Efficient System-Level Design Space Exploration for Large-Scale Embedded Systems

Yuki Ando

Abstract

This dissertation presents system-level design space exploration method for large-scale embedded systems.

Embedded systems have been increasing their complexities, and they are getting more and more functionalities. Since the functionalities of embedded systems have increased, embedded systems need more computational power and communications capacity. Thus, several processing elements (PEs) such as dedicated hardware and multi-core processors have been used in embedded systems.

Embedded systems are generally required to satisfy strict requirements of the system performances such as execution time, chip area, and power consumption since they are used in limited environments. During the design of embedded systems using several PEs, designers must determine a mapping which indicates the allocation of functions to these PEs. Since the system performances depend on the mapping, it is important for designers to find appropriate mappings efficiently from the large design space.

In order to design complex embedded systems, system-level design has been proposed. The key points of system-level design are to design a system at a high level of abstraction and to explore the design space. Designers first describe the system in particular model at a high level of abstraction. Then, the model is converted to a simulation description and a target implementation in order to evaluate the system performances. If the evaluation results of the system performances do not satisfy the requirements, the designers modify the model, and they again evaluate the system performances. Designers iterate modification of the model and evaluation of the system performances in order to explore the design space. The exploration of design space continues until the system performances can satisfy the requirements.

System-level design tools have been developed to realize system-level design. One of the tools is SystemBuilder. SystemBuilder automatically generates a target implementation from the model according to a mapping decided by designers. The generated implementation can be executed on both simulation tools and Field-Programmable Gate Array. Since SystemBuilder automatically synthesizes the target implementation and the communication interfaces among software and dedicated hardware, it is easy for designers to evaluate system performances of several mappings.

The author first evaluated the design efficiency of SystemBuilder throughout a case study of AES encryption system design. It is clarified that SystemBuilder is effective for designing pipelined systems. In addition, the author unveiled three problems to design systems efficiently, which are 1) limited mapping, 2) long evaluation time, 3) lack of support to improve the system performances. The first problem is limited mapping. Most of system-level design tools can allocate a function to either a software or a hardware module. The tools, however, cannot support different mappings such as several functions share a hardware module. The second problem is long evaluation time. Even though SystemBuilder automatically generates the implementations, it takes very long time for the generation of the implementations and the evaluation of system performances. Thus, with only SystemBuilder, it is hard to evaluate the system performances of a lot of mappings in order to find appropriate mappings. The third problem is lack of support to improve the system performances. SystemBuilder clarifies a bottleneck of execution time by profiling tools. However, it is hard to consider how to improve the system performances of current complex embedded systems with only profiling tools. These are problems of not only Sys-

temBuilder but also the other system-level design tools. Therefore, it is important to solve the problems in order to realize more efficient system-level design.

The author proposes three tools which can overcome the problems above. Three tools are Extended SystemBuilder, Mapping Explorer, and Improvement Analyzer. Extended SystemBuilder is an extension of SystemBuilder, which can generate a hardware module which is shared by several functions. Since Extended SystemBuilder increases the type of mapping, it solves the first problem. Mapping Explorer uses an efficient algorithm named pareto-update search. Since pareto-update search drastically decreases the number of exploration of mappings to find appropriate mappings. Thus, Mapping Explorer can decrease the time to evaluate the system performances, which is the second problem. Even if Mapping Explorer cannot find an appropriate mapping, Improvement Analyzer helps the designer to identify bottlenecks. In addition, Improvement Analyzer lists several candidates of the ways to improve the system performances so that the designer can consider how to modify the model. Improvement Analyzer, therefore, can overcome the third problem.

Throughout the three tools, exploration of mappings and bottleneck analysis are done automatically to accelerate the design space exploration at system-level. Therefore, system designers can efficiently design large-scale embedded systems with multi-core processors and dedicated hardware.

This dissertation describes the detail of three tools and evaluates them using case studies. In the case studies, efficient design space exploration is demonstrated.

CONTENTS

ABSTRACT i

1 INTRODUCTION 1

1.1 Backgrounds 1

1.2 Proposed Method 5

1.3 Outlines of the Dissertation 9

2 SYSTEM-LEVEL DESIGN AND THE BASE TOOL 11

2.1 General Design Flow of Embedded Systems 11

2.2 System-Level Design 15

2.3 Design Space Exploration in System-Level Design 20

2.4 Related Works 23

2.5 SystemBuilder 24

2.5.1 Input description 25

2.5.2 Automatic synthesis of target implementation 26

2.5.3 Performance Analysis 28

2.6 Overview of Proposed Method 28

3 A CASE STUDY OF AES ENCRYPTION SYSTEM WITH SYSTEMBUILDER 33

3.1 Introduction 33

3.2 AES Encryption System Design 35

3.2.1 Usage of SystemBuilder 35

3.2.2 Single Process Software Design 36

3.2.3 Five Processes Hardware Design 36

3.2.4 Pipelined Design 38

3.2.5 Operation Refined Design 40

3.2.6 Performance of Design 41

3.2.7 Design Time of AES Encryption System 41

3.3 Evaluation of SystemBuilder 43

3.4 Conclusions 44

4 Extended SystemBuilder: Communication Synthesis for Hardware Sharing 45

4.1 Introduction 45

4.2 Communication Synthesis for Hardware Sharing 46

4.2.1 The Design Flow with Hardware Sharing 46

4.2.2 Implementation of Communication for Hardware Sharing 48

4.2.3 Mapping of Processes with Hardware Sharing 50

4.3 A Case Study 51

4.3.1 Target Systems 51

4.3.2 Design Space of Hardward Sharing 53

4.3.3 The Reduction of Hardware Area 58

4.3.4 The Execution Time with Polling Policy 58

4.4 Conclusion 61

5 Mapping Explorer: A Fast Design Space Exploration by Pareto-Update Search 63

5.1 Introduction 63

5.2 Related Works 67

5.3 Definition of a Pareto Solution 69
5.4 Pareto-Update Search 69
5.4.1 Assumptions and Singular Mappings 70
5.4.2 Generation of New Mappings 71
5.4.3 Overview of the Pareto-Update Search 73
5.4.4 The Algorithm of Pareto-Update Search 77
5.5 A Case Study on MPEG-4 Decoder Application 78
5.5.1 Target Systems and Evaluation of Pareto Solutions 78
5.5.2 Exploration Results 80
5.6 Conclusion 83

6 Improvement Analyzer: Performance Improvement with Bottleneck Analysis 85
6.1 Introduction 85
6.2 Overview of the Design Flow 88
6.3 Exploration of Improvement Rate of the Bottleneck 89
6.3.1 Definition of Bottleneck Process 89
6.3.2 Definition of Improvement Rate 91
6.3.3 Exploration of the Improvement Rates on Bottleneck Processes . . 91
6.3.4 Detail of Cost Function 94
6.4 A Case Study 97
6.4.1 Calibration of Coefficients in the Cost Function 97
6.4.2 Initial Design of AES Encryption and Decryption 97
6.4.3 Improvement of System-Level Description 99
6.4.4 Improvement of System-Level Description with Different Requirements 102
6.4.5 Design Efficiency 104

6.5 Conclusion 105

7 Conclusions 107

7.1 Summary of Contributions 107

7.2 Future Directions 109

Acknowledgment 111

Bibliography 112

List of Publications by the Author 121

List of Figures

1.1 Relationships of three tools. 6

1.2 Overview of proposed design flow. 7

2.1 General design flow of embedded systems. 12

2.2 Overview of system-level design flow. 14

2.3 An example of system specification model. 16

2.4 An example of a software/hardware partitioning. 18

2.5 An example of communication model. 19

2.6 A typical flow of design space exploration in system-level design. 20

2.7 Result A: design space exploration for a single processor with dedicated hardware. 21

2.8 Result B: design space exploration for dual processors with dedicated hardware. 21

2.9 An example of trade-off relation between execution time and hardware area. 22

2.10 Overview of SystemBuilder. 25

2.11 An example of process profiler waveforms. 28

2.12 Entire design flow of the proposed method. 29

3.1 System partitioning and processes. 37

3.2 Waveform of the five processes hardware design. 38

3.3 Waveform of the pipelined design. 39
3.4 Waveform of the operation refined design. 40
3.5 Design time and summary of work on the AES encryption system. 42

4.1 Design flow of Extended SystemBuilder with/without hardware sharing. . . 47
4.2 Detail of the wrapper generated by Extended SystemBuilder. 49
4.3 Mappings of hardware sharing supported by Extended SystemBuilder. . . . 50
4.4 Construction of Quad-AES. 52
4.5 Trade-offs between performance and hardware area: Dual-AES. 56
4.6 Trade-offs between performance and hardware area: Triple-AES. 56
4.7 Trade-offs between performance and hardware area: Quad-AES. 57
4.8 Execution time of AES applications: Dual-AES. 59
4.9 Execution time of AES applications: Triple-AES. 59
4.10 Execution time of AES applications: Quad-AES. 60

5.1 Simulation-based design space exploration methods: exhaustive search. . . 64
5.2 Simulation-based design space exploration methods: pareto-update search. . 65
5.3 Example of exploration from ALLSW. 74
5.4 The pareto-update algorithm. 77
5.5 Processes in MPEG-4 decoder application. 78
5.6 Calculation on the area of a pareto solution. 80
5.7 Pareto solutions of MPEG-4 on Quad-core-HW architecture. 81

6.1 Entire design flow. 86
6.2 An example of IRs and bottleneck processes. 90
6.3 Exploration flow of Improvement Analyzer to identify IRs of bottleneck processes. 92
6.4 The structure of AES encryption and decryption application. 98

List of Tables

3.1 Performance improvement of the AES encryption system. 41

4.1 Mappings and the polling policies of Dual-AES. 53

4.2 Mappings and the polling policies of Triple-AES. 54

4.3 Mappings and the polling policies of Quad-AES. 55

5.1 An example set of processes with their execution time (Exe) and hardware area. 73

5.2 Execution time and hardware area of the entire system for the example set of processes. 75

5.3 Number of ALUTs. 79

5.4 Exploration results on MPEG-4 decoder application. 81

6.1 Summary of improvement on AES application. (Exe. and Area indicate execution time and hardware area, respectively) 98

6.2 A list of IRs for Initial design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 18,000) 100

6.3 A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 14,000) 101

6.4 A list of IRs for Initial design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 17,000) 102
6.5 A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 15,000) 103
6.6 A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 13,000) 103
6.7 The quantity of work spent by designers. 104

Chapter 1

Introduction

1.1 Backgrounds

Embedded systems have been widely used around the world in order to improve our lifestyles. An embedded system is a computer system which realizes dedicated functions within a large system. An example of embedded systems is cell phone which is an essential communication tool for our lifestyles. Cell phones are computer systems which realize dedicated functions such as talk and message within a large telecommunication system.

Embedded systems have been increasing their complexities. For example, cell phones used to have functions of talk and message. Recently they have functions of camera, movie, and even Internet browsing. Along with the increase of the complexities of embedded systems, the time to design embedded systems also tends to delay. This causes longer time-to-market. Time-to-market is the time between the planning of a product and the release of it. In general, the early released products have more chances in the market. Therefore, the system designers should shorten design time in order to achieve a shorter time-to-market.

Embedded systems are generally required to satisfy strict requirements of system performances such as execution time, chip area (hardware area), and power consumption since they are used in limited environments. For example, cell phones are required to work under low power and to be run fast for longer battery life and user-friendliness. In addition, embedded systems consist of software and hardware. In order to realize the requirements of system performances, both software and hardware should be optimized. System designers are facing a problem to design systems efficiently under strict requirements of system performances.

Presently embedded systems are getting more and more functionalities in order to realize complex systems. Since the functionalities of embedded systems have increased, embedded systems need more computational power and communications capacity. For that reason, several processing elements (PEs, hereafter) such as dedicated hardware and multi-core processors have been used in embedded systems. During the design of embedded systems using several PEs, designers must determine a mapping that indicates the allocation of functions to these PEs. Since the system performances depend on the mapping, it is important for designers to find appropriate mappings from the large design space. Therefore, it is important for designers to efficiently iterate the evaluation of system performances of different mappings.

In order to design complex embedded systems, system-level design has been proposed. The key points of system-level design are to design a system at a high level of abstraction and to explore the design space. Designers first describe the system in particular model at a high level of abstraction. Then, the model is converted to a simulation description and a target implementation in order to evaluate the system performances. If the evaluation results of the system performances do not satisfy the requirements, the designers modify the model, and they again evaluate the system performances. Designers iterate modification of the model and evaluation of the system performances in order to explore the design space. The exploration of design space continues until the system performances can satisfy the requirements.

