.Add;
  /* Product: '<Root>/Product3' incorporates:
   *  Inport: '<Root>/In5'
   */
  mbd_B.Product3 = mbd_B.Product * mbd_U.In5;
  /* Sum: '<Root>/Add1' */
  mbd_B.Add1 = mbd_B.Product2 + mbd_B.Product3;
  /* Switch: '<Root>/Switch' */
```

```
  if (mbd_B.Add1 > 0.0) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In6'
     */
    mbd_Y.Out1 = mbd_U.In6;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In7'
     */
    mbd_Y.Out1 = mbd_U.In7;
  }
  /* End of Switch: '<Root>/Switch' */
}
```

The thesis researches and improves some code-level optimization methodologies to facilitate and speed up code running. Although automatic code generation frees developers from heavy coding, the efficiency of the generated code is brutal to guarantee and can affect the performance and throughput of the entire system. Addressing this issue can speed up the generated code running for the same model. It will be able to better cope with the increasingly complex system, unified design, and implementation, shorten the product development time, improve the system's reliability to a certain extent, and reduce the maintenance cost of the product in the later period—development test time and expense.

1.1.3 Tricore Pipeline

The Infineon Tricore processors, which have a 6-stage pipeline, are the hardware used in the thesis project. Tricore uses a pipeline architecture different from the x86-architecture. TriCore implements a Harvard architecture with separate addresses and data buses for program and data memories. Instruction fetches can be handled in parallel with data accesses. The super-scalar pipeline has six stages, as shown in Figure 1.2. It consists of Instruction Fetch (IF), Instruction Pre-decode (PreD), Instruction Decode (ID), Instruction Execution 1 (Ex1), Instruction Execution 2 (Ex2), and Write Back (WB). To simplify the analysis, it is simplified as four stages.

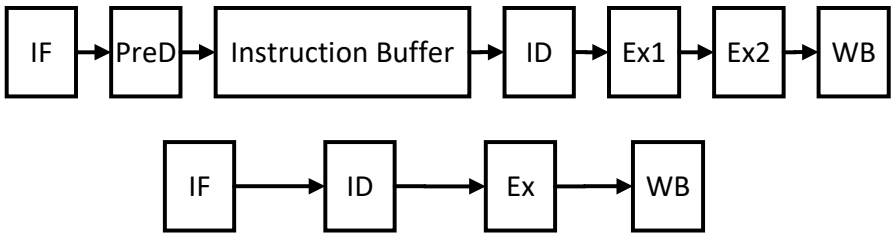

Figure 1.2: Partition of Tricore Pipeline.

1.1.4 Pipeline Stall

Here are two instructions to demonstrate what a pipeline stall is. The first instruction is an addition operation, which adds d1 and d2 and then stores the result in d0. The second instruction is the multiplication operation, which multiplies d0 with d3 and then stores the result in d4. Table 1.1 shows the pipeline analysis diagram for executing these two instructions. The value of the register at the d0 position is available on the back edge of Clock4. At that time, the new value is written back to the register heap, but I2 needs this value at the front edge of Clock4. If the pipeline is not stalled, as shown in Table 1.1. I2 will read an old value in the register at the d0 position and then do the calculation, which causes an RAW hazard.

Table 1.1: RAW Hazard for Instruction 1&2.

Clock	C1	C2	C3	C4	C5	C6
I1: MUL d0, d1, d2	IF	ID	EX	WB		
I2: ADD d4, d3, d0		IF	ID	EX	WB	

To avoid this RAW hazard, the CPU will control the I2 stay in the ID segment for an extra CPU cycle when it detects this hazard, as shown in Table 1.2. Because of this extra cpu cycle, this situation is called a pipeline stall.

Table 1.2: Sloved RAW Hazard for instruction 1&2 by Pipeline Stall.

Clock	C1	C2	C3	C4	C5	C6
I1: MUL d0, d1, d2	IF	ID	EX	WB		
I2: ADD d4, d3, d0		IF	ID	ID	EX	WB

1.2 Aim

1.2.1 Research Significance

Current design tools like Ptolemy-II and Simulink have strong modeling capabilities. However, the increasing complexity of control requirements challenges their simulation and code generation functionalities.

These coder generator tools ignore code compatibility with low-level processor architectures, especially instruction pipelines [8]. As a result, instruction pipeline stall occurs frequently, resulting in additional delays in instruction execution and limited efficiency in embedded software deployment. In other words, all of these generators transform the model into code that contains multiple parts with data dependencies that run in serial without interleaving to execute independent instructions. Therefore, stalls in the instruction pipeline often occur. Specifically, the execution of one instruction needs to wait for other instructions to be processed, resulting in additional delays in the execution of the instruction. In addition, it will research algorithms for task assignments.

This thesis studies the optimization effects of different methods for the given problems. Test and compare the speed of the code optimized by the methods against the code generated by the Simulink Coder.

1.2.2 Problem Definition

This thesis aims to optimize the code generated by Simulink Coder to improve its runtime performance. The main research contents of this paper are as follows:

1) Study the code generation mechanism of Simulink Coder and optimize it on the embedded platform.

2) Please make sure to study how a model is ultimately executed on the hardware device.

3) Finding a code synthesis with less data dependency to arrange the instruction to avoid read-after-write hazards and data relay.

4) The influence of parallelization on multi-core architecture. Propose an optimized task assignment algorithm to improve the multi-core parallelization rate.

1.2.3 Evaluation Metric

The index measured by the standard is the time of code run time and clock cycles. CPU clock cycles refer to the fundamental unit of time in a CPU's operation. Every instruction executed takes a certain number of clock cycles to complete. The CPU's speed determines the clock cycle, typically measured in gigahertz (GHz) or megahertz (MHz).

The CPU used in this thesis has a clock speed of 300 MHz, completing 300 million clock cycles per second. Thus, each clock cycle takes about 33.3 nanoseconds.

1.3 Limitation

The first limitation concerns the way to transform from model to code. In order to consider compatibility, Simulink Coder roughly generates the codes of the model, i.e., the order in which modules appear in the model is the order in which the corresponding functions of the module run in the code. However, this method does not consider the effects caused by data dependence. It needs to be avoided in research. After the code reorders, the result of the function should be the same as the model.

Another limitation concerns the compiler. Some commercial platforms, such as Hightec or Tasking, are available for code editing and compilation of the Tricore architecture. These powerful compilers may come with some performance optimization options. These optimization methods need to be turned off during the experiment.

1.4 Ethical Considerations

The ethical and moral issues to be considered in this thesis mainly include the following aspects.

1) Automotive Safety: The study of safety issues in automotive design, development and testing. Traffic accidents may occur during vehicle road testing, which should avoided as much as possible.

2) Environmental protection: The development process impacts the environment and energy. For example, early testing will be conducted in wind tunnels, consuming more energy. Thus, it is better to simulate it on a computer, which is more environment friendly.

3) Privacy protection: Some test data may derived from personal user data collected and shared from the vehicle, and individual characteristics data needs to be erased to protect the owner and passenger's privacy.

1.5 Report Structure

The first chapter introduces the subject's background and the research's purpose and significance. Then, realize automatic code generation, the research status and development trend, and the limitations of the thesis project.

The second chapter introduces the structure and scheme of the automatic code generation tool, including the customization of the Simulink Coder and the functional components of the automatic code generation tool. It also briefly introduces the standards the tool refers to, such as the AUTOSAR standard and the Infineon Tricore platform.

The third chapter introduces some related work about two optimization directions. In addition, it expounds on the drawbacks of these works and the need for improvement in the thesis's approach.

The fourth chapter elaborates on the method of the thesis in theory. How the thesis method works, and its feasibility is analyzed theoretically.

In the fifth chapter, multiple models are developed and tested using various methodologies, and the results are thoroughly analyzed and compared.

The sixth chapter is the conclusion of this thesis project.

2

Preliminary

This chapter introduces the comprehensive preparatory knowledge that is needed for the project. It begins with an introduction of data hazards in Section 2.1. Then, it introduces the V-shape design model in Section 2.2. Section 2.3 delves into the mechanics of MBD, offering insights into its workings and applications within the industry. Following this, Section 2.4 elucidates the intricacies of the Infineon platform architecture. In Section 2.5, the principle of the code generator takes center stage, elucidating its significance in automating the code generation process from models. Furthermore, Section 2.6 introduces AUTOSAR, underscoring its indispensable role in ensuring standardization and compatibility across the industry. Section 2.7 navigates the path from C code to hardware deployment, presenting a systematic approach to the final implementation phase. Lastly, Section 2.8 introduces an industry model sample.

2.1 Data Hazard

In general, there are data dependencies on register and memory operands. In a machine that executes one instruction at a time, data dependencies do not cause any hazard since the results of all preceding instructions are available before an instruction is fetched. However, when multiple instructions are in execution simultaneously, data dependencies can cause data hazards, which must be resolved for correct operation. Data hazards are caused by data dependencies and the micro-architecture's structure and instruction flow. Three types of data dependencies are as follows [9].

1) Read After Write (RAW) is a subsequent instruction that needs the current instruction's result. If the dependent instruction does not wait for the operand, it may read an error value.

2) Write After Read (WAR) is a subsequent instruction written to an operand that is read by the current instruction. If the write instruction can race ahead of the read, the current instruction reads a future value instead of the current value.

3) Write After Write (WAW) is a subsequent instruction written into the same operand as the current instruction. If the second write is allowed to race ahead of the first write, then the final value of the operand is the former value and, therefore, is an error.

2.2 Automotive Software Development

The automotive software design process follows the V-model process. The V-model of software development does not follow a linear approach. Its process gradually goes down before the source code stage and progressively goes up after the source code stage.

In developing software for complex automotive systems, developers seek flexibility, speed, and the means to enhance the code. Following the steps along the "V" of the model, development begins with design at the upper-left end, continues down to implementation at the bottom, and ends with final testing at the upper-right end.

In this thesis, the approach for performance acceleration is based on software. During testing, verifying the consistency of results between the code and design is essential.

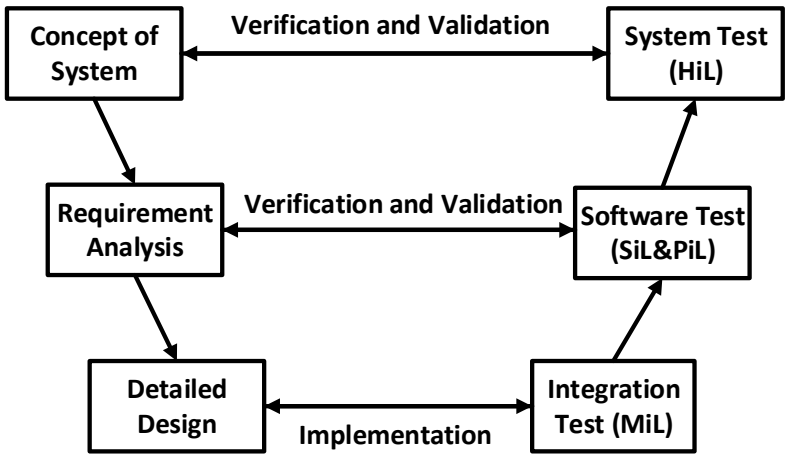

Figure 2.1: V Model in Automotive Development.

2.3 Model-based Software Design

The emergence of MBD is mainly due to the increasing complexity and size of automotive software. MBD makes it easy for engineers to design complex models. The core of MBD development is the model, and the way to describe the model is the model Language. The widely used Modeling Language in MBD is Executable Modeling Language (XML). XML has emerged because it can represent the model accurately. For example, Figure 2.2 is the visualized model of the division calculation in Simulink. The file stored in the computer system is an XML file. Each block and line is described accurately in the XML file.

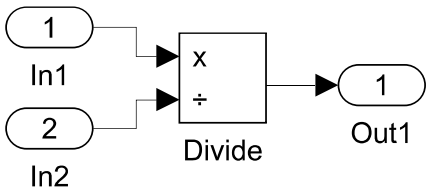

Figure 2.2: A Sample Model for XML File.

```
<Block BlockType="Inport" Name="In1" SID="1">
    <P Name="Position">[300, 263, 330, 277]</P>
    <P Name="ZOrder">1</P>
</Block>
<Block BlockType="Inport" Name="In2" SID="3">
    <P Name="Position">[320, 303, 350, 317]</P>
    <P Name="ZOrder">3</P>
    <P Name="Port">2</P>
</Block>
<Block BlockType="Product" Name="Divide" SID="4">
    <PortCounts in="2" out="1"/>
    <P Name="Position">[370, 267, 400, 298]</P>
    <P Name="ZOrder">4</P>
    <P Name="Inputs">*/</P>
</Block>
<Block BlockType="Outport" Name="Out1" SID="2">
    <P Name="Position">[445, 233, 475, 247]</P>
    <P Name="ZOrder">2</P>
</Block>
<Line>
    <P Name="ZOrder">1</P>
    <P Name="SrcBlock">In1</P>
    <P Name="SrcPort">1</P>
    <P Name="Points">[10, 0; 0, 5]</P>
    <P Name="DstBlock">Divide</P>
    <P Name="DstPort">1</P>
</Line>
<Line>
    <P Name="ZOrder">2</P>
    <P Name="SrcBlock">In2</P>
    <P Name="SrcPort">1</P>
    <P Name="DstBlock">Divide</P>
    <P Name="DstPort">2</P>
</Line>
<Line>
    <P Name="ZOrder">3</P>
    <P Name="SrcBlock">Divide</P>
    <P Name="SrcPort">1</P>
```

```
    <P Name="Points">[18, 0; 0, -45]</P>
    <P Name="DstBlock">Out1</P>
    <P Name="DstPort">1</P>
</Line>
```

For example, the code generator generated the code based on the XML file. Based on the information about the block and line, the generator gets the information about the production step. For example, based on the "BlockType" and "Name" of the block, it gets the type of calculation is division; based on the name of "Name=SrcBlock" and "Name=DstPort", it is aware of the inputs and output variable names of function.

