CHALMERS

A Driving Assistance System with Hardware Acceleration

Master of Science Thesis in Computer Systems and Networks

GONGPEI CUI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, January 2015

A Driving Assistance System with Hardware Acceleration

Gongpei Cui

Examiner: Ioannis Sourdis
Supervisor: Vasileios Papaefstathiou
Anders Svensson (Volvo AB)

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden January 2015

Abstract

Nowadays, active safety has become a hot research topic in vehicle industry. Active safety systems play an increasingly important role in warning drivers about and avoiding a collision or mitigating the consequences of the accident. The increased computational complexity requirement imposes a great challenge for the development of advanced active safety applications using the traditional Electronic Control Units (ECUs). One way to tackle this challenge is to use hardware offloading, which has the capability of exploiting massive parallelism and accelerating such applications. A hardware accelerator combined with software running on a general purpose processor can compose a hardware/software hybrid system.

Model Based Development (MBD) is a common development scheme that reduces development time and time-to-market. In this project, we evaluate different MBD workflows for the hardware/software co-design and propose a general workflow for MATLAB/Simulink models. We investigate key techniques for hybrid system design and identify three factors to assist hardware/software partitioning. Moreover, several essential techniques for hardware logic implementation, such as pipelining, loop unrolling, and stream transmission, are analyzed based on system throughput and hardware resources.

This project describes the workflow for hardware/software co-design based on MBD and finds methods to improve the system throughput combining hardware accelerators and software. Using the proposed profiling methods and partitioning roles, a matrix multiplication function is selected to be implemented by a hardware accelerator. Having optimized the hardware implementation scheme of the accelerator, a 5.4x speedup is achieved on a Zynq evaluation board.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my examiner and supervisor Prof. Yiannis Sourdis and Dr. Vassilis Papaefstathiou. Their guidance helped me a great deal throughout the project process and the writing of this thesis. Besides, I warmly thank Dr. Anders Svensson at Volvo ATR for his kind support, guidance and advice during this project. Fruitful discussions with him helped me to find better and better solutions and his great enthusiasm to research was my source of inspiration. A special acknowledgement is given to Henrik Lönn who offered me opportunities at Volvo and gave me the feeling of home warmth. Finally, I am particularly grateful for the assistance given by Serkan Karakis, Hans Blom, Hamid Yhr, Mattias Wallander, Håkan Berglund, Daniel Karlsson and all the colleagues at Volvo GTT. I own my deepest gratitude to my parents, who support me to study abroad and live in China alone. Last but not the least, my sincere thanks goes to my wife, Jingya. Thank you for your positive mind, prays and all your efforts. I wish you all the best.

Gongpei Cui, Gothenburg 8/8/14

Contents

1

Introduction

1.1 Introduction

Over the past few decades, vehicles, such as cars, buses and trucks, have dramatically improved people's lives. Now, vehicles have become an essential part of everyday life. They have made it easier and faster for users to get from place to place. They bring convenience to people but at the same time can endanger people's lives. Surveys show that about 50000 Europeans were killed in car accidents in 2001 [1]. Nowadays, customers select vehicles based on two major factors: *(a)* safety and *(b)* fuel consumption; they reference (a) and (b) with 54% and 53% respectively.

The automotive industry is continuously working to improve vehicle safety. The safety systems developed in vehicles can be divided into two categories: passive safety systems and active safety systems. Passive safety is a technology to decrease the damage to the driver and passengers in an accident. For example, seat-belts can hold passengers in place so that they will not be thrown forward or ejected from the car; airbags can provide a cushion to protect the drive and passengers during a crash. These passive safety systems have saved thousands of lives and are milestones in the automotive industry. Active safety refers to systems that help keep a car under control and use an understanding of the state of the vehicle to predict and avoid accidents. For example, anti-lock brakes can prevent the wheels from locking up when the driver brakes, enabling the driver to steer while braking. Advanced Driving Assistance System (ADAS) can alert the driver to potential problems, or to avoid collisions by implementing safeguards and taking over the control of the vehicle.

In current vehicles, increasingly more mechanical components are replaced by Engine Control Units (ECUs), sensors, and actuators. More sophisticated active safety systems, such as Anti-lock Brake Systems (ABS) and Electronic Stability Control (ESC), are deployed in the vehicles. According to the eSafety effects database, ABS has prevented 10-33% of single-vehicle accidents and ESC has reduced all kinds of crashes by 19.3% [2].

It is expected that active safety systems will play an increasing role in collision avoidance and mitigation in the future. However, as more sophisticated systems are deployed, the deployment of active safety systems becomes more challenging.

As mentioned before, ADAS is an example of active safety applications. Figure 1.1 shows three typical applications of ADAS. Figure 1.1(a) presents an example of predictive pedestrian protection. Based on video streams from cameras installed on vehicles, a pedestrian detection algorithm marks out the location of pedestrians and their distance from the vehicle. If the distance is shorter than a threshold, then ADAS stops the vehicle automatically. Figure 1.1(b) depicts a diagram of lane assist systems. By analyzing the lane marks, algorithms of ADAS detect whether vehicles are crossing lane marks in reasonable situations; this function detects whether the driver is concentrated on driving. Figure 1.1(c) shows an example of radar system which measures the distance between its front vehicle and itself. When the distance becomes shorter than a safe distance threshold, ADAS triggers an emergency braking system to stop the vehicle. Besides the above three applications, more new applications of ADAS have been developed recently, such as surrounding view, automatic parking system, etc.

(a) Predictive Pedestrian Protection (b) Lane Assist Systems (c) Emergency Braking System

Figure 1.1: Applications of Advanced Driving Assistance System (a), (b) and (c) [3]

Data fusion is an essential module for ADAS. If each application supported by ADAS requires its private sensor(s), then adding new applications will require more sensors, which is inefficient. Figure 1.2 illustrates an example of a vehicle with several ADAS applications, where multiple sensors are needed for supporting different services. There may be significant overlapping between different sensors. Moreover, different sensors may have different observation capabilities and various detection properties. The goal of a data fusion application is to reduce duplicated sensors and organize different sensors so as to get higher observation accuracy and efficiency. This thesis focuses on the implementation problems of the data fusion application.

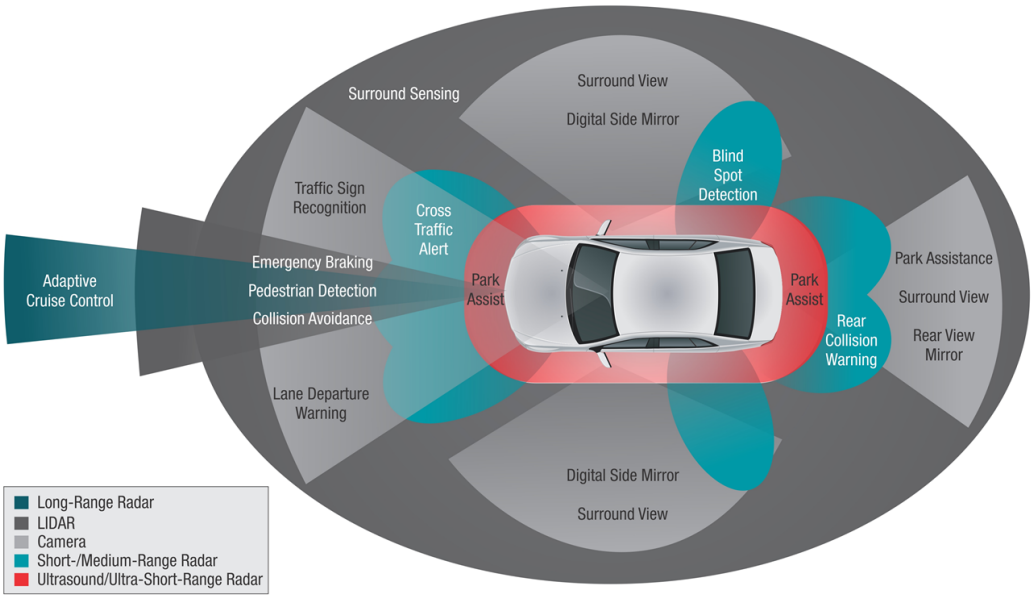


Figure 1.2: Sensors and Applications on Vehicles [4]

1.2 Problem Statement

The ever increasing number of new applications in ADAS imposes a great challenge for supporting high computational complexity. In the past, as shown in Figure 1.3, only some simple and independent applications ran on ECUs, such as ABS and anti-lock braking system. These applications have a limited number of sensor inputs and use very simple algorithms, which ECUs could support. However, once ADAS has been introduced, a substantial amount of video and image streams is feed to such systems. Moreover, the algorithms used in such applications are significantly more complex than in the past. In particular, for the data fusion application considered in this thesis, execution has to be completed within a tight deadline so that the actors can handle the risk as early as possible.

Instead of using a single processor, Systems-on-Chip (SoCs), which contains both a fixed General Purpose Processor (GPP) and a Programmable Logic (PL) section, can be used to handle the high computational complexity problem. As in shown Figure 1.4, an SoC consists of both hardware (i.e., PL) and software (i.e., GPP) parts. Hardware accelerators can also be implemented in Field-Programmable Gate Array (FPGA) with PL. Systems which execute some parts of an application in software and some other parts in hardware accelerators are called hybrid hardware/software systems. Integrating software and hardware, the performance of the entire system can be improved significantly.

The design of hardware/software hybrid system that uses SoCs technology also faces some challenges: partitioning and development workflow. The system needs to be partitioned into software and hardware parts properly. Software and hardware systems

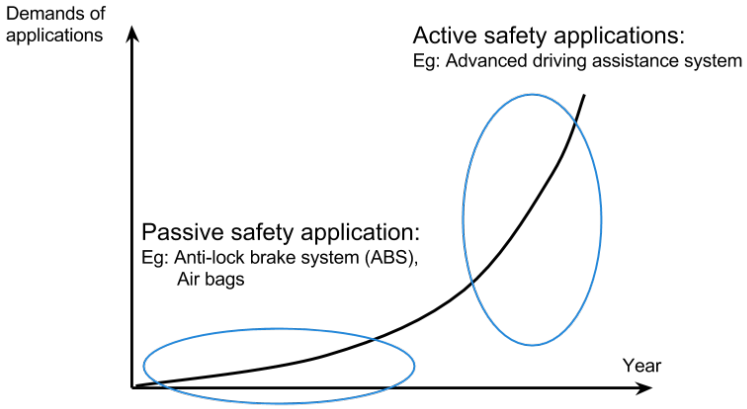


Figure 1.3: Demands of Applications

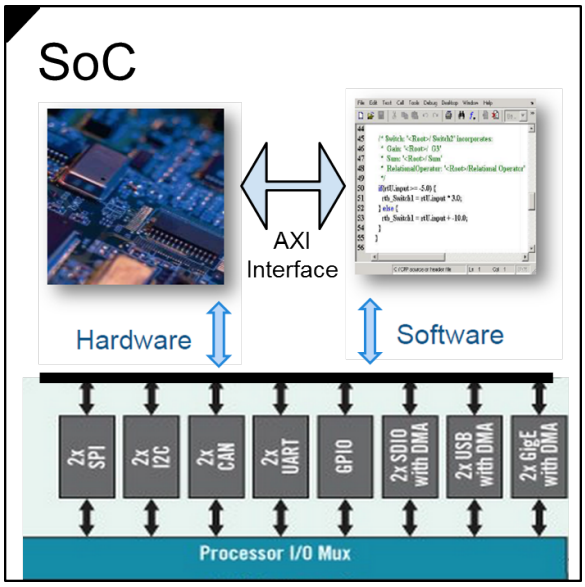


Figure 1.4: Overview of System-on-Chip

have their own advantages and disadvantages. For instance, some algorithms are more suitable for pure software environments while other algorithms are more efficient when executed by specialized hardware logic. An improper implementation of an algorithm can result in low throughput and unnecessarily high resource utilization. The communication between processors and hardware accelerators is also a key factor that limits the performance of hybrid systems. The total execution time of a hybrid system consists of two parts: the overhead caused by the communication between the processor and the accelerator, and hardware execution latency. Therefore, in order to reduce the total execution time, both the communication scheme and the hardware implementation need to be optimized. Note that for hardware implementations, a parallel design can achieve

high system throughput, while a serial structure might use resources more efficiently. For developers it is time-consuming to find the proper tradeoff between the performance and the resource consumption for hardware design.

Round-Trip Engineering (RTE) is a traditional development flow for the design of hybrid systems [5]. As shown in Figure 1.5, three development roles are involved in the cycles. First, system designers define system models in specialized tools, such as MATLAB/Simulink and LabVIEW. Then, software and hardware designers implement their components in other design tools, for example, Eclipse SDK, Quartus, and Vivado. The separate software and hardware designs are then combined to form hybrid systems. Finally, the HW/SW co-design is verified based on the results of the models. In conclusion, RTE takes a significant amount of time and is not flexible if any change occurs in any of the development phases.

Model Based Development (MBD) is a method where models are used in all phases of development. Model designers define systems in modeling tools based on requirement analyses. Based on the models, prototyping, visualization and testing are conducted. In the end, executable code is generated from these models automatically. The advantage of MBD is that the whole development cycle is centrally modeled so that it reduces the development time. Ideally, only one development role, model designer, is necessary and all the other tasks are done by modeling tools. There are several options of the modeling tools and code generators and this thesis tries to answer the following two questions: (a) Which workflow works best for general models? (b) Does the MBD fulfill the expectations for automatic flows?

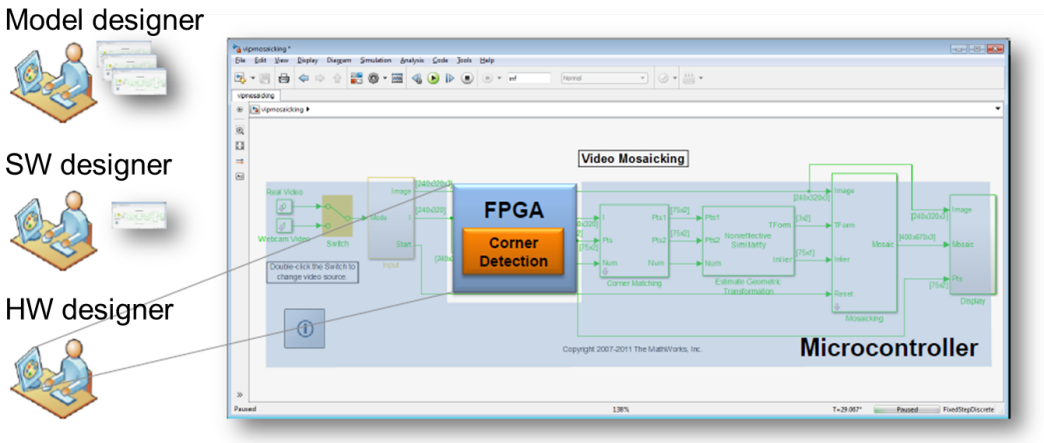


Figure 1.5: Three Development Roles for Traditional Development [6]

1.3 Thesis Objectives

In this thesis, a driving assistance system will be implemented based on SoC technology. Moreover, different MBD workflows will be investigated.

The first objective is to find a suitable workflow for MBD. A good workflow should shorten the develop cycle as much as possible. It should also support the most common source models including different language syntax and function blocks. The work of the project starts from the second phase of MBD; thus, the inputs of the project are MATLAB code and Simulink models. We use two MathWorks' tools, namely the Embedded Coder and the HDL Coder, as candidates for generating software and hardware code respectively. Xilinx's High-Level Synthesis is another alternative, which can generate hardware code based on C code. Therefore, the first goal is to evaluate different tools and suggest the best workflow for hardware/software co-design of the given application.

The second objective is to investigate methods for efficient hardware/software co-design, by which the hybrid systems can achieve higher performance with low cost. Specifically, high performance translates to high processing throughput, so the design will meet the timing constraints. The cost represents the silicon area on the FPGA and the utilization of processors. In order to achieve this goal, three steps are taken, profiling, partitioning, and optimization of hardware design. Criteria for partitioning will be summarized to assist developers to select the blocks that should be implemented in hardware. During the hardware code generation, different optimization settings, which will affect the performance of the generated hardware blocks, should also be considered.

1.4 Overview of the Report

The rest of this thesis is organized as follows. Chapter 2 introduces the background of the project and related work. Chapter 3 describes different methodologies for MBD and analyzes their advantages and limitations. Chapter 4 proposes methods to analyze and implement applications, such as profiling, partitioning, and RTL optimization. Finally, Chapter 5 presents the evaluation results for the test cases and a real ADAS application.

2

Background

This chapter provides the background for this thesis and discusses related work. First, we present the background for ADAS applications and describe the basic processes and algorithms. Next, we introduce the concept of Hardware/Software co-design and MBD. The state-of-the-art of both areas are introduced. Finally, we present the evaluation platform used in this work.

2.1 Application

The main goal of this project is to implement a data fusion application on SoC using MBD. Although the application model has been designed in other previous works, it is still necessary to understand the basic processes so that we can explain the results of application profiling and the complexity of the application.

Fusion is widely used in nature, where human beings combine the senses of vision, touch and auditory to help themselves to recognize the world [7]. With the development of sensor technology, signal processing and advanced hardware, data fusion is becoming increasingly popular in industry [8, 9]. In generally, multisensor data fusion is to associate information from different sensors and provides more accurate observation of objects than what a single sensor does. Consequently, the better representation is provided, the more precise decisions are made [10, 11].