System-level design tools have been developed in order to realize system-level design [1]. One of the tools is SystemBuilder [2]. SystemBuilder takes a system-level description, a target architectures template, and a mapping as inputs, and automatically generates simulation descriptions and target implementations including software, hardware, and interfaces among them. The generated simulation descriptions and the generated target implementations can be executed on simulation tools and Field-Programmable Gate Array (FPGA, hereafter), respectively. Since SystemBuilder automatically synthesizes the target implementations and interfaces from the system-level description, it is easy for designers to evaluate the system performances of several mappings. In addition, SystemBuilder cooperate with profiling tools so that designers can easily analyze bottlenecks of the systems [3].

Even though SystemBuilder makes evaluation of system performances easy by automatic synthesis of the target implementations, SystemBuilder still has following problems:

- Limited mapping
- Long evaluation time of a lot of mappings
- Lack of support to improve the system performances

First problem is limited mapping. A function in the system-level description can be allocated to a software or a hardware module. SystemBuilder, however, cannot support different mappings such as several functions share a hardware module. There is a possibility that the system performances are improved by the other mappings. Thus, SystemBuilder should support the other mappings to realize better system performances. Second problem is long evaluation time. Even though SystemBuilder automatically generates the implementations, it takes very long time for the generation of the implementations and the evaluation of system performances. With only SystemBuilder, it is hard to evaluate the system performances of a lot of mappings in order to find appropriate mappings. Thus, a strategy is needed to support the exploration of mappings. The third problem is lack of support to improve the system performances. SystemBuilder clarifies a bottleneck of execution time by profiling tools. However, it is hard for designers to consider how to improve the system performances of current complex embedded systems with only profiling tools. Designers need another support to improve system performances in order to design systems efficiently.

Not only SystemBuilder but also the other system-level design tools have been developed in order to realize efficient system-level design. SCE [4], Artemis [5], PeaCE [6], Metropolis [7], and ARTS [8] are system-level design tools which support modeling and evaluation of the system performances. These tools take their own system-level model and automatically generate simulation descriptions. These tools, however, only support modeling and automatic synthesis of simulation descriptions. On the other hand, SystemBuilder automatically generates both the simulation descriptions and the target implementations on FPGAs. SystemBuilder can evaluate the system performances accurately by the target implementations on FPGAs. This is an advantage of SystemBuilder against the other tools. In addition, the other tools do not support complex simulation descriptions which has a shared hardware module. Thus, as with SystemBuilder, these system-level design tools also have three problems above.

There are different tools which support system-level design. System-level performance estimation tools [9, 10, 11] have been developed in order to overcome the second problem. As performance estimation tools use profiles recorded by system-level design tools and FPGAs, the simulation accuracy is sufficiently high, and the evaluation time is short. Thus, the performance estimation tools can reduce the evaluation time of a mapping. However, despite fast simulation speed, the total evaluation time becomes longer if the tools evaluate the system performances of a lot of mappings in order to find appropriate mappings. Therefore, an efficient method to explore appropriate mappings is necessary even though the performance estimation tools can reduce the evaluation time of a mapping.

Profiling and analyzing tools [12, 13, 14] also have been developed in order to support system-level design. As with the profiling tools of SystemBuilder, these tools can trace the behaviors of the system and visualize the traces. With the tools, it is easy for designers to identify the bottlenecks of the system. Designers, however, must analyze how to improve the bottlenecks in order to get better system performances. Thus, another tool that efficiently assists designers to improve the system performances is required.

As mentioned above, not only SystemBuilder but also the other current system-level design tools have three problems above. Therefore, it is necessary to solve the problems in order to realize efficient system-level design.

1.2 Proposed Method

The main objective of proposed method is to realize the efficient design of embedded systems that satisfy the requirements. Proposed method consists of three tools below:

- Extended SystemBuilder
- Mapping Explorer
- Improvement Analyzer

Figure 1.1 shows relationships of three tools. The gray colored parts are works of the author. Extended SystemBuilder is an extension of SystemBuilder. Extended SystemBuilder has a regular synthesis flow provided by SystemBuilder as shown 1-1 in Figure 1.1. Extended SystemBuilder also has an extension synthesis flow to synthesize communication for shared hardware modules as shown 1-2 in Figure 1.1. If the mapping indicates shared hardware, the extension synthesis flow takes place after the regular synthesis flow. Mapping Explorer is an exploration tool of mappings. It can finds trade-off between execution time and hardware area efficiently. Improvement Analyzer is an analysis tool of the system

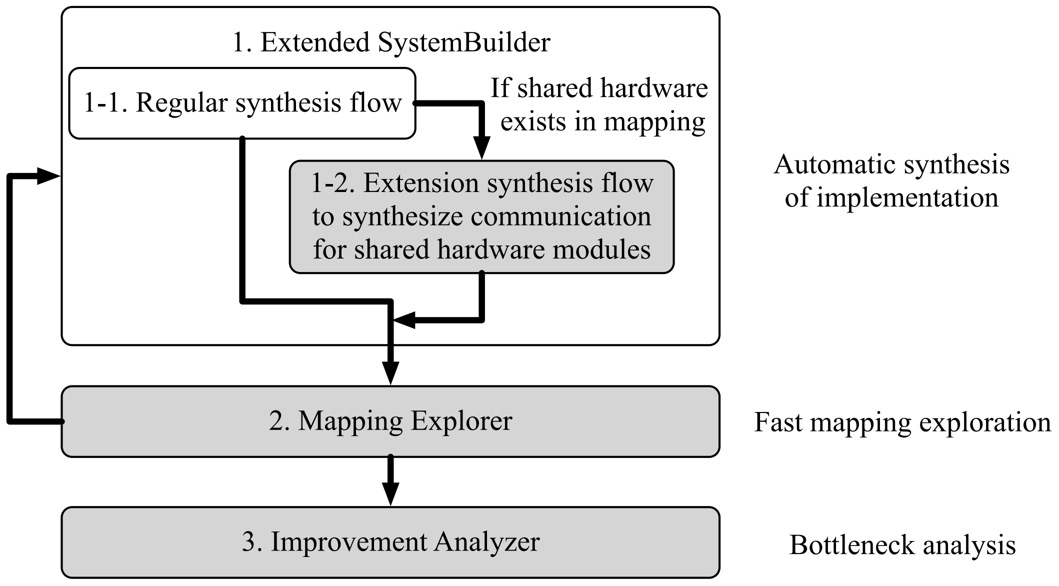

1. Extended SystemBuilder
1-1. Regular synthesis flow
If shared hardware exists in mapping
1-2. Extension synthesis flow to synthesize communication for shared hardware modules
Automatic synthesis of implementation
2. Mapping Explorer
Fast mapping exploration
3. Improvement Analyzer
Bottleneck analysis

Figure 1.1: Relationships of three tools.

performances. It automatically identifies bottlenecks of the system. In addition, Improvement Analyzer lists several candidates of the ways to improve the system performances so that the designer can consider how to modify the system-level description.

Figure 1.2 shows the overview of proposed design flow. Extended SystemBuilder takes a system-level description, a target architecture template, and mappings as inputs, and it automatically generates target implementations including software, hardware, and interfaces among them. The system-level description consists of a set of applications, each of which in turn consists of a set of processes and channels. Processes represent the functions of the system and channels represent the communication among the processes. Also, Extended SystemBuilder provides communication APIs to partition the system. Designers easily partition the system into processes because both communication APIs and processes are written in C language. The generated implementations can be executed on both simulation tools and FPGAs to evaluate the system performances. Mapping Explorer uses the evaluation results of the system performances.

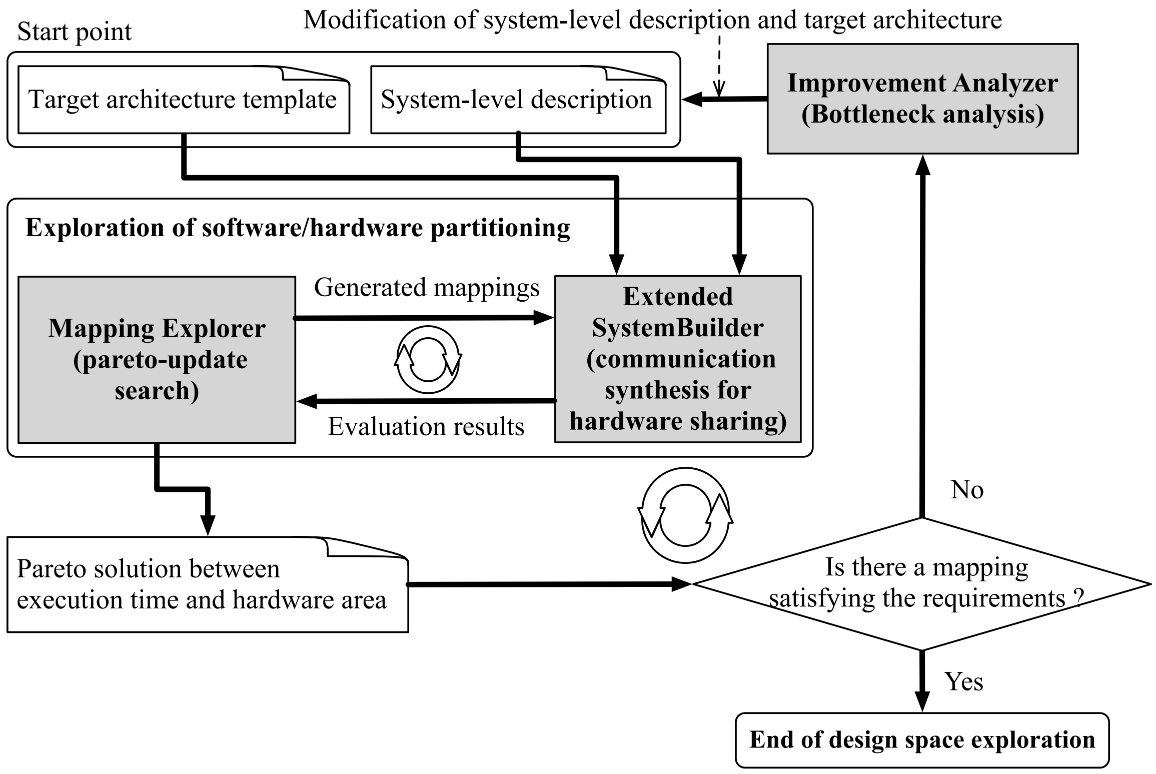

Start point
Modification of system-level description and target architecture
Target architecture template
System-level description
Improvement Analyzer (Bottleneck analysis)
Exploration of software/hardware partitioning
Mapping Explorer (pareto-update search)
Generated mappings
Evaluation results
Extended SystemBuilder (communication synthesis for hardware sharing)
No
Pareto solution between execution time and hardware area
Is there a mapping satisfying the requirements ?
Yes
End of design space exploration

Figure 1.2: Overview of proposed design flow.

As the numbers of PEs and functions increase, the number of possible mappings increases exponentially. Mapping Explorer can accelerates design space exploration to find appropriate mappings from an enormous number of possible mappings. This tool uses an efficient algorithm named pareto-update search. In pareto-update search, the concept of pareto solution is used to explore appropriate mappings [15]. In addition, pareto-update search focuses on the trade-off relationship between execution time and hardware area. By using this relationship, pareto-update search can reduce the number of exploration of mappings compared to exhaustive search.

If Mapping Explorer finds appropriate mappings that satisfy the requirements, it is the end of the design space exploration. However, Mapping Explorer may not find a mapping that satisfies the requirements of the system performances. For example, there may be a case that a mapping only satisfies the requirement of the execution time. In fact, design space exploration of mappings does not always find a mapping that satisfies the all requirements. Then, designers must improve the system-level description.

Improvement Analyzer is a tool that assists the designers to improve the system-level description. It automatically identifies bottlenecks of the system, and it lists several candidates of the ways to improve the system performances. Designer, then, modify the system-level description with the list of candidates of the ways to improve the system performances.

This design flow is iterated until the designers find appropriate mappings. With three tools above, most of design flow are automated. Therefore, system designers can efficiently design large-scale embedded systems with multi-core processors and dedicated hardware.

This dissertation describes the detail of three tools and evaluates them using case studies. The case studies demonstrate efficient design space exploration and easiness of design analysis using the proposed method.

1.3 Outlines of the Dissertation

The organization of this dissertation is as follows. First, Chapter 2 summarizes the system-level design methodologies, related works, and a base tool named SystemBuilder. Then, Chapter 3 shows a case study of Advanced Encryption Standard (AES, hereafter) encryption system in order to evaluate the basic design efficiency of SystemBuilder. Chapter 4 describes the detail of Extended SystemBuilder that can synthesize communication for shared hardware modules. Chapter 5 presents Mapping Explorer and pareto-update search which accelerate design space exploration to find appropriate mappings. Chapter 6 proposes Improvement Analyzer that analyzes bottlenecks of the system and supports modification of the system-level description. Finally, Chapter 7 concludes this dissertation with a summary.

Chapter 2

System-Level Design and the Base Tool

This chapter explains the backgrounds of system-level design, related works, an overview of proposed method, and a base tool.