```
void xml_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Divide'
   */
  xml_Y.Out1 = xml_U.In1 / xml_U.In2;
}
```

The modular Model-Based Development (MBD) process for software can be divided into three main parts: modeling, simulation, and code generation. Modeling is used to build models graphically according to user requirements. Simulation is used for model debugging and functional correctness verification.

Code generation is critical in turning a model into code for deployment on an embedded device. More specifically, according to the system's function to be realized, it is independent of any particular platform to describe the system's function. At the same time, transformation rules are formalized according to the specific details of different platforms, and the independent model is transformed into the model related to the particular platform details. Then, the platform-related model is automatically generated into the code implemented by the target language.

2.4 Development Platform and Architecture

For this thesis project, the software will run on embedded hardware. This hardware is based on the specified architecture, which will be introduced below. AURIX (Automotive Realtime Integrated Next Generation Architecture) is a 32-bit Infineon microcontroller family that targets the automotive industry. It is based on the multi-core architecture of up to three independent 32-bit TriCore CPUs. The TriCore architecture refers to the kernel software computing architecture. The AURIX and TriCore are the basis of the hardware platform that will execute the software. Therefore, understanding and leveraging hardware resources is the potential technology basic to help with improving performance.

AURIX platform architecture refers to the hardware architecture of the chip, including

1) Bus communication structure;

2) Peripheral resources;

3) Storage and memory architecture;

4) Computing Cores (Microcontroller, RISC Processor, Digital Signal Processor (DSP)). Each computing unit has different advantages. The microcontroller can support fast context switches and interrupts. The RISC processor supports fast load/store instructions and super-scalar pipelines. DSP supports multiply-accumulate (MAC) instructions and complex mathematical calculation instructions. The Tricore used in this thesis has three computing cores, each with these three units.

The computing cores architecture refers to the kernel software computing architecture called Tricore. Figure 2.3 is the most essential part of the architecture of TC297 that is used in this thesis. Its component is tightly related to thesis optimization methodology.

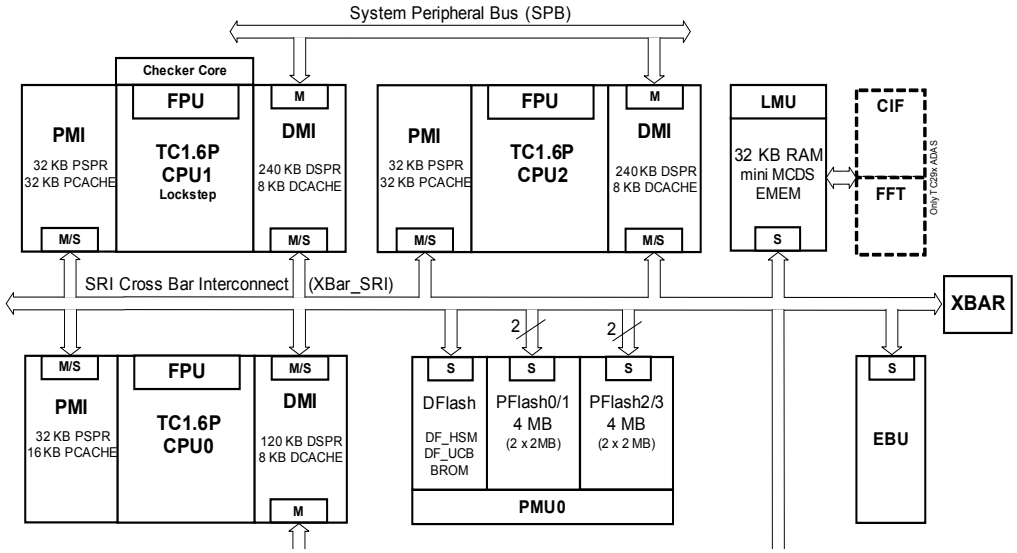

Figure 2.3: TriCore TC297 Architecture without System Peripheral Devices [10].

2.5 Principle of Coder Generator

With the wide application and development of the MBD, the automatic model-to-code generation technology can be wholly and accurately mapped to the code. The model-to-code generation process can be regarded as the model mapping process of mapping one model language to another model language, using a language to describe the mapping process, which allows the mapping rules to be modified and reused like ordinary code.

2.5.1 C/C++ Coder Generator based on Simulink Coder

The Simulink tool is integrated with the Matlab development environment, with control system simulation and visual graphics development characteristics. Simulink integrates Simulink Embedded Coder to generate embedded code.

By Simulink Coder, models can automatically generate production-grade embedded code, which can be customized and tested. Embedded Coders and Tester have the following characteristics and applications:

1) Generate product-grade C code. Simulink Coder is fully compatible with Simulink and can generate production-grade C code from discrete-time Simulink models.

2) Create a code generation report. The Simulink Coder can create a code generation report during automatic code generation. The report describes the code module information and the code optimization performance analysis information, such as function call times and time. With this information, the slow section of the program code can be found, which can potentially be valuable for optimization.

2.5.2 Workflow from Model to Hardware

Figure 2.4 illustrates the workflow for models to code and deploy to the hardware. The code generation is the most critical period. The Simulink Coder integrates the coder build and target language compiler.

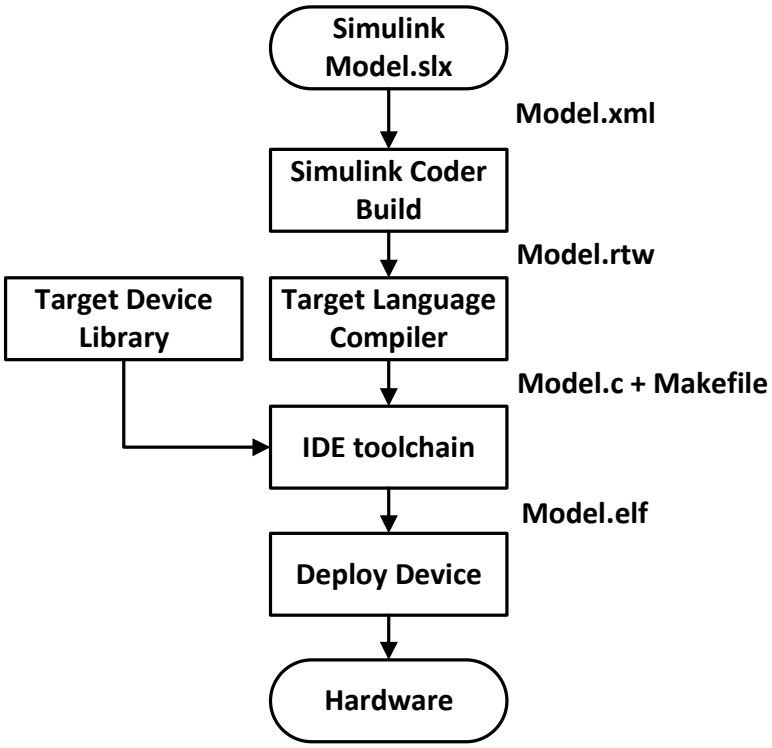

Figure 2.4: The Workflow from Visualize Models to Hardware.

After customizing the embedded target and automatically generating the code using the Simulink Coder tool, the processing parameters that describe the configuration information related to the model and generated code need configure in advance. Then, the model information and configuration parameters are passed into the TLC environment through the generated intermediate file for automatic code generation, and the code generation process is controlled through the top-level control file to realize the creation of the code generation report and the control of the cross-development environment. Subsequently, the code generation process for specific embedded goals has been completed. Finally, the code needs to be compiled in an IDE to create a binary file, which is then deployed to the hardware.

2.6 Automotive Software Architecture

AUTOSAR (Automotive Open System ARchitecture) is a worldwide development partnership of vehicle manufacturers, suppliers, and other companies from the electronics, semiconductor, and software industries [11]. AUTOSAR standardizes automotive electronic systems, provides a standard software architecture model and component-based design to simplify the development, deployment, and integration process, and ensures compatibility and reliable communication and interaction between different systems. It employs a set of communication protocols and interface definitions for communication and interaction between components of different automotive electronic systems. These protocols and interfaces ensure correct and

reliable communication between components.

Although unrelated to performance optimization, it is the reference standard for the code in the thesis project. The Simulink model conforms to Autosar standards in AUTOSAR XML data format. The basic principle of the calculation model to code is the same as Section 2.3.

2.7 Development Toolchain

The primary function of the toolchain is the compiler tool. The compiler will compile the code files into binary files, which can be deployed to the embedded device. The development platform Tasking and HighTec includes compilers such as LLVM or GCC that support TriCore instructions. This type of toolchain will be used as the compiler toolchain in this thesis. Development platforms such as HighTec leverage open-source technologies based on GCC and LLVM to develop C/C+ compiler suites.

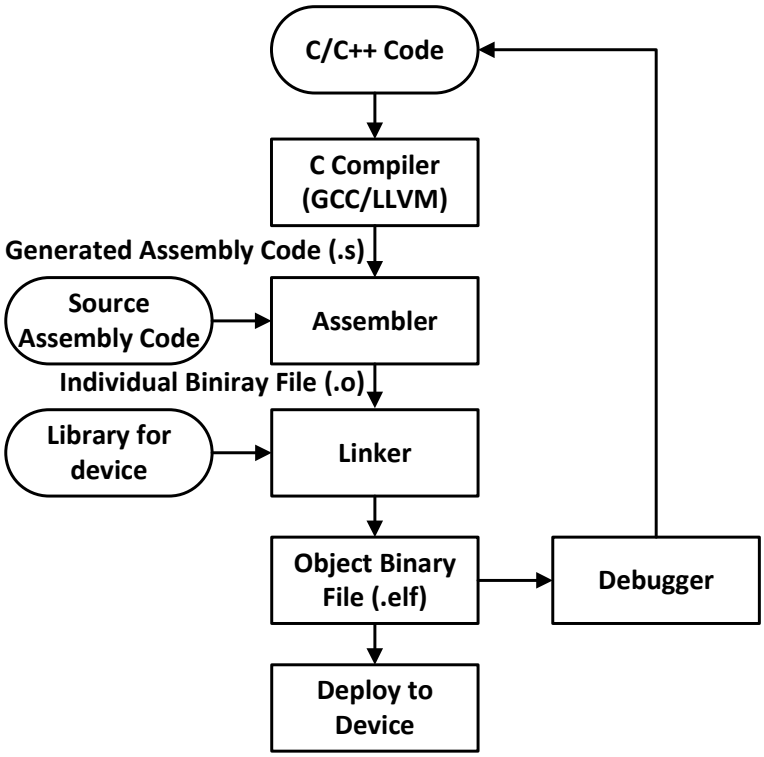

Figure 2.5: Hightec C/C++ Development Platform [12].

2.8 Industry Model Sample

Zeekr Technology Europe AB (Zeekr Tech EU) developed the Joule thermal control system [13]. Joule comprises a battery coolant circuit, an electric drive, a heating, ventilation, air conditioning, and a refrigerant circuit. A schematic overview of the thermal control model is shown in Figure. This model is developed in the Simulink,

and the Simulink Coder generates the code. Then, compile with the toolchain introduced in Section 2.7 and deploy it into ECU.

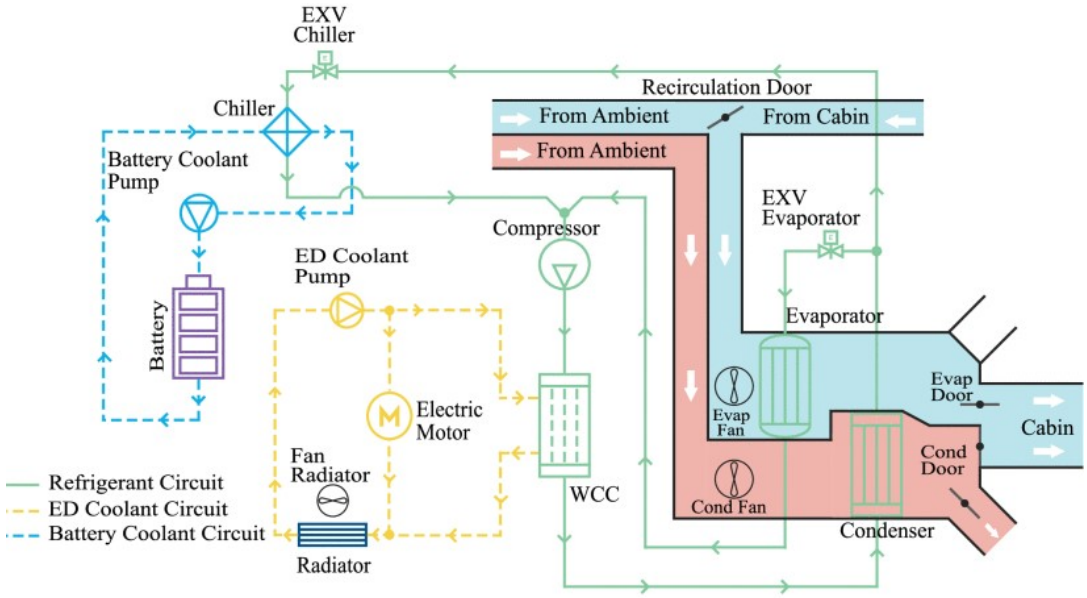

Figure 2.6: ZeekrTecEU Thermal System Model [13].

3

Related Work

When the MBD method was first introduced, it was generally limited to simple model. There was less demand for automatic code generators. The corresponding code generator research gradually increased until MBD was widely used in automotive software development for complex models. This thesis mainly studies two optimization directions, one for the pipeline of single-core and another for multi-core architecture. Section 3.1 introduces data hazards that cause pipeline stalls and discusses how research has solved this problem. Section 3.2 presents the optimization by parallelization on multi-core at the task or function level.

3.1 Optimization of Pipeline Stall

The execution order of the generated code may affect pipeline stall [9]. The current code generators ignore the compatibility between code and low-level processor architectures, especially instruction pipelines. Generators transform the model into code containing multiple parts with data dependencies, which run in serial. Randomly arranged code can lead to frequent stalls in the instruction pipeline.

For example, suppose simultaneously translate modules that have data dependencies. However, since the data required by one module may still be occupied or waiting to be calculated by other modules, it can lead to write after write hazard and read after write hazard. To avoid these hazards, the pipeline must stall and wait for these data to be sent back to the cache or memory, thus causing additional overhead. Therefore, it is essential to select the most appropriate module for the translation sequence to improve the performance of the generated code.

To address this problem, Canedo et al. implemented a skewed pipeline that can use the stall time to improve utilization and reduce communication overhead [14]. This method reduces the communication overhead by delaying some data dependencies for a few iterations while concurrently executing other independent code of the program.

The skewed pipeline reduces communication overhead by delaying specific data dependencies by several iterations. The state change rate that benefits from dynamic system simulation is prolonged, ignoring the fact that several feedback input updates for some modules do not affect the output. This coarseness fine-grained programs to delay communication over multiple iterations to reduce communication costs.

However, the data dependency module of this control-based program is the data

generated by the last loop, which is very rare in the streaming program in this thesis. The data dependency in this thesis is mainly the data generated by calculating the previous module.

With a similar principle, Mercury is implemented by Yu et al. Mercury improves the code selection running algorithm [8]. It is an instruction-aware code generator for Simulink models. Mercury traverses the model data stream to collect data dependencies and estimate execution delays for each participant. Using the information gathered, Mercury generates code that can avoid pipeline stalls.

Mercury consists of three main steps. First, it identifies and records the attributes of each actor in the Simulink model. Then, it obtains the necessary instructions from the model library to execute the corresponding actor, get the data dependencies between the actors, and estimate the execution delay for each actor. A topology-based approach is also used to analyze the collected data dependencies to obtain candidate participants in each translation iteration. Finally, based on the information gathered, it uses the lowest penalty priority, iterative selecting the order in which the instruction pipeline pauses least often to generate code.

However, the problem with Mercury is that they use the greedy algorithm of global iteration to find the code structure with the most negligible data dependence to reduce pipeline stops inside the generated code. However, in some situations, the cost of context switching for loading and storing data is sometimes longer than the pause time of the pipeline. Therefore, in this thesis, it is necessary to consider dividing a suitable range to apply the greedy algorithm to find the local optimal execution order instead of global iteration.

3.2 Parallelization at Task Level

Computer system parallelism refers to the ability to perform multiple tasks or operations simultaneously in a computer system. Parallel computing accelerates the execution of functions by utilizing numerous processors, cores, or computing resources simultaneously. The primary purpose of parallel computing is to improve performance, throughput, and efficiency. Parallelism in computer systems can be divided into multiple levels:

1) Instruction-Level Parallelism: Increases execution speed by executing multiple instructions simultaneously within a single processor or core. It is usually implemented by pipeline technology, superscalar processor, out-of-order execution, and Single Instruction Multiple Data.

2) Thread-Level Parallelism: Improves the performance of a system by executing multiple threads. Threads can be executed parallel within or between the same cores.

3) Task-level Parallelism: The decomposition of a task into multiple independent sub-tasks and the execution of these sub-tasks to speed up the completion of the overall task. Task-level parallelism can be applied to multi-core, distributed, and cluster systems.

The parallelization in this thesis is called Model-based Parallelization (MBP), which assigns modules or subsystems of the Simulink model to the multi-processor. There are two types of MBP: block level and code level.

Block-level MBP draws on task-level parallelism, which sees subsystems as tasks. It is a relatively simple parallelization method. In block-level MBP, different individual subsystems of Simulink are assigned to one core. For example, in the top part of Figure 3.1, Subsystem1 and Subsystem2 can run in parallel on two different cores. This method can extract and parallel simple tasks. However, it does not solve the problem of data dependency. For example, in the bottom part of Figure 3.2, the output of Subsystem2 is one input of Subsystem3. These two subsystems will be assigned to run on identical cores since they are seen as a single unit.

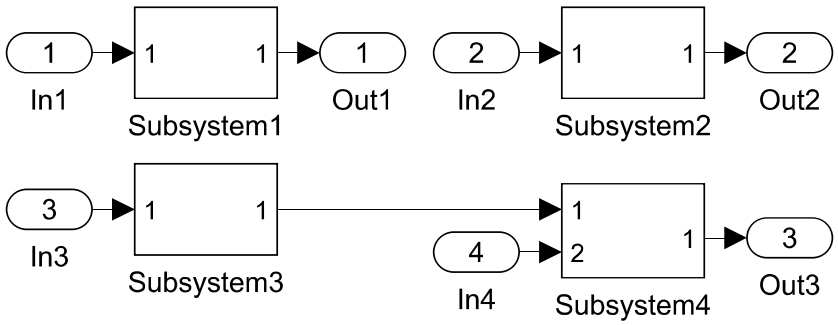

Figure 3.1: A Example of Multi-subsystem.

To solve this problem, Zhong et al. proposed a hierarchical clustering method that facilitates task-parallel code generation [15]. It is a set of algorithms for mapping functions to cores on a heterogeneous platform.

For code-level MBP, Xu et al. propose one MBP algorithm that exploits function parallelism and assigns to cores by the hierarchical clustering method [16]. This approach treats functions as basic units and assigns them. The code needs to be written manually by Simulink's custom block. It is more complicated for data communication due to the overhead of context switching and thread creation or destruction; the acceleration is poor when the task granularity is fine [17]. When the granularity is finer, the system may require a large number of resources for inter-processor communication (IPC), such as shared memory, message passing, and message pipes [18], [19]), which can reduce the overall system performance.

Fine-grained task models can utilize parallelism to achieve high performance of multi-processor system-on-chip. However, fine-grained models face high communication overhead and difficult scheduling decisions, and these two challenges are interdependent.

Despite more communication costs when granularity is finer, it can provide more optimization space and opportunities for multi-core parallelization. When the granularity is finer, the barrier wait time can be reduced, and the parallelism rate can be improved to make the multi-core operation more efficient. This fine-grained

generates communication overhead, so communication optimization techniques have been developed. Lisane et al. proposed a multi-threaded code generation method based on Simulink, which applied message aggregation optimization technology to reduce the number of resources for IPC [20].

Using Lisane's method, Kai et al. combined message aggregation and communication pipelines to reduce IPC costs. This approach minimizes the communication overhead in terms of communication and execution time by increasing the number per message. Besides, it reduces the memory size by decreasing the number of channels [21].

This research can help minimize the overhead of communication costs. The communication cost caused by scheduling at a functional level of granularity can not be ignored. For the model of this thesis, the coarser the granularity of scheduling, the communication required will decrease exponentially. This thesis will reconsider the trade-off of the communication costs and parallelism rate on multi-core CPUs to make a better level of grain. This level of grain can reduce communication complexity, and the costs of IPC can be ignored as much as possible when scheduling.

This thesis aims to design a static task assignment algorithm that can consider the causality of each module and divide the code into data-dependent parts and non-data-dependent parts based on the module's input and output information and computation information. Then, assign these tasks to different cores to achieve acceleration.

4

Methodology

4.1 Optimization of Pipeline Stall

4.1.1 Data Hazard of Simulink Model

In this thesis, the main hazard is RAW. A Simulink model shown in Figure 4.1 was used to analyze the occurrence of RAW hazards in the code generated by the Simulink Coder. Table 4.1 is a pipeline state table illustrating how it happened in Tricore. The Simulink code generates the code as below.

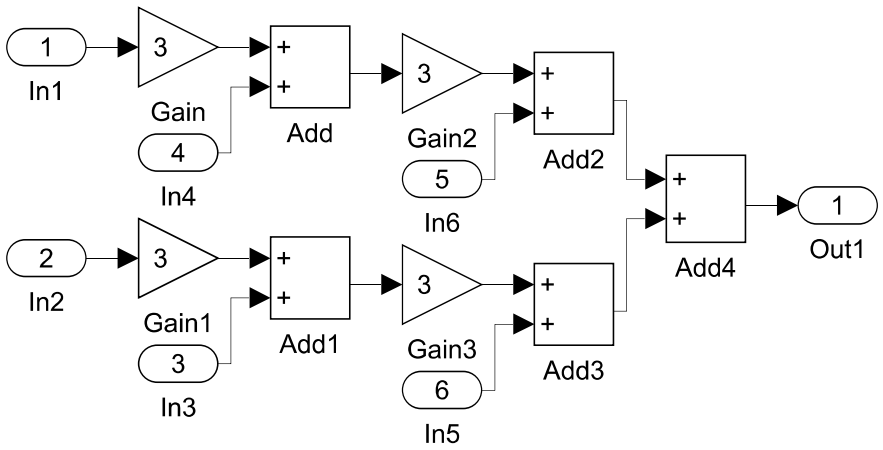

Figure 4.1: Example Model with RAW Hazard.