Specifically, in vehicle industry, sensor data fusion (sDF) provides a way to use different sensors in active safety applications of ADAS as shown in Figure 2.1. Various sensors are used because different sensors have distinct properties, for instance, radars are sensitive to longitudinal distance but they have lower accuracy in lateral measurements [12]. Information from cameras is useful to detect the direction of objects, while it is difficult to measure how far the objects are. In addition, the rate of traffic accidents is 17 times higher in bad weather conditions than in good conditions. Moreover, laser and supersonic-wave radar are not robust in foggy weather compared to mm-W radar [13].

However, the combined information from mm-W radars and cameras can improve the object recognition rate by 84% as compared to human judgments.

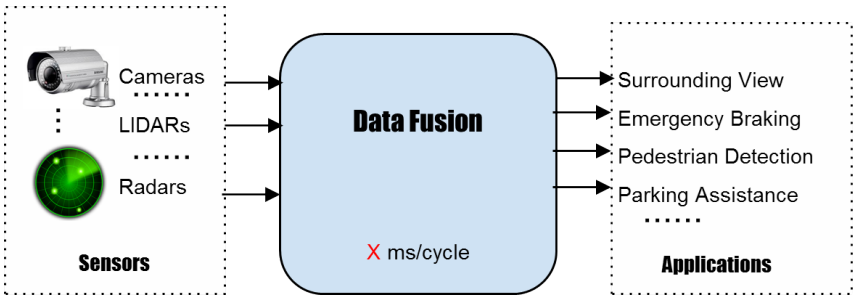


Figure 2.1: Overview of Data Fusion

Sandblom and Sorstedt present the general process of sDF [12], which can be summarized in Figure 2.2. Observational data from different sensors is represented in various models, such as the constant acceleration (CA) model and the coordinate turn constant acceleration (CT) model [14]. The first step is to parameterize them into a unique model and then alignment follows to shift time delay from different sensors. The main part of the procedure is state fusion that consists of four steps. First, data from different sensors is associated by using Global Nearest Neighbor method, where data representing the same objects is clustered [15]. Second, based on previous results a local trace management algorithm is executed, e.g. Multiple Hypothesis Tracking [16]. The next two steps are priority selection and state updating. Finally, according to the updated state vector the objects' movement is estimated and the final results are delivered to the ADAS applications.

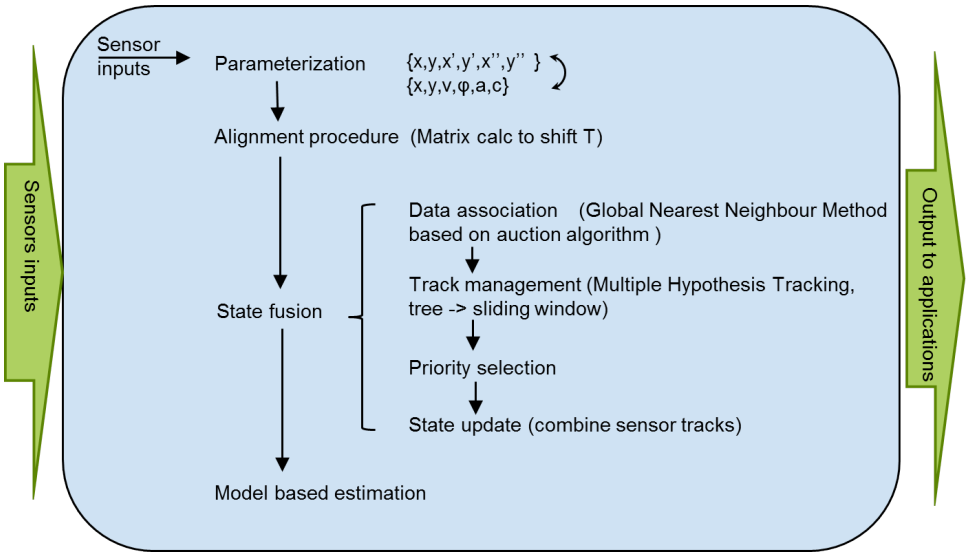


Figure 2.2: Process of Data Fusion Algorithm

2.2 Hardware/Software Co-design

Hardware/software co-design refers to the simultaneous design of both software and hardware in a system. Software is usually executed in processing elements, such as, Central Processing Units (CPU) and Digital Signal Processors (DSP). Hardware logic is commonly implemented in Application-Specific Integration Circuit (ASIC) or FPGA. Since software and hardware systems have different characteristics, a combined system has the potential to take the best of both worlds. Teich analyzes the major properties of the hardware/software co-design: co-ordination, concurrency, correctness, and complexity [17].

Hardware/software co-design has largely evolved since the first time it was used in electric systems. The earliest use case of the co-design emerged in the 1980s [18]. In the first generation of co-design, partitioning was the main issue to be solved. Two approaches were proposed: (a) start from pure software and then migrate software functions to hardware [19], and (b) start with a hardware-only system and end up with a co-designed system [20]. In the second generation, multicore and multiprocessor have been utilized and thus, instead of a single thread, multi-threads were used. Thread scheduling become one of the main challenges for the area. Moreover, hardware/software interface and communication are critical since they dramatically affect system performance and design space [21]. According to [17], the co-design is now in its third generation, where the time-to-market cycles are shortened by optimizing the hardware/software development flow. Moreover, languages for hardware/software co-design are necessary. The ideal language would be suitable for both hardware and software development. It seems that neither Hardware Description Languages (HDLs) nor C/C++ can replace the other. On the other hand, MBD (Section 2.3) is a promising solution.

Compared with pure-software design, the major difference of hardware/software co-design are listed in [18]. 1) Allocation: a lot of options are available for the architecture design of SoC and designers need to select suitable resources for their systems among different processors, ASICs, FPGAs, DSPs and so on. 2) Binding: Having different resources in SoC(s), developers have to bind applications, tasks or variables to specific resources; partitioning, mentioned in the first generation, was one kind of binding. In [22], it has been proved that the process of binding is a NP-complete problem, thus it's time consuming to find an optimal solution. 3) Scheduling: several resources are shared – e.g. processors, memory and communication bus – thus, real-time analysis is necessary when complex co-design is being implemented.

Nowadays, additional factors are affecting the hardware/software co-design technology [17]. Heterogeneous SoCs is a new trend, where much more resources could be integrated in a single chip, e.g. one multi-billion transistor chip contains multiprocessors, DSPs, FPGA, analog intellectual property (IP), memory and peripherals. Next, the complexity of SoC(s) systems is increasing dramatically since multiple SoCs need to cooperate together. Last but not least, once systems become more and more complex, it's very common that different subsystems are implemented by different suppliers, thus, public standards shared by all companies are absolutely essential.

2.3 Model Based Development

Beyond the hardware/software co-design implementation for embedded systems, some other challenges come from aspects of the development flow [23]. First of all, it's time consuming to code manually. After system models are done by system designers, hardware developers and software developers implement HDL code and C code respectively, according to model descriptions. Note that hardware code, HDL, is a low-level language and it takes significant amount of time to implement. Moveover, some models are difficult to be represented by HDL codes.

Next, the co-design integration is a manual step. In the verification phase, test benches need to be implemented for the co-design platform. In addition, bit-accurate and cycle-accurate simulations are executed to verify whether the results from co-design are consistent with the simulation results. System designers and software/hardware developers need to cooperate closely and most of work is done manually. Therefore, this thesis explores Model-based development as a potential approach to solve the problems and challenges discussed above.

MBD is a design workflow, where all the phases of development are based on models. MBD is widely used in industry, for example, the automotive industry [24]. The typical steps of MBD for co-design are: modeling, synthesis, integration, and verification. Using MBD, behavior models are first designed according to requirement analyses. The advantage of behavioral models is that designers can concentrate on the high-level features of systems instead of the low-level implementation details of the system [25]. Consequently, system designers can analyze whether their systems satisfy the requirements even before they are really implemented, thus, shorten development cycles significantly. Next, MBD code generation for both hardware and software is an automatic process handled by the tools. Developers need only to set some parameter for the code generation tool so as to tune the implementation. During the integration and verification phases, important issues are interconnection between hardware and software sections of hybrid systems, and communication between boards and simulators. Ideally, MBD can automate these interfaces based on models.

MBD has been a hot research field for several decades and there are a lot of widely used solutions. MathWorks contains several tools for MBD [26]. HDL Coder is a tool that generates RTL code (VHDL and Verilog) from MATLAB functions and Simulink blocks. On the other hand, Embedded Coder is another tool from MathWorks that synthesizes C/C++ code, however only a subset of functions and blocks can be converted to C code and hardware logic [27]. System Generator (XSG) is another solution supplied by Xilinx [28] as a blockset for Simulink. Using predesigned blocks, developers can build their models in Simulink and synthesize them into RTL code. Another alternative is LabVIEW [29], a graphical programming tool that includes LabVIEW Real-time and LabVIEW FPGA, where developers can implement their design by predefined blocks. LabVIEW synthesizes C/C++ code and VHDL code [30] and today supports advanced features such as cloud synthesis [31]. The typical compilation process of hardware projects takes from minutes to days, but with the cloud service, remote ex-

ecution on NI's cluster servers yields very fast results. Other examples include OpenCL: a programming language supporting parallelism models for CPUs, GPUs, FPGAs and so on. OpenCL compilers can use C programs to generate RTL code for FPGAs [32]. Recently, OpenCL has been integrated into Altera's SDK, by which embedded CPUs can communicate with accelerators implemented in FPGAs [33].

2.4 Hardware Platform

In this thesis we use a Xilinx device (Zynq XC7Z020), which contains both a Processing System section (PS) and a PL [34]. Configurable Logic Blocks, Block RAM (BRAM) and DSP Blocks are the main parts of PL. Configurable Logic Blocks consist of Loop-Up Tables (LUT) and Flip-flops (FF), which are the basic logic units of FPGAs that are connected according to user design. BRAMs are memory elements used to store data and support several configurable organizations such as: single/dual port(s) RAMs/ROMs or FIFOs. DSP Blocks are specialized hardware components handling mathematical calculations. Configurable Logic can also be assigned to the same computations, however they achieve lower performance than specific DSPs. LUTs, FFs, BRAMs and DSPs are the most important hardware resources of an FPGA design and our resource utilization analysis focuses on these.

The PS section of Zynq contains a dual-core A9 processor together with a rich set of peripheral connectivity interfaces and interconnection between PL and PS sections. Figure 2.3 shows how PL, PS and peripherals are organized inside the Zynq FPGAs. Table 2.1 presents the architectural parameters of the specific Zynq device. Figure 2.4 illustrates the Zynq chip and the evaluation board used for this thesis.

Table 2.1: Zynq 7020 All Programmable SoC [34]

PS	Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
	Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor
	Maximum Frequency	667 MHz (-1)
	L1 Cache	32 KB Instruction, 32 KB Data per processor
	L2 Cache	512 KB
	On-Chip Memory	256 KB
	External Memory Support	DDR3, DDR3L, DDR2, LPDDR2
	External Static Memory Support	2x Quad-SPI, NAND, NOR
	DMA Channels	8 (4 dedicated to Programmable Logic
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
	Peripherals w/ built-in DMA	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
	Security	RSA Authentication, and AES and SHA 256b Decryption and Authentication for Secure Boot
PL	Look-Up Tables (LUTs)	53,200
	Flip-Flops	106,400
	Extensible Block RAM	560 KB (140)
	Programmable DSP Slices	220

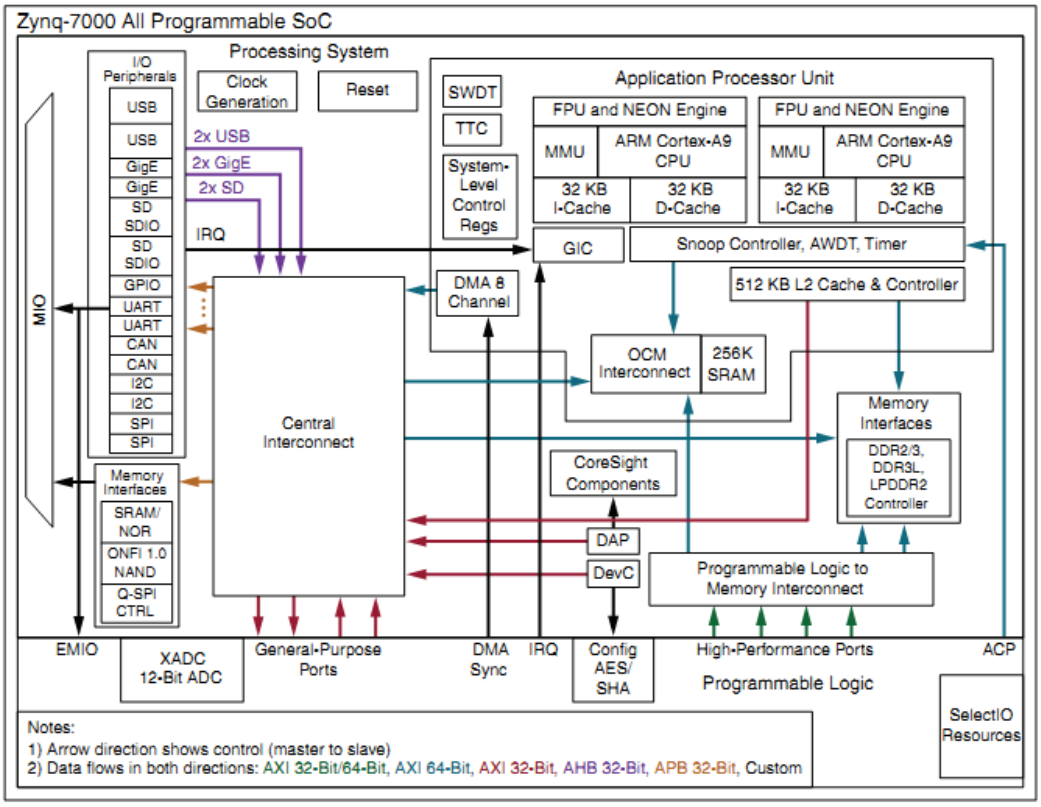


Figure 2.3: Zynq-7000 All Programmable SoC Overview [34]

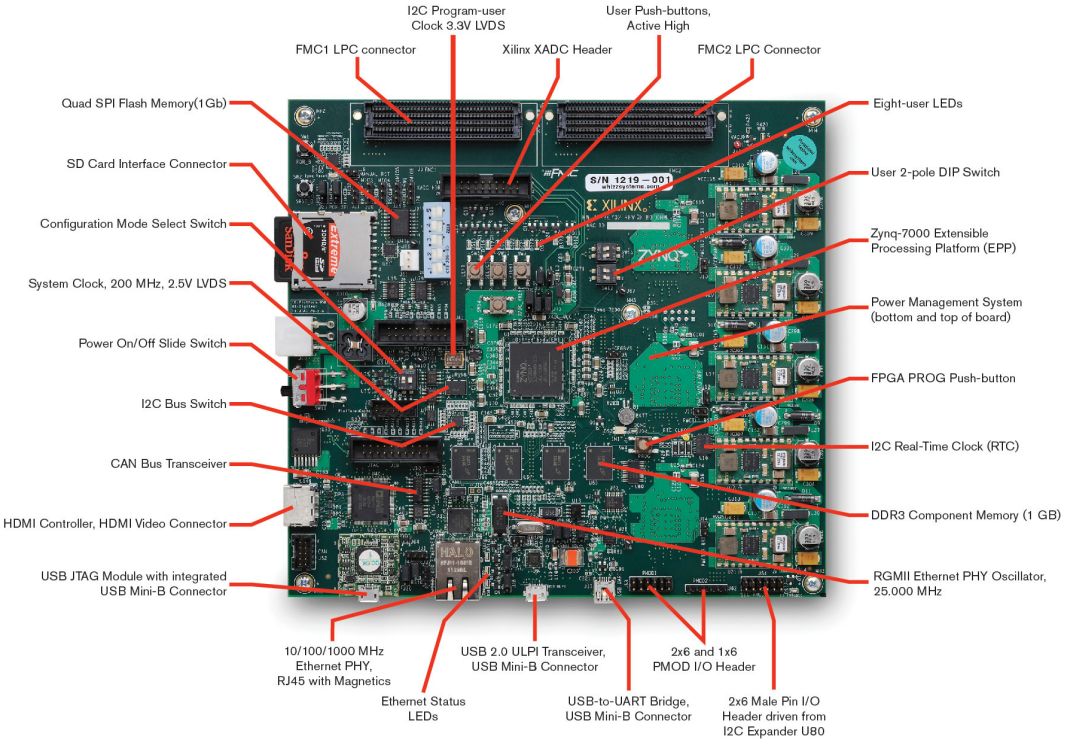


Figure 2.4: Overview Zynq Evaluation Board [34]

2.5 Conclusion

This chapter introduced the background for this thesis and the main concept of multi-sensor data fusion. We discussed the two main areas of the thesis: (i) hardware/software co-design and (ii) MBD. We presented the motivation, gave definitions, and discussed the key issues and the most common solutions. In the end, we presented details about the evaluation platform and the FPGA development board.

Although many MBD methods for hardware/software co-design exist, in this thesis we use predesigned models in MATLAB/Simulink and focus on the associated workflows. For the hardware/software co-design we focus and optimize the system throughput, hardware architecture and interconnection schemes. Other remaining key factors such as design space exploration and energy consumption will be studied in the future.

3

Methodologies for MBD

Nowadays, MATLAB/Simulink is one of the most popular modeling tools, which has been widely used in industry. In this chapter, we present the findings of our study considering different workflows. In Section 3.1, we analyze a workflow for hardware/software co-design based on Simulink models. In Section 3.2, based on MATLAB models, a similar workflow is described. Moreover, a different workflow for hardware code generation, namely High-Level Synthesis (HLS) is presented. The advantages and limitations of different workflows are also discussed in this chapter.