At first, section 2.1 represents the general design flow of embedded systems. Then section 2.2 shows the details of system-level design before describing details of design space exploration at system-level in section 2.3. Section 2.4 briefly shows related works and section 2.5 introduces a base tool named SystemBuilder. Finally, section 2.6 represents the overview of the proposed method.

2.1 General Design Flow of Embedded Systems

Figure 2.1 shows the general top-down design flow of embedded systems. The general design flow starts from a step of system design at a high level of abstraction at which software and hardware cannot be distinguished. Then, the flow goes to software design step and hardware design step separately as shown at the middle of the figure since embedded systems consist of software and hardware. One side corresponds to the software design step, whereas the other side corresponds to the hardware design step. At last, software and

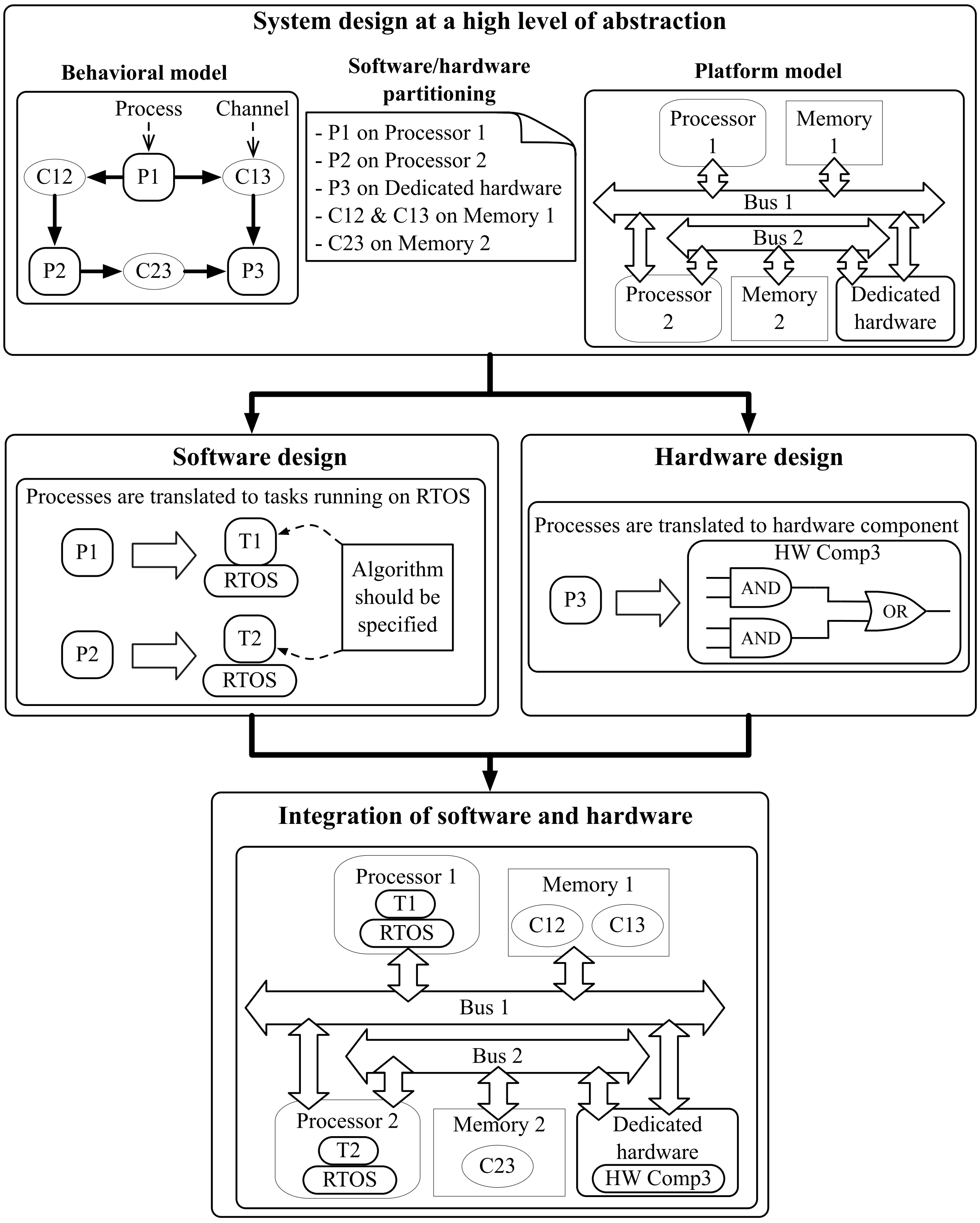


Figure 2.1: General design flow of embedded systems.

hardware are integrated to make a whole system as shown at the bottom of the figure.

In the step of system design at a high level of abstraction, system designers describe the systems by behavioral models that consist of processes communicating via channels. An example of behavioral models is Kahn Process Network [16]. The platform model is a typical structural model. The platform model consists of architectural components such as processors, busses, memories, and dedicated hardware. In the step of system design at a high level of abstraction, system designers must select an appropriate platform model, and they determine a software/hardware partitioning which indicates the allocation of processes and channels onto the selected platform model. The result is a refined model which contains all decisions for software design and hardware design such as throughput, latency, or hardware area. These refined models are then used as input to software design step and hardware design step.

In the software design step, processes mapped to software are translated into the tasks run on Real-Time Operating System (RTOS, hereafter) or custom runtime environments. At this step, the algorithm of the tasks should be specified. Then the tasks are synthesized as instructions of the selected processor. This synthesis process is typically performed by a compiler and linker tool chain for the selected processor and RTOS.

In the hardware design step, processes which are selected to be realized as hardware are converted to hardware components. Then, system designers design the logics of the hardware components. The logics are synthesized and implemented as Flip-Flops and logic gates.

Finally, instructions of software, and implementations of hardware are integrated in the last step.

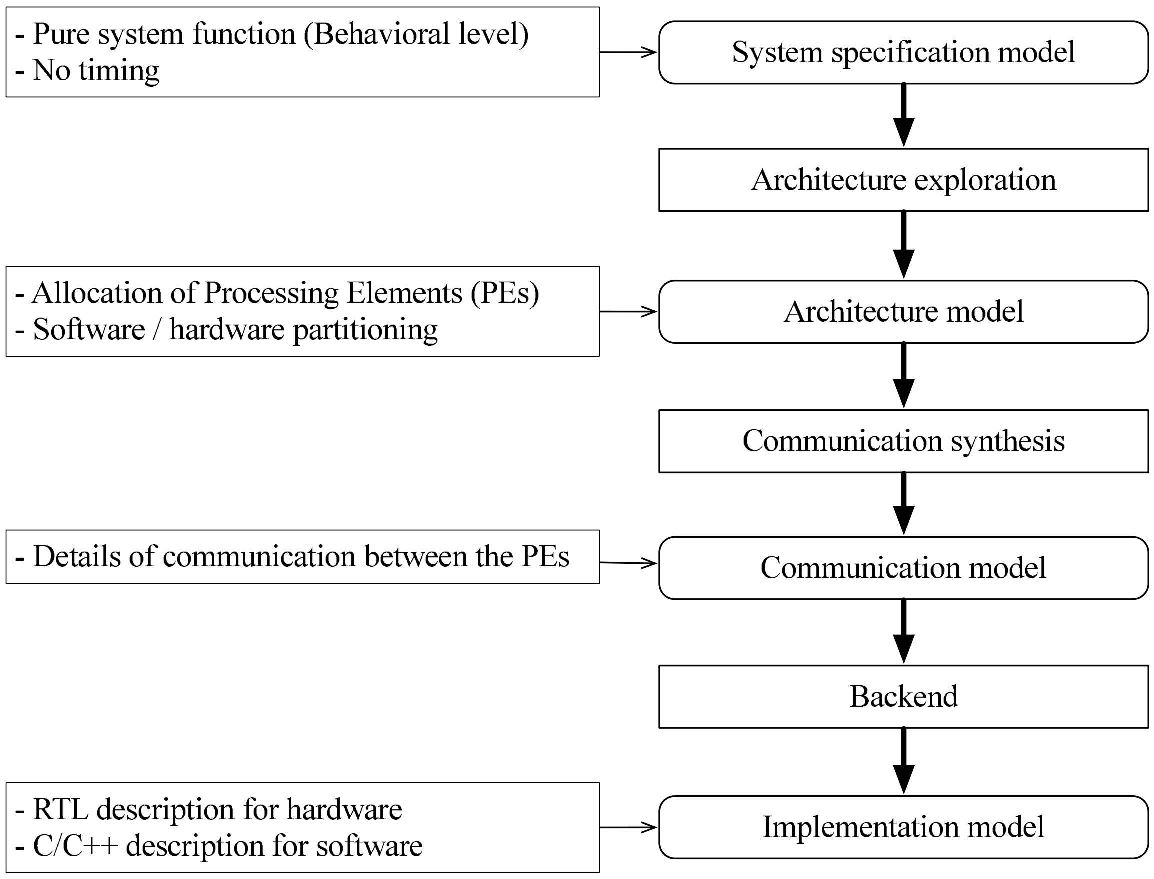

- Pure system function (Behavioral level)
- No timing
System specification model
Architecture exploration
- Allocation of Processing Elements (PEs)
- Software / hardware partitioning
Architecture model
Communication synthesis
- Details of communication between the PEs
Communication model
Backend
- RTL description for hardware
- C/C++ description for software
Implementation model

Figure 2.2: Overview of system-level design flow.

2.2 System-Level Design

System-level design is a design methodology that focuses on higher abstraction level first and foremost. System-level design takes place in the step of system design at a high level of abstraction in Figure 2.1. The basic idea is to model the behavior of the entire system at high level and evaluate system performances by a rapid prototyping in an early stage of system design. By the evaluation of system performances in the early stage of system design, the setback in latter design stage (e.g., software design step in Figure 2.1) can be reduced. Thus, system-level design can bring better efficiency of system design.

Figure 2.2 shows an overview of system-level design flow. System-level design flow consists of system specification model, architecture model, communication model, and implementation model. The design flow starts at system specification model, and it goes top-to-bottom by converting the models through architecture exploration, communication synthesis, and backend steps.

System specification model defines pure system function. Since the model only defines pure system function, the model does not considers the timing. System specification model is converted to architecture model by architecture exploration. Architecture model defines architectural components such as PEs. This model also indicates the allocation of functions to PEs. This is called software/hardware partitioning. By communication synthesis, architecture model is converted to communication model. Communication model has the detail of communication between the PEs. For example, this model defines the driver of an interface circuit and a bus protocol. Finally, communication model is converted to implementation model by backend step. Implementation model consists of C/C++ and RTL descriptions for software side and hardware side, respectively. The following describes the details of system-level design flow.

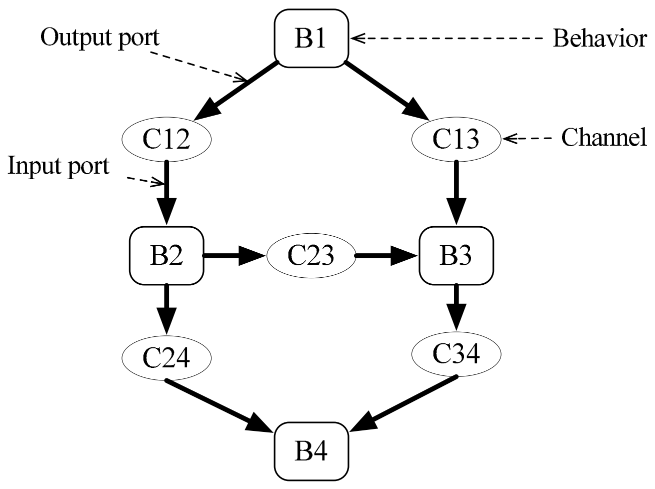


Figure 2.3: An example of system specification model.

System Specification Model

In the system-level design, a set of functions represents the functional specification, and the execution sequences of the functions are defined. The functions are called behavior in order to clarify differences from a function of the program. The behaviors can be executed in sequential and in parallel. The behavior has input and output ports in order to communicate with the outside by changing the value of the ports. Basically, the behavior gets values from input port, calculates them, and changes the value of output ports. The behavior communicates with the other behaviors through channels. A channel is an object which connects several behaviors' ports.

System specification model is a model that several behaviors execute their intended action and communicate each other. Figure 2.3 shows an example of system specification model. The model consist of four behaviors (B1, B2, B3, and B4) and five channels (C12, C13, C23, C24, and C34). At first, B1 starts its execution and outputs the result to C12 and C13 throughout the output port. B2 and B3 can run in parallel after they get data from C12 and C13 throughout the input port, respectively. They can communicate by C23 during their run. Additionally, C23 may synchronize B3 with B2 in order to set up an order of the execution of B2 and B3. In general, an event is used to synchronize behaviors. An event is produced in previous behavior (B2) and consumed by the following behavior (B3). The following behavior must wait until the event is produced. In this manner, an order of the execution of behaviors can be set up.