```
void RAW_step(void)
{
  RAW_B.Gain = 3.0 * RAW_U.In1;          //I1
  RAW_B.Add = RAW_B.Gain + RAW_U.In4;    //I2
  RAW_B.Gain2 = 3.0 * RAW_B.Add;         //I3
  RAW_B.Add2 = RAW_B.Gain2 + RAW_U.In6;//I4
  RAW_B.Gain1 = 3.0 * RAW_U.In2;         //I5
  RAW_B.Add1 = RAW_B.Gain1 + RAW_U.In3;//I6
  RAW_B.Gain3 = 3.0 * RAW_B.Add1;        //I7
  RAW_B.Add3 = RAW_B.Gain3 + RAW_U.In5;//I8
  RAW_Y.Out1 = RAW_B.Add2 + RAW_B.Add3;//I9
}
```

For the program segmentation, the instruction pipeline state table will be stalled five times (the stall to avoid data hazard is expressed as "X," and the stall to prevent structure hazard is represented as "Y"). Structural conflict refers to the conflict that occurs because hardware resources cannot meet the requirements of overlapping instruction execution. For example, Instruction 5 and 3 can not run in the same pipeline stage.

Table 4.1: RAW hazard Analysis for Model in Figure 4.1.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
I1	IF	ID	EX	WB													
I2		IF	ID	X	EX	WB											
I3			IF	ID	X	X	EX	WB									
I4				IF	ID	X	X	X	EX	WB							
I5					IF	ID	Y	EX	WB								
I6						IF	ID	X	X	EX	WB						
I7							IF	ID	X	X	X	EX	WB				
I8								IF	ID	X	X	X	X	EX	WB		
I9									IF	ID	X	X	X	X	X	EX	WB

According to the table, the extra pipeline stalls as I1 and I2, I2 and I3, I3 and I4, I5 and I6, I6 and I7, I7 and I8, I8 and I9. In the seven extra pipeline stalls, one is a structural hazard from Instruction 5 that can not be avoided, and the other are RAW hazards. Avoiding the pipeline stall can improve the utilization of the CPU and the program's performance.

The easiest way to solve this problem is to pause the above pipeline. Waiting for the result of the previous instruction to be written back to the register, and then decode the next instruction. Nevertheless, pipeline stalls will reduce the efficiency of the pipeline.

The second approach that can avoid this situation is to reorder the instructions, such as adjusting the order of the instructions and inserting other independent instructions between the two instructions.

The third solution is to add hardware to the pipeline. At the end of the execution of the first instruction, the result of the addition to be written back is passed on to the execution phase of the second instruction. This hardware avoids pauses and requires no additional adjustment of the order of instructions. This hardware is called forwarding or bypassing [9]. However, it often cannot be seamless in the actual pipeline for every instruction, and sometimes, it still takes several cycles to stall the data from the bypass. Besides, while this function is available in modern desktop processors, not all processors on embedded devices have this hardware component.

Therefore, modern computer architecture usually uses Out-of-Order (OoO) technology to solve RAW hazards. OoO can used at different levels. The following section will introduce how this technology works at these levels.

4.1.2 Methodology

4.1.2.1 Out-of-Order at CPU Architecture Level

At the computer micro-architecture level, the processor can use OoO execution as Tomasulo algorithm [22]. It is one type of dynamically scheduled pipeline technology

used in the modern CPU. For example, Intel's sixth-generation CPU (Pentium Pro) introduced OoO execution in 1995 [23], [24]. Tricore executes instructions in order, as incoming instructions are pre-fetched and aligned by the instruction fetch unit from the Program Memory Interface (PMI). Instructions are placed in program order in the 6-Instructions buffer and then issued to the pipeline in the First-Come-First-Out (FIFO) order [10].

4.1.2.2 Out-of-Order at Compiler Level

Except at the CPU micro-architecture level, modern compilers such as LLVM and GCC can utilize OoO compiling to optimize code and minimize data hazards [25], [26]. Compilers can rearrange the instructions in a different order than what is specified in the code.

After opening the compiler for OoO optimization, the generated assembly code does not strictly follow the logical order of the code as below.

```
int a, b;
void foo(void)
{
    a = b + 1;
    b = 0;
}
```

Compile the above code without the optimization option (-O0) using arm64-gcc (version 13.2.0), and view the disassembly result of the function foo() using the objdump tool [27].

```
a:
        .zero   4
b:
        .zero   4
foo():
        adrp    x0, b
        add     x0, x0, :lo12:b
        ldr     w0, [x0]
        add     w1, w0, 1
        adrp    x0, a
        add     x0, x0, :lo12:a
        str     w1, [x0]
        adrp    x0, b
        add     x0, x0, :lo12:b
        str     wzr, [x0]
        nop
        ret
```

The GCC compiler supports OoO compiling. The GCC compiler can use OoO compiling after enabling the "-fschedule-insns" optimization option or enable "-O2" or

"-O3" optimization [28]. Without the optimization, the order of variables "a" and "b" are written to memory follows program code order. However, after optimization"-O2 ", the written order of variables "a" and "b" is the opposite of program order. The optimized assembly code converted into C language can be regarded as the following form.

```
int a, b;
void foo()
{
    register int reg = b;
    b = 0;
    a = reg + 1;
}
```

OoO at the compiler level can improve performance by rearranging operations without changing the behavior or the program's result. Most OoO instruction reordering does not cause program execution logic exceptions because the compiler analyzes the context before reordering instructions to ensure that the results of the in-ordering instructions and the reordering instructions are equivalent.

However, the compiler can only guarantee the correctness of the execution results in a single thread, and it is possible to cause occasional bugs due to instruction rearrangement in a multi-threaded execution environment.

For example, a global variable "f" flags whether the shared variable "data" is ready. Problems may be introduced due to compiler reordering without lock programming.