3.1 Simulink Models

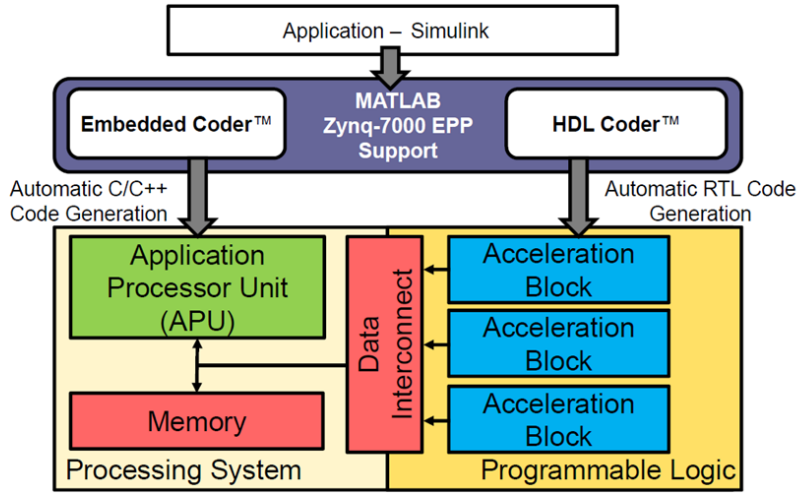


Figure 3.1: Overview of Automatic Code Generation [6]

Simulink is a block-based modeling environment. As illustrated in Figure 3.1, the main goal for MBD based on Simulink models is to automatically generate C code and Register Transfer Level (RTL) code using the Embedded Coder and HDL Coder, respectively. The generated C code will run on General Purpose Processors (GPPs) and the RTL code will run on programmable hardware. Figure 3.2 shows the Simulink workflow, which consists of two parts: the process of generating RTL code and the chain to synthesize C code.

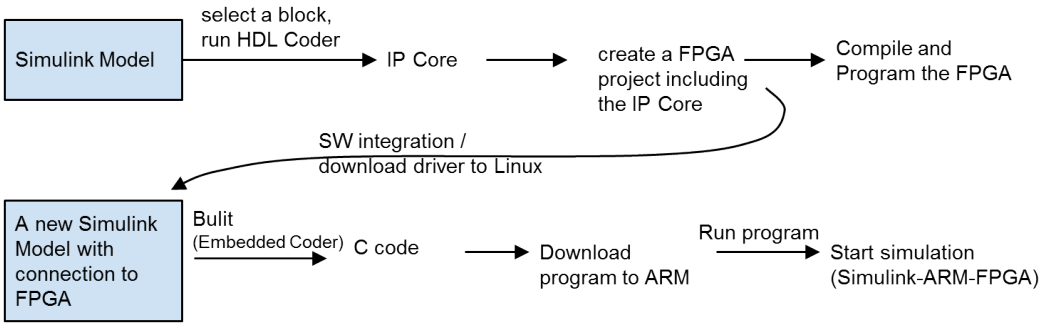


Figure 3.2: Overview of Simulink Workflow

HDL code can be generated by using 'HDL Workflow Advisor' of HDL Coder. First, the block which will be implemented by hardware logic is selected from the original Simulink model. This block is used to start the HDL Workflow Advisor. Then, in the advisor, an IP Core for the selected block is generated. In the next step, the generated IP Core together with a processor are inserted into an FPGA project. At the same time, the connection between the IP Core and the processor is established automatically. Based on the FPGA project, the advisor generates a bit file, which can be used for programming a target board. Finally, a new Simulink model is produced by replacing the selected block with an interface between the Simulink model and the generated IP Core. This new model will be used to synthesize C code in the second part. Figure 3.3 indicates the overview of the advisor.

The second part of the Simulink workflow is to synthesize C code by using Embedded Coder. In the first step, the interface between Simulink and a target processor is configured. The second step is to build the Simulink model and generate the C code. Then, the C code can be downloaded to the target processor. Finally, via Simulink, the C code can be run on the processor. We can control input valuables easily in Simulink and check simulation result in Simulink. This step is called a System-In-the-Loop (SIL) co-simulation. By this feature, developers can verify the generated RTL and C code to confirm that the behavior of the hybrid system matches the behavior of the model.

3.1.1 Discussion

Advantages:

- Convenient HDL Workflow Advisor: The advisor covers the whole process of RTL

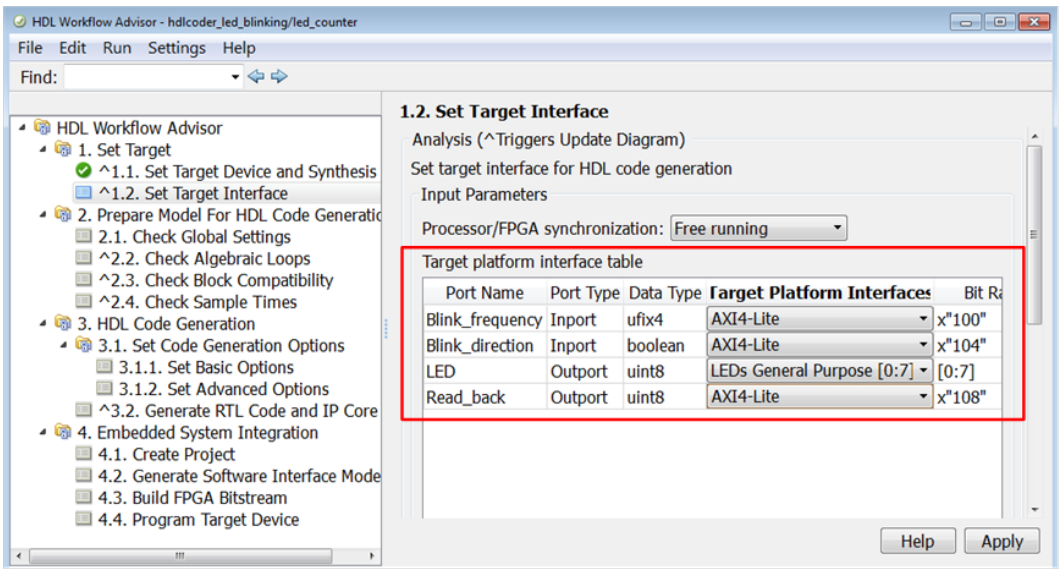


Figure 3.3: Overview of HDL Workflow Advisor

code generation. It produces IP Core, FPGA project, and replaces selected block in the original Simulink model with some specific interface. Following the advisor, the users can performs all the essential steps for hardware/software co-design.

- Broad blocks support: HDL Coder and Embedded Coder have sufficient support for Simulink blocks. Different from the MATLAB workflow, almost all the Simulink pre-defined blocks can be synthesized to RTL and C code.

- Co-simulation: Another advantage is that the SIL co-simulation can coordinate target boards and Simulink. Therefore, it becomes possible for a system model developer, without extensive C coding and RTL coding experience, to implement hardware/software hybrid systems.

Limitations:

- Single top-level block: The advisor can only support one block for hardware implementation. We tried to execute the advisor in second round, where we select the second block to be implemented in hardware, but the advisor cannot work correctly in second round. Since it is very common that several hardware blocks (hardware accelerators) exist in one design, thus, users cannot use the Simulink workflow to complete the whole hybrid system design.

- Limited compatibility: Only a limited number of Simulink blocks are supported by Embedded Coder and HDL Coder, Thus, before starting a new hybrid system design, it is better to check the compatibility and supported block libraries for these tools [35, 36].

Conclusion:

In summary, Embedded Coder and HDL Coder are good code generators for Simulink models. The generated code can be easily verified. However, if a system contains more than one acceleration blocks then the developers need to integrate the hardware/software co-design manually.

3.2 MATLAB Models

MATLAB is another modeling environment from MathWorks. Compared to the Simulink platform, MATLAB is more script-based instead of block-based. The difference between these two environments is that a MATLAB script is more sequential, while Simulink blocks can executed in parallel. Before synthesizing RTL and C code, some preparation needs to be done as illustrated in Figure 3.4. Developers need to select a top-level function that is the model to be synthesized into RTL or C code. The top-level function can contain sub-functions. Moveover, a testbench is necessary, where the top-level function is called.

The workflows to generate RTL and C code in MATLAB are shown in Figure 3.5. In general, the inputs for boths workflow are the same: a testbench and a top-level function. If we want to generate RTL code for the top-level function, we use HDL Coder. If the C code is needed, we insert the testbench and the top-level function to Embedded Coder.

The HDL workflow advisor is still available for HDL Coder. This process can be divided into five main steps: (a) data format setting, (b) target interface setting, (c) IP Core generation, (d) FPGA project integration, and (e) FPGA programming. Note that step (a) is a new step compared to previous Simulink workflow. In this step, a fixed-point conversion tool will help us to set data representation for each variable. It has been shown in [37] that a fixed-point algorithm can be transformed into a more area efficient RTL implementation. The conversion feature included by HDL workflow advisor shortens development cycles dramatically. Based on the testbenches, all the variables are analyzed so that statistical results are produced and developers can easily decide the suitable fixed-point format for them. After setting the fixed-point configuration, the tool helps designers to analyze the precision loss and overflow of the fixed-point version; this is a very useful feature for HDL Coder.

For the C code generation, the workflow is quite simple. After setting a top-level function and a testbench, Embedded Coder generates C code directly for the top-level function. However, the workflows for the hardware and software generation do not interact with each other. Thus, the software and hardware design cannot be integrated automatically.

3.2.1 Discussion

Advantages:

- Easy mapping between the m-script and the C code: The way of C programming is very similar to writing a m-script and therefore it is very easy to find one to

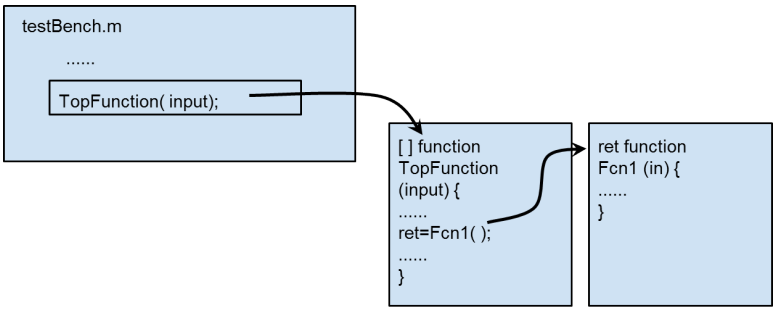


Figure 3.4: Preparation of MATLAB Workflow

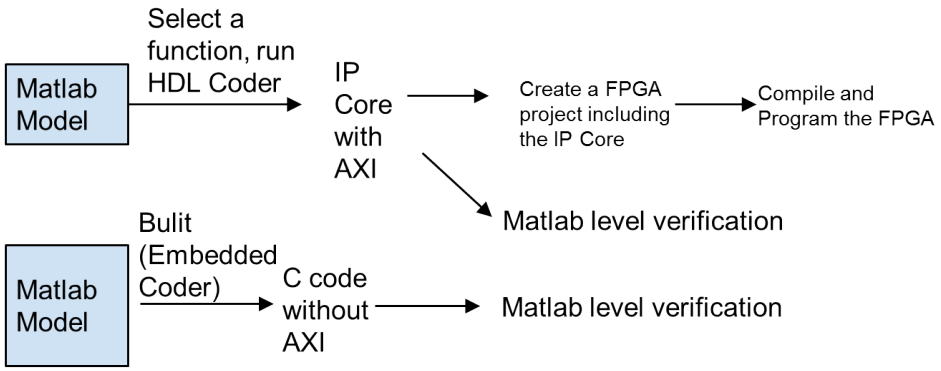


Figure 3.5: Overview of MATLAB Workflow

one mapping between them. This is a good feature that helps developers to better understand the generated code.

- Good support from m-script to C code: Embedded Coder has a good support for MATLAB functions. Almost all functions included in m-script can be synthesized to C code.

- Fixed-point conversion: The biggest advantage of HDL Coder is the feature of fixed-point conversion. It evaluates the fixed-point format for each variable based on real testbenches. The floating-point C code is transformed into a compatible fixed-point version, which is in turn used to derive the hardware implementation.

Limitations:

- No integration features: There is no software/hardware integration features for MATLAB models. Embedded Coder does not create any interaction functions that communicate with hardware parts. The HDL Coder does not modify the original models to indicate that some functions will be implemented by hardware accelerators.

- Limited supported functions: Only a small set of the pre-defined functions is supported by HDL Coder [38] and some language syntax is not supported by Embedded Coder [39]. For example, in our real model, data with structural format is used as input and output parameters, but structural format is not allowed by HDL Coder. Therefore, we suggest that designers should check which MATLAB language features, functions, classes, and System objects are supported by Embedded Coder and HDL Coder before the model design [38, 39].

- Hard to find reference designs: Most of the documents are only available to users with product licenses. For some features, even with a license, it is still difficult to find their corresponding reference designs. For instance, HDL Coder can support the following three communication schemes: the basic non-blocking AXI interface, blocking AXI interface, and AXI streaming interface, but we can only find the reference design using the basic setting.

- Blocked fixed-point conversion: As mentioned before, fixed-point conversion is a useful feature of HDL Coder. However, a limitation has been found in a real user case. HDL Coder projects have to be located in the same folder as all the other m-script files. In our application, m-files are located in different folders, so when fixed-point conversion is used, an error will occur and the process will be terminated.

Since the MATLAB workflow does not include any interactive design, which establishes the communication between modeling tools and boards, it is not possible to run a co-simulation like what we can do in the Simulink workflow.

Figure 3.6 presents one way we proposed to verify the generated code. In this case, 'Function1' is the initial top-level function.A 'main function' is created including 'Function1', input data, and output data. The input and output data are pre-generated from MATLAB and the output data will be used as a reference for verification. The input data is fed to 'Function1' and the result of 'Function1' is compared with the output data. Both the target function ('Function1') and the verification data are synthesized when 'MainFunction' is sent to the code generators. By doing so, we can verify the generated code automatically.

Conclusion: For MATLAB models, Embedded Coder and HDL Coder can generate C functions and RTL Code. The C code and IP Cores are modules useful for the further hybrid system development. However, developers need to do the further integration work manually.

3.2.2 C to HDL

Due to the limitations of the MATLAB workflow discussed in Section 3.2, the HDL Coder cannot satisfy our requirements. Another potential solution is to use High-Level Synthesis (HLS). HLS is a tool designed by Xilinx and it can synthesize RTL code based on C code. The C code can be generated by Embedded Coder mentioned in the beginning

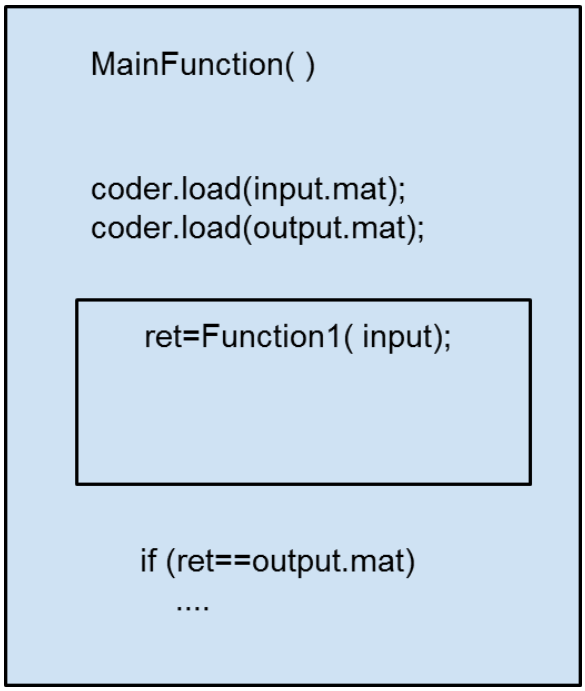


Figure 3.6: Verification Methodology Based on MATLAB Models

of this section. We can get an improved MATLAB workflow by combining Embedded Coder and HLS as shown in Figure 3.7.

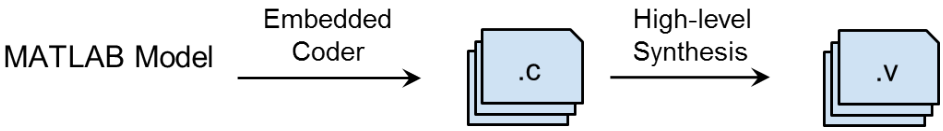


Figure 3.7: Overview of MATLAB Workflow Based on HLS

The way of using HLS is very similar to that of using HDL Coder. A top-level function and a testbench are given to HLS, and then it builds C code, synthesizes RTL code, and exports IP cores. Similar to HDL Coder, HLS has some interface and optimization settings, which can be configured by TCL scripts.

Advantages:

- Good support for SoC chips: HLS is part of the Xilinx design flow, so it has a good support for hardware logic implementation, namely it offers very flexible optimization options and interfaces. The Xilinx website has a comprehensive set of tutorials and documents describing how to use these features. In Chapter 4 we will show how to use three types of implementation settings to improve the performance of the RTL Code.

Limitations:

- Manual fixed-point design: Fixed-point code is supported by HLS, but developers need to manually set the particular data representation for each variable. In addition, HLS does not offer any accuracy analysis feature, so developers need to run simulations in HLS and verify the precision loss and overflows manually.

3.3 Testing Experience Comparison

In this chapter, we introduced and compared different workflows for hybrid system design based on MBD. In the following two tables, the testing experience for these workflows is given. Here, we use '✓', '◯' and '✗' to represent good, medium and bad experience, respectively.