As described above, system specification model only represents pure system functions at behavioral level. Therefore, it does not have information of timing and details of implementation.

Architecture Exploration

In general, architecture exploration is a step to convert a system specification model into an architecture model. On this step, the allocation of PEs and a software/hardware partitioning are decided. First, system designers decide the allocation of PEs. In particular, system designers decide the number and kind of architectural components such as processors, dedicated hardware, busses, and memories. Note that the details of the implementation of PEs are not decided at this step. Then, system designers decide a software/hardware partitioning that indicates the allocation of behaviors to PEs. Figure 2.4 shows an example of software/hardware partitioning of Figure 2.3. In the example, B1 and B2 are allocated to PE1 which is a processor, and B3 and B4 are allocated to PE2 which is a dedicated hardware.

Architecture Model

Architecture model has information of architectural components such as PEs (processors and dedicated hardware), busses, and memories. It also has information of PEs on which each behavior realizes.

The implementations of PEs are decided at this model. For example, some PEs are newly designed as dedicated hardware, or existing dedicated hardware are used to realize

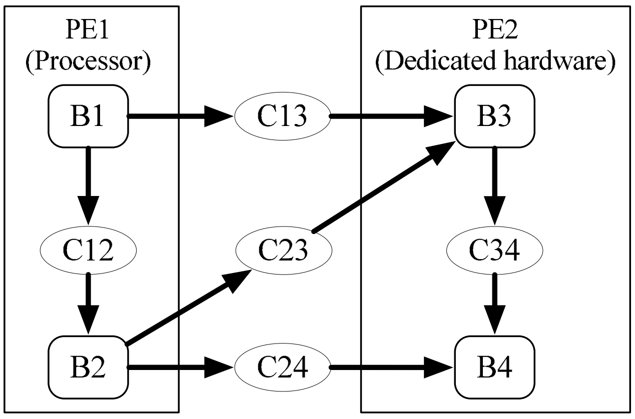


Figure 2.4: An example of a software/hardware partitioning.

them. On the other hand, other PEs may be implemented as software. It is also decided whether PEs can run in sequential or parallel. Since implementations of PEs are decided, the execution time of each PE can be roughly predicted with the knowledge of past experience. Moreover, it is possible to simulate the model if the model is described C based design language such as SystemC [17]. Using the prediction and simulation, system designers explore an appropriate architecture during architecture exploration.

Communication Synthesis

Before converting the architecture model into the communication model, communications among PEs must be synthesized. In particular, implementations of channels are synthesized.

There are five channels in Figure 2.4. Since B1 and B2 are allocated to a processor, channels C12 can be implemented as a shared variable of software program. However, channels C13, C23, and C24 represent communication between software and hardware. It is not easy to synthesize their implementation since they must be converted into a driver for software side and an interface circuit for hardware side. It is also not easy to implement a channel C34 representing communication in hardware. In the communication synthesis,

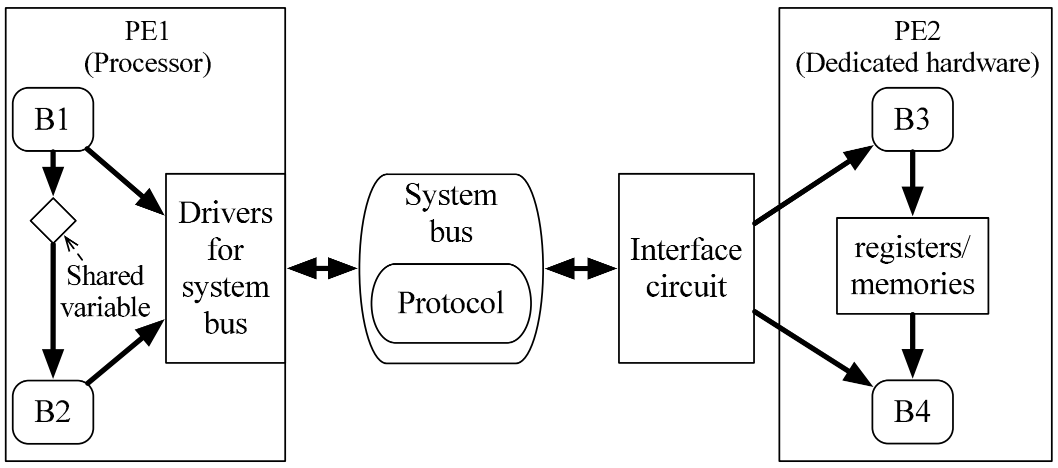


Figure 2.5: An example of communication model.

the implementation of channels are decided and synthesized.

Communication Model

Figure 2.5 shows an example of a communication model of Figure 2.4. In this example, a system bus is used for communication between software and hardware. Channels C12 is converted into a shared variable of software program. C34 is converted into registers, memories, and control signals since it represents the communication in hardware. Channels C13, C23, and C24 are communication between software and hardware. Thus, they are allocated to the system bus. Since these channels are allocated to the same system bus, they are integrated into a common driver for software side and a common interface circuit for hardware side. The driver and the interface circuit should be suitable for the protocol of the system bus.

Backend and Implementation Model

Implementation model is converted from communication model at backend step. In the backend step, synthesis tools are used to generate the implementation. The lowest model of system-level design is implementation model. In the implementation model, behaviors

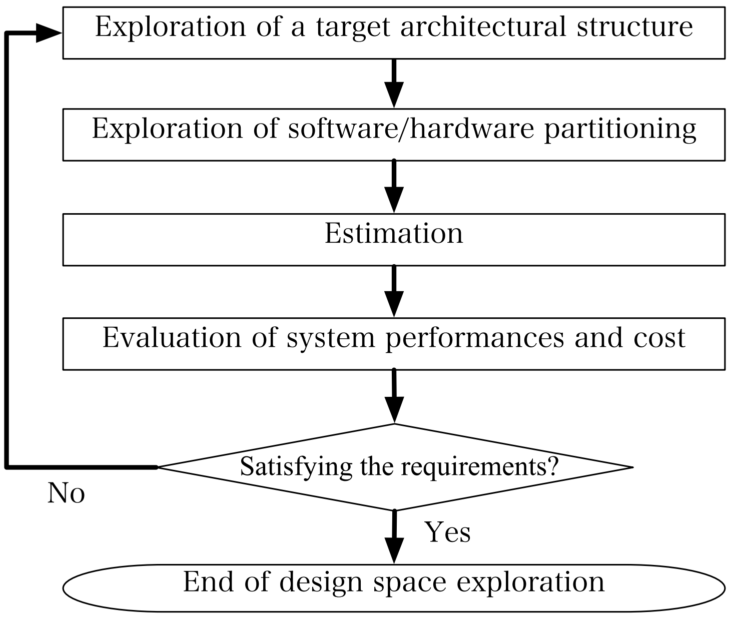

Exploration of a target architectural structure
Exploration of software/hardware partitioning
Estimation
Evaluation of system performances and cost
Satisfying the requirements?
No
Yes
End of design space exploration

Figure 2.6: A typical flow of design space exploration in system-level design.

allocated to software and the drivers synthesized during communication synthesis are converted into C/C++ descriptions. Then, the C/C++ descriptions are converted into binary by a compiler and linker tool chain. On the other hand, behaviors allocated to hardware and the interface circuits synthesized during communication synthesis are converted into Register Transfer Level (RTL, hereafter) descriptions. The RTL descriptions are synthesized as Flip-Flops and logic gates through logic synthesis and place and route.

2.3 Design Space Exploration in System-Level Design

For current large-scale embedded systems, earlier phase in the design flow has more influences in final products. Therefore, design space exploration in system-level design is one of key parts to design current large-scale embedded systems. The main purpose of design space exploration is to find an appropriate architectural structure and a software/hardware partitioning that satisfies the requirements of the system performances.

Figure 2.6 shows a typical flow of design space exploration which is generally per-

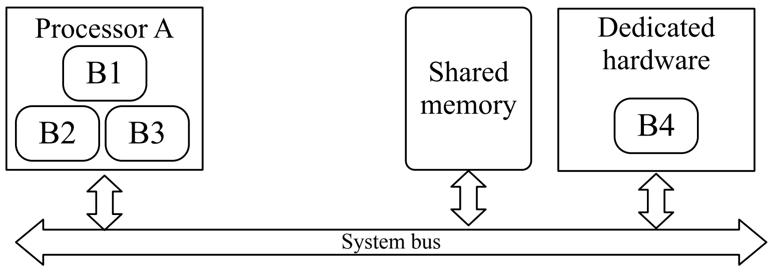


Figure 2.7: Result A: design space exploration for a single processor with dedicated hardware.

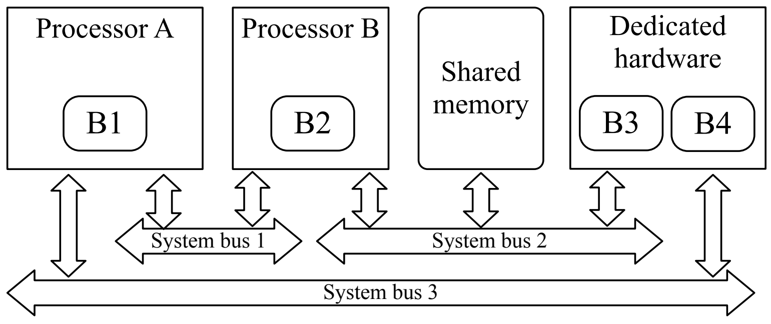


Figure 2.8: Result B: design space exploration for dual processors with dedicated hardware.

formed at architecture exploration step in Figure 2.2. The flow of design space exploration consists of four steps. At first, the number and kind of architectural components such as processors, dedicated hardware, busses, and memories are explored. Then, a software/hardware partitioning of behaviors is explored. After that, the system performances and costs are estimated. The estimation results are used to evaluate them. Finally, it is judged whether design space exploration continues or not. If the system satisfies the requirements, it is the end of design space exploration. If not, system designers should return to the first step of the flow. These four steps continue until system designers find an appropriate target architectural structure and a software/hardware partitioning.

Figures 2.7 and 2.8 show different results of design space exploration of four behaviors. Figure 2.7 shows a result that the target architecture has a single processor with dedicated hardware. In this result, three behaviors, B1, B2, and B3, are allocated to processor A, and a behavior B4 is allocated to dedicated hardware. Figure 2.8 shows a result that the

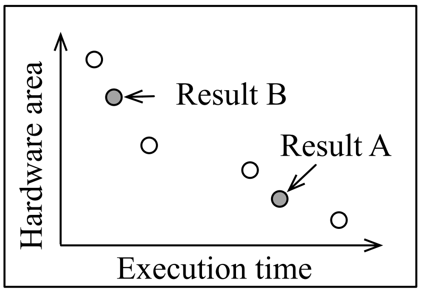


Figure 2.9: An example of trade-off relation between execution time and hardware area.

target architecture has dual processors with dedicated hardware. In this result, B1 and B2 are allocated to processor A and processor B, respectively, and B3 and B4 are allocated to dedicated hardware.

In terms of the execution time, Result B would be faster than Result A since B1 and B2, and B3 and B4 are allocated to different processors and dedicated hardware, respectively, in Result B. On the other hand, Result A would have less hardware area than Result B since Result A has only a single processor and B4 is only allocated to dedicated hardware. Figure 2.9 shows an example of trade-off relation between execution time and hardware area. In the figure, each circle indicates a software/hardware partitioning. Two gray colored circles represent Result A and Result B. As described above, the execution time of Result B is faster than that of Result A, and hardware area of Result A is less than that of Result B. There are the other software/hardware partitioning as shown white colored circles in the figure. In this manner, the explorations of the target architecture and software/hardware partitioning are closely related. Therefore, system designers should repeat these explorations to find an appropriate architecture and software/hardware partitioning.

At the estimation and evaluation steps, simulation tools and rapid prototyping are necessary. It is possible to simulate the architecture model if the model is described in C based design language. In order to simulate software, behaviors in the model are compiled and linked to run on a processor simulator. In order to simulate hardware, nowadays, behaviors are synthesized as Hardware Description Language (HDL, hereafter) by behavioral or high-level synthesis tools [18]. Recent growth of high-level synthesis tools has enabled designers to develop hardware modules at behavioral level using C/C++ like languages [19, 20, 21, 22]. By means of compilers and high-level synthesis tools, system designers can describe systems in a single behavioral language. Also, high-level synthesis tools support designers to develop hardware by using optimization technics of parallelizing compilers such as loop pipelining [23, 24]. Then the HDL description is simulated on HDL simulators. Backend program takes care of the communication between a processor simulator and a HDL simulator. By combining several simulators, the system model is simulated. Note that the accuracy of the estimation depends on the simulators.