```
int f, data;
    void writedata(int value)
    {
        data = value;   //Statement 1
        f = 1;          //Statement 2
    }
    void readdata(void)
    {
        int res;
        while (f == 0); //waiting for flag f
        res = data;
        f = 0;
        return res;
    }
```

The program has two threads. One updates the value of the data and uses the variable "f" to indicate that the data is ready for other threads to read. The other thread keeps calling readdata(), waiting for the flag "f" to be set, and then returns the read data. Suppose the assembler code produced by the compiler is "f" written to memory before data (Statement 2 before Statement 1). Even on a single-core system, there's a problem. Preemption of readdata() may occur when "f" is set to 1

but before "data" has been written. Because the readdata() finds that the "f" is set to "1" and considers that the value in "data" is ready. However, the actual value of the read data is not the updated value (it may be the last historical data or initialization data).

This anomaly is because the compiler does not know that there is a strict dependency between "data" and "f". This dependency relationship is artificially imposed on the code by the programmer.

4.1.2.3 Out-of-Order on Coder Generator

The key to OoO optimization is correctly identifying which code can be reordered, which can lead to unforeseen and unpredictable bugs as miscalculations occur, as in the above example [9]. For the function safety requirement for AUTOSAR in Section 2.6. Unpredictable data problems may influence the system. The OoO optimization will not be used at the compiler level. Furthermore, OoO also can not used at the processor level since Tricore only supports issue instruction in program order.

OoO optimization should be applied at the program code level during code generation from the model. Some research in this field is introduced in Section 3.2. The first method only optimizes for cyclic variable dependence [14]. It does not optimize for RAW dependence for intermediate variables. The second method, Mercury, optimizes the whole model but produces negative optimizations for the large models running on resource-limited devices due to cache misses.

The cache is effective due to the locality characteristic when the program accesses memory. This locality includes both spatial locality (if one piece of data is used, there is a high probability that the adjacent data will also be used) and temporal locality (if one piece of data is accessed, it will be re-accessed quickly). The temporal locality usually uses the least recently used (LRU) caching mechanism, which can keep frequently accessed data in memory in the modern computer architecture. Taking advantage of this locality cache can achieve a high shot rate [9].

However, the temporal locality is not considered in Mercury. A cache miss is an event in which a system or application makes a request to retrieve data from a cache, but that specific data is not currently in cache memory. If the cache is missing, it must fetch data from the main memory (DDR-SDRAM in x86, Data Flash in Tricore), which has much more overhead than accessing the data from the cache. For example, when executing the addition instruction in the x86 architecture, the data corresponding to a variable is not in the L1 cache. Then, when executing the instruction, the CPU needs to read the data from memory. Accessing this data leads to additional time overhead for hundreds of clock cycles. Under modern computer architecture, according to the memory pyramid structure principle, memory closer to the CPU is faster, and latency is lower. Therefore, when all required data is stored in memory close to the CPU, it can achieve high CPU utilization [9].

The Tricore architecture used in this thesis has Memroy resources, including Level-1 (including Scratch-Pad RAM (SPRAM), cache) and Level-2 (Flash, Electrically Erasable Programmable Read-Only Memory (EEPROM)) as table 4.4 [10].

Table 4.2: Memory on TC297 [10].

core0	32KB Program SPRAM, 16KB Instruction Cache 120KB Data SPRAM, 8KB Data Cache
core1	32KB Program SPRAM, 32KB Instruction Cache 240KB Data SPRAM, 8KB Data Cache
core2	32KB Program SPRAM, 32KB Instruction Cache 240KB Data SPRAM, 8KB Data Cache
Shared	128 KB EEPROM, 8MB Program Flash, 728KB Data Cache

The Data SPRAM (DSPR), Data Cache (DCACHE), and Data Cache (DFlash) are used to store data. DSPR and DCACHE are regarded as L1-memory, and DFlash managed by Program Memory Unit (PMU) is viewed as L2-memory [29]. The access time for these memories is shown in Table 4.5.

Table 4.3: CPU Access Latency for TC29x [10].

CPU Access Mode	latency (CPU clock cycles)
Data read/write access to own DSPR	0
Data read/write access to own DCache	0
Data read/write access to DSPR of other core	8
Data read/write access to DCache of other core	8
Data read/write to flash	20 (typical, may slower)

Figure 4.2 is a model with more middle-layer variables data to illustrate why Mercury can easily cause cache misses in resource-limited devices.

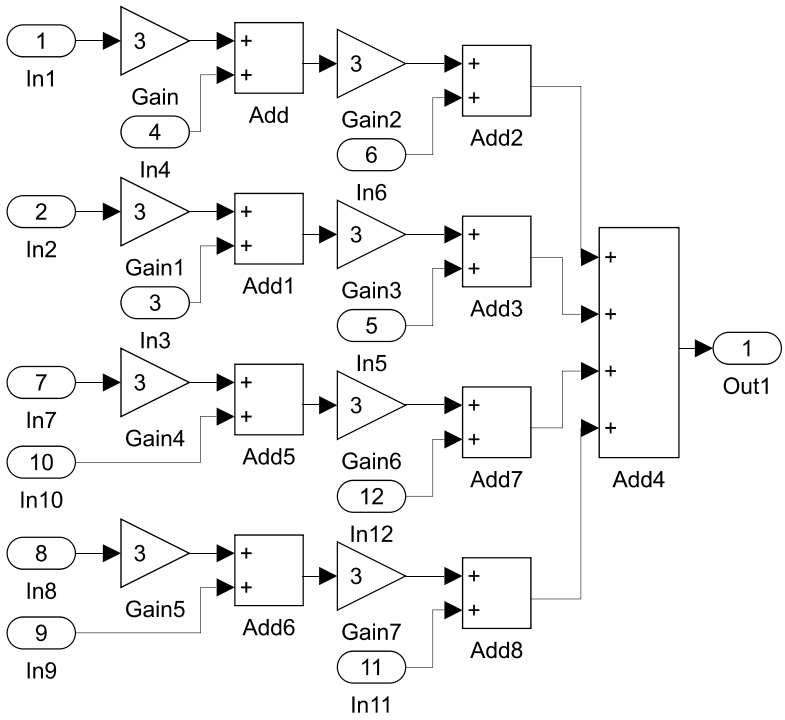

Figure 4.2: Model with Swap Out using Mercury.

The Simulink Coder generates the code as below.

```
/* Model step function */
void a_step(void)
{
  a_B.Gain = 3.0 * a_U.In1;          //Statement 1
  a_B.Add = a_B.Gain + a_U.In4;      //Statement 2
  a_B.Gain2 = 3.0 * a_B.Add;         //Statement 3
  a_B.Add2 = a_B.Gain2 + a_U.In6;    //Statement 4
  a_B.Gain1 = 3.0 * a_U.In2;         //Statement 5
  a_B.Add1 = a_B.Gain1 + a_U.In3;    //Statement 6
  a_B.Gain3 = 3.0 * a_B.Add1;        //Statement 7
  a_B.Add3 = a_U.In5 + a_B.Gain3;    //Statement 8
  a_B.Gain4 = 3.0 * a_U.In7;         //Statement 9
  a_B.Add5 = a_B.Gain4 + a_U.In10;   //Statement 10
  a_B.Gain6 = 3.0 * a_B.Add5;        //Statement 11
  a_B.Add7 = a_B.Gain6 + a_U.In12;   //Statement 12
  a_B.Gain5 = 3.0 * a_U.In8;         //Statement 13
  a_B.Add6 = a_B.Gain5 + a_U.In9;    //Statement 14
  a_B.Gain7 = 3.0 * a_B.Add6;        //Statement 15
  a_B.Add8 = a_U.In11 + a_B.Gain7;   //Statement 16
  a_Y.Out1 = ((a_B.Add2 + a_B.Add3) + a_B.Add7) + a_B.Add8;
}
```

Mercury proposes the optimization by reordering the code in five-layer order. The

order is as the layers of Figure 4.3. This approach eliminates pipeline stalls as much as possible. The optimized code is as follows.

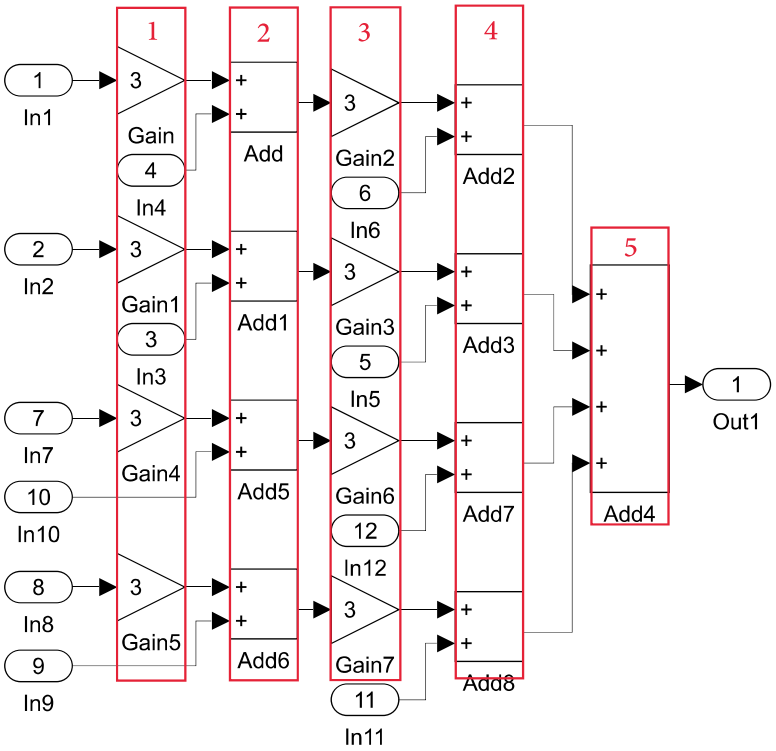

Figure 4.3: Model with Swap Out using Mercury.

```
/* Model step function */
void a_step(void)
{
  /* Layer1 Start */
  a_B.Gain = 3.0 * a_U.In1;          //Statement 1
  a_B.Gain1 = 3.0 * a_U.In2;         //Statement 5
  a_B.Gain4 = 3.0 * a_U.In7;         //Statement 9
  a_B.Gain5 = 3.0 * a_U.In8;         //Statement 13
  /* Layer1 End */

  /* Layer2 Start */
  a_B.Add = a_B.Gain + a_U.In4;      //Statement 2
  a_B.Add1 = a_B.Gain1 + a_U.In3;    //Statement 6
  a_B.Add5 = a_B.Gain4 + a_U.In10;   //Statement 10
  a_B.Add6 = a_B.Gain5 + a_U.In9;    //Statement 14
  /* Layer2 End */

  /* Layer3 Start */
  a_B.Gain2 = 3.0 * a_B.Add;         //Statement 3
  a_B.Gain3 = 3.0 * a_B.Add1;        //Statement 7
  a_B.Gain6 = 3.0 * a_B.Add5;        //Statement 11
```

```
  a_B.Gain7 = 3.0 * a_B.Add6;        //Statement 15
  /* Layer3 End */

  /* Layer4 Start */
  a_B.Add2 = a_B.Gain2 + a_U.In6;    //Statement 4
  a_B.Add3 = a_U.In5 + a_B.Gain3;    //Statement 8
  a_B.Add7 = a_B.Gain6 + a_U.In12;   //Statement 12
  a_B.Add8 = a_U.In11 + a_B.Gain7;   //Statement 16
  /* Layer4 End */

  a_Y.Out1 = ((a_B.Add2 + a_B.Add3) + a_B.Add7) + a_B.Add8;
}
```

When the Mercury method is used for optimization, the first layer of the program generates four intermediate variables.

1) First, running Statement 1 and Statement 5, middle-layer variables "a_B.Gain" and "a_B.Gain1" will be stored in the cache.

2) After statement 9 is executed, "a_B.Gain4" needs to be stored in the cache, but the cache capacity is insufficient. In this case, the LRU algorithm on the Tricore cache architecture will swap out the "a_B.Gain" to data flash.

3) Similarly, after running Statement 13, "a_B.Gain5" will be stored in the cache, and "a_B.Gain1" will be swapped out to data flash.

4) When running Statement 2, the CPU finds that "a_B.Gain" is not in the cache, so it will read from data flash, which incurs a time overhead, which may be even higher than recalculating by Statement 1.

5) Same situation as step 4 when running remaining statements.

To solve this problem, this thesis proposes an optimization method that combines the advantages of the locality of the sked-pipeline and the globality of Mercury [8], [14]. It considers the limitations of embedded devices with small L1 memory capacity. In simple terms, a threshold is set when the code is reordered. The threshold is set according to the L1 memory size and the model characteristics (in addition to intermediate variables, data such as global variables and input/output variables also need to take the L1 memory). This threshold will affect the size of the reorder area.

For the same example as Figure 4.2, assume the cache for the middle-layer variables is 8 bytes. Each middle-layer variable is a single-precision floating-point format that takes 4 bytes. The algorithm in this thesis is implemented on the model. The result is shown in Figure 4.4, and the code is as follows.

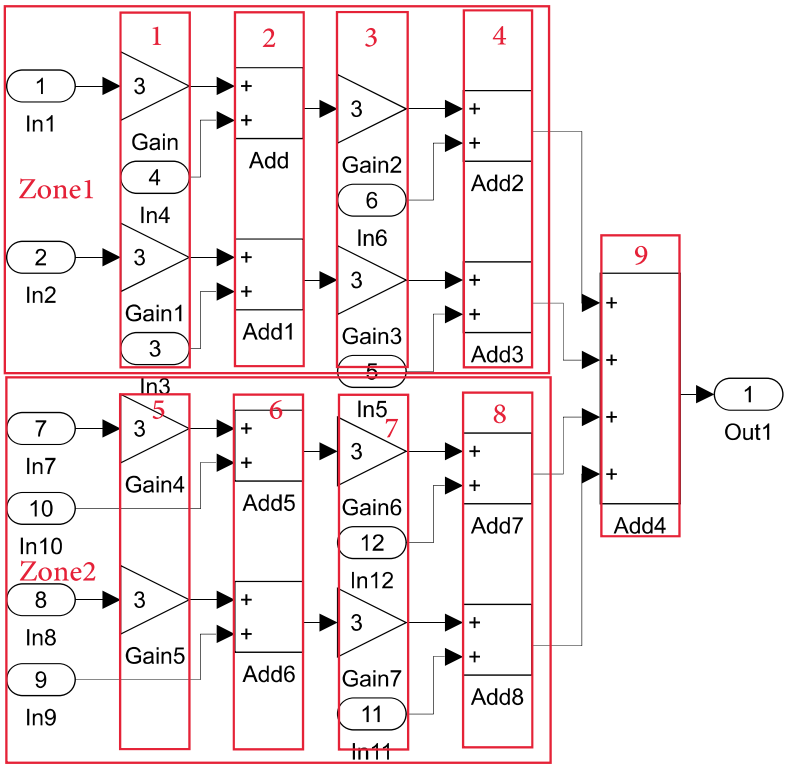

Figure 4.4: The Reorder Optimization of Thesis Algorithm.

```
/* Model step function */
void a_step(void)
{
  /* Zone1 Start */
    /* Layer1 Start */
    a_B.Gain = 3.0 * a_U.In1;          //Statement 1
    a_B.Gain1 = 3.0 * a_U.In2;         //Statement 5
    /* Layer1 End */

    /* Layer2 Start */
    a_B.Add = a_B.Gain + a_U.In4;      //Statement 2
    a_B.Add1 = a_B.Gain1 + a_U.In3;    //Statement 6
    /* Layer2 End */

    /* Layer3 Start */
    a_B.Gain2 = 3.0 * a_B.Add;         //Statement 3
    a_B.Gain3 = 3.0 * a_B.Add1;        //Statement 7
    /* Layer3 End */

    /* Layer4 Start */
    a_B.Add2 = a_B.Gain2 + a_U.In6;    //Statement 4
    a_B.Add3 = a_U.In5 + a_B.Gain3;    //Statement 8
    /* Layer4 End */
  /* Zone1 End */
```

```
  /* Zone2 Start */
    /* Layer5 Start */
    a_B.Gain4 = 3.0 * a_U.In7;         //Statement 9
    a_B.Gain5 = 3.0 * a_U.In8;         //Statement 13
    /* Layer5 End */

    /* Layer6 Start */
    a_B.Add5 = a_B.Gain4 + a_U.In10;   //Statement 10
    a_B.Add6 = a_B.Gain5 + a_U.In9;    //Statement 14
    /* Layer6 End */

    /* Layer7 Start */
    a_B.Gain6 = 3.0 * a_B.Add5;        //Statement 11
    a_B.Gain7 = 3.0 * a_B.Add6;        //Statement 15
    /* Layer7 End */

    /* Layer8 Start */
    a_B.Add7 = a_B.Gain6 + a_U.In12;   //Statement 12
    a_B.Add8 = a_U.In11 + a_B.Gain7;   //Statement 16
    /* Layer8 End */
  /* Zone2 End */

  a_Y.Out1 = ((a_B.Add2 + a_B.Add3) + a_B.Add7) + a_B.Add8;
}
```

Unlike Mercury rearranging the entire module at once, the thesis algorithm divides the model into two zones and rearranges the code within each zone at a single time. This method maximizes the utilization of temporal locality and eliminates pipeline stalls as much as possible while maintaining the L1-memory hit rate. The algorithm of this method is as follows:

```
Mercury(); //as Mercury to divide the model into n layer
for(i=1; i<=n; i++){
    //total middle variables in this layer
    for(j=1; j<=n; i++) {
        data[n]=data[n]+blcok*4; //majority of data is 32bit float
    }
    if (data[n]>max){
        max=data[n]
        maxflag=n
    }
}
int averagesegment = cache/max;
if (averagesegment != 0) {
    int segment=averagesegment++;
}
```