The first part of the hybrid system design is the C coding. Embedded Coder is used to synthesize C code. The Simulink workflow and the MATLAB workflow are evaluated based on Simulink models and MATLAB models respectively. According to the description in this chapter, five aspects are compared between these two workflows and the testing experience is shown in Table 3.1.

Table 3.1: Performance of C/C++ Coder

C/C++ Coder:	Embedded Coder	
	Simulink	MATLAB
Mapping	✗	✓
Platform	◯	✓
Model limitation	✓	✓
Verification	✓	✗
Automation	✓	◯

The second part of the hardware/software co-design is the RTL coding. In this chapter, we described how to use HDL Coder based on Simulink models and MATLAB models. Furthermore, we also introduced HLS as another candidate to generate RTL code. Table 3.2 summarizes our testing experience of these three methods.

Table 3.2: Performance of RTL Coder

RTL Coder:	HDL Coder		HLS
	Simulink	MATLAB	
Mapping	✓	✗	✗
Fixed point	○	✓	✗
Interface flexibility	○	○	✓
Model limitation	✓	✗	○
Integration	✓	✗	○
Verification	✓	✗	✗
Platform mobility	✓	✓	○
Automation	○	✗	✗

3.4 Conclusion

This chapter analyzed several potential methodologies for MBD. Both Simulink models and MATLAB models can be synthesized to software and hardware code. The biggest advantage of Simulink workflow is that it supports co-simulation between Simulink and target boards. This means that the software and hardware code can be downloaded to the boards and the generated codes can be verified through Simulink. Having MATLAB models, developers can use MathWorks tools to generate code, but MATLAB workflow does not support co-simulation, so developers need to verify the code manually. Given the limitations we found in the MATLAB workflow, we investigated another potential solution with MATLAB models called HLS. MATLAB models are first synthesized to C code and then developers select specific parts of C code to be transformed into HDL code by using HLS.

In Figure 3.8, we summarize all three alternative workflows for hybrid systems design. First, C code is generated with the Embedded Coder and then the applications run on the processors for profiling. Based on the profiling results, HDL Coder or HLS can be used to synthesize RTL code and IP cores. In the end, the IP cores are integrated to become a hybrid hardware/software system.

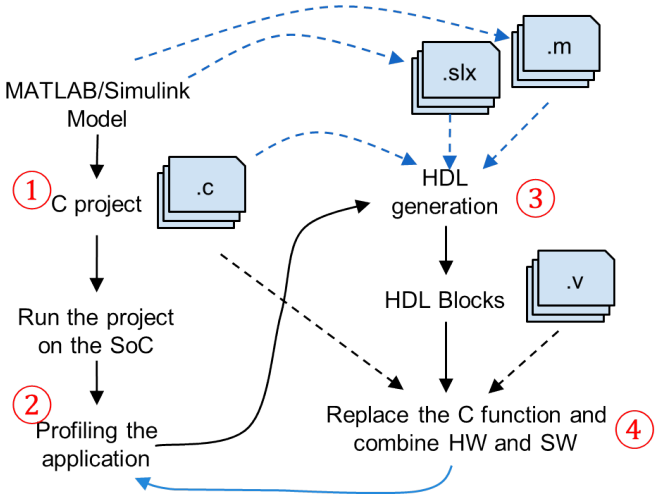


Figure 3.8: Overview of the Workflow for Co-design

4

Application Analysis and Implementation

In this chapter, the methodologies of system optimization will be introduced. First, the profiling methods are introduced in Section 4.1, where three different profiling methods are compared. Next, according to the results obtained from the profiling methods, we propose three criteria for selecting the function(s) to be implemented in PL. Finally, the optimization of the hardware implementation is discussed in Section 4.3, where we evaluate loop pipelining, loop unrolling, and stream based interface schemes.

4.1 Profiling

Profiling is an important step that needs to be done before applications are separated into hardware and software parts. The performance in terms of execution time, memory usage and function call graphs are usually analyzed during the profiling process. In this project, the system throughput is our main criterion. In the following parts, we discuss and compare three different profiling methods, i.e., target profiling, host profiling, and MATLAB profiling, for analyzing the execution time of applications.

4.1.1 Introduction of A Test Case

Note that the model of our ADAS application is developed in MATLAB, we create a test model in the same environment for comparing different profiling methods. The MATLAB model can be profiled in MATLAB, or it can be transformed into C code by Embedded Coder, an then be profiled either on a target board of in a host environment.

Figure 4.1 depicts the structure of the proposed test model. In the following sections, the results of three profiling methods will be introduced and compared.

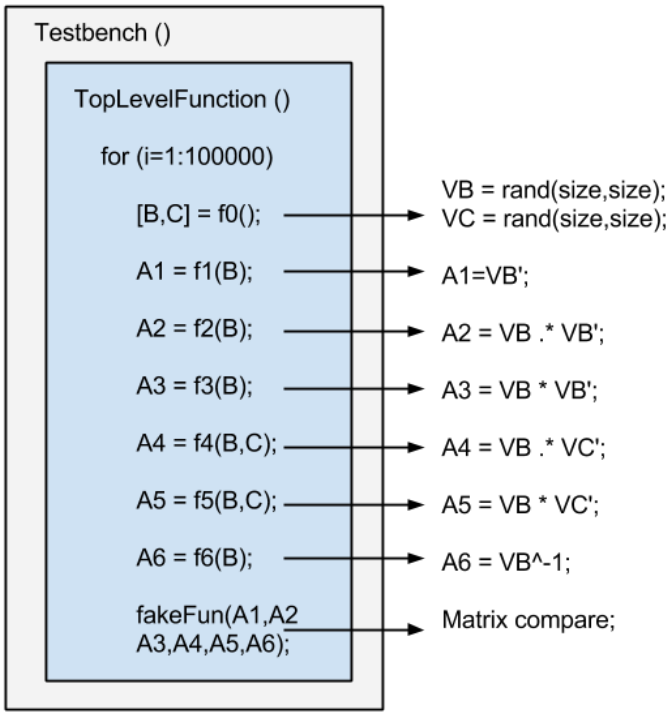


Figure 4.1: Test Code

4.1.2 Target Profiling

For the target profiling method, the test MATLAB model is first transferred into C code and then profiled on target boards. The execution time of each function is measured based on a hardware platform.

The steps of target profiling are described in [40]. The output results of the target profiling includes two files: a text file, which shows the statistic result of each function and the subfunctions in it (see Figure 4.2) and a graph generated by an open source application (gprof2dot [41]).

In Figure 4.2, there are six columns. The first column shows the indexes of different functions. The second column lists the percentage of the time taken by running each function. The next two columns indicate the time the self-code takes and how much time is taken by sub-function calls in second. The number of calls is shown in the fifth column. In the last column, a program structure is shown for each function. For example, for the 'main' function shown in the top of Figure 4.2, its parent(s) is a 'start' function. Note that the parent functions can be more than one, if several functions invoke the same function. The children of the 'main' function are 'topLevelFunction' and 'init platform'.

Figure 4.3 shows two call graphs for our test model. For each block, four elements

```
index % time    self  children    called     name
                0.02   14.09       1/1           _start [2]
[1]     96.6    0.02   14.09       1         main [1]
                0.01   14.08   10000/10000       topLevelFunction [3]
                0.00    0.00       1/1           init_platform [70]
-----------------------------------------------
                                                 <spontaneous>
[2]     96.6    0.00   14.11                 _start [2]
                0.02   14.09       1/1           main [1]
-----------------------------------------------
                0.01   14.08   10000/10000       main [1]
[3]     96.4    0.01   14.08   10000         topLevelFunction [3]
                0.00    4.69   10000/10000       f6 [5]
                0.28    2.86   10000/10000       f5 [10]
                0.21    2.86   10000/10000       f3 [11]
                0.01    2.42   10000/10000       f0 [12]
                0.35    0.00   10000/10000       f2 [22]
                0.21    0.00   10000/10000       f4 [25]
                0.15    0.00   10000/10000       f1 [27]
                0.01    0.00   10000/10000       fakeFun [45]
                0.01    0.00   70000/100000      emxInit_real_T [44]
                0.01    0.00   70000/100000      emxFree_real_T [52]
-----------------------------------------------
...
-----------------------------------------------
...
```

Figure 4.2: Result of Target Profiling 1

are shown in the graph: the name of functions, percentage of time taken to run the functions (including sub-functions), percentage of time that self-code takes (excluding sub-functions), and number of calls. From the graphs, it is very convenient to find the relationships between different function blocks and the proportion of the execution time for each function. Moreover, from the color of the diagram, developers can realize which parts require more computational resources.

4.1.3 Host Profiling

When we use the target profiling, applications are run on target boards. The target boards are the actual platform where the applications will be implemented, so target profiling can provide the most accurate results. However, the disadvantage of target profiling is that developers have to work with boards. Another alternative is using host profiling, which means software developers can profile their designs on computers. The

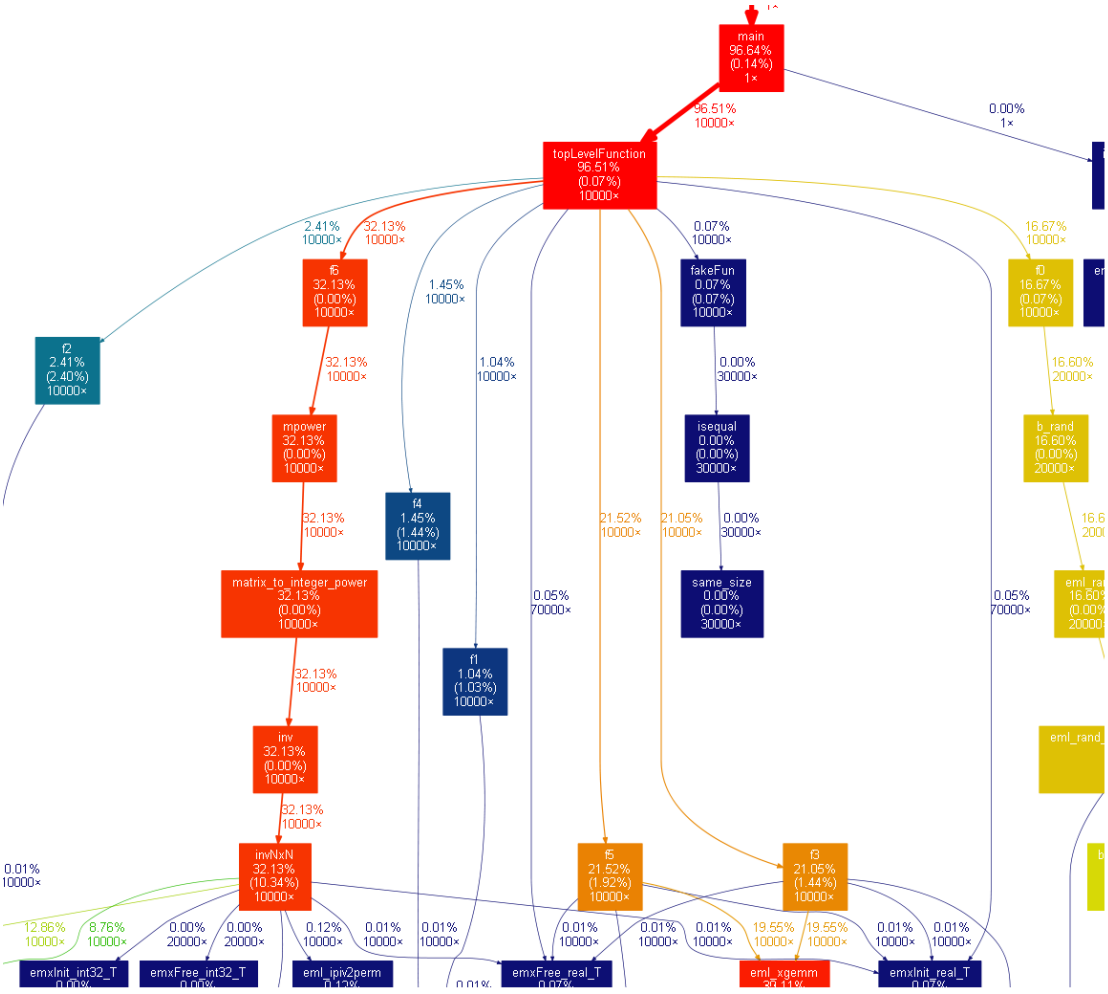


Figure 4.3: Result of Target Profiling 2b

steps for host profiling are summarized in Appendix B. In general, host profiling uses the same profiling feature as target profiling and produces similar outputs, namely one text file and one calling graph. The comparison results between host and target profiling will be discussed in Section 4.1.5

4.1.4 MATLAB Profiling

The third method of profiling is MATLAB profiling, by which model developers can optimize their design during the model development phase. In the next section, we will compare MATLAB profiling with the other two methods.

Figure 4.4 shows the percentage for execution time of each sub-function and the actual execution time. The figure provides similar information as Figure 4.2. Additional information offered by MATLAB profiling is line-based analysis shown in Figure 4.5

Children (called functions)

Function Name	Function Type	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
f6	function	100000	4.668 s	0.00 Kb	84.00 Kb	0.00 Kb	29.5%	
fakeFun	function	100000	1.995 s	36.00 Kb	4.00 Kb	36.00 Kb	12.6%	
f0	function	100000	1.609 s	0.00 Kb	36.00 Kb	0.00 Kb	10.2%	
f5	function	100000	1.061 s	0.00 Kb	20.00 Kb	0.00 Kb	6.7%	
f3	function	100000	0.908 s	0.00 Kb	0.00 Kb	0.00 Kb	5.7%	
f2	function	100000	0.692 s	0.00 Kb	0.00 Kb	0.00 Kb	4.4%	
f4	function	100000	0.545 s	20.00 Kb	0.00 Kb	20.00 Kb	3.4%	
f1	function	100000	0.429 s	0.00 Kb	0.00 Kb	0.00 Kb	2.7%	
Self time (built-ins, overhead, etc.)			3.912 s	88.00 Kb	0.00 Kb	84.00 Kb	24.7%	
Totals			15.820 s	144.00 Kb	144.00 Kb	84.00 Kb	100%	

Figure 4.4: Result of MATLAB Profiling 1

and Figure 4.6, where the execution time and memory usage per line are illustrated respectively.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time	Time Plot
18	`A6=f6(VB); %% A6 = VB^-1;`	100000	5.129 s	4.00 Kb	84.00 Kb	4.00 Kb	32.4%	
20	`ret=fakeFun(A1,A2,A3,A4,A5,A6)...`	100000	2.569 s	120.00 Kb	4.00 Kb	84.00 Kb	16.2%	
4	`[VB, VC] = f0(size); %% VB = r...`	100000	1.926 s	0.00 Kb	36.00 Kb	0.00 Kb	12.2%	
16	`A5 = f5( VB, VC); %% A5 = VB *...`	100000	1.466 s	0.00 Kb	20.00 Kb	0.00 Kb	9.3%	
12	`A3=f3(VB); %% A3 = VB * VB';`	100000	1.274 s	0.00 Kb	0.00 Kb	0.00 Kb	8.1%	
All other lines			3.455 s	20.00 Kb	0.00 Kb	84.00 Kb	21.8%	
Totals			15.820 s	144.00 Kb	144.00 Kb	84.00 Kb	100%	

Figure 4.5: Result of MATLAB Profiling 2

4.1.5 Comparison

In this section, we compare the three profiling methods. We assume that the results from target profiling will be the final performance on the real system. Then, by comparing with the results of target profiling, we can verify the accuracy of the results provided by host profiling and MATLAB profiling.

Table 4.1 shows the profiling results of target, MATLAB and host profiling. Considering the test case designed in Section 4.1.1, the rank of each function is listed for each considered profiling method. The left table gives the result obtained from target profiling. We see that 'f6', 'f5' and 'f3' are the most heavy functions, which take more than 70% of execution time in total, while 'f2', 'f4' and 'f1' are three light weight functions. The table in the middle shows the result from MATLAB profiling. It can be seen that the ranks of functions are almost the same as the ones shown for target profiling.

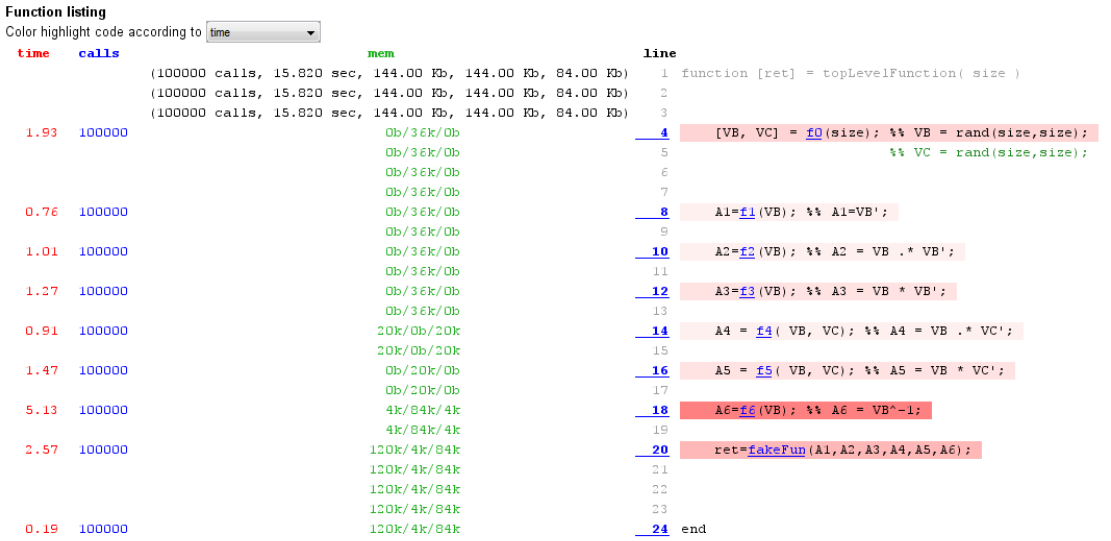


Figure 4.6: Result of MATLAB Profiling 3

However, we observe that 'f5' and 'f3' take much smaller proportion of execution time than that of target profiling. Note that both 'f5' and 'f3' are matrix multiplication. So one possible explanation is that MATLAB has better optimized algorithms for matrix computations. Finally, the right table presents the results obtained from host profiling. The execution time for all functions except 'f6' are similar to the results from target profiling. However, the most heavy time consuming function 'f6' (a matrix division) takes almost zero execution time in the host profiling. The reason for this result is not clear. Note that the total execution proportion is not 100%. This might imply that some operations are done in operating system level and these operations cannot be sampled by Eclipse.