Instead of the simulation tools, a rapid prototyping is used to evaluate the system performances of the model. An example is a FPGA-based rapid prototyping [25]. On the FPGA-based rapid prototyping, prototypes of software and hardware are realized on FPGA. Since FPGA is a real hardware device, the accuracy of evaluation is very high.

It takes time to prepare a simulation description and realize a FPGA-based prototype. For that reason, system-level design tools have been developed [4, 5, 6, 7, 8]. These tools automatically generate simulation descriptions or FPGA-based prototypes. Moreover, some tools support the explorations of target architectural structure and software/hardware partitioning. System-level design tools can support and accelerate the design space exploration for system designers.

2.4 Related Works

Various researches have been conducted on system-level design tools. The tools mainly assume heterogeneous multi-core processor system-on-a-chip as a target architecture.

System-On-Chip Environment [4] is a system-level design framework based on the

SpecC language [26]. It realizes an interactive and an automated design flow with a consistent and a seamless tool chain, and supports all the way from a specification of the system to a hardware/software implementation.

Artemis [5] provides modeling and simulation methods, and tools for efficient performance evaluation and exploration of heterogeneous embedded multimedia systems. Artemis's design flow start at a sequential application specification, and it is transformed to a concurrent application specification. Then, Artemis allows designers to estimate performance through co-simulation of a concurrent application specification.

PeaCE (Ptolemy extension as a Codesign Environment) [6] is a hardware-software codesign environment that provides seamless codesign flow from functional simulation to system prototyping. Its target application is multimedia applications with real-time constraints. Unlike other system-level design tools, PeaCE is a reconfigurable environment into which other design tools can be easily integrated.

Metropolis [7] is a modeling and simulation environment based on the platform-based design paradigm. It provides a general, proprietary metamodel language that is used to capture separate models for behavioral model, platform model, and their binding and scheduling. Metropolis itself does not define any specific design tools but rather a general framework and language for modeling with the support for simulation, validation, and analysis.

ARTS [8] provides a simulation platform for modeled in SystemC. It supports multiple PE models and network model among PEs. ARTS assumes that the application model simulated on it is already developed and separated properly in order to explore allocation to PEs.

2.5 SystemBuilder

In this section, the author shows a brief overview of SystemBuilder [2].

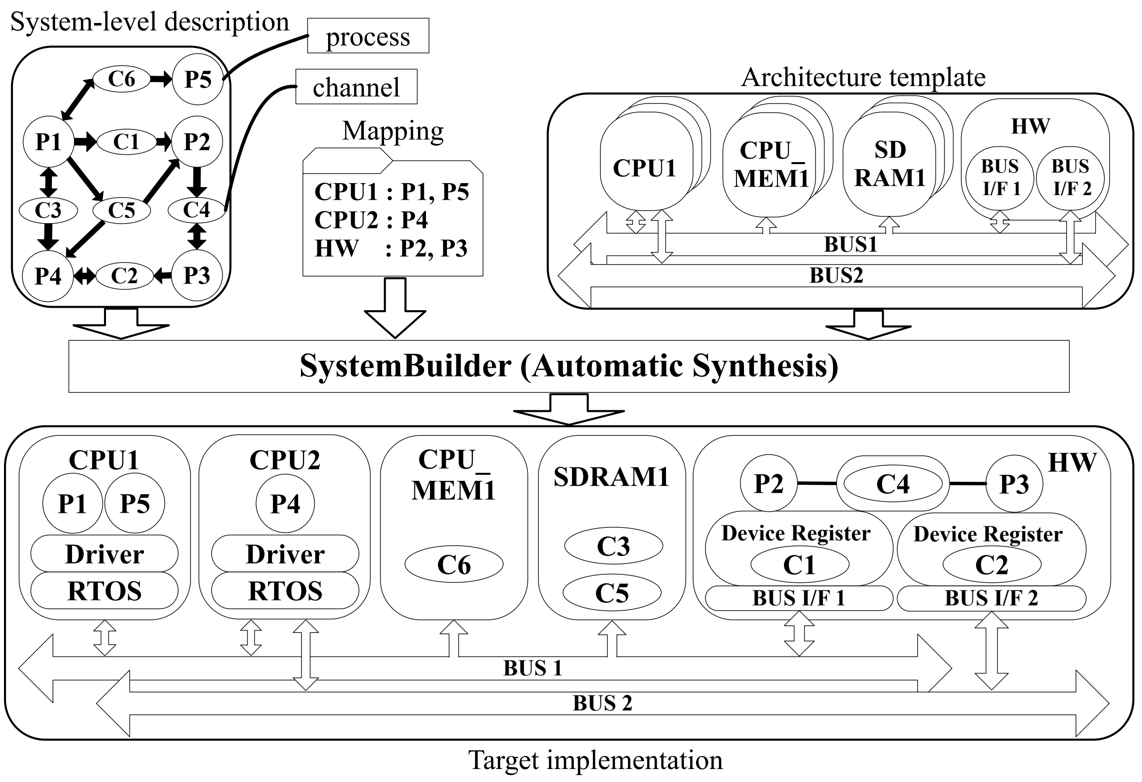


Figure 2.10: Overview of SystemBuilder.

2.5.1 Input description

Figure 2.10 shows input descriptions and the synthesis overview of SystemBuilder. SystemBuilder takes a system-level description, an architecture template, and a mapping as inputs, and it generates a target implementation of the system.

The system-level description represents the system functionalities. The system-level description consists of a set of applications, each of which in turn consists of a set of processes and channels. The processes are written in the C language and the communication APIs which are interfaces to the channels.

The channels represent communications among the processes. The channels are generally classified into two types: one is asynchronous and the other is synchronous.

Asynchronous channels are used to transfer data among processes. Non-blocking channels are one of asynchronous channels which represent a small data storage. Two or more processes can access non-blocking channels. Memory channels are another asynchronous channels which are used to transfer a large data between two processes. On the other hand, synchronous channels are mainly used to transfer events for activation of processes as well as synchronization between processes. Blocking channels are synchronous channels. In order to store multiple events, blocking channels have buffers. Typically, blocking channels and memory channels are combined to behave as FIFOs with large data, and non-blocking channels are used like global variables.

An architecture template represents a target architecture. The processors (CPUs) are assumed to be homogeneous, and the number of the processors must be greater than or equal to one. There is no limit of the number of the processors. The processors, dedicated hardware (HW), and memories (CPU_MEM, SDRAM) are connected through standard on-chip buses. There is also no limit of the number of memories and buses.

A mapping represents an allocation of processes to PEs such as CPUs and HW. Depending on the mapping, the process is implemented as either a software task on a real-time OS or a hardware module.

2.5.2 Automatic synthesis of target implementation

SystemBuilder automatically synthesizes a target implementation from system-level description depending on the mapping specified by the designers. The process mapped to a processor is compiled and linked with a RTOS as a task. The process mapped to HW is first translated to HDL from C language by a behavioral synthesis tool. Then the implementation of the HDL is synthesized by a logic synthesis tool for a target FPGA. SystemBuilder uses TOPPERS/ASP kernel [27] and TOPPERS/FMP kernel [28] for a single processor RTOS and multi processors RTOS, respectively. CyberWorkBench [19] and eXCite [20] are used as behavioral synthesis tools, and Altera Quartus II [29] is used as logic synthesis tool and a place and route tool for a target FPGA.

The interfaces of the channels written in the processes are translated into either interface programs or hardware logics depending on the mapping so that the processes can communicate with each other through the channels. The buffers in the channel are mapped onto the memory. In detail, the interfaces of the channels among the processes on the processors are implemented as API calls provided by the RTOS. Also the interfaces of the channels among the processes on HW are converted to additional hardware logics that realize FIFO and block RAMs in the target FPGA. The interfaces of the channels between the processes on the processors and those on HW are realized by the device drivers for the processors and by hardware logics that generate interrupts for the activation of the device drivers.

Figure 2.10 shows an example of the target implementation. In the figure, uni-directional arrows mean that the modules work as slaves of the connected buses. Also, bi-directional arrows mean that the modules work as slaves and masters of the connected buses. Processes P1 and P5 are mapped to CPU1, and a process P4 is mapped to CPU2. These processes are converted to RTOS tasks. Then they are compiled and linked with the device drivers. Processes P2 and P3 are mapped to HW. These processes are converted to hardware modules by a behavioral synthesis tool. Channels C1 and C2 represent communication between processors and HW. Thus, these two channels are synthesized as device registers and bus interface circuits. A channel C4 represents communication between hardware modules. It is synthesized as a communication circuit between hardware modules. Channels C3 and C5 are allocated to SDRAM1 which is shared memory since they are accessed from CPU1, CPU2, and HW. A channel C6 represents an inner processor communication. Thus, it is allocated to CPU_MEM1. Note that the decision of the mapping should be done by the designers.

Although SystemBuilder presently only supports Altera's FPGAs and their associated architectures, SystemBuilder can potentially support the other devices and architectures. In

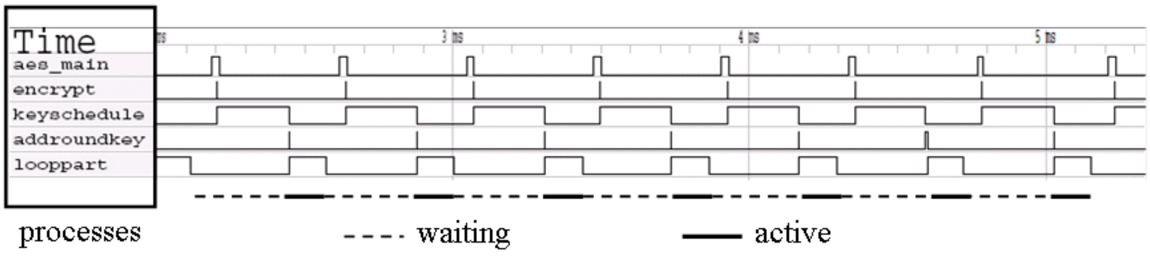


Figure 2.11: An example of process profiler waveforms.

actual, an earlier version of SystemBuilder supported Xilinx ’ s architectures with Microblaze soft-core processors [30] and the OPB bus [31].

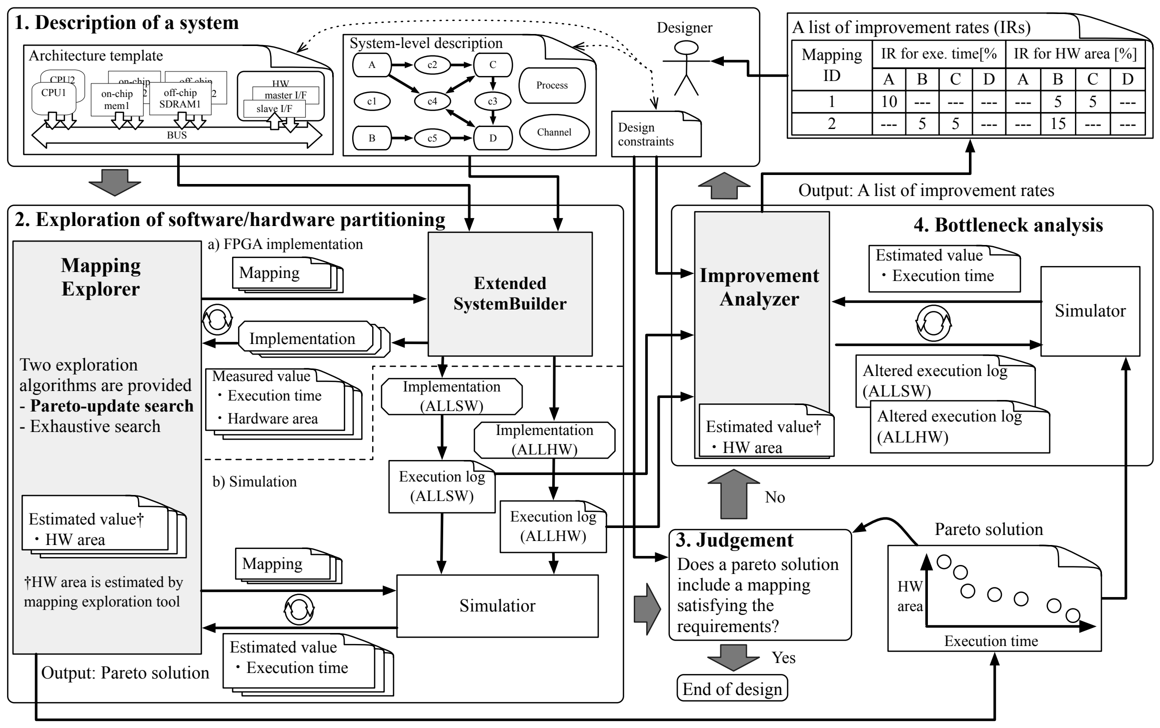
2.5.3 Performance Analysis

In order to refine system within a short time, SystemBuilder provides a profiling tool which is denoted as “process profiler”. Process profiler assists the designers so that they can find bottlenecks out from the processes executing concurrently by visualizing the execution histories, including the activation/suspension timing and the period of each process.