```
else{
    int segment=averagesegment;
}
int partition[segment];
int remainder = cache % max;
for(m=1; m<=segment; m++){
    partition[m]=averagesegment;
}
for(f=1; f<=remainder; f++){
    partition[f]++;
}
//segment the layer with maximum data by partition[segment]
segment();
//by data dependency of Section 2.2 to create the context layer
reorderthecode();
```

4.2 Parallelization of Task

4.2.1 Hierarchy Simulink Model

The industry Simulink Coder is good at generating reliable and high-quality embedded code but is unable to take advantage of the natural concurrency in programs. It is unable to efficiently utilize modern multi-core architectures. The initial program is designed to perform a single task. The code can not be created with various threads or tasks. The Tricore processor used in this thesis has three cores, and the current functions are distributed on Core0 or assigned to 3 cores in a coarse granularity approach.

The concepts of multilevel dependencies and multilevel subsystems are introduced to explain the thesis algorithm. The thesis approach breaks some data dependency relationships while retaining the original semantics of the Simulink model. The thesis proposes a fine-grained task assignment method that breaks the first-level data dependency. In simple terms, the model is divided into several subsystems when building the model, and analyze the running time of each subsystem. Then, an algorithm is proposed to assign different subsystems to different cores.

Figure 4.5 is a Simulink model with 3-Levels subsystems to illustrate the multi-level dependencies and multilevel subsystems. Simulink models have a hierarchical structure. The subsystem as a single unit consists of a model. Each subsystem can contain other lower-level subsystems and a large amount of calculation units.

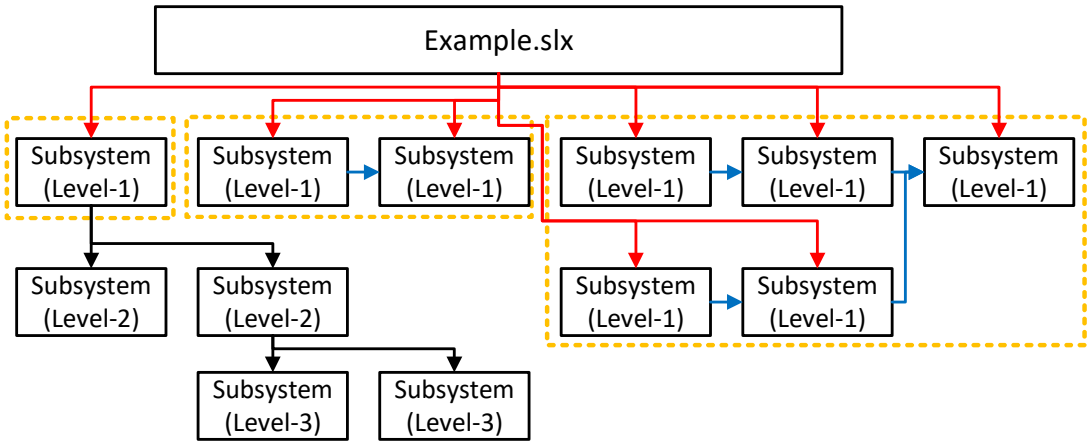

Figure 4.5: 3-Levels Simulink Model.

The file "Example.slx" is a Simulink model that contains various subsystems. Subsystems have different levels. The vector arrow lines represent the subordinate relationship. The horizon arrow lines represent the data dependency relationship. And the blue lines represent the Level-1 data dependency. In theory, a Simulink model file can have infinite levels of subsystems.

It sets some terms to make it easier to understand and analyze the following. The "system" is created as a virtual collection. Each system unit combines all the Level-1 subsystems with data dependencies by the blue line. If the system is scheduled as a single unit, it is called first-level granularity. If the Level-1 subsystem is planned as a single unit by breaking the Level-1 data dependency relationship, it is called second-level granularity scheduling.

The current academic research schedules the task with the finer granularity (function level), as discussed in Section 3.2. The industry possesses a tool that can schedule essential tasks [30]. However, it only uses the coarse granularity method for scheduling. For example, these data-dependent Level-1 subsystems are seen as a single unit to be scheduled and assigned to the core. One commonly used technique for assigning tasks is the greedy algorithm.

4.2.2 Greedy Algorithm

The greedy algorithm is used to solve optimization problems. The basic idea of the greedy algorithm is to ensure that each step is selected optimally to achieve global optimally. It is usually used to solve those problems with optimal substructure. The optimal solution to the problem can be obtained by combining a series of locally optimal solutions. The greedy algorithm selects the optimal solution in the current state at each step without considering the future consequences. Although the greedy algorithm can not guarantee the perfect global optimal solution, it can obtain the optimal or near-optimal solution in many situations.

Applying such a greedy algorithm to assign tasks to cores is as follows:

1) Measure the run time for each system using the test 5.1.3. Instead of operating system dynamic scheduling [18], [19], since the total number of computing tasks in the system is fixed, its running time is also basically fixed.

2) Sort these systems in order of running time. Assign the task with the longest time to the core with the shortest total existing tasks each time.

3) The update iterates through the total duration of existing tasks in this core.

4) Loops through the second step until all systems have been assigned.

```
// Define task structure
typedef struct {
    char name;
    int time;
} Task;
// Define core structure
typedef struct {
    Task *tasks;
    int sum;
} Core;
int main() {
    // for example, has five tasks, three cores
    Task tasks[] = { {'A', 10}, {'B', 5},
    {'C', 8}, {'D', 3}, {'E', 7} };
    int n = sizeof(tasks) / sizeof(tasks[0]);
    int m = 3;
    // Allocate memory
    Core *cores = (Core*)malloc(m * sizeof(Core));
    // Assign tasks to cores
    assignment(tasks, n, cores, m);
    // Free memory
    free(cores);
}
void assignment(Task *tasks, int n, Core *cores, int m) {
    // Initialize cores queue
    for (int i = 0; i < m; i++) {
        cores[i].tasks = (Task*)malloc(n * sizeof(Task));
        cores[i].sum = 0;
    }
    // Sort tasks by their time from long to short
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (tasks[j].time > tasks[j+1].time) {
                Task temp = tasks[j];
                tasks[j] = tasks[j+1];
                tasks[j+1] = temp;
            }
```