Table 4.1: Comparison of Profiling Results

Rank in Target	%
f6	37.06
f5	17.54
f3	17.46
f4	2.24
f2	2.16
f1	1.1
self	0.1
total	96.4

Rank in MATLAB	%
f6	29.5
f5	6.7
f3	5.7
f2	4.4
f4	3.4
f1	2.7
self	24.7
total	100

Rank in Host	%
f3	17.8
f5	17.7
f2	1.7
f1	1.4
f4	1.4
f6	0
self	0
total	40

4.2 Application Analysis and Partitioning

After profiling, the next step is to partition the application into hardware and software parts. Different criteria can be used to decide how to partition hybrid systems. In this project, the system throughput is our main criterion. Thus, one goal of partitioning is to find the optimal way to get the highest speedup of execution time. Designers can always test blocks one by one to evaluate whether it is faster to implement the block in software, or it should be implemented in hardware logic. However, this process takes long development time, since each trail contains several steps, including RTL code generation, RTL optimization, IP Core integration, and verification. Therefore, it is not efficient to partition the options one by one.

In this section, we introduce three criteria: execution time, communication overhead, and potential for speedup, which can be used to assist designers to decide whether a block is a good candidate for going through the process.

4.2.1 Execution Time

The execution time for a block i can be calculated by $T_i^{exe} = T * P_i$, where T is the execution time for the whole system, and P_i is the percentage of the time taken by block i. Let S_i denote the speedup factor for block i, the new execution time for the whole system will be T'.

Equation 4.1 is the expression of Amdahl's Law and is a method to estimate the system execution time when one part of the system is accelerated. According to Equation 4.1, the lower bound of the new system execution time T' is $T * (1 - P_i)$, which is achieved as S_i goes to infinite. Therefore, potential candidates should have higher values of P_i, which can be found from the profiling results mentioned in the last section.

$$T' = \frac{T * P_i}{S_i} + T * (1 - P_i) \tag{4.1}$$

Equation 4.2 is another expression of Equation 4.1 using T' as the function of T_i^{exe}. Assumed that S_i is the constant for all candidates and it is larger than 1, the longer T_i^{exe}, the faster execution time T' is on the new system. Consequently, the 1st criterion is longer execution time T_i^{exe}.

$$T' = f(T_i^{exe}) = \frac{T_i^{exe}}{S_i} + T - T_i^{exe} = T_i^{exe}(\frac{1}{S_i} - 1) + T \tag{4.2}$$

4.2.2 Communication Overhead

Once software blocks are replaced by hardware accelerators, the new execution time will consist of two parts: the computation time (T_i^{exe}/S_i) and the communication time (T_i^{tran}), that is,

$$T' = f(T_i^{exe}, T_i^{tran}) = \frac{T_i^{exe}}{S_i} + T_i^{tran} + T - T_i^{exe} \tag{4.3}$$

According to the experience from this project, the communication overhead can be easily estimated based on the number of data, clock frequency and selected communication protocol. Let n_i^i and n_i^o denote the number of input and output data respectively. Then, the communication overhead is $(n_i^i + n_i^o) * t^{tran}$. Using the basic AXI protocol, t^{tran} is 16 clock cycles, so it takes 160 ns to send or receive one data if the clock frequency is set to 100MHz.

4.2.3 Computation Characteristics / Potential for Speedup

From Equation 4.3, we see that, the execution time of the new system depends on three factors, i.e., T_i^{exe}, T_i^{tran}, and S_i. The first two factors, i.e., the execution time (T_i^{exe}) and the communication overhead (T_i^{tran}) has been discussed in the previous two subsections. Now, we turn our attention to the third factor, i.e., the speedup factor (S_i). The goal of this step is to find which blocks are time consuming in processors but run efficiently in hardware, resulting in a larger value of S_i. The comprehensive expression of T' is given in Equation 4.4.

$$T' = f(T_i^{exe}, T_i^{tran}, S_i) = \frac{T_i^{exe}}{S_i} + T_i^{tran} + T - T_i^{exe} \tag{4.4}$$

One property of hardware (FPGA) is that it can support massive parallel computations. For example, one Zynq chip contains 220 DSP units, but a processor contains much less Arithmetic Logic Units (ALU). 220 arithmetic calculations can be executed at the same time in Zynq, while few calculations can run in parallel on the processor. As a result, algorithms that contain a lot of parallelism can get high speedup when they are implemented in hardware. Furthermore, loops with independent iterations can also be executed in parallel on hardware to achieve high speedup.

In order to explain what kinds of computation characteristics are suitable for hardware implementation, six examples are examined:

Function 1: $\vec{A} = \vec{B}'$;
Function 2: $\vec{A} = \vec{B} \odot \vec{B}'$;
Function 3: $\vec{A} = \vec{B}\vec{B}'$;
Function 4: $\vec{A} = \vec{B} \odot \vec{C}$;
Function 5: $\vec{A} = \vec{B}\vec{C}$;
Function 6: $\vec{A} = \vec{B}^{-1}$;
where $\odot$ *denotes for elementwise multiplication.*

In Appendix A, generated C code for cases 1-4 is illustrated. The code generator produces C code as expected. Table 4.2 shows the computation demands and the execution time obtained based on different profiling methods, considering test cases 1 to 5. Assuming that addition, subtraction, multiplication, and division are all floating point calculations taking the same cycles for ALU, the value for computation demands is the sum of four values. All the values for different test cases are normalized based on the computation demands of case 2.

Table 4.2: Computation Demands and Results of Profiling

	+/-	*/÷	Inputs	Computation demands	Time in Target	Time in MATLAB	Time in Host
f1	0	0	225	0	0.51	0.61	0.82
f2	0	225	225	1	1.00	1.00	1.00
f3	15*225	15*225	225	30	8.08	1.30	10.47
f4	0	225	2*225	1	1.04	0.77	0.82
f5	15*225	15*225	2*225	30	8.12	1.52	10.41

Case 1 does not require any computation as matrix transpose is implemented by memory operations (here address calculation is not taken into consideration). Note that vector B is a 15*15 matrix, so Case 2 requires 225 multiplications, and Case 3 requires 15*225 additions and 15*225 multiplications. Consequently, the ratio of computation demands between Case1, Case2, and Case3 is 0:1:30. From Table 4.2, we see that the ratios of the execution time for these three cases are 0.51:1:8.08 and 0.82:1:10.47 according to target profiling and host profiling respectively, which are close to the computation ratio. The result from MATLAB profiling has the same rank but the ratio is not linear at all. This is probably because algorithms are optimized in MATLAB. Case 1,2 and 3 have the same number of input data, so they have the same value of T_i^{tran}. Since Case 3 has the biggest computation demands, it can get the biggest potential speedup S_i if all its computations are implemented in parallel. Cases 4 and 5 have the same demands as Cases 2 and 3 respectively, can be seen from profiling results. Although Cases 3 and 5 have the same potential speedup based on computation demands, Case 5 is not a good candidate for compared to Case 3, since requires a larger amount of input data.

In conclusion, blocks that have higher computation demands, less data exchange, and better speedup factors are good candidates for hardware implementation.

4.3 HDL implementation techniques

Table 4.3: Options of Implementation

Implementation Options	Example(s)
Interface Management	specifying bus interfaces
Design Optimization	arbitrary precision data types
Function Optimization	function inlining
Loop Optimization	unroll, pipeline
Array Optimization	memory resource selection
Logic Structure Optimizations	struct packing

According to the previous evaluation of different workflows, HLS is the tool selected to synthesize RTL code. The steps of HLS has been introduced in Section 3.2.2. There are a lot of options by which designers can set and customize the implementation of IP Cores. A list of options and example settings can be found in Table 4.3. Detailed description of all settings could be found in a Xilinx user guide document [42].

In this section, two useful settings for loop optimization and an interface scheme, Direct Memory Access (DMA), will be introduced.

4.3.1 Pipelining

Loops are commonly used in algorithms. In hardware implementations, the input data is fed into a pipeline and results are delivered to the output port iteration by iteration. Figure 4.7 shows a simple example that contains loop iterations. Ideally, each operation in the loop takes 1 clock cycle in default setting. Therefore, one iteration takes 3 cycles to finish. The next iteration will not start until the previous one is finished, thus, the second iteration starts in cycle 4 and it takes 6 cycles in total to finish 2 iterations.

Assuming that there is no dependence between these two iterations, then in pipelining mode the second iteration does not need to wait for previous iteration to complete all operations. It can start as long as no resource conflict exists. In Figure 4.8, the iteration latency is still 3 cycles, but the initiation interval between two iterations is reduced from 3 cycles to 1 cycles. The blocks with the same color are executed at the different times, so no resource conflict exists. As shown in Figure 4.8, it takes 4 cycles in total to finish 2 iterations, which is 2 cycles faster than the normal setting.

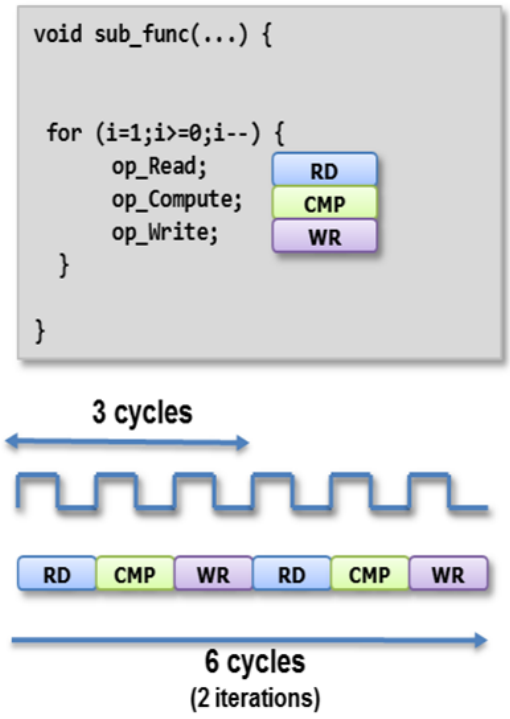


Figure 4.7: An Example Loop And Time Schedule of The Loop in Normal Mode

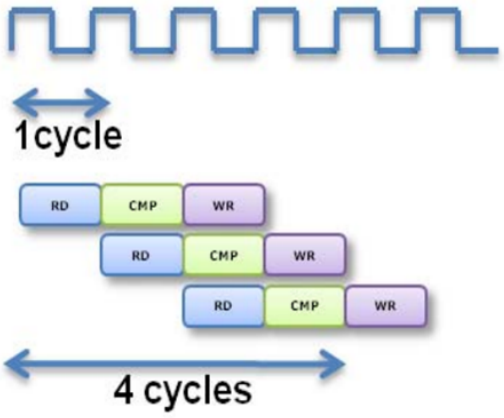
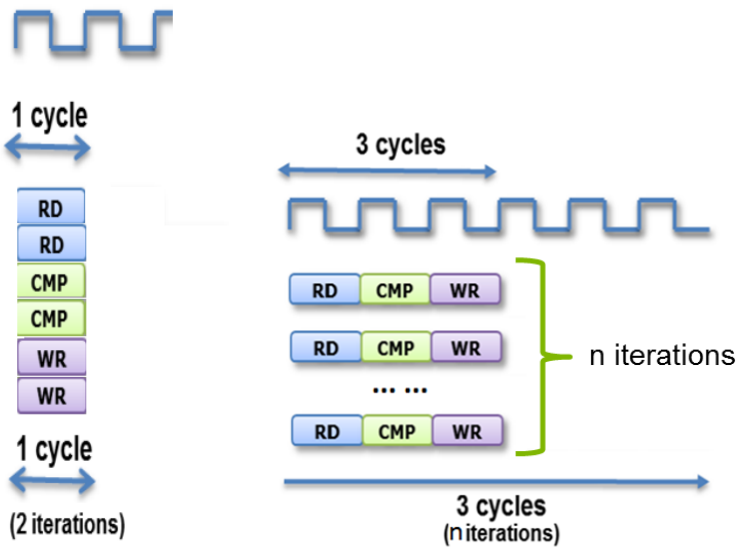


Figure 4.8: Time Schedule of The Loop in Pipelining Mode

4.3.2 Loop Unrolling

Another way for the loop optimization is loop unrolling. If there is no dependency between iterations and the operations within one iteration are independent, then in the unrolled mode, ideally all operations can run at the same time, as shown in Figure 4.9(a).



(a) Independent operations

(b) Dependt operations

Figure 4.9: Time Schedule of The Loop in Loop Unrolling Mode (a) and (b)

No matter how many operations are contained in one loop, the latency of the loop is 1 cycle. HLS duplicates hardware logic so that all operations execute in parallel. However, if there is dependency between operations within the iteration, those operations have to execute sequentially. In this example, the three operations within one iteration (read, calculate and write) need to execute one by one. Therefore, the latency of one iteration is 3 cycles as shown in Figure 4.9(b). Since the hardware implementation is parallelized, the loop duration is always 3 cycles no matter how many iterations are contained in this loop. HLS also allows the designers to set a partially unrolled loop. In that case, the scale of parallel execution is decided by the unrolling factor.

4.3.3 AXI DMA Data Transfers

In Section 4.2.2, it has been discussed that the rate of data communication is 0.16 us/data for sending and receiving data between the processor and the hardware accelerator. The maximum width of each data is 32 bits using AXI Master interfaces (MGP) [43]. The interconnection between PL and PS of Zynq is shown in Figure 4.10. An IP Core with AXI Slave is located in PL and connected to PS via MGP0. In general, every data communication process consists of address configuration, data write, data read and protocol overhead, so the efficiency of MGP connection is low. In our application, one block has an input array with 225 elements. Hence, it takes $225 * 0.16us = 36us$ for input data transmission.

High Performance ports (HP) and Accelerator Coherency Port (ACP) are the other

two options for data transfer. Both of them belong to the category of DMA schemes. By using HP, IP Cores can communicate with DDR memory directly. Hardware accelerator can also exchange data with the L2 cache of the processor via ACP. The first advantage of DMA schemes is that exchanging data does not need to go through L1 Cache and the core of processors. Thus, the time for memory flush could be saved, especially for the case with large set of data streams. The second advantage is that one address is enough for transmitting one array. Since DMA schemes are stream-based, the protocol can access a sequence memory area according to its base address and the size of the space. A detailed tutorial of DMA connection for Zynq-7000 can be found in [44].

After we implemented HP connection for our IP Core, the transmission time is reduced from $36us$ to $16us$. It has been shown that the speedup can be even higher if a larger stream is exchanged [45].

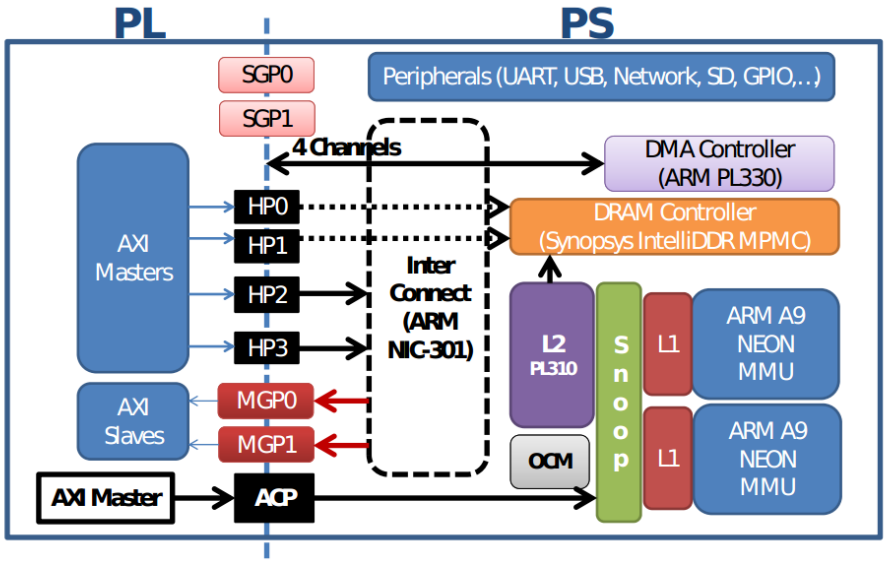


Figure 4.10: Interconnection between PL and PS of Zynq [45]

4.4 Conclusion

In this chapter, using the workflow suggested in Chapter 3, we discussed how to analyze applications and implement hardware IP Cores. Hardware/software partitioning and hardware logic optimization are the two main parts of this chapter.

Profiling is the step before partitioning. We introduced and compared three ways of profiling, i.e., target, host and MATLAB profiling. According to our evaluation, the results of these three profiling methods are similar to each other. Because systems will finally run on target boards, target profiling is suggested, but the other two ways are also acceptable for developers.

Next, three criteria: execution time, communication overhead, and potential speedup, were proposed for assisting designers to decide which blocks are better to be implemented in hardware when partitioning hybrid systems.