Process profiler can gather the histories of all processes allocated to software and hardware. The histories are provided in VCD file format. The VCD file can be visualized as waveforms by using tools such as GTKWave [32]. Figure 2.11 shows an example of waveforms which are provided by process profiler. With the waveforms, the designers can find out the processes that are bottleneck of the system. Thus, the designers can easily decide a mapping. Process profiler is implemented in HW, and it gathers the histories at the cycle level.

2.6 Overview of Proposed Method

This dissertation proposes an efficient design space exploration method at system-level. Proposed method consists of three tools which are Extended SystemBuilder, Mapping Ex-



Mapping ID	IR for exe. time[%				IR for HW area [%]			
	A	B	C	D	A	B	C	D
1	10	---	---	---	---	5	5	---
2	---	5	5	---	---	15	---	---

Figure 2.12: Entire design flow of the proposed method.

plorer, and Improvement Analyzer. By using the proposed method, system designers efficiently explore the large design space and can find a system configuration that satisfies the system requirements in short time. This section shows an overview of the proposed method.

Figure 2.12 shows an entire design flow of the proposed method.

First of all, system designers describe the system functions as a system-level description. They also decide the target architectural structure which is defined by the target architecture template. The system-level description and the target architecture template are inputs of Extended SystemBuilder used in next step.

Secondly, system designers explore software/hardware partitioning. Mapping Explorer generates mappings which indicates software/hardware partitioning of the system functions. Mapping Explorer generates mappings by two algorithms. Pareto-update search is an efficient algorithm which finds a pareto solution between execution time and hardware area. However, pareto-update search does not ensures an ideal pareto solution. On the other hand, exhaustive search surely find an ideal pareto solution since it generates all possible mappings.

The generated mappings are inputted to Extended SystemBuilder. Then, Extended SystemBuilder generates the implementations of generated mappings in order to evaluate the system performances of them. Unlike the other system-level design tools, Extended SystemBuilder can synthesize communication for shared hardware modules in order to realize better implementations. There is another choice, simulator, to evaluate the system performances of the generated mappings. In order to use the simulator, two particular execution logs, all software (hereafter, ALLSW) and all hardware (hereafter, ALLHW), are needed. The execution logs are profiled by process profiler from two particular implementations (ALLSW and ALLHW) which are generated by Extended SystemBuilder. ALLSW allocates all processes to a single processor, whereas ALLHW allocates all processes to dedicated hardware whenever possible. The simulator takes two execution logs and estimates the execution time of inputted mappings. By using the simulator, designers can evaluate the system performances of the generated mappings in short time. After the evaluation, Mapping Explorer gets feedback of the evaluation results in order to generate new mappings for the further exploration. Until Mapping Explorer finds a pareto solution, the generation of mappings, and the evaluation of system performances are repeated.

After the exploration of software/hardware partitioning, system designers have the pareto solution between execution time and hardware area. Since the pareto solution is a set of mappings on a trade-off curve, system designers can find appropriate mappings. Thus, they can judge whether the pareto solution includes mappings that satisfy the system requirements.

If mappings that satisfy the system requirements exist, it is the end of the design space exploration. However, mappings that satisfy the system requirements may not exist since the exploration of software/hardware partitioning only changes mappings. In such case, system designers need to modify the system-level description in order to improve bottlenecks of the system.

Improvement Analyzer supports bottleneck analysis. Improvement Analyzer uses an algorithm to automatically identify not only bottlenecks of the system but also a list of improvement rates. The list of improvement rates shows the ratio of improvement that are necessary for processes to satisfy the system requirements. Thus, the list of improvement rates is useful for system designers to consider how to modify the system-level description in order to improve the system performances.

Finally, system designers modify the system-level description to improve the system performances. If necessary, they may modify the target architecture template.

The flow of design space exploration is repeated until system designers find an appropriate design that satisfies the system requirements. Extended SystemBuilder assists designers to get better target implementations. Mapping Explorer and Improvement Analyzer assist designers to explore mappings and identify bottlenecks of the system, respectively. Therefore, the proposed method accelerates design space exploration at system-level.

Chapter 3

A Case Study of AES Encryption System with SystemBuilder

3.1 Introduction

In this chapter, the author demonstrates a case study on an AES encryption system design with SystemBuilder which is described in section 2.5. The main purpose of this case study is to evaluate the design efficiency of SystemBuilder. In addition, this case study clarifies good and bad points of SystemBuilder. It is important to know them in order to make system-level design more efficient.

The main objective of SystemBuilder is to help designers efficiently explore software/hardware partitioning in short time based on iterative evaluations of system performances by executing systems actually on an FPGA. SystemBuilder takes a system-level description, a target architecture template, and a mapping of functions to architectures as inputs, and it automatically generates target implementations including software, hardware, and interfaces among them. Also, SystemBuilder provides communication APIs to partition the system functions in the system-level description. It is easy for designers to partition the system functions because communication APIs and the system-level description are written in C language. Since SystemBuilder automatically generates the target implementations and interfaces from divided system-level descriptions and APIs, it is possible to easily design coarse-grain pipelined system.

SystemBuilder has an advantage in the design time. Since interfaces among software and hardware should be built on every change of software/hardware partitioning decision, the interface synthesis capability especially affects time to design. With SystemBuilder, designers can avoid describing the interfaces such as device driver programs and hardware control logics. As a result, SystemBuilder shortens the design time. With these capabilities, system designers do not need to consider the details of the interfaces. Thus, they can develop a coarse-grain pipelined system in short time.

The author selected AES encryption as a target system since encryption is an essential technology to keep a large amount of data in safety these days. In addition, an AES algorithm has become the default choice for various security services in numerous applications. For example, wide band Internet is spread and a lot of data are transferred through the Internet. There are a lot of risks to be stolen the data through the Internet. In order to handle a lot of data in short time, a fast AES encryption is needed. However an AES encryption algorithm has many steps and takes long time to handle many inputs. One of the solutions to speed up an AES encryption system is designing the system as hardware [33]. The author has decided to improve the performance of AES encryption system by a coarse-grain pipelined hardware implementation.

Starting from a sequential software program, the author incrementally developed a pipeline-structured system-level description with SystemBuilder. The author present a whole design process aiming to develop a fast AES encryption system. The AES encryption system with a pipelined hardware implementation achieved 5.0 times higher performance than that with software implementation. This case study shows the effectiveness of

SystemBuilder on system-level design.

This chapter is organized as follows. Section 3.2 presents a case study on an AES encryption system design. Section 3.3 evaluates the effectiveness of SystemBuilder through the case study, and section 3.4 concludes this chapter.

3.2 AES Encryption System Design

This section shows the four steps of developing the AES encryption system. This case study aims to make an AES encryption system faster.

The author started an AES encryption system design from converting a software program into a system-level description. A software program of AES is selected from CHStone Benchmark Suite [34], and the author confirmed that it is correctly executed on a Nios II processor with an RTOS. Next, the author partitioned the AES encryption system into six processes. One of them can be allocated to only software, and the others can be allocated to software and hardware. Then five processes allocated to hardware are designed with pipeline manner. Finally some multiple, division, and modulo operations were changed into bit operations.

3.2.1 Usage of SystemBuilder

SystemBuilder was used in two means in this case study. One is to design a software/hardware intermingled system. SystemBuilder automatically generates RTL descriptions of hardware processes, execution files of software processes and communications among processes. The other is to generate a pipelined system. Since Communication APIs are written in C language, they can be replaced in short time. A system is partitioned into several processes by SystemBuilder so that the processes can run in parallel. In addition, SystemBuilder provides waveforms by process profiler in order to find bottleneck processes.

These features help a designer to let the processes run in parallel.

3.2.2 Single Process Software Design

Starting point of the AES encryption system design is an AES program written in C language. Through this case study, any algorithm of the original AES program was not changed. In this design, the AES encryption system consists of a single process, aes_main. The process can run as software. This design is called a "single process software design". The single process software design was executed on Nios II, and it took 1,236 msec.

3.2.3 Five Processes Hardware Design

The author partitioned the system into several processes before designing the coarse-grain pipelined system. The author decided the principle to translate each function in C program into a process. Some of the partitioned processes are designed as hardware implementations to run faster than the single process software design. The AES encryption system was partitioned into six processes, aes_main, encrypt, keyschedule, addroundkey, looppart, and endpart, as shown in Figure 3.1. The author added communication APIs in each process in order to communicate among six processes. Blocking channels were used for starting a process, memory channels were used for sending encrypted data, and non-blocking channels were used for parameters. The author decided that all blocking channels have no buffer in order to debug easily. If both partitioning of the system-level description and the addition of buffer are performed, it is difficult to determine the cause of system bugs. In this design, encrypting data was sent as same as the single process software design. The endpart process has a blocking channel connected to the aes_main process to notify that encryption of an integer data is completed. The aes_main process is software implementation, and the other five processes are hardware implementation. This design is called a "five processes

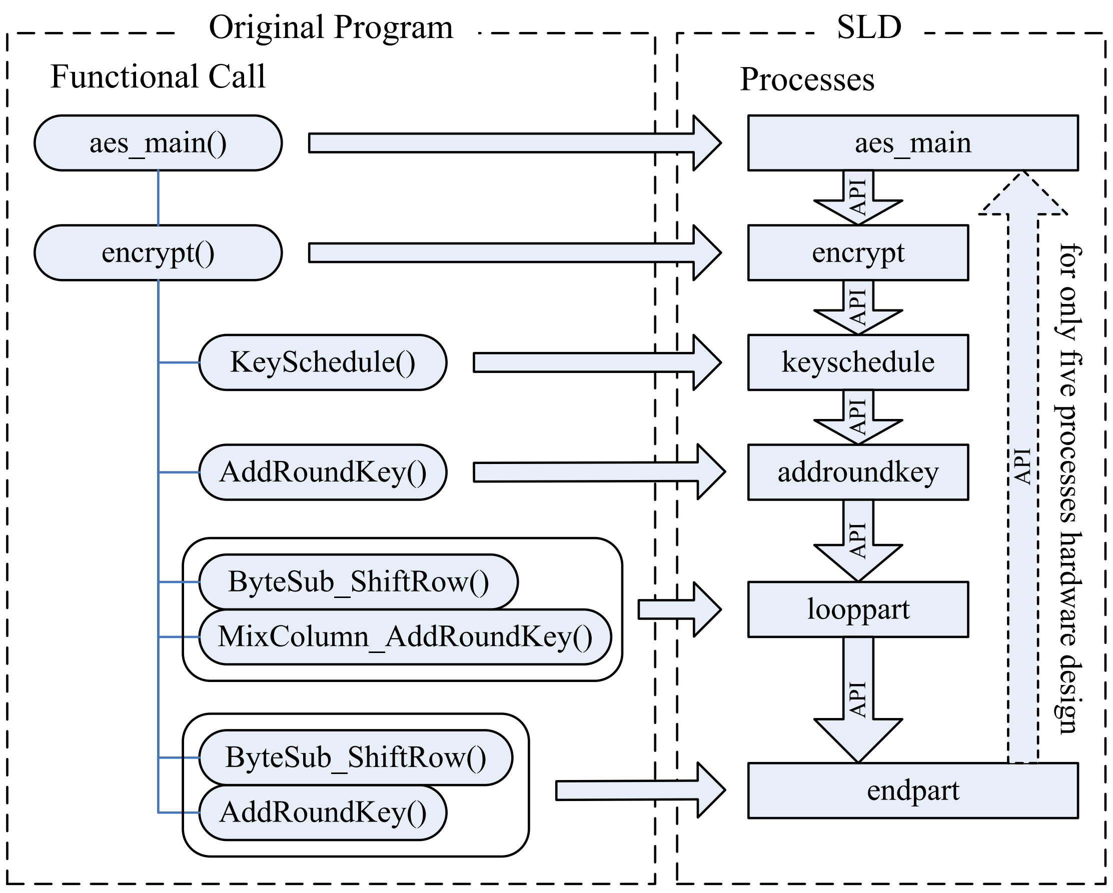


Figure 3.1: System partitioning and processes.

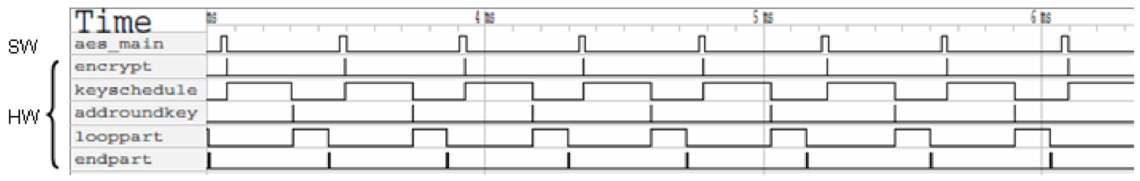


Figure 3.2: Waveform of the five processes hardware design.

hardware design".