```
        }
    }
    // Assign tasks to cores by Greedy Algorithm
    for (int i = 0; i < n; i++) {
        int min_index = 0;
        //Find the core with the least task time
        for (int j = 1; j < m; j++) {
            if (cores[j].sum < cores[min_index].sum) {
                min_index = j;
            }
        }
        //update task of the core
        cores[min_index].tasks[i] = tasks[i];
        //update total time of the core
        cores[min_index].sum += tasks[i].time;
    }
}
}
```

Figures 4.6 and 4.7 visually illustrate that five tasks were assigned to 3 cores by the greedy algorithm. Here are five tasks of varying lengths of time that must be scheduled to the three cores to ensure minimal completion time.

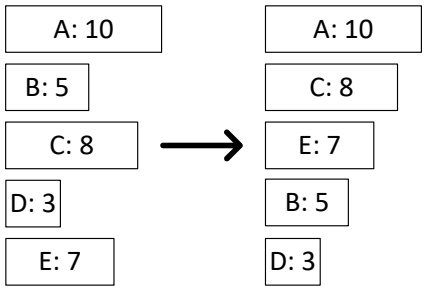

Figure 4.6: Sort 5 Task Units by Time.

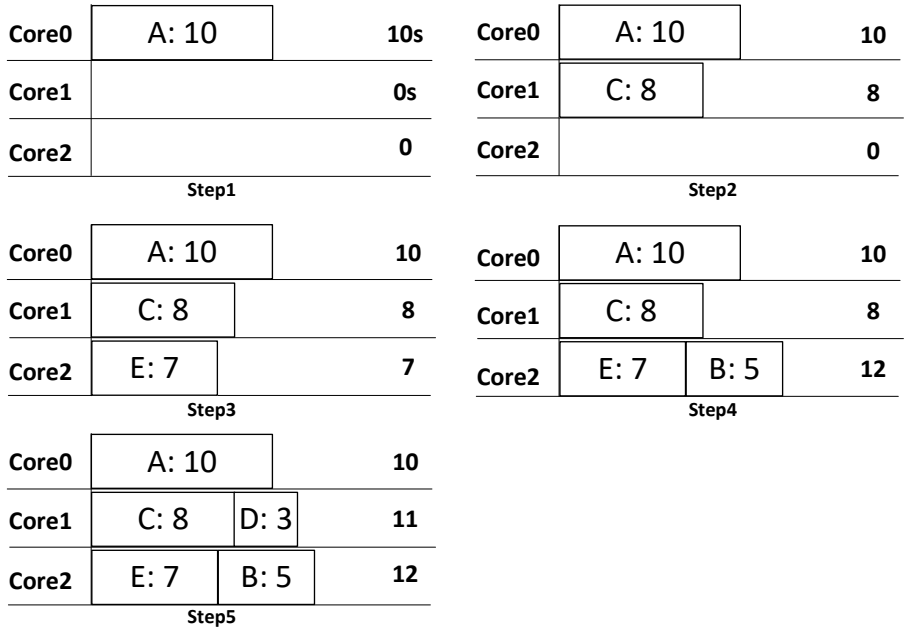

Figure 4.7: Assigned to 3 Cores by Greedy Algorithm.

4.2.3 Hybrid Flow Shop Scheduling Problem

The scheduling of subsystems in the single system in the model can be viewed as a classic hybrid flow shop scheduling problem (HFSP). There are a series of n machines, and each task must follow a specific processing path. After processing on one machine, the following task enters the subsequent machine. It is usually assumed that all queues follow the rule of first in, first out while waiting in the data relationship. This scheduling order is called the arrangement flow shop.

When the task can be processed on multiple machines at a particular processing stage, this flow shop is called a flexible or hybrid flow shop. Hybrid flow combines a flow shop and a parallel core environment.

The difference is that every machine in this thesis is the same; every task can be assigned to any core. For this classic problem, there have been many algorithms for past decades, including reinforcement learning and heuristic algorithms [31], [32].

Heuristic algorithms are based on intuitionistic logic or empirically constructed algorithms that provide a feasible solution for each instance of the combinatorial optimization problem to be solved at an acceptable time and memory cost. One of the most straightforward heuristics is scheduling rules, also known as scheduling policies or list scheduling algorithms. These are simple rules of thumb for sorting and assigning tasks to cores. Many papers are devoted to the study and comparison of various issues [33], [34]. For example, ten scheduling rules for M-level problems are compared with the maximum delay criterion [35].

Scheduling rules have been used to solve many theoretical and practical problems. Many heuristics use a divide-and-conquer strategy, in which the original problem is divided into smaller sub-problems that are solved one at a time, and their solutions

are integrated into the overall solution to the original problem. This thesis draws on the idea of scheduling rules in the heuristic algorithm to improve. Specifically, in this thesis, the following rules are set out.

1) Tasks in the beginning direction of the data flow have a higher priority to avoid deadlocks.

2) Systems with more data dependency layers have better priorities for minimizing barrier wait times.

4.2.4 Thesis Algorithm

To increase the parallelization rate, the thesis suggests a task assignment method based on a greedy algorithm and prior rule. This hierarchical clustering method partitions subsystems with the same attribute into a cluster set, and a greedy algorithm is used for each set to find the local optimal assignment. It has a finer granularity than the method used in industry. Compared to existing academic research, it has a much coarser granularity, which reduces complexity and overhead for communication.

This method has the following characteristics;

- Offline analysis. Each level-1 subsystem is analyzed before distribution to get its run time. Offline algorithms are pre-designed, and all the input data needs to be known before processing the problem. This means the algorithm has complete information about the issue before execution. The advantage of offline algorithms is that they can use all the data for accurate calculation and analysis to obtain an optimal or near-optimal solution.

- Multi-priority, multi-partition. Different priorities are given depending on the number of layers in each system. According to the internal data dependency of each system, different level-1 subsystems have different priorities. These priorities affect the partition on which this l1subsystem is located and, therefore, the priority it is assigned.

- Fine granularity. Compared with packaging and distributing the entire data-dependent system, this algorithm has a finer granularity. As shown in Figure 4.8, multi-to-one, one-to-multi multi-to-one, and multi-to-multi data-dependent subsystems can be scheduled with finer granularity.

- Low IPC overhead. The coarser granularity significantly reduces IPC costs. The model has only dozens of intermediate variables at the level-1 subsystem. This data access overhead can be ignored compared to the time of calculations.

- Modularity. Merging modules with One-to-One dependency.

- Non-preemptive scheduling. The order in which each task runs on each core is fixed.

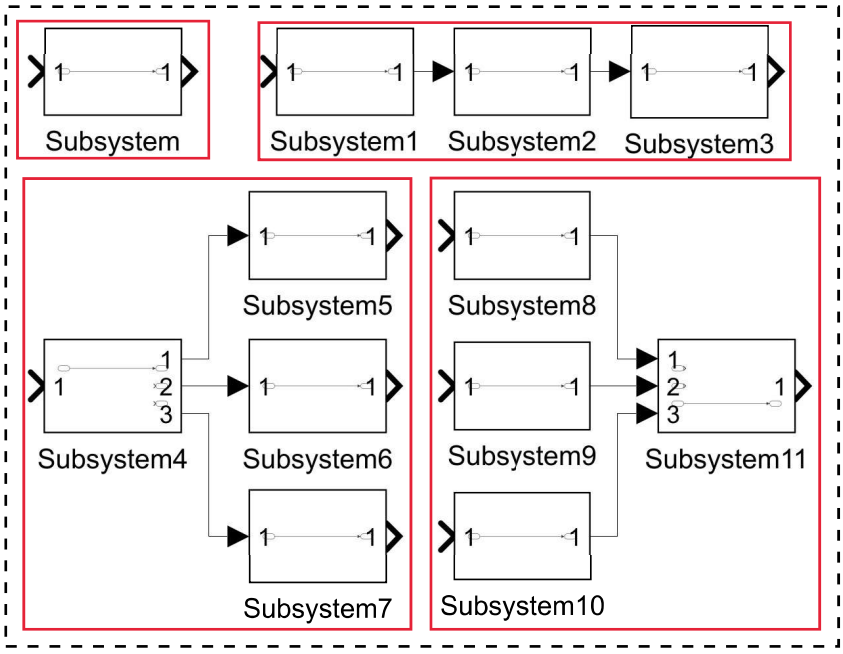

Figure 4.8: 4 Types of System (Single, One-to-One, One-to-Multi or Multi-to-One, and Multi-to-Multi).

The algorithm proposed in this thesis has the following steps.

1. The subsystems in the Simulink model are analyzed as a data flow graph. The data dependence relationship is as lines, and the subsystems are represented as task blocks.
2. Uses the PiL test the run time of each level-1 subsystem. The data of run time will be used to schedule.
3. Merge the modules of the Level-1 subsystem of one-to-one data dependencies. All level-1 subsystems with data dependencies have been seen as virtual system.
4. Analyzes the number of layers of all virtual systems. Grouping all systems with the same number of layers into a zone gives higher priority to zones with more layers and lower priority to zones with fewer layers.
5. Analyzes all system data dependencies relationship and groups them in a data flow direction. The end of the data flow is treated as a group and has the lowest priority.
6. The level-1 subsystem in the group with the highest priority in the highest priority zone is assigned by greedy algorithm
7. Assign level-1 subsystem with the highest priority in the second highest priority zone.
8. When the highest priority group in the lowest priority zone is assigned, go back to step 6 and loop until all level-1 subsystems are assigned.
9. Outputs schedule information for each core.

Figure 4.9 is a sample Simulink model that has fourteen tasks. There are different computing units in each task. It will demonstrate how to schedule these tasks across the three cores using the thesis assignment method.

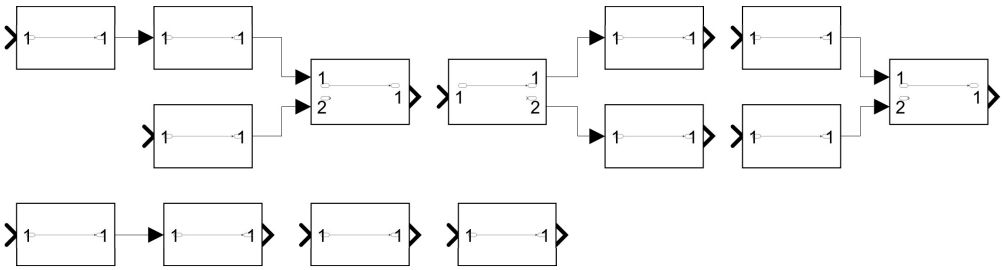

Figure 4.9: Sample Simulink Model

First, the Simulink model is analyzed as a task graph with data flow.

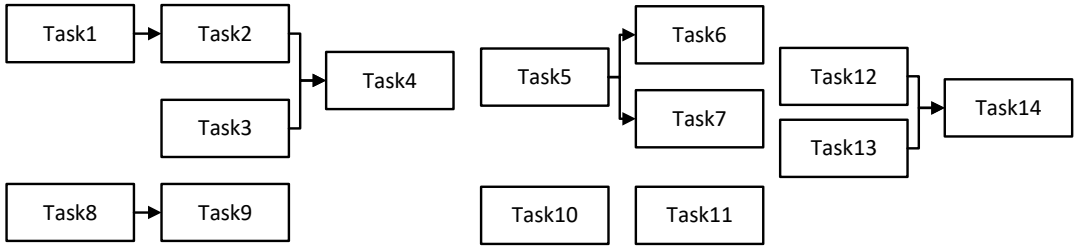

Figure 4.10: Abstracted Tasks from Level-1 Subsystems

Then, using the PiL test to measure the run time of each task. And merge the modules of the level-1 subsystem of one-to-one data dependencies. Abstract the task with data relationship as the system.

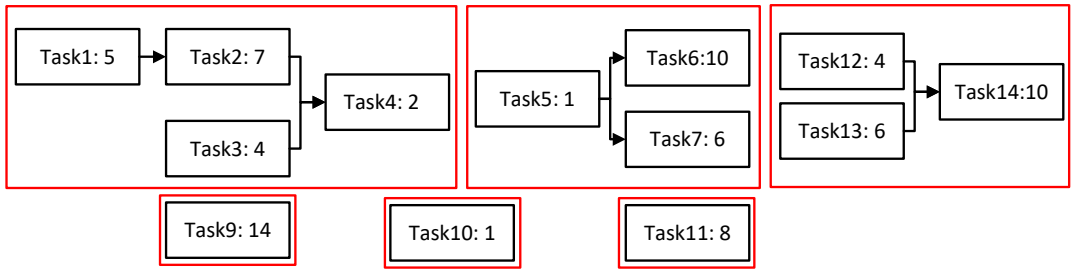

Figure 4.11: Tasks After Pil and Merging

Analyzes the number of layers for each system. Grouping all systems with the same number of layers into a zone. And giving higher priority to zones with more layers and lower priority to zones with fewer layers.

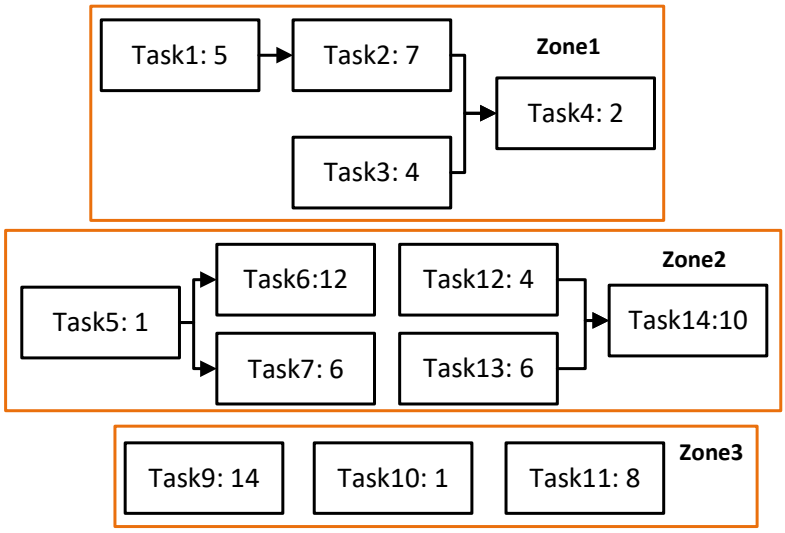

Figure 4.12: Task Partitions by Zone with Priority

Analyzes all system data dependencies relationships and groups them in a data flow direction. The end of the data flow is treated as a group and has the highest priority.

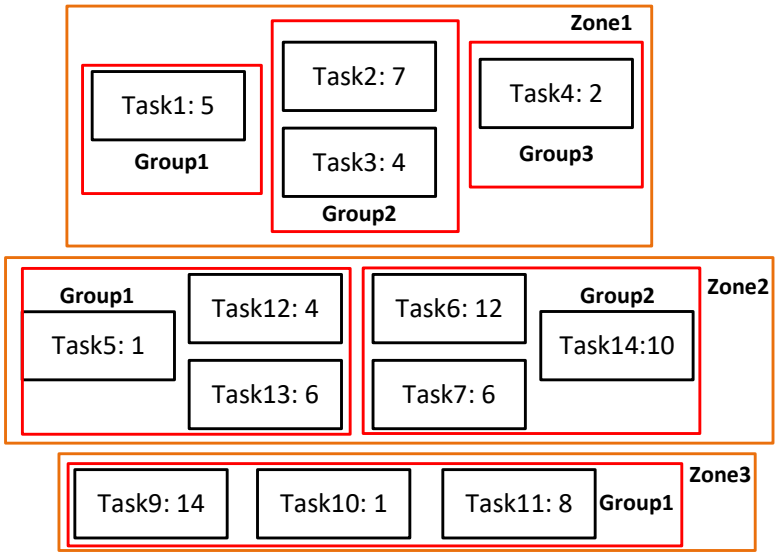

Figure 4.13: Task Partitions by Zone and Group with Priority

Finally, assign the groups in the below order. The greedy algorithm is used inside the group. The assignment process is as follows.

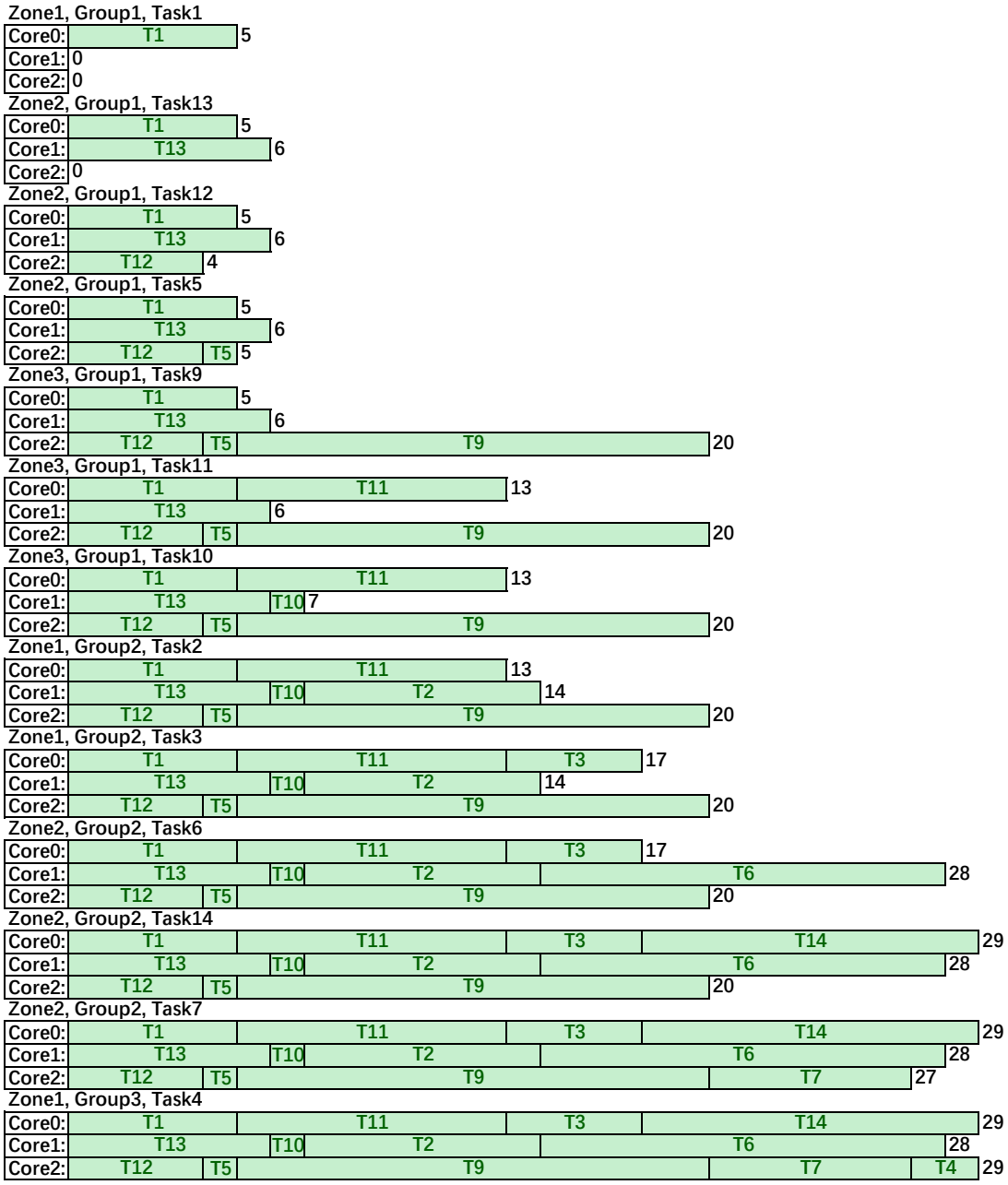

Figure 4.14: Tasks Assignment Process

5

Evaluation

This chapter describes the results and process of experiments on the methodology. Section 5.1 describes the compilation and experimental environment. From Section 5.2 to Section 5.3, two different methods are tested, each with a different scaling and model style. The results are then analyzed and discussed.

5.1 Profiling Environment

To ensure that other factors except the code will not affect the experiment's results, all code generator and compiler settings must remain consistent for all experiments.

5.1.1 Simulink Coder

The version of Simulink and Simulink Coder is 2022b. Some critical parameters of the Simulink Coder are set in Table 5.1.

Table 5.1: Parameters of Simulink Coder.

Default parameter behaviour	Inlined
Hardware instruction set extensions	None
Optimization levels	Debugger
Remove initialization code	Yes
Removing termination code	Yes
Removing unnecessary data support	Yes
Conditional input branch execution	Yes
Signal storage outputs	Yes
Enable local block outputs	Yes

5.1.2 Compiler

The compiler computer is running, as shown in Table 5.2.

Table 5.2: Parameters of Compiler Computer.

CPU	Intel i7-9850H, x86-64, 6-core, 2.6GHz
Memory	16GBx2, 2667MHz
Disk	512GB SSD

The integrated development environment (IDE) used in this thesis is AURIX Development Studio [36]. The specific version is set in Table 5.3. The external compiler used in AURIX Development Studio is the compiler of Hightec C/C++ Development Platform [12], [37]. The compiler's optimization options are set in Table 5.4.

Table 5.3: Parameters of Development Tools.

Development Platform	AURIX Development Studio 1.9.20
Flashing tool	AURIX Flasher Software Tool 1.0.8.0
Infineon AURIX Drivers	DAS V8.0.5
Infineon Low Level Drivers	iLLD 1.0.1.17.0
Java Runtime Environment	OpenJDK 17
Compiler	GCC11 with TriCore support
Compiler toolchain	HighTec GCC toolchain 4.9

Table 5.4: Optimization Options of Compiler.

Coalescer: remove unnecessary moves	Yes
Common sub-expression elimination	Yes
Generic assembly code optimizations	Yes
Control flow simplification	Yes
Common sub-expression elimination	Yes
Automatic function inlining	Yes
Loop transformations	Yes
Unroll small loops	Yes
Convert IF statements using predicates	Yes

5.1.3 Test Environment

Tests are needed before the experiment can be carried out. The purpose of the test is to verify that the results of the generated code are consistent with the results of the model in Simulink. The input data of the test is provided by the various sensors, such as the ambient temperature, battery temperature, and motor temperature data generated by the temperature sensor. The test output is that the model needs to control the object's behavior, such as the refrigerator switch. For example, if the temperature of the battery and motor is beyond the limit, it is necessary to control the opening of the refrigerator to reduce the temperature of these components in the

vehicle. If the program's output matches the model's output when the same input data is used in the code test, then the code passes the test.

There are two types of tests: software in loop (SiL) and processor in loop (PiL). The code in SiL runs on the Windows platform. However, this does not guarantee that the results of the code run on the target processor will be still consistent with the model. The second PiL is also required for testing to ensure that the results of the code run on the target processor are consistent with the model. The only difference between SiL and PiL is the target processor. In the PiL test, the target processor is the Tricore platform.

The SiL test tool is integrated into Simulink, whose version is 2022b [38]. A SiL simulation compiles generated source code and executes the code as a separate process on the host computer. By comparing the results of the model and SiL, it can evaluate the equivalence of the model and the generated code. The running environment of SiL is shown in Table 5.1. The parameter of SiL is shown in Table 5.5.

Table 5.5: Test Configuration.

System Under Test	Top model
Simulation Mode	Normal

After the software and model results are equal, transplant the code to Aurix Development Studio to compile it into a binary file (.elf). Then, it will be deployed on the developer board to do the PiL test. During a PiL test, it can collect code coverage and execution-time metrics for the code. The running environment of the PiL code is shown in Table 5.6.

Table 5.6: Parameter of Developer Board for PiL.

Developer Board	KIT AURIX TC297 TFT
SoC	SAK-TC297TF-128F300N BC
CPU	3 TriCore1.6P with 300MHz
Cache and Scratch-Pad RAM	As Table 4.4
ROM	As Table 4.4
Debugger	USB miniWiggler JTAG Debugger
Connector	Micro USB

5.1.4 Evaluation Metric Method

The SiL and PiL tests can measure the run time of the entire program. The CPU clock cycle will be used to measure the run time of a particular code segment of the program more accurately. The Tricore supports to measure the performance as the CPU Clocks Counter (CCNT) and Instruction Counter (ICNT) [39]. The example is as follows.