Finally, the hardware implementation was optimized in order to get better performance and three common settings for HLS projects were investigated.

5

Evaluation

In this chapter, we use two test cases to evaluate the performance of the proposed MBD workflow, profiling methods, and optimization schemes. Using the first case, we show how the performance, in terms of throughput of the blocks, is affected by the implementation settings. The second case is selected from our data fusion application, where an AXI DMA scheme is used to reduce the communication overhead. The throughput performance and hardware resource utilization are compared and analyzed for different optimization schemes.

5.1 Implementation Results

The first case is a simple loop, which contains 15 iterations. Each iteration consists of one multiplication operation. The C source code of this case is shown as follows:

Case 1:

```
void testCase(const float v[15],float d[15]){
    char i;

    for (i = 0; i < 15; i++) {
        d[i]=v[i]*v[i];

    }
}
```

The second case is a function selected from the data fusion application. The C source code of this case is generated from the MATLAB Code *covariance = srCovariance*srCovariance'* and *variances = diag(covariance)*. Here, the input (srCovariance) of the former MATLAB function (multiplication) is one matrix with size of 15*15. The output of the former MATLAB function is then used as the input for the latter one

(diag), which produces a new output, the vector "variance" with size of 15*1. By combining these two MATLAB functions together, the number of elements for the output of the function block shown in Case 2 reduces to 15, which is much smaller than the case where only the first MATLAB function is contained in the C function block.

Case 2:

```
void matrixProduct(const float v[225],float d[15]){

    char i,j;
    float b_StateVectors[225];

    //covariance = srCovariance*srCovariance';
    loop1: for (i = 0; i < 15; i++) {
        loop2: for (i111 = 0; i111 < 15; i111++) {
          b_StateVectors[i + 15 * i111] = 0.0F;
          loop3: for (i112 = 0; i112 < 15; i112++) {
            b_StateVectors[i + 15 * i111]
            += v[i + 15 *i112] * v[i111 + 15 * i112];
          }
        }
      }

    //variances = diag(covariance);
    for (j = 0; j < 15; j++) {
        d[j] = b_StateVectors[j << 4];
    }

}
```

5.1.1 Performance Analysis

Case 1: We consider three different configurations for loop implementation, i.e., default loop, pipelined loop, and unrolled loop. The latency of the block is analyzed and compared between these configurations.

If we set the default configuration for the loop, we will observe the control step diagram in Figure 5.1. The bar in Line 1 indicates the time interval for each iteration of the loop. As Lines 2-6 show the information for different operations within each iteration, for instance, reading vector 'v', multiplication, and writing back to vector 'd'. Each iteration starts from 'C1' and ends with 'C7', so the iteration latency is 7 clock cycles. Since the function contains 15 iterations, the total latency is 15*7+1=107 cycles, where one additional cycle is used for loop status initialization.

When the loop is configured in a pipeline mode, according to the concepts mentioned in Section 4.3.1, the computation in the loop is pipelined. The diagram of control steps is the same as Figure 5.1. However, the initiation interval decreases from 7 clock cycles

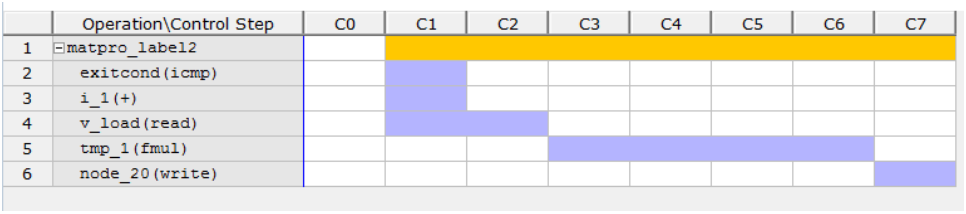


Figure 5.1: Control Steps of Case 1 in Default Setting and in the Pipeline Mode

to 1 clock cycles as can be seen in Figure 5.2, since each iteration starts 1 clock cycle after the its prior iteration. The latency for each individual iteration is still 7 clock cycles. Therefore, the total latency of the block is 1*15+7+1=23 clock cycles.

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
matpro	-	22	23	-	-
matpro_label2	yes	20	1	7	15

Figure 5.2: Latency Calculation for Case 1 in the Pipeline Mode

When the loop is configured in unrolled mode, HLS generates a total parallel hardware logic, so we can see 15 parallel control steps in Figure 5.3. Table 5.1 summarizes the latency and speedup factors for three different settings. We see that compared to the default mode, the pipelining scheme speeds up the block by a factor of 4.65 and the unrolling scheme speeds up it by a factor of 26.75. The reason why it gets more than 15 times speedup in the unrolled mode is that it not only parallelizes the hardware logic but also saves a lot time for data multiplexing.

Table 5.1: Total Latency and Speedup for Case 1

Case 1:	Latency (cycles)	Speedup
Default	107	1
Pipeline	23	4.65
Unrolled	4	26.75

Case 2:

Initially, the function takes around 10% (314 us) of the total execution time for the whole ADAS application. We try to use hardware accelerator to implement the selected function block as given in Case 2. Table 5.2 shows the execution time and speedup factors when considering different loop configurations. Note that there are three nested loops in Case 2, and the iterations of different loops have dependencies between each other. Thus, the effects of different loop configurations on the latency performance is not straight forward to analyze as in Case 1. For example, if we unroll a loop, the execution time does reduce, but the speedup factor does is not equal to the unroll factor. As it shown in Table 5.2, although the block needs only 366 clock cycles to execute the

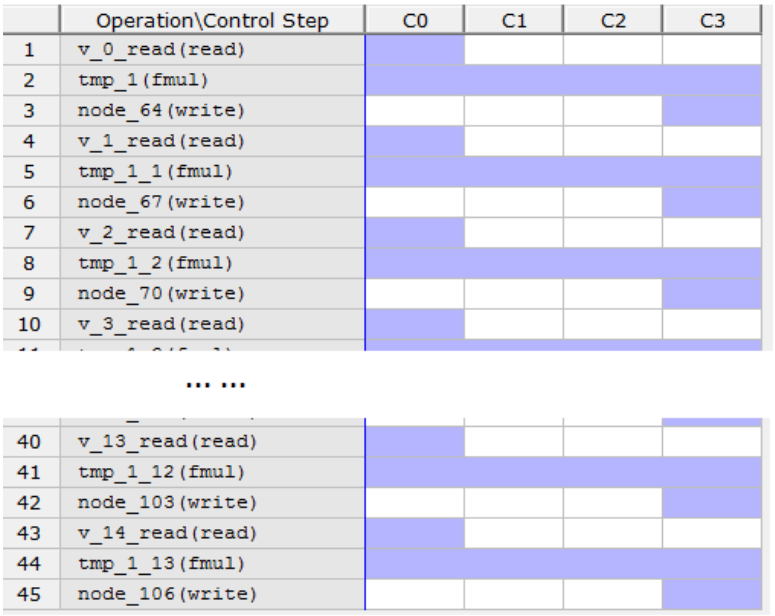


Figure 5.3: Control Steps of Case 1 in Loop Unrolled Mode

computation (4 us @ 100MHz), the total execution time for the block is 44 us. This is because the communication takes 40 us.

Table 5.2: Execution time for Case 2 without DMA Interface

Case 2:	Latency	Execution time	Speedup	Loop1	Loop2	Loop3
noDMA1	41012	450 us (410+40)	0.69	default	default	default
noDMA2	8012	120 us (80+40)	2.61	default	unrolled	unrolled
noDMA3	366	44 us (4+40)	7.13	default	pipelined	default

By utilizing DMA communication as discussed in Section 4.2.2, we can further reduce the communication overhead. The DMA scheme is evaluated for case 2. Using DMA communication, the execution time of the block is 58.1 us. Note that the way of data storage for DMA block is different from the non-DMA scheme. Input/output arrays are stored in a Block RAM instead of the registers organized with Look Up Table (LUT). Therefore, 'Case 2/DMA1' does not work in parallel, since one block RAM can only support one pipeline. In order to parallelize the hardware pipeline, the input array is partitioned into several RAMs, see 'DMA2' and 'DMA3' in Table 5.3. By doing so, the execution time can be further reduced. As can be seen from Table 5.4, the execution time does not linearly decrease with the number of partitioned blocked, since only the computation time decreases while the time for data communication does not change.

Table 5.3: Execution time for Case 2 with DMA Interface

Case 2:	Latency	Execution time	Speedup	Loop (DMA Stream Pop)	Loop2	Partition Array 'v'
DMA1	3729	58.1 us	5.41	pipelined	pipelined	1 block
DMA2	1498	35.0 us	9.74	pipelined	pipelined	3 block
DMA3	602	27.0 us	12.62	pipelined	pipelined	15 block

Table 5.4: Latency Breakdown of Hardware Blocks with DMA Interface

Case 2:	DMA1	DMA2	DMA3
latency (cycles)	3729	1498	579
total execution time for the block (us)	58.1	35	27
read input data (us)	7	7	7
format input data (us)	9	9	9
computation (us)	38	15	6
write back and format output data (us)	5	5	5

5.1.2 Hardware Utilization Analysis

Case 1: The resource utilization of Case 1 is shown in Table 5.5 for different implementation settings. When the loop is pipelined, the generated RTL code consumes almost the same amount of hardware resources, which could be explained by the concept of pipelining. When the loop is unrolled by a factor of 15, all hardware logic are duplicated so that the numbers of DSP units and LUT are increased by almost 15 times. We define a new criterion Speedup/Utilization, for selecting the best optimization scheme. The value of this criterion can be calculated as:

$$Speedup/Utilization = \frac{Speedup}{\frac{Num_{BRAM}}{TotalNum_{BRAM}} + \frac{Num_{DSP48E}}{TotalNum_{DSP48E}} + \frac{Num_{FF}}{TotalNum_{FF}} + \frac{Num_{LUT}}{TotalNum_{LUT}}} \tag{5.1}$$

Note that we prefer higher speedup and lower hardware resource utilization. Therefore, larger Speedup/Utilization is better. The last column of Table 5.5 shows the value of Speedup/Utilization for the three solutions. The results indicate that the pipelined loop is the best solution for Case 1, since it achieves the highest value of Speedup/Utilization.

Case 2: The resource utilization and speedup of Case 2 are listed in Tables 5.6, 5.7 for non-DMA solutions and DMA based solutions. In non-DMA mode, solution 'noDMA2' and 'noDMA3' have the same value of Speedup/Utilization. With DMA configuration, when parallel hardware structure is set, only computation time decreases linearly, so speedup does not increase relatively. The best Speedup/Utilization value appears when

Table 5.5: Area and Speedup of Case 1

Case 1:	BRAM	DSP48E	FF	LUT	Speedup	Speedup/Utilization
Default	0	3	225	338	1	45.23
Pipeline	0	3	222	333	4.65	211.53
Unrolled	0	45	2147	4815	26.75	84.85
Total Resource	280	220	106400	53200		

the array is stored in a single block RAM, which means 'DMA1' is the most efficient solution for case 2. If FPGA resource are available, the second optimized solution, 'DMA2', is also a good candidate due to its high speedup. In the end, the tradeoff between the throughput, performance and resource utilization needs to be considered in real system design.

Table 5.6: Area and Speedup of Case 2 without DMA Interface

Case 2:	BRAM	DSP48E	FF	LUT	Speedup	Speedup/Utilization
noDMA1	1 (0.4%)	5 (2.3%)	1114 (1.0%)	6022 (11.3%)	0.69	4.65
noDMA2	1 (0.4%)	15 (6.8%)	3840 (3.6%)	9124 (17.2%)	2.61	9.37
noDMA3	1 (0.4%)	75 (34.1%)	9285 (8.7%)	17581 (33.0%)	7.13	9.36
Total	280	220	106400	53200		

Table 5.7: Area and Speedup of Case 2 with DMA Interface

Case 2:	BRAM	DSP48E	FF	LUT	Speedup	Speedup/Utilization
DMA1	2 (0.7%)	5 (2.3%)	2216 (2.1%)	2412 (4.5%)	5.41	56.37
DMA2	4 (1.4%)	15 (6.8%)	6023 (5.7%)	5728 (10.8%)	9.74	39.48
DMA3	16 (5.7%)	75 (34.1%)	9481 (8.9%)	12071 (22.7%)	12.62	17.68
Total	280	220	106400	53200		

5.2 Conclusion

In this chapter, we analyzed the performance and resource utilization of hardware implementation. We found that different hardware implementation configurations in HLS can dramatically affect the system throughput. Using pipeline loop configuration together with the AXI DMA scheme, a speedup of 10 can be achieved for one block selected from our data fusion application. This implies that hardware/software co-design can improve

the performance of embedded systems.

6

Conclusion

6.1 Contributions

This thesis explores a software/hardware hybrid system for ADAS using MBD. An FPGA-based SoC was selected as the platform of the system because standalone processors are not powerful enough to satisfy the high demands of ADAS applications. Using hardware accelerators on SoC is one potential solution to improve the performance of applications. Moreover, MBD is a methodology widely used in vehicle industry, by which can improve the development workflow and shorten the time-to-market cycle.

The thesis project has two objectives. First, MBD methodologies are evaluated for MATLAB and Simulink models. The aim is to identify a suitable workflow, which synthesizes C and RTL code automatically, minimizes development time and supports most source models. Second, the key technique for hardware/software co-design is investigated based on MBD workflow. We found methods for profiling the hybrid system, criteria for partitioning the system into hardware and software, and ways for optimizing SoC designs to get better system performance.

The workflow for MBD was presented in Chapter 3. Embedded Coder and HDL Coder are two tools from MathWorks. Embedded Coder and HDL Coder can generate C and RTL code from Simulink models. The SoC design was created using the HDL workflow advisor. System-In-the-Loop simulation is integrated in the Simulink environment, which is a helpful feature to reduce the coding and verification time. From MATLAB models, C code can be synthesized by Embedded Coder, but some limitations were found when RTL code was generated using the HDL Coder. Due to these limitations, we selected HLS as an alternative to generate RTL Code from C code. Subsequently, we proposed a workflow that combines Embedded Coder and HLS.

In order to develop hardware/software hybrid systems, we proposed three steps: (1) Profiling, (2) Partitioning, and (3) Implementation optimization. For profiling, we evaluated and compared different profiling methods in Section 4.1. Developers can profile

applications on hosts, target boards or modeling tools. For partitioning, some key factors were listed and explained in Section 4.2. For instance, execution time, communication overhead, and the potential for speedup. Using these factors, developers can decide which blocks are candidates to be implemented hardware accelerators. For implementation optimization, a lot of detailed hardware implementation schemes can be used that affect the performance of hardware accelerators. We investigated loop pipelining, loop unrolling and DMA interfaces in Section 4.3.

After evaluating the workflow of MBD and methods of hardware/software co-design, a real active safety application was implemented. The application was developed in MATLAB. C code was generated by the Embedded Coder. Next, it was executed and profiled on a target board. According to the proposed partitioning criteria, we selected a specific function, which takes around 10% of the total execution time, and migrated it from software to hardware implementation. After the block was optimized in HLS using pipelining and DMA, we achieve up to 12 times speedup. Section 5.1 analyzed different settings, speedups and resource utilization. The ADAS application cannot be further improved because most of the algorithms in the source models are not suitable for hardware implementation. However, the main goals of this thesis have been achieved: we found a suitable workflow for hardware/software co-design based on MDB and we proved that hardware accelerators can significantly improve system throughput.

6.2 Discussion and Suggestions

Based on our investigations, we have some suggestions for MBD.

First, for model selection. Simulink models are more suitable for RTL design while MATLAB models can be mapped to software design easily. Therefore, the model selection mainly depends on whether the project has more software functions or hardware blocks. Additionally, code generation tools have better support for Simulink models than for MATLAB models, especially for integration and verification steps. This support can be very helpful for the developers who do not have long experience in hardware/software co-design, since the tools can automate the whole development flow.

Second, for hardware coder selection: both HDL Coder and HLS can synthesize RTL code. The advantage of HDL Coder is that it works for both Xilinx and Altera products, note that HLS can only work for Xilinx products. Moreover, the HDL Coder synthesizes MATLAB/Simulink models directly; on the other hand, C code needs to be generated before HLS workflow can start. In addition, the fixed-point conversion of the HDL Coder is a very useful feature for MBD. Based on the above discussion, we believe that it is better to select the HDL Coder. However, HLS is still a good option as it has wide and flexible interconnection schemes for Xilinx products, based on which we can select the most suitable interface for the hardware/software communication.

Third, for function interface design: communication overhead occupies a large proportion of execution time of hardware accelerators. Therefore, it is important to simplify data communications between function blocks so that the data exchanging between processors and hardware accelerators is not slow. Furthermore, struct data is not supported

by HDL Coder. Although HLS supports it, elements in the struct are splitted into individual data ports by HLS, therefore, the struct data should be avoided for the blocks that might be implemented in hardware accelerators.

6.3 Future Works and Directions

For the Simulink workflow, hardware implementation options can be further investigated. Like HLS, HDL Coder also offers some options to customize the hardware implementation. Designers can set different structures for the hardware logic (e.g., serial or parallel), select the way for data storage, etc. Those settings have not been investigated in this project.

As mentioned in Section 3.1, one limitation of the Simulink workflow is that only one hardware accelerator is supported by the workflow advisor. We have tried to import two hardware accelerators into an FPGA project manually. The test was successful, but a lot of manual work needs to be done. Therefore, it would be better if some automation workflow can be found to support more than one hardware accelerators.

The fixed point conversion is a useful feature of HDL Coder in the MATLAB workflow. However, it requires all the source code to be on the same folder. This restriction blocks the workflow, thus the fixed point conversion wasn't evaluated in this project.