The author measured the performance of the five processes hardware design. The design took 429 msec. Its performance is 2.9 times higher than the single process software design. Figure 3.2 shows the profiler waveform of the five processes hardware design. In the figure, as same as the author had expected, the author confirmed three things: the aes_main process started after the endpart process had ended, only a single process runs at the same time, and every time all processes are executed in the same order. So far, the system was partitioned into six processes, and the data were encrypted correctly. Because of the blocking channel connecting the endpart process and the aes_main process, six processes execute sequentially like the function call of software program. Actually, all six processes were possible to run in parallel because one of them can be allocated to software, whereas the others can be allocated to hardware. Next, the author shows how to design pipelined hardware implementation.

3.2.4 Pipelined Design

The author changed the configuration of APIs and the system-level description in order to design coarse-grain pipelined system. First the author removed the blocking channel connecting the endpart process and the aes_main process shown in Figure 3.1. The blocking channel was for debugging, and it was no longer necessary to design the pipelined system. The blocking channels have buffers, and their size is configurable. Then the author increased the number of buffers in the blocking channels. The author only needed to change

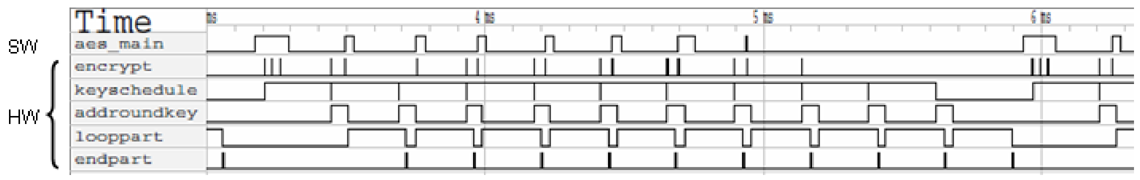


Figure 3.3: Waveform of the pipelined design.

the system-level description to increase the buffer size of the blocking channel. The author changed the buffer size from zero to one. Therefore, a process sending a message can write data to the blocking channels until the buffer becomes full, while a process receiving a message runs. Also the memory channel APIs and the system-level description need to be changed in order to protect the data. If memory size is equal to transferred data size, the data is overwritten by the producer process while it is being used by the consumer process. To prevent that, the author added two same size memories by changing the system-level description and adding APIs for the memories. The blocking channel which starts the process sends the address data of memory. The address data was used in order to prevent processes from overwriting the encrypting data.

As same as the five processes hardware design, the aes_main process is software implementation, and the other five processes are hardware implementation. This design is called a “pipelined design”. The pipelined design took 276 msec. Its performance is 4.5 times higher than the single process software design and also 1.5 times higher than the five processes hardware design.

Figure 3.3 shows a profiler waveform of the pipelined design. In the figure, the author can see that multiple processes ran in parallel. Since that, the total execution time of the system was shortened. The author analyzed that the keyschedule process ran longer than the other processes and the other processes had to wait for long time. The author confirmed that the bottleneck is the keyschedule process and the execution time of the keyschedule process need to be improved in order to shorten the total execution time of the system.

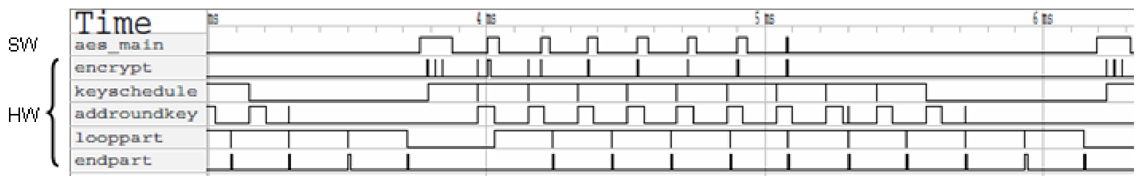


Figure 3.4: Waveform of the operation refined design.

3.2.5 Operation Refined Design

The author checked operations used in the keyschedule process in order to shorten its execution time. Multiply, division and modulo operations are used a lot, and most of their operands are 2^n. These three operations become multi-cycle operations when they are generated as hardware. Since most of their operands are 2^n, they could be changed to bit operations which are right shift, left shift, and AND operations. Bit operations become single cycle operations when they are realized as hardware. Therefore, the author converted multiply operations, division operations, and modulo operations into left shift operations, right shift operations, and AND operations, respectively.

The author only changed keyschedule process's operations. Then the author generated five processes as hardware implementation and a single process as software implementation as same as the pipelined design. This design is called a "operation refined design". The operation refined design took 243 msec. Its performance is 5.0 times higher than the single process software design.

Figure 3.4 shows a profiler waveform of the operation refined design. In the figure, the author found out that the keyschedule process of the operation refined design ran faster than that of pipelined design. Since the keyschedule process runs faster, waiting time of the looppart process was reduced compared to the pipelined design. Thus, the total execution time of the system was reduced.

Table 3.1: Performance improvement of the AES encryption system.

designs	performance	improvement	#LE	memory
Single Process Software	1,236 msec	1.0 x	7,672	1.09 Mbit
Five Processes Hardware	429 msec	2.9 x	23,866	1.26 Mbit
Pipelined	276 msec	4.5 x	29,088	1.39 Mbit
Operation Refined	243 msec	5.0 x	28,164	1.39 Mbit

3.2.6 Performance of Design

The system performance was measured by its total execution time to encrypt 10 integer data for 100 times. Table 3.1 shows the implementation results of four designs. It shows performances, total execution time, improvement, the number of Logic Element (LE, hereafter), and amount of memory usage of FPGA. Improvement of the performances is based on the single process software design. High-level synthesis tools can generate a hardware implementation like the five processes hardware design, and the performance of the hardware implementation is improved from a software implementation. However, high-level synthesis tools cannot generate coarse-grain pipelined system. SystemBuilder realizes to design the coarse-grain pipelined system such as the pipelined design. The pipelined design is faster than the five processes hardware design. Therefore, the generation of the pipelined hardware implementation is effective to shorten the total execution time of the system. The operation refined design is not improved its performance a lot from the pipelined design. The changes of operations seem to be not effective. The design of a pipelined system is more important than the change of operations.

3.2.7 Design Time of AES Encryption System

This section shows detail of work time and contents at each section. Figure 3.5 shows the design time and the summary of work at each step. The author debugged the system

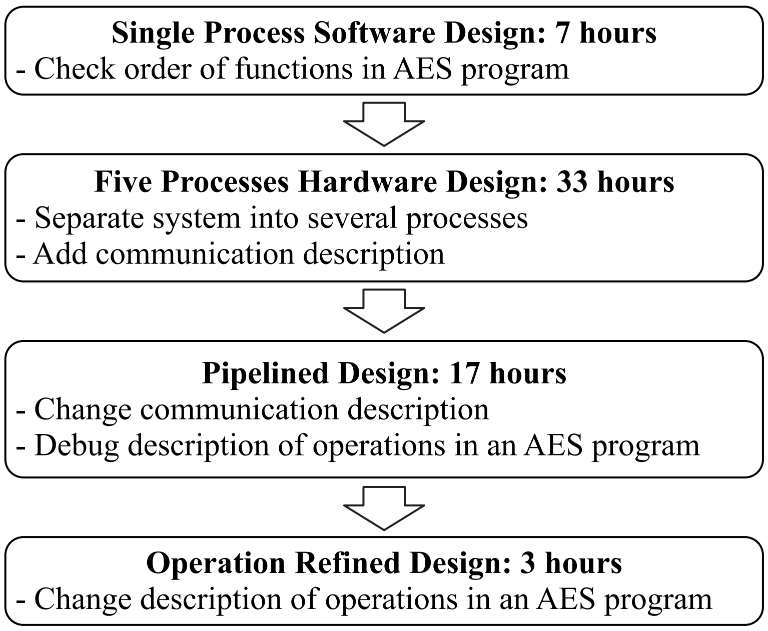

Single Process Software Design: 7 hours
- Check order of functions in AES program
Five Processes Hardware Design: 33 hours
- Separate system into several processes
- Add communication description
Pipelined Design: 17 hours
- Change communication description
- Debug description of operations in an AES program
Operation Refined Design: 3 hours
- Change description of operations in an AES program

Figure 3.5: Design time and summary of work on the AES encryption system.

with software/hardware cosimulation at each step. First step is development of the single process software design. It took seven hours to develop. In this step, the author checked the order of functions in AES program.

Second step is development of the five processes hardware design. This step took 33 hours. In this step, the author partitioned the system into six processes and added APIs to communicate among processes.

Third step is development of the pipelined design. This step took 17 hours. The author changed only the system-level description and APIs at this step. The author had a mistake in description of memory APIs and took time to debug it.

The last step is development of the operation refined design. This step took three hours. In this step, the author converted multiply, division, and modulo operations into bit operations in C language.

The author totally took 60 hours to develop the AES encryption system in this case study. If the author were supposed to work eight hours a day, the pipelined AES encryption system would be developed in 7.5 days, about a week.

3.3 Evaluation of SystemBuilder

The author has developed 1,000 lines of C program and has experienced RTL design in student experiment. With SystemBuilder, an unskilled designer even can design the pipelined hardware system in almost a week. SystemBuilder has high effectiveness to design systems, especially to design the pipelined system. The author has shown the advantages of SystemBuilder as follows:

- Easy to partition a system into several processes.
- Inexpensive to change and add communication API calls.
- Automatic generation of communication channels.
- Easy to develop the pipelined systems with APIs.
- Easy to check and analyze the system performances.

However, the author found some problems below during this case study.

- Limited mapping
- Long evaluation time of a lot of mappings
- Lack of support to improve the system performance

The first problem is limited mapping. There are the same functions in different processes in AES encryption system. SystemBuilder, however, cannot support a mapping that several processes share a hardware module. Sharing a hardware module among the processes has possibility to decrease the hardware area. Thus, SystemBuilder should support the other mappings to realize better system performances. The second problem is long evaluation time. Even though SystemBuilder automatically generates the implementations, it takes very long time to generate the implementations and evaluate the system performances. With only SystemBuilder, it is hard to evaluate the system performances of many mappings in order to find appropriate mappings. The third problem is lack of support to improve the system performances. SystemBuilder clarified a bottleneck of execution time by profiling tools. However, it is hard for designers to consider how to improve the system performances of current complex embedded systems with only profiling tools. Designers need another support to improve the system performances in order to design systems efficiently.

3.4 Conclusions

This chapter presented a case study of an AES encryption system design with System-Builder. SystemBuilder can generate a target implementation of the system given as a system-level description, an architecture template, and a mapping. The AES encryption system has developed by converting a sequential software program. In order to achieve better performance, the author took 4 steps to design pipelined system. During the design, the author analyzed the behaviors of the processes by process profiler to find the bottleneck. As a result, the author designed a system which achieve 5.0 times faster than software program by a hardware implementation with pipelined manner on an FPGA. The author took about 60 hours for overall design. This was realized with an automatic synthesis capability of SystemBuilder. The author conclude that designing a pipelined system with System-Builder is effective. In addition, this case study clarifies three problems of system-level design.

Chapter 4

Extended SystemBuilder: Communication Synthesis for Hardware Sharing

4.1 Introduction

The three problems of system-level design were clarified by a case study of AES encryption system with SystemBuilder as described in Chapter 3. One of the three problems is that system-level design tools support limited mapping. A number of system-level design tools supporting process and channel mapping were proposed in the past [1]. However, process-level hardware sharing, i.e., a mapping of processes which exist in different applications onto a single hardware module, is not supported by most of the existing system-level design tools. This limitation lost opportunities of designers for getting better implementations.

Because most of the tools assume single-application systems, existing system-level design tools have not supported process-level hardware sharing. Although some tools assume multiple applications, they do not allow process-level hardware sharing. Even if it is allowed, the tools do not automatically synthesize the interface circuits which realizes mutually exclusive accesses to the shared hardware module. Therefore, the designers need to implement the interface circuits manually.

This chapter presents an automatic synthesis of communications for hardware modules which are shared by multiple applications. SystemBuilder has been extended in order to support process-level hardware sharing. This is named Extended SystemBuilder. Extended SystemBuilder automatically generates the interface circuits for the shared hardware module. Since the applications may run concurrently, the interface circuits generated by Extended SystemBuilder realizes mutually exclusive accesses to the shared hardware module.

The rest of this chapter is organized as follows. Section 4.2 presents the detail of the communication synthesis for hardware sharing. Section 4.3 shows the effectiveness of hardware sharing through a case study, and section 4.4 concludes this chapter.