```
typedef struct
{
    IfxCpu_Counter instruction;       // Instruction counter
    IfxCpu_Counter clock;             // CPU clock counter
} IfxCpu_Perf;
// Set as a global variable to display in the debugger
IfxCpu_Perf g_perfCounts;
float32 time_ns;
float32 IPC;
// Reset and start counters in normal mode
IfxCpu_resetAndStartCounters(IfxCpu_CounterMode_normal);
// Run program segment to measure clock cycles and instructions
test();
// Stop counters and get results
g_perfCounts = IfxCpu_stopCounters();
//cacualt real time()
//The CPU frequency is 300MHz,so per cycle need 33.3 ns
time_ns=g_perfCounts.clock*33.3;
IPC=g_perfCounts.instruction/cpu_clock_cycle;
```

For example, Figure 5.1 is the variable before running the program. Figure 5.2 is the variable after running the program. The total instruction for this program is 1342. The total CPU clock cycle is 1405. The IPC is 0.955, and the real time is 46786.5ns.

Variables × Breakpoints

Name	Type	Value
'CPU_Perf_Counters.c'::g_perfCounts	struct	{ instruction = ; clock = ; counter1 = ; cou
instruction	struct	{ counter = 0; overlfow = 0 '\0'; }
counter	unsigned long	0
overlfow	unsigned char	' '
clock	struct	{ counter = 0; overlfow = 0 '\0'; }
counter	unsigned long	0
overlfow	unsigned char	' '

Figure 5.1: Example of Before Evaluation.

Variables × Breakpoints

Name	Type	Value
'CPU_Perf_Counters.c'::g_perfCounts	struct	{ instruction = ; clock = ; counter1 = ; cou
instruction	struct	{ counter = 1342; overlfow = 0 '\0'; }
counter	unsigned long	1342
overlfow	unsigned char	' '
clock	struct	{ counter = 1342; overlfow = 0 '\0'; }
counter	unsigned long	1405
overlfow	unsigned char	' '

Figure 5.2: Example of After Evaluation.

5.2 Optimized Pipeline

The thesis conducted experiments on the pipeline stall optimization method. The anchoring groups and the experimental group were set up. The first anchoring group is code automatically generated by Simulink Coder, and the second anchoring group is code manually written using Mercury's out-of-order optimization algorithm based on the code of the first anchoring group. The third experimental group is the code written manually using the thesis method based on the code of the first anchoring group.

This section will implement the algorithm in Section 4.1.2.3 by the model with different characteristics as Table 5.7 and some industry examples in Figures 5.5 and 5.6. All the code was executed 100 times to avoid randomness, and the average execution cycles were obtained.

Table 5.7: Test Model with Different Characteristics .

Characteristic	Can't be Reorder	Reorder
In L1-Memory	Figure 5.3	Figure 5.4
L1-Memory Overflow	Figure 5.3	Figure 5.4

Figure 5.3 is an all-serial Simulink model with eight layers. It has twelve inputs and one output. The model can be divided into eight layers, and each layer needs the results of the previous layer at the beginning of the operation. So when it run on the pipeline, no individual instruction are available can be used for OoO optimization to break the data dependency.

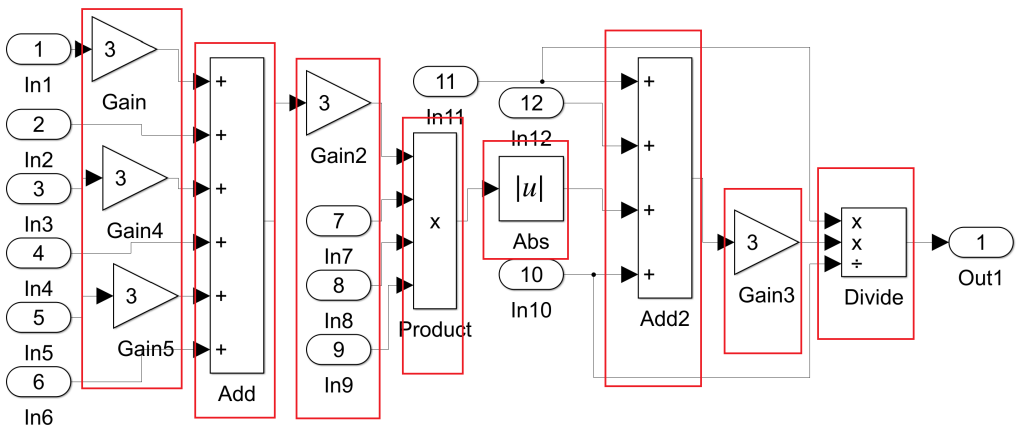

Figure 5.3: Example of Totally Serial Model.

The experiment result is shown in Table 5.8. When the L1 memory is sufficient for intermediate variables, the CPU clock cycles of Simulink, Mercury, and thesis method are the same. This is because calculating each layer needs data from predecessor blocks. When turning off the L1 memory to simulate the capacity of L1 memory is not enough, the CPU clock cycles of the three methods are still the same. However,

it is a performance loss compared to having enough L1 memory. This verifies the theoretical analysis that accessing data from L2 memory is slower than accessing L1 memory, as shown in Section 4.1.2.

Table 5.8: Test Model with Different L1 memory size.

	Simulink	Mercury	Thesis
In L1-Memory	304 cycles	304 cycles	304 cycles
Over L1-Memory	404 cycles	404 cycles	404 cycles

Figure 5.4 is a sample Simulink model with 11 layers.

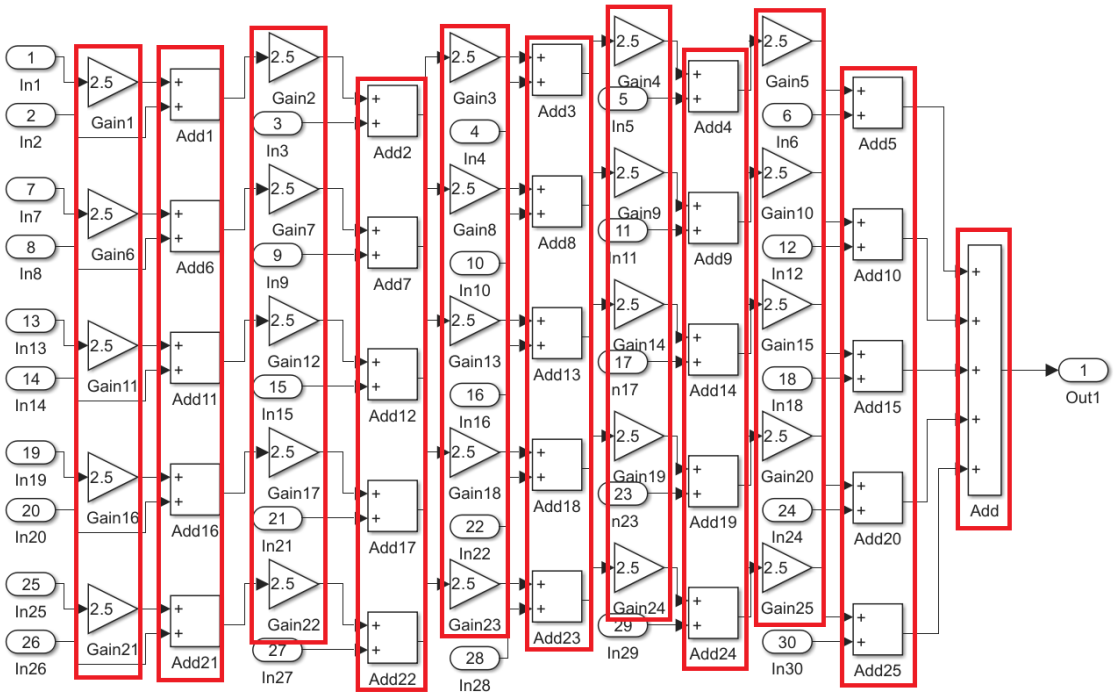

Figure 5.4: Sample Model for Evaluation.

The experiment result is shown in Table 5.9. It can be seen that when the L1 memory is big enough for middle-layer variables, the Mercury and thesis methods have the same performance improvement as Simulink. This is because all the variables are in the L1 memory. There will be no extra overhead for accessing the L2 memory.

Mercury has performance loss compared to Simulink when limiting the L1-memory size, which is insufficient to store all the middle-layer variables. On the contrary, the thesis algorithm still has performance improvement compared to Simulink and uses much fewer cycles than Mercury. Compared to the case of enough L1 memory, the rate of performance improvement is reduced. The reason for this phenomenon is that the cycles of pipeline stalls that can be avoided are the same, but the overall task time is longer because of more access time from L2 memory.

Table 5.9: Test Model with Different Characteristics.

	Simulink	Mercury	Thesis	vs Simulink	vs Mercury
In L1-Memory	737 cycles	649 cycles	649 cycles	11.2%	0.0%
Over L1-Memory	879 cycles	1055 cycles	802 cycles	8.75%	23.9%

Figures 5.5 and 5.6 are two kinds of proportional integral derivative (PID) controller models widely used in industrial control systems' controller models. The mechanism of these two controllers is the same. They need to set three control parameters: proportional, integral, and derivative coefficient. Here, the three parameters are set to 50, 12, and 13, respectively. The controller has two inputs. The first input is the set temperature of the controlled object (battery), and the second is the real-time temperature of the battery detected by the sensor. The output of the controller is the voltage of the conditioner. When the difference between the real-time and set temperatures is too large, the output voltage will increase with this difference.

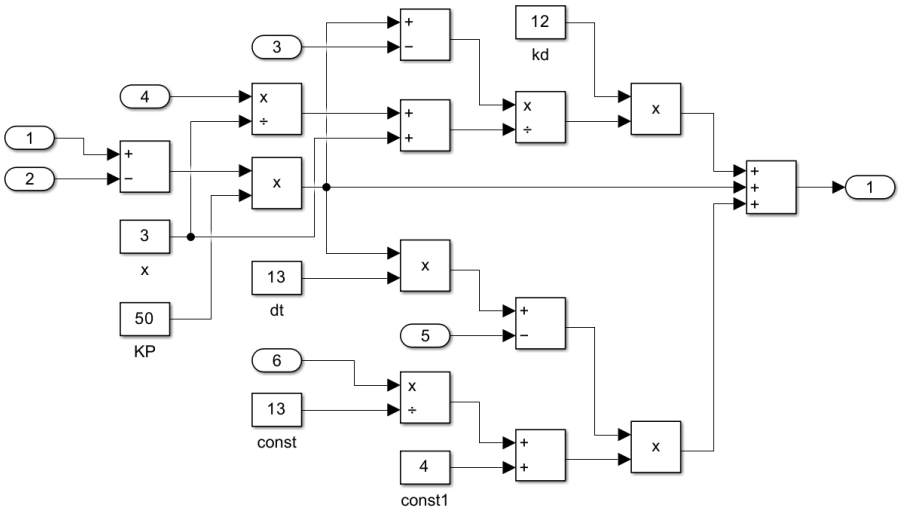

Figure 5.5: Industry Example of PID1 controller.

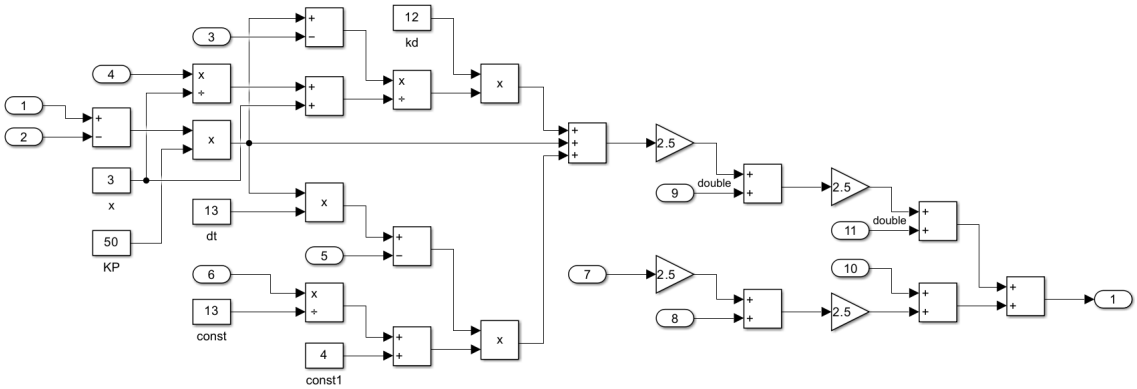

Figure 5.6: Industry Example of PID2 controller.

The evaluation result is shown in Table 5.10. Compared to the PID2 model in Figure 5.6, the PID1 has less data dependency relationship. The improvement of PID1 is not as obvious as PID2. When the L1 memory is limited, the improvement of the thesis method is better than Mercury.

Table 5.10: Result of Two Types PID Model.

	Simulink	Mercury	Thesis	vs Simulink	vs Mercury
PID1 In L1-Memory	356 cycles	336 cycles	336 cycles	5.6%	0.0%
PID1 Over L1-Memory	453 cycles	525 cycles	422 cycles	6.8%	19.6%
PID2 In L1-Memory	528 cycles	483 cycles	483 cycles	8.5%	0.0%
PID2 Over L1-Memory	702 cycles	884 cycles	677 cycles	3.6%	23.4%

5.3 Parallelization at Task Level

Figure 5.7 is a sample model set to test three assignment approaches. This model is used to control an integrated thermal management model. The output of these models includes the fan radiator and battery coolant pump voltage, as well as the compressor switch and cabin fan switch. The input consists of cabin temperature, battery temperature, and motor temperature measured from the sensor. Different tasks are responsible for controlling different parts of the model. Different tasks have different execution times. The execution time of each task is measured using the PiL test. In order to facilitate task scheduling, the minimum time required for the task is set as a time unit in the thesis. For instance, task 10 requires around 200 clock cycles, its time unit is considered as 1. Task 1 has a time unit of 5, its actual time is about 1000 clock cycles.

Three groups are also set to optimize multi-task scheduling methods for multi-core platforms. In the first anchoring group, multiple tasks are run on a single core. In the second anchoring group, multiple tasks are scheduled to run on three cores using the coarse-grained schedule method. In the third experimental group, multiple tasks are scheduled to run on three cores using the thesis method.

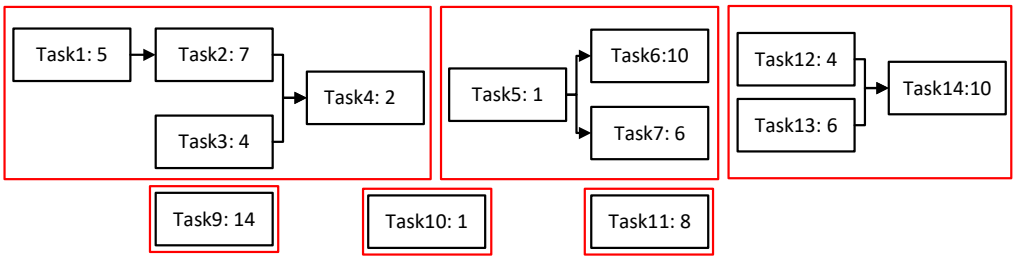

Figure 5.7: Sampel Model.

Figures 5.8 and 5.9 are the results of assigning tasks to three cores using the system-level coarse granularity approach and the thesis's fine-grained approach, respectively. The core with the longest time is the time the entire task ends. According to the

Gantt chart, it can be calculated that theoretically, the thesis method can increase the speed of the entire task by 14.7%.

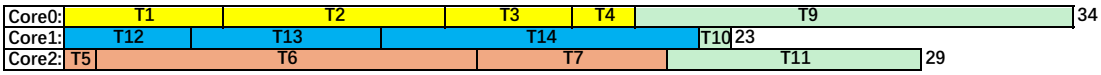

Figure 5.8: Gantt Chart Analysis for System Level Assignment Method.

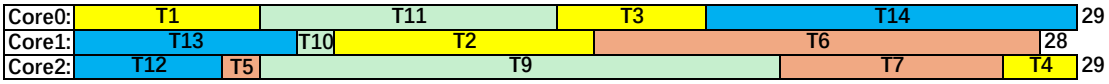

Figure 5.9: Gantt Chart Analysis for Thesis Assignment Method.

Then, using the assignment results, the development board tests these tasks. Every task is programmed on one public c file. In this c file, each task is seen as a single function that needs to be executed. The main function of each core will call this function in the sequence, as shown in Figure 5.8 and Figure 5.9. The result of the different assignment methods is shown in Figure 5.10. It can be seen that the core with the longest time in the thesis method is smaller than that in the coarse-grained method.

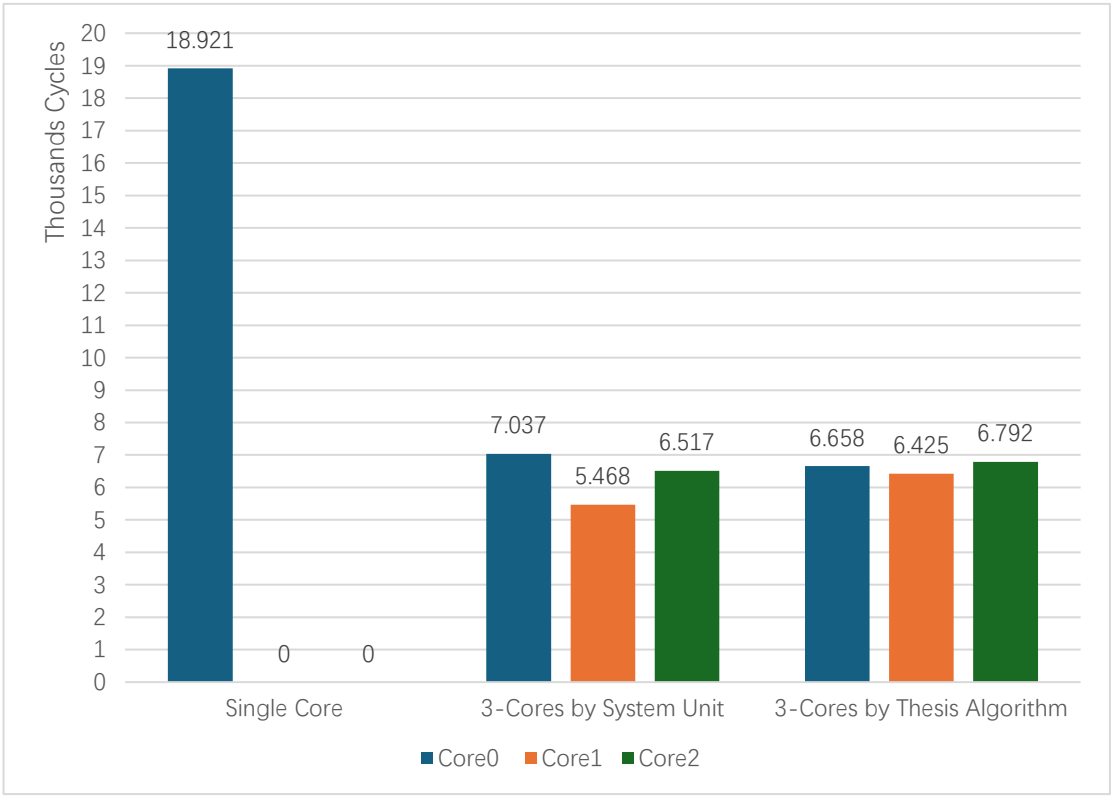

Figure 5.10: Result of Different Assignment Approach for each core.

Table 5.11 shows the running time of single-core, multi-core coarse-grained assignment methods and thesis fine-grained methods. It can be seen that the thesis algorithm has improved compared to the coarse grain method.

Table 5.11: Result of Different Assignment Approach.

	Single	3-Cores System	3-Cores Thesis	vs Single	vs System
Total	18921	19022	19875	-5.0%	-4.5%
Final	18921	7037	6792	62.8%	3.6%