Last but not least, other applications could be evaluated using the proposed workflow. In this project, we mainly focused on a data fusion application, but we found that some algorithms are not suitable for hardware implementation. Therefore, only one function was implemented in the hardware accelerator. It necessary to try other applications, where more suitable functions can be found and higher speedup can be achieved.

Bibliography

[1] A. Jarašūniene, G. Jakubauskas, Improvement of road safety using passive and active intelligent vehicle safety systems, Transport 22 (4) (2007) 284–289.

[2] K. J. Kingsley, Evaluating crash avoidance countermeasures using data from fmcsa/nhtsa's large truck crash causation study, in: Proceedings of the 21s International Technical Conference on the Enhanced Safety of Vehicles Conference (ESV)-International Congress Center Stuttgart, Germany, 2009.

[3] Bosch Automotive Technology, Driver assistance systems (Jul. 2014).
URL `http://www.bosch-automotivetechnology.com/en/de/driving_safety/driving_safety_systems_for_passenger_cars_1/driver_assistance_systems/driver_assistance_systems_2.html`

[4] TEXAS INSTRUMENTS, Advanced safety and driver assistance systems paves the way to autonomous driving (Jul. 2014).
URL `http://e2e.ti.com/blogs_/b/behind_the_wheel/archive/2014/02/04/advanced-safety-and-driver-assistance-systems-paves-the-way-to-autonomous-driving.aspx`

[5] T. Hettel, M. Lawley, K. Raymond, Model synchronisation: Definitions for round-trip engineering, in: Theory and Practice of Model Transformations, Springer, 2008, pp. 31–45.

[6] MathWorks, Simulink coder (Jul. 2014).
URL `http://www.mathworks.se/`

[7] D. L. Hall, J. Llinas, An introduction to multisensor data fusion, Proceedings of the IEEE 85 (1) (1997) 6–23.

[8] D. Hall, J. Llinas, A challenge for the data fusion community i: research imperatives for improved processing, in: Proc. 7th Nat. Symp. on Sensor Fusion, Vol. 16, 1994.

[9] J. Llinas, D. Hall, A challenge for the data fusion community ii: Infrastructure imperatives, in: Proc. 7th Natl. Symp. on Sensor Fusion, Vol. 16, 1994.

[10] B. Khaleghi, A. Khamis, F. O. Karray, S. N. Razavi, Multisensor data fusion: A review of the state-of-the-art, Information Fusion 14 (1) (2013) 28–44.

[11] L. A. Klein, Sensor and data fusion concepts and applications, Society of Photo-Optical Instrumentation Engineers (SPIE), 1993.

[12] F. Sandblom, J. Sörstedt, Sensor data fusion for multiple sensor configurations, in: Submited to 2014 IEEE Intelligent Vehicles Symposium.

[13] K. Mori, T. Takahashi, I. Ide, H. Murase, T. Miyahara, Y. Tamatsu, Recognition of foggy conditions by in-vehicle camera and millimeter wave radar, in: Intelligent Vehicles Symposium, 2007 IEEE, IEEE, 2007, pp. 87–92.

[14] M. Brannstrom, F. Sandblom, L. Hammarstrand, A probabilistic framework for decision-making in collision avoidance systems, Intelligent Transportation Systems, IEEE Transactions on 14 (2) (2013) 637–648.

[15] D. P. Bertsekas, The auction algorithm for assignment and other network flow problems: A tutorial, Interfaces 20 (4) (1990) 133–149.

[16] S. S. Blackman, Multiple hypothesis tracking for multiple target tracking, Aerospace and Electronic Systems Magazine, IEEE 19 (1) (2004) 5–18.

[17] J. Teich, Hardware/software codesign: The past, the present, and predicting the future, Proceedings of the IEEE 100 (Special Centennial Issue) (2012) 1411–1430.

[18] C. U. Smith, G. A. Frank, J. Cuardrado, An architecture design and assessment system for software/hardware codesign, in: Design Automation, 1985. 22nd Conference on, IEEE, 1985, pp. 417–424.

[19] R. Ernst, J. Henkel, T. Benner, Hardware-software cosynthesis for microcontrollers, Readings in hardware/software co-design (2002) 18–29.

[20] R. K. Gupta, G. De Micheli, Hardware-software cosynthesis for digital systems, Design & Test of Computers, IEEE 10 (3) (1993) 29–41.

[21] P. H. Chou, R. B. Ortega, G. Borriello, The chinook hardware/software co-synthesis system, in: Proceedings of the 8th international symposium on System synthesis, ACM, 1995, pp. 22–27.

[22] T. Blickle, J. Teich, L. Thiele, System-level synthesis using evolutionary algorithms, Design Automation for Embedded Systems 3 (1) (1998) 23–58.

[23] S. Sharma, W. Chen, Using model-based design to accelerate FPGA development for automotive applications, Tech. rep., SAE Technical Paper (2009).

[24] P. Struss, C. Price, Model-based systems in the automotive industry, AI magazine 24 (4) (2003) 17.

[25] F. Paterno, Model-based design and evaluation of interactive applications, Springer, 2000.

[26] D. F. Bacon, R. Rabbah, S. Shukla, FPGA programming for the masses, Communications of the ACM 56 (4) (2013) 56–63.

[27] H. N. Abdullah, H. A. Hadi, Design and implementation of FPGA based software defined radio using simulink hdl coder, Engineering and Technology Journal, Iraq, ISSN (2009) 1681–6900.

[28] I. n. Mhadhbi, S. B. Saoud, Model-based design approach for embedded digital controllers design, International Journal of Automation and Control 5 (3) (2011) 267–283.

[29] G. W. Johnson, LabVIEW graphical programming: practical applications in instrumentation and control, McGraw-Hill School Education Group, 1997.

[30] NATIONAL INSTRUMENTS, Getting started with LabVIEW FPGA (Jul. 2014).
URL `http://www.ni.com/tutorial/14532/en/`

[31] NATIONAL INSTRUMENTS, NI LabVIEW FPGA compilation options (Jul. 2014).
URL `http://www.ni.com/white-paper/11573/en/`

[32] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, D. P. Singh, From opencl to high-performance hardware on fpgas, in: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, IEEE, 2012, pp. 531–534.

[33] D. Singh, S. P. Engineer, Higher level programming abstractions for fpgas using opencl, in: Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing, 2011.

[34] Xilinx, Znyq-7000 all programable soc (Jul. 2014).
URL `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/`

[35] MathWorks, Block support and compatibility checks, supported block libraries (Jul. 2014).
URL `http://www.mathworks.se/help/releases/R2013b/hdlcoder/block-support-and-compatibility-checks.html`

[36] MathWorks, Check for blocks not supported by code generation (Jul. 2014).
URL `http://www.mathworks.se/help/rtw/ref/embedded-codersimulink-coder-checks.html#btpdhno-1`

[37] E. Monmasson, M. N. Cirstea, Fpga design methodology for industrial control systems—a review, Industrial Electronics, IEEE Transactions on 54 (4) (2007) 1824–1842.

[38] MathWorks, Matlab language syntax and functions for hdl code generation (Jul. 2014).
URL http://www.mathworks.se/help/releases/R2013b/hdlcoder/matlab-language-support.html

[39] MathWorks, Matlab language features, functions, classes, and system objects supported for c and c++ code generation (Jul. 2014).
URL http://www.mathworks.se/help/ecoder/language-supported-for-code-generation.html

[40] Xilinx, A guide to profiling in edk (Jul. 2014).
URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/edk_prof.pdf

[41] José Fonseca's utilitities, Convert profiling output to a dot graph (Jul. 2014).
URL https://code.google.com/p/jrfonseca/wiki/Gprof2Dot

[42] Xilinx, ug902-vivado-high-level-synthesis (Jul. 2014).
URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[43] Xilinx, Zynq-7000 all programmable soc overview (Jul. 2014).
URL http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[44] Xilinx, Zynq-7000 all programmable soc accelerator for floating-point matrix multiplication using vivado hls (Jul. 2014).
URL http://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf

[45] M. Sadri, C. Weis, N. Wehn, L. Benini, Energy and performance exploration of accelerator coherency port using xilinx zynq, in: Proceedings of the 10th FPGAworld Conference, FPGAworld '13, ACM, New York, NY, USA, 2013, pp. 5:1–5:8.
URL http://doi.acm.org/10.1145/2513683.2513688

[46] GNU, Gnu gprof documents (Jul. 2014).
URL https://sourceware.org/binutils/docs-2.20/gprof/index.html

A

Generated C Code

Below is the C Code generated by the Embedded Coder from Matlab script. Four functions from Section 4.1.1 are shown below, which are useful for explaining the profiling results in Section 4.2.3.

```
//A1=VB';
void f1(const emxArray_real_T *VB, emxArray_real_T *A1)
{
  int i1;
  int loop_ub;
  int b_loop_ub;
  int i2;
  i1 = A1->size[0] * A1->size[1];
  A1->size[0] = VB->size[1];
  A1->size[1] = VB->size[0];
  emxEnsureCapacity((emxArray__common *)A1, i1,
  (int)sizeof(double));
  loop_ub = VB->size[0];
  for (i1 = 0; i1 < loop_ub; i1++) {
    b_loop_ub = VB->size[1];
    for (i2 = 0; i2 < b_loop_ub; i2++) {
      A1->data[i2 + A1->size[0] * i1]
      = VB->data[i1 + VB->size[0] * i2];
    }
  }
}

//A2 = VB .* VB';
void f2(const emxArray_real_T *VB, emxArray_real_T *A2)
```

```
{
  int i3;
  int loop_ub;
  int b_loop_ub;
  int i4;
  i3 = A2->size[0] * A2->size[1];
  A2->size[0] = VB->size[0];
  A2->size[1] = VB->size[1];
  emxEnsureCapacity((emxArray__common *)A2, i3,
  (int)sizeof(double));
  loop_ub = VB->size[1];
  for (i3 = 0; i3 < loop_ub; i3++) {
    b_loop_ub = VB->size[0];
    for (i4 = 0; i4 < b_loop_ub; i4++) {
      A2->data[i4 + A2->size[0] * i3]
      = VB->data[i4 + VB->size[0] * i3] *
        VB->data[i3 + VB->size[0] * i4];
    }
  }
}

//A3 = VB * VB';
void f3(const emxArray_real_T *VB, emxArray_real_T *A3)
{
  emxArray_real_T *b;
  int i5;
  int loop_ub;
  int b_loop_ub;
  int i6;
  int c_loop_ub;
  int i7;
  unsigned int unnamed_idx_0;
  unsigned int unnamed_idx_1;
  emxInit_real_T(&b, 2);
  i5 = b->size[0] * b->size[1];
  b->size[0] = VB->size[1];
  b->size[1] = VB->size[0];
  emxEnsureCapacity((emxArray__common *)b, i5,
  (int)sizeof(double));
  loop_ub = VB->size[0];
  for (i5 = 0; i5 < loop_ub; i5++) {
    b_loop_ub = VB->size[1];
    for (i6 = 0; i6 < b_loop_ub; i6++) {
      b->data[i6 + b->size[0] * i5]
```

```
      = VB->data[i5 + VB->size[0] * i6];
    }
  }

  if ((VB->size[1] == 1) || (b->size[0] == 1)) {
    i5 = A3->size[0] * A3->size[1];
    A3->size[0] = VB->size[0];
    A3->size[1] = b->size[1];
    emxEnsureCapacity((emxArray__common *)A3, i5,
    (int)sizeof(double));
    loop_ub = VB->size[0];
    for (i5 = 0; i5 < loop_ub; i5++) {
      b_loop_ub = b->size[1];
      for (i6 = 0; i6 < b_loop_ub; i6++) {
        A3->data[i5 + A3->size[0] * i6] = 0.0;
        c_loop_ub = VB->size[1];
        for (i7 = 0; i7 < c_loop_ub; i7++) {
          A3->data[i5 + A3->size[0] * i6]
          += VB->data[i5 + VB->size[0] * i7] *
            b->data[i7 + b->size[0] * i6];
        }
      }
    }
  } else {
    unnamed_idx_0 = (unsigned int)VB->size[0];
    unnamed_idx_1 = (unsigned int)b->size[1];
    i5 = A3->size[0] * A3->size[1];
    A3->size[0] = (int)unnamed_idx_0;
    emxEnsureCapacity((emxArray__common *)A3, i5,
    (int)sizeof(double));
    i5 = A3->size[0] * A3->size[1];
    A3->size[1] = (int)unnamed_idx_1;
    emxEnsureCapacity((emxArray__common *)A3, i5,
    (int)sizeof(double));
    loop_ub = (int)unnamed_idx_0 * (int)unnamed_idx_1;
    for (i5 = 0; i5 < loop_ub; i5++) {
      A3->data[i5] = 0.0;
    }

    eml_xgemm(VB->size[0], b->size[1], VB->size[1],
    VB, VB->size[0], b, VB->
              size[1], A3, VB->size[0]);
  }
```

```
  emxFree_real_T(&b);
}

//A4 = VB .* VC';
void f4(const emxArray_real_T *VB,
const emxArray_real_T *VC, emxArray_real_T *A4)
{
  int i8;
  int loop_ub;
  int b_loop_ub;
  int i9;
  i8 = A4->size[0] * A4->size[1];
  A4->size[0] = VB->size[0];
  A4->size[1] = VB->size[1];
  emxEnsureCapacity((emxArray__common *)A4, i8,
  (int)sizeof(double));
  loop_ub = VB->size[1];
  for (i8 = 0; i8 < loop_ub; i8++) {
    b_loop_ub = VB->size[0];
    for (i9 = 0; i9 < b_loop_ub; i9++) {
      A4->data[i9 + A4->size[0] * i8]
      = VB->data[i9 + VB->size[0] * i8] *
        VC->data[i8 + VC->size[0] * i9];
    }
  }
}
```

B

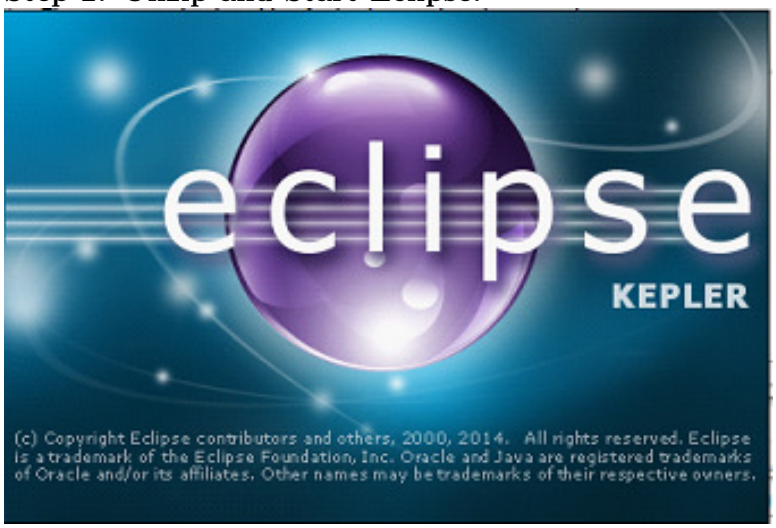
User Guide for Host Profiling

In this thesis, three methods of profiling are proposed. MATLAB profiling and target profiling are two ways that developers can easily find tutorials on the internet. In this section, we present the way to perform host profiling that is a method we summarized in this project. The steps to setup the profiling are the following:

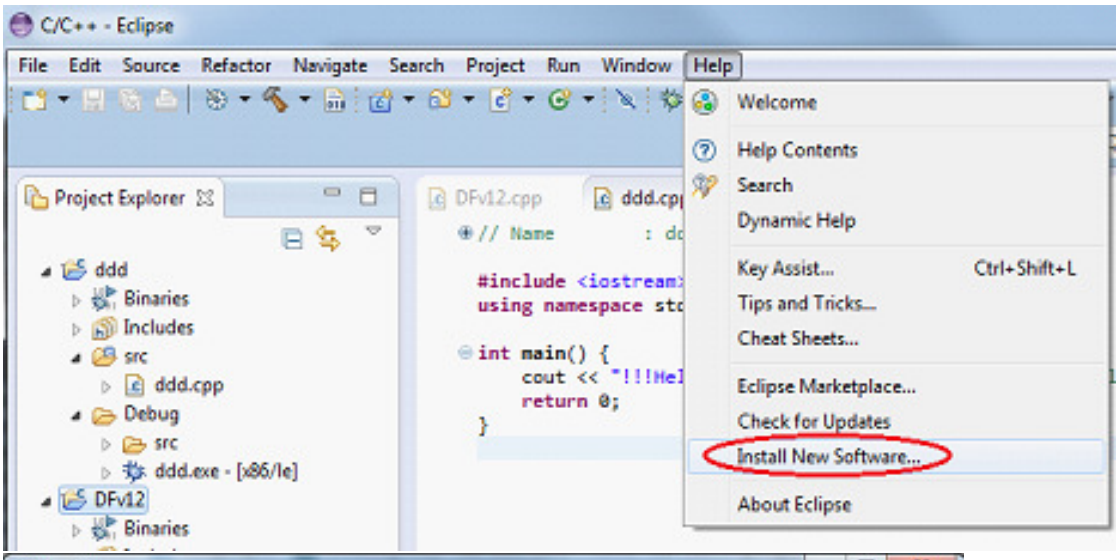
Step 1: Download Eclipse.