By analyzing the Gantt chart and bar graph of these two methods, it can be found that the total time of the thesis assignment method is longer than that of the system assignment method. More time is needed for a core to read the output of the L1-level subsystem from another core. For the same reason, the improvement of the experimental final time is lower than the anticipated theoretical value by the Gantt chart, which is 14.7%. If the parallelism rate needs to be improved, the task granularity must be finer. However, this will increase the cost of communication. Thus, obtaining a trade-off in traffic time and multi-core utilization is necessary.

6

Conclusion

Model-based design (MBD) tool has been widely used in the automotive industry. It has the advantages of convenience and speed, but there is optimization space of the generated code. The thesis project analyzed code generated by automated code tools and revealed that the code has data dependencies relationship that cause the instruction pipeline stall.

The thesis studies academic research Mercury. Mercury proposes a methodology to solve the overhead of pipeline stall by breaking the data dependencies relationship of two adjacent instructions. In the exploring process of Mercury, one shortcoming is found. Mercury is easy to exhaust cache space when processing complex models with large amounts of intermediate data. However, when the cache space required is larger than the cache of processor, it negatively affects the code generated by the original Simulink due to the access time of flash. According to this cache miss problem, the thesis project proposes an optimized method. It calculates the data layer that takes up the most space, and if the fast cache is exceeded, it splits the entire model into parts based on the size of the fast cache that the processor has. Make the maximum cache used by each section more minor than the processor's resources. Finally, the code is reordered in each section. This improved algorithm can also achieve good results on resource-limited embedded devices. Experiments are conducted on hardware for the thesis method and Mercury. Compared to Mercury, when the cache resources are insufficient, the thesis method achieves about 20% improvements compared to Mercury for the sample model and two industry control models.

In addition, based on the characteristics of multi-core hardware devices, the thesis proposes a task assignment scheme based on greedy algorithm and prior rules. It minimizes barrier wait times and ensures the shortest completion time for cores. Compared with the coarse-grained optimization scheme, this method shows a more parallelism rate to improve performance. The experiment shows, compared to the existing coarse-grain scheduling method, the thesis fine-grain method achieves about 4% improvement.

The two methods of the thesis achieve visible optimization effects. These results demonstrate that reducing the occurrence of instruction pipeline stalls can improve CPU efficiency. Using fine-grained scheduling methods can improve the parallelism rate and reduce execution time.

Bibliography

[1] A. Shaout and S. Pattela, "Model based approach for automotive embedded systems," in *2021 22nd International Arab Conference on Information Technology (ACIT)*, 2021, pp. 1–7. DOI: 10.1109/ACIT53391.2021.9677298.

[2] B. University of California. "Ptolemy ii home page." (http://ptolemy.berkeley.edu/ptolemyII), [Online]. Available: http://website-url.com.

[3] MathWorks. "Homepage of simulink - simulation and model-based design." (), [Online]. Available: https://se.mathworks.com/products/simulink.html.

[4] Ansys. "Home page of ansys scade suite." (), [Online]. Available: https://www.ansys.com/products/embedded-software/ansys-scade-suite.

[5] dSpace. "Home page of targetlink." (), [Online]. Available: https://www.btc-embedded.com/test_environments/dspace-targetlink.

[6] Vector. "Home page of davinci configurator." (), [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/davinci-configurator-classic.

[7] T. Miyazaki and E. Lee, "Code generation by using integer-controlled dataflow graph," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1997, 703–706 vol.1. DOI: 10.1109/ICASSP.1997.599865.

[8] Z. Yu, Z. Su, Y. Yang, *et al.*, "Mercury: Instruction pipeline aware code generation for simulink models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4504–4515, 2022. DOI: 10.1109/TCAD.2022.3199967.

[9] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. USA: Cambridge University Press, 2012, ISBN: 0521886759.

[10] Infineon. "Tc29x user manual." (2015), [Online]. Available: https://www.infineon.com/dgdl/Infineon-TC29x_B-step-UM-v01_03-EN.pdf?fileId=5546d46269bda8df0169ca1bdee424a2.

[11] MathWorks. "Autosar support in matlab and simulink." (), [Online]. Available: https://se.mathworks.com/solutions/automotive/standards/autosar.html.

[12] H. E.-S. GmbH. "C compiler for tricore, powerpc and arm." (), [Online]. Available: https://hightec-rt.com/en/products/development-platform.

[13] P. Lokur, K. Nicklasson, L. Verde, M. Larsson, and N. Murgovski, "Modeling of the thermal energy management system for battery electric vehicles," in *2022 IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2022, pp. 1–7. DOI: 10.1109/VPPC55846.2022.10003328.

[14] A. Canedo, T. Yoshizawa, and H. Komatsu, "Skewed pipelining for parallel simulink simulations," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, Dresden, Germany: European Design and Automation Association, 2010, pp. 891–896, ISBN: 9783981080162.

[15] Z. Zhong and M. Edahiro, "Model-based parallelizer for embedded control systems on single-isa heterogeneous multicore processors," in *2018 International SoC Design Conference (ISOCC)*, 2018, pp. 117–118. DOI: 10.1109/ISOCC.2018.8649919.

[16] P. Xu, M. Edahiro, and K. Masaki, "Code generation from simulink models with task and data parallelism," *INTERNATIONAL JOURNAL OF COMPUTERS amp;amp; TECHNOLOGY*, vol. 21, pp. 1–13, Apr. 2021. DOI: 10.24297/ijct.v21i.9004. [Online]. Available: https://rajpub.com/index.php/ijct/article/view/9004.

[17] M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim, "Deriving high-performance real-time multicore systems based on simulink applications," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, 2011, pp. 267–274. DOI: 10.1109/DASC.2011.64.

[18] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA: Pearson, 2014, ISBN: 978-0-13-359162-0.

[19] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th. Wiley Publishing, 2012, ISBN: 1118063333.

[20] L. Brisolara, S.-i. Han, X. Guerin, *et al.*, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous mpsoc," in *Proceedingsof the 10th International Workshop on Software & Compilers for Embedded Systems*, ser. SCOPES '07, Nice, France: Association for Computing Machinery, 2007, pp. 81–89, ISBN: 9781450378345. DOI: 10.1145/1269843.1269855. [Online]. Available: https://doi.org/10.1145/1269843.1269855.

[21] K. Huang, M. Yu, R. Yan, *et al.*, "Communication optimizations for multi-threaded code generation from simulink models," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, May 2015, ISSN: 1539-9087. DOI: 10.1145/2644811. [Online]. Available: https://doi.org/10.1145/2644811.

[22] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[23] Intel. "Pentium-pro 1995." (), [Online]. Available: https://timeline.intel.com/1995/pentium-pro.

[24] Wiki. "Pentium-pro introduction." (), [Online]. Available: https://en.wikipedia.org/wiki/Pentium_Pro.

[25] L. Community. "Llvm document." (), [Online]. Available: https://llvm.org/docs/.

[26] F. S. Foundation. "Gcc, the gnu compiler collection." (), [Online]. Available: https://gcc.gnu.org/.

[27] Linux. "Objdump(1) — linux manual page." (), [Online]. Available: https://man7.org/linux/man-pages/man1/objdump.1.html.

[28] F. S. Foundation. "3.11 options that control optimization." (), [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[29] Infineon. "Memory access time in tc1m based systems." (2015), [Online]. Available: https://www.infineon.com/dgdl/ap3206511_TC1M_MemoryAccessTime.pdf?fileId=db3a304412b407950112b40f72d413eb.

[30] dSpace. "Real-time interface for multiprocessor systems." (2015), [Online]. Available: https://www.dspace.com/en/ltd/home/products/sw/impsw/rtimpblo.cfm#179_25062.

[31] R. Ruiz and J. A. Vázquez-Rodríguez, "The hybrid flow shop scheduling problem," *European Journal of Operational Research*, vol. 205, no. 1, pp. 1–18, 2010, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2009.09.024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221709006390.

[32] W. Han, F. Guo, and X. Su, "A reinforcement learning method for a hybrid flow-shop scheduling problem," *Algorithms*, vol. 12, no. 11, 2019, ISSN: 1999-4893. DOI: 10.3390/a12110222. [Online]. Available: https://www.mdpi.com/1999-4893/12/11/222.

[33] L. Xu, J. Yeming, and H. Ming, "Solving hybrid flow-shop scheduling based on improved multi-objective artificial bee colony algorithm," in *2016 2nd International Conference on Cloud Computing and Internet of Things (CCIOT)*, 2016, pp. 43–47. DOI: 10.1109/CCIOT.2016.7868300.

[34] J. V. Moccellin, M. S. Nagano, and A. R. e. a. Pitombeira Neto, "Heuristic algorithms for scheduling hybrid flow shops with machine blocking and setup times," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 40, p. 40, 2018. DOI: 10.1007/s40430-018-0980-4.

[35] S. A. BRAH, "A comparative analysis of due date based job sequencing rules in a flow shop with multiple processors," *Production Planning & Control*, vol. 7, no. 4, pp. 362–373, 2007. DOI: 10.1080/09537289608930364.

[36] Infineon. "Aurix development studio." (), [Online]. Available: https://www.infineon.com/cms/en/product/promopages/aurix-development-studio/.

[37] Infineon. "External gcc for tricore™ applications for aurix development studio." (), [Online]. Available: https://docs.hightec-rt.com/ads-with-hightec-toolchains/1.5/chapter/external-gcc.html.

[38] Matlab. "Software-in-the-loop simulation." (), [Online]. Available: https://in.mathworks.com/help/ecoder/ug/configuring-a-sil-or-pil-simulation.html.

[39] Infineon. "Code examples for aurix development studio." (2021), [Online]. Available: https://github.com/Infineon/AURIX_code_examples.