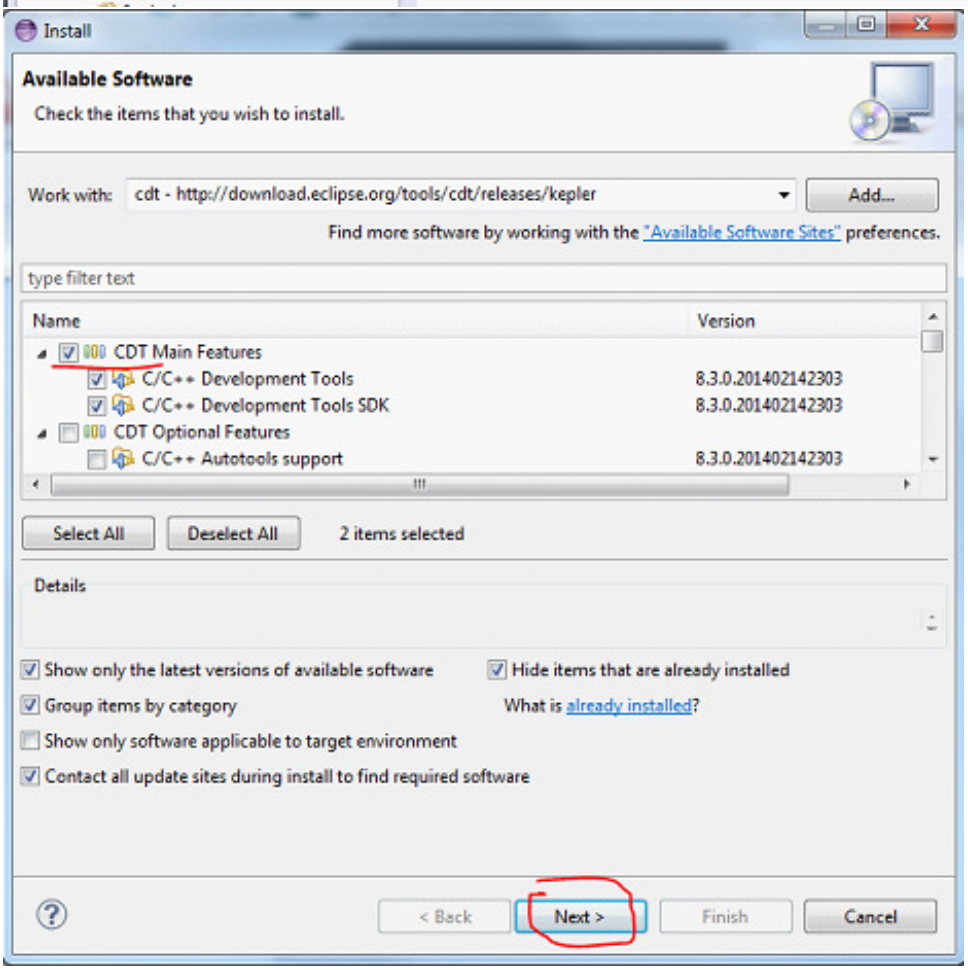
Eclipse Standard 4.3.2

Step 2: Unzip and Start Eclipse.

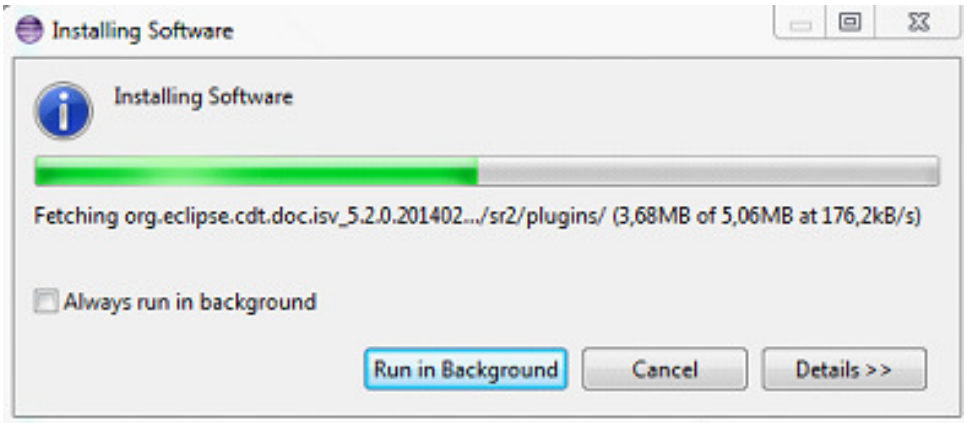


Step 3: Install new software (CDT C Develop Tool).

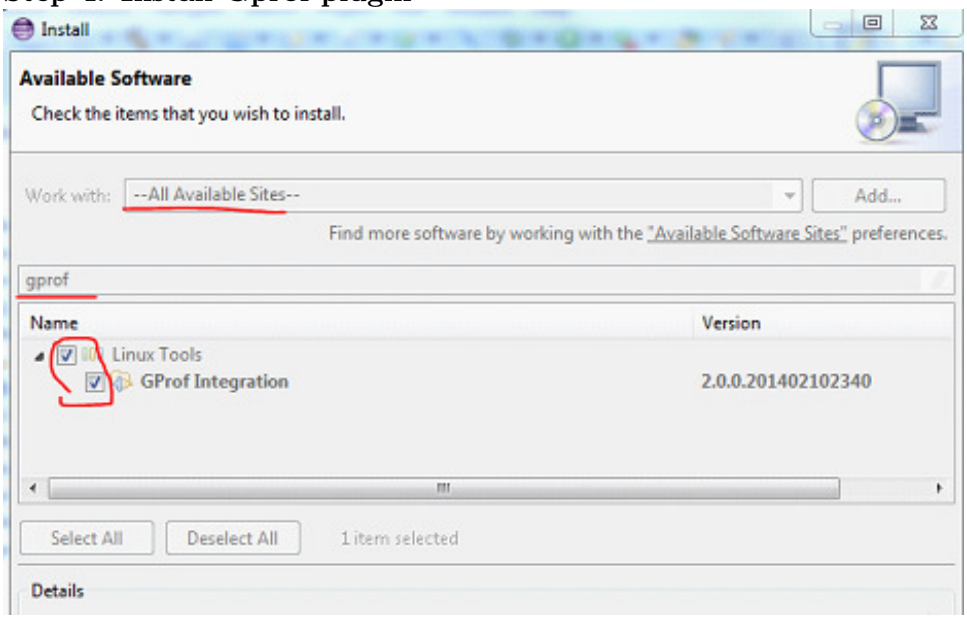




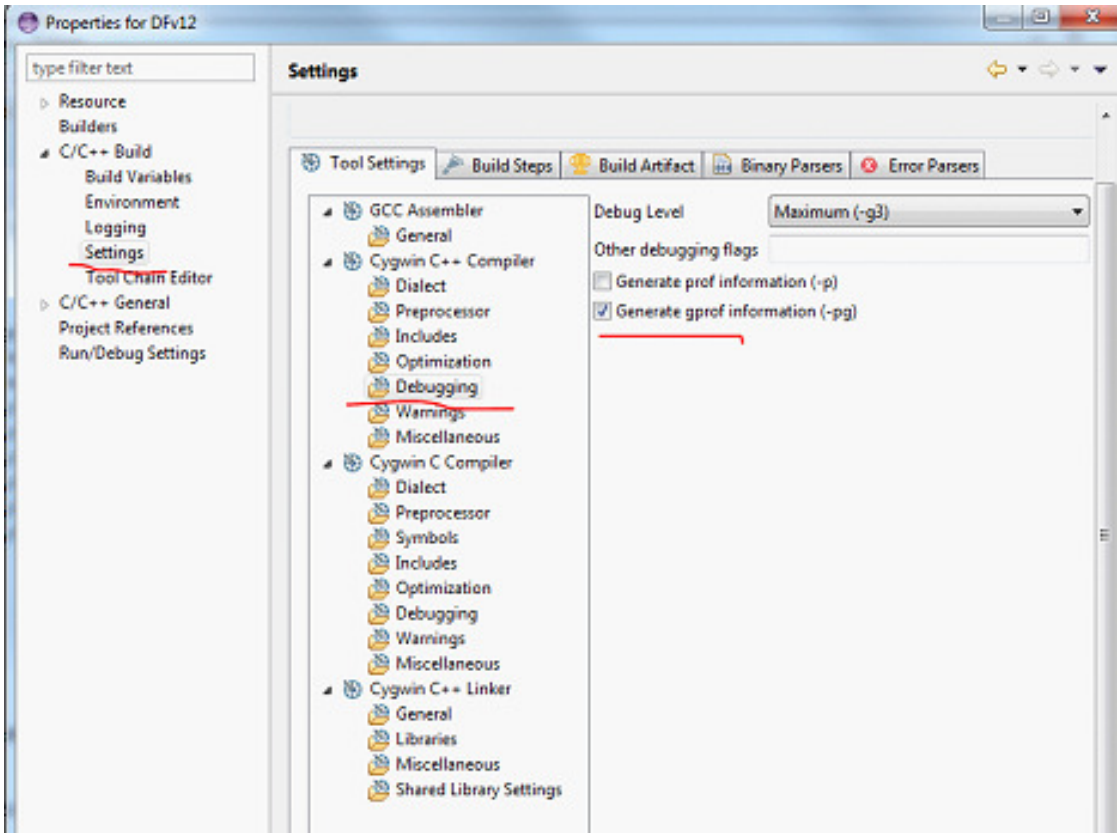
Wait for the installation to be finished.



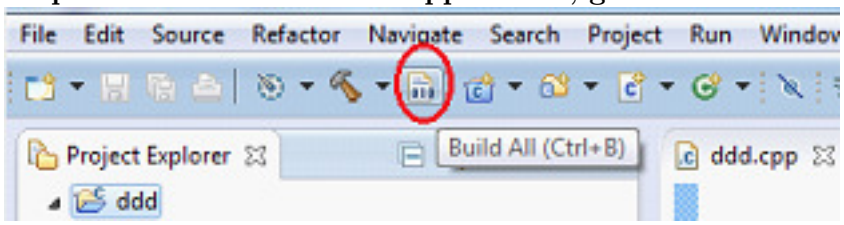
Step 4: Install Gprof plugin

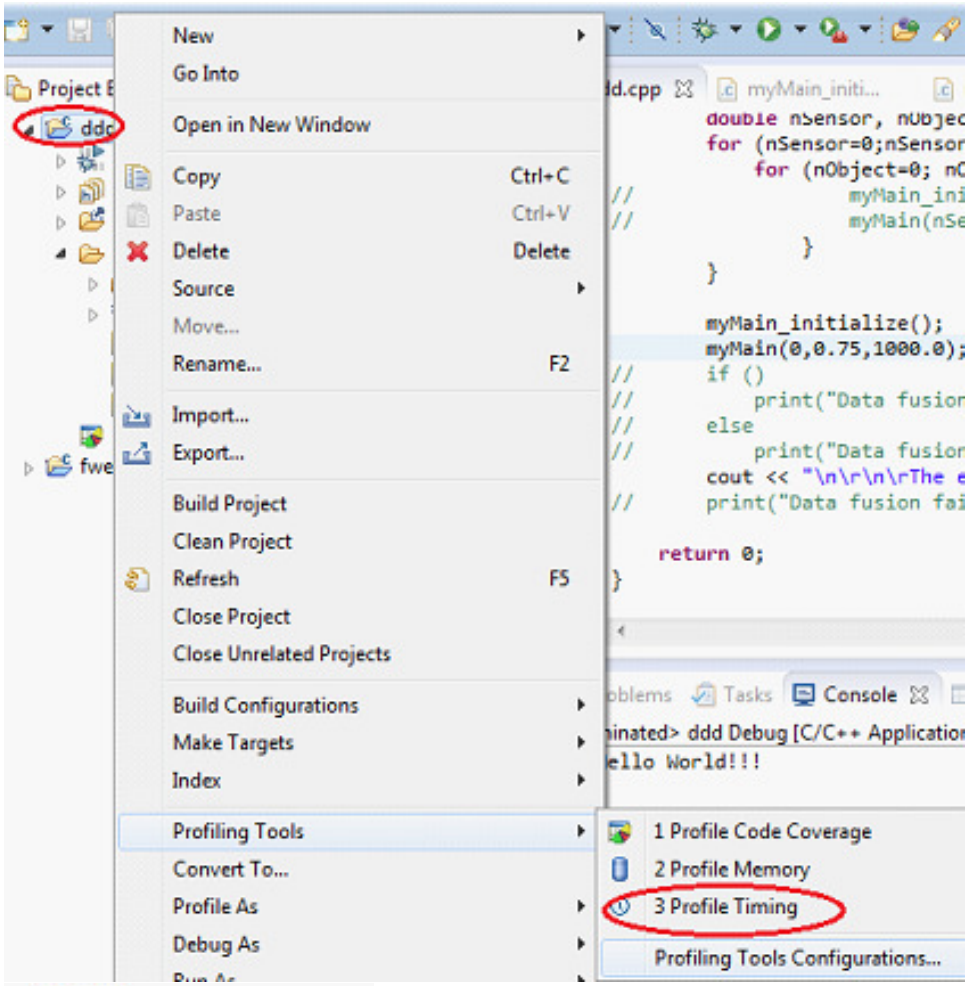


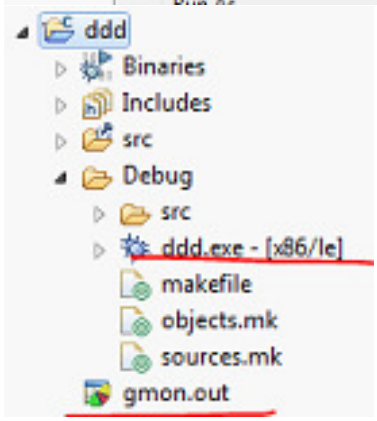
Step 5: Create a C/C++ project, and set debug flags:



Step 6: Build and Run the application, get .exe and .out files.





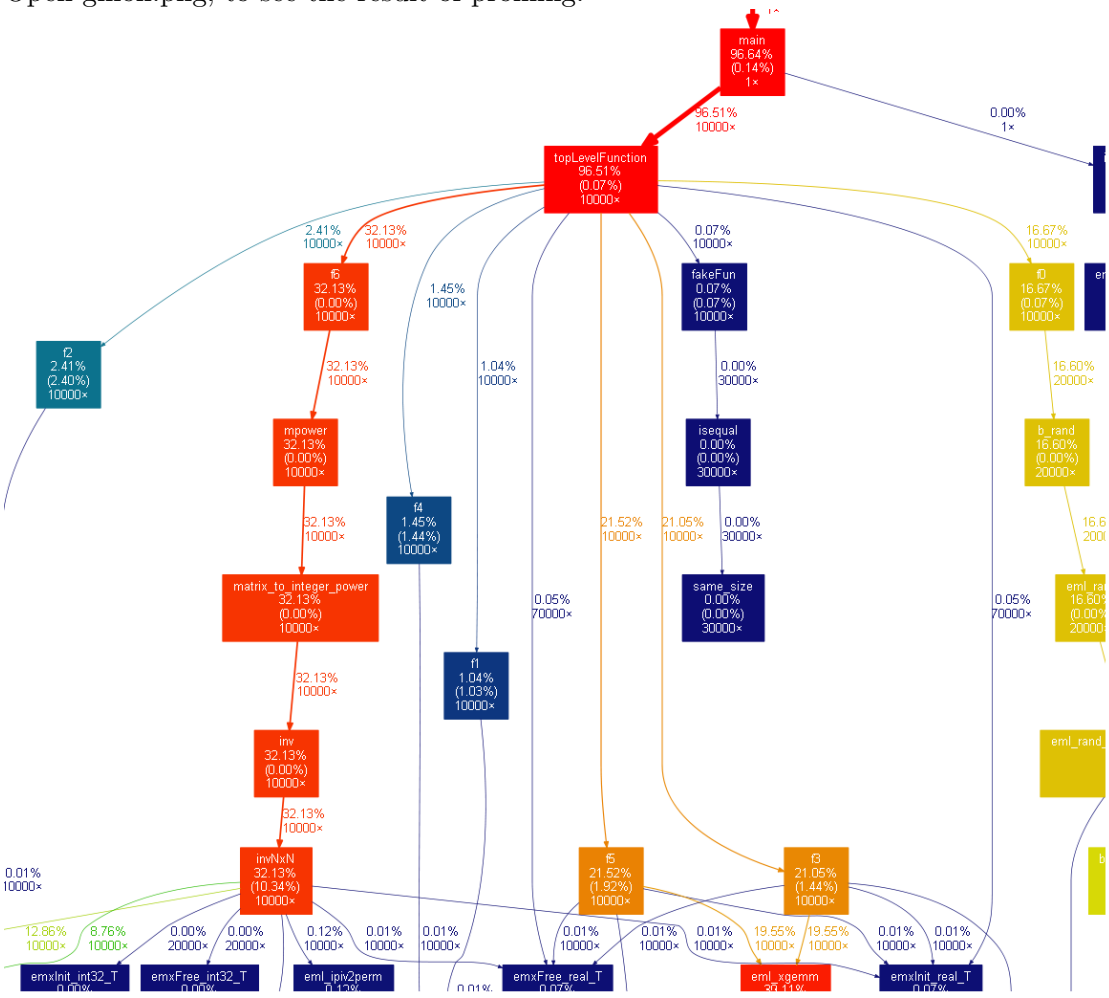


Step 7: Copy .exe and .out into a folder together with gprof2dot.py.
Step 8: Execute the following commands.
:: Process Profile daa recalled from target
gprof ddd.exe gmon.out > gmon.txt
:: Transform into a Dot format

python gprof2dot.py -e 0.1 -n 0.1 –skew 0.0001 -o gmon.gv gmon.txt
:: Visualise
dot -q -T png -o gmon.png gmon.gv

Step 9: Open results.

Open gmon.png, to see the result of profiling.



The detailed results are in gmon.txt.

More information about how to understand the txt file can be found in [46].