4.2 Communication Synthesis for Hardware Sharing

4.2.1 The Design Flow with Hardware Sharing

Figure 4.1 shows the design flow of Extended SystemBuilder. Designers first design applications independently as shown in (a). If hardware is not shared, Extended SystemBuilder generates the system implementation without hardware sharing as shown in (b).

It is assumed that a system consists of more than one application. It is also assumed that some processes in the different applications have the same functions. In the description (a), there are two applications, Application1 and Application2, and processes P_B and P_Y have the same function. If hardware is shared, Extended SystemBuilder automatically converts the description (a) into an internal description (c). During the conversion, processes P_B and P_Y are merged into a new process P_S with the same function, where process P_S is shared by the two applications. In the proposed hardware sharing method, the number of channels in Application1 and Application2 does not change. In other words, Applica-

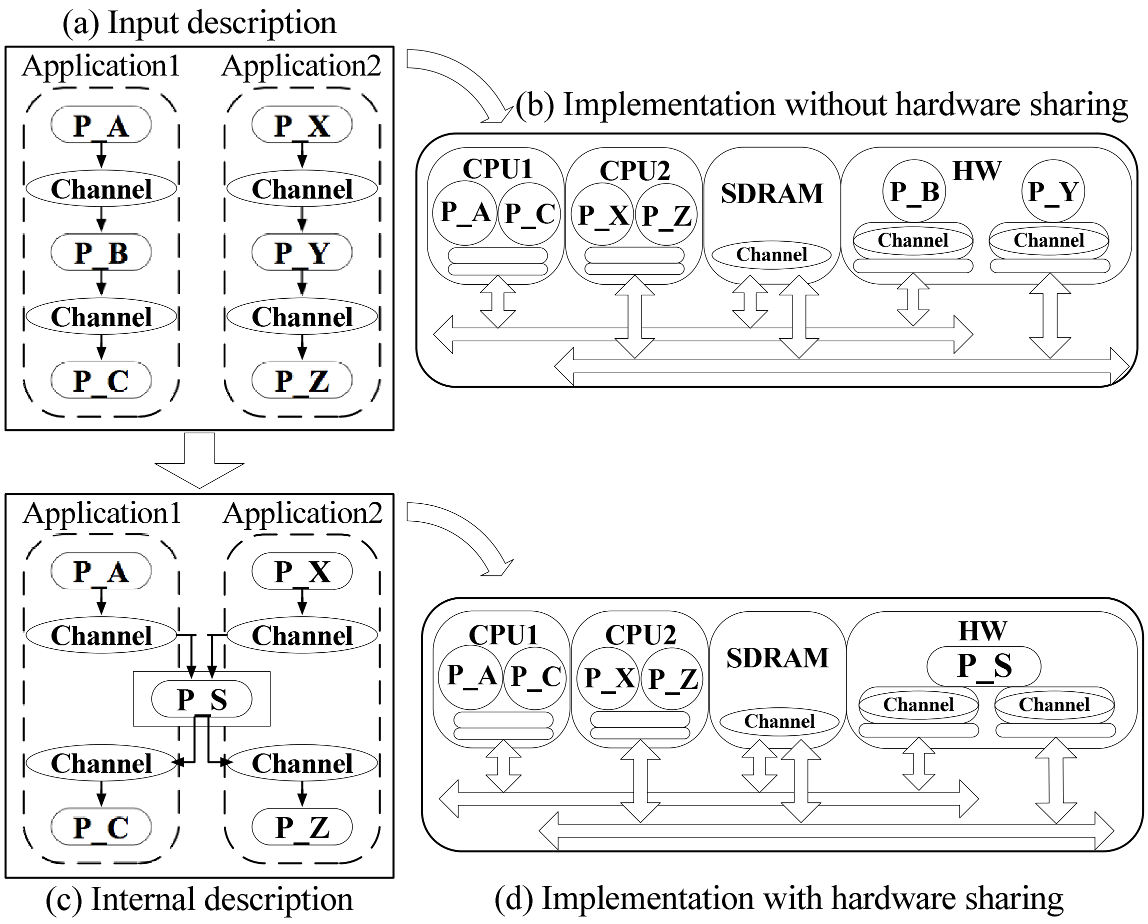


Figure 4.1: Design flow of Extended SystemBuilder with/without hardware sharing.

tion1 and Application2 do not share the channels. Thus, no data conflict occurs within the communication channels. Then, Extended SystemBuilder automatically synthesizes the implementation with hardware sharing, as shown in (d) from the internal description (c).

Extended SystemBuilder automatically completes the design flow from (a) to (d) in Figure 4.1 if a hardware sharing option is enabled in the mapping. Designers only need to turn on the option so that the two applications share a hardware module. The designers, therefore, will be able to explore a wider design space in short time. Note that more than two applications can share a hardware module although Figure 4.1 only shows two applications.

4.2.2 Implementation of Communication for Hardware Sharing

Process P_S in Figure 4.1(c) is shared by Application1 and Applicaiton2, and thus process P_S requires two sets of channels, one for Application1 and another for Application2. However, note that the process originally has only a single set of the channels. Also note that the functions inside the process should not be modified for reusability and easiness of debugging.

Extended SystemBuilder automatically inserts a wrapper to the shared process as shown in Figure 4.2. The wrapper has two sets of external channels, i.e., one for each application. In addition, the wrapper provides an interface to the shared process. The wrapper realizes mutual exclusion and selects a channel to which the shared process should access. Also, Extended SystemBuilder inserts a signal to channels which are connected to the shared process. The signal indicates whether a buffer in the channel is empty or not.

The wrapper works as follows. First, the wrapper polls the signals whether the channels have valid data or not. If more than one channels have valid data, the wrapper selects an application to be served. Extended SystemBuilder supports two types of polling, priority-based polling and round-robin one. Designers select the polling policy and decide the

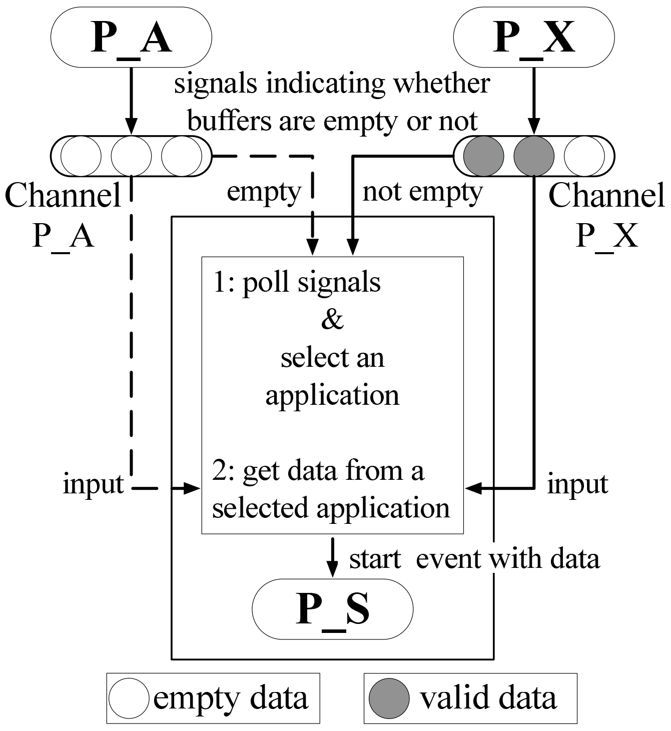
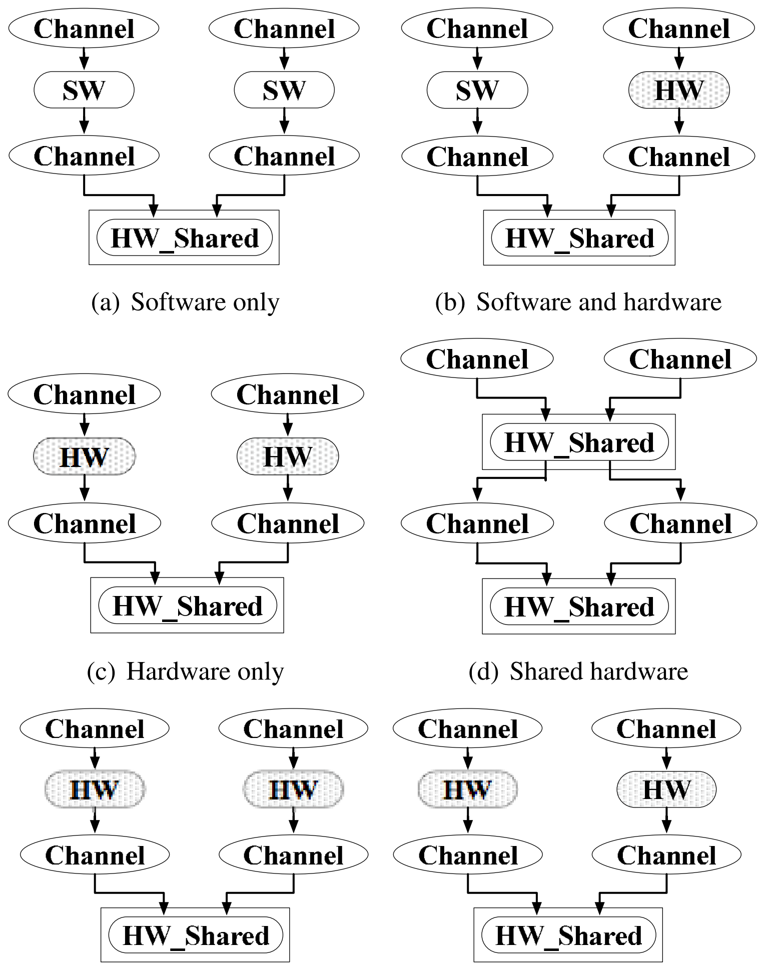

P_A
P_X
signals indicating whether buffers are empty or not
Channel P_A
Channel P_X
empty
not empty
1: poll signals & select an application
2: get data from a selected application
input
input
start event with data
P_S
empty data
valid data

Figure 4.2: Detail of the wrapper generated by Extended SystemBuilder.

priorities of the applications in the mapping. With priority-based polling, every time the shared process starts polling, the channel of the highest priority application is checked first. If its signal indicates empty, the next highest priority application will be checked. With round-robin polling, the channels are checked in a round-robin manner. The polling continues until non-empty signal is found.

Next, data are read from the channel of the selected application, and the wrapper sends a start event and the data to the shared process. Then, the shared process starts its execution.

The shared process may communicate with the other processes not only at entry and exit points of its execution but also during its execution. Every time the shared process communicates with the other processes, the wrapper passes the data between the shared process and the channel of the selected application.



(a) Software only (b) Software and hardware

(c) Hardware only (d) Shared hardware

(e) Four hardware share two shared hardware

Figure 4.3: Mappings of hardware sharing supported by Extended SystemBuilder.

4.2.3 Mapping of Processes with Hardware Sharing

Figure 4.3 shows five mappings of hardware sharing supported by Extended SystemBuilder. The proposed hardware sharing method does not restrict hardware/software mapping possibilities. This means that the shared processes are able to communicate with the processes to be implemented in software as well as ones to be implemented in hardware as shown in Figure 4.3(a), Figure 4.3(b), and Figure 4.3(c). Furthermore, the shared processes can communicate with the other shared processes as shown in Figure 4.3(d). Also, the proposed hardware sharing method does not restrict the number of shared processes. For example, there can be two shared processes among four processes as shown in Figure 4.3(e).

4.3 A Case Study

In this section, a case study is presented in order to show the effectiveness of the proposed hardware sharing method. Section 4.3.1 explains the target systems. Section 4.3.2 shows the evaluation of design space exploration with hardware sharing method. Section 4.3.3 indicates the reduction of hardware area, and section 4.3.4 makes clear the relation between polling policy and the execution time of each application.

4.3.1 Target Systems

A case study has been conducted on three systems, Dual-AES, Triple-AES, and Quad-AES. Dual-AES, Triple-AES, and Quad-AES consist of two, three, and four AES applications [34], respectively. Each application in the systems is numbered from 1 through 4. In other words, there are four applications named AES1, AES2, AES3, and AES4 as shown in Figure 4.4. The four AES applications are identical, and each application encrypts and decrypts data which consist of 16 integers for 1000 times. The AES applications consist of four processes, aes_mainX, encX (encryption), decX (decryption), and check_resultX. In the name of processes, X differs from 1 to 4 depending on the AES application. It is assumed that each AES application runs on a dedicated processor. In case of Quad-AES, it consists of four processors, and the four AES applications run on their own processors.

In this case study, the system designer has explored different mappings on Dual-AES, Triple-AES, and Quad-AES as summarized in Tables 4.1, 4.2, and 4.3, respectively. The system designer has varied the allocations of all encX processes on either a software (SW, hereafter) or a hardware (HW, hereafter). Similarly, all decX processes are mapped to either a SW or a HW. Thus, there are mainly four patterns of mappings below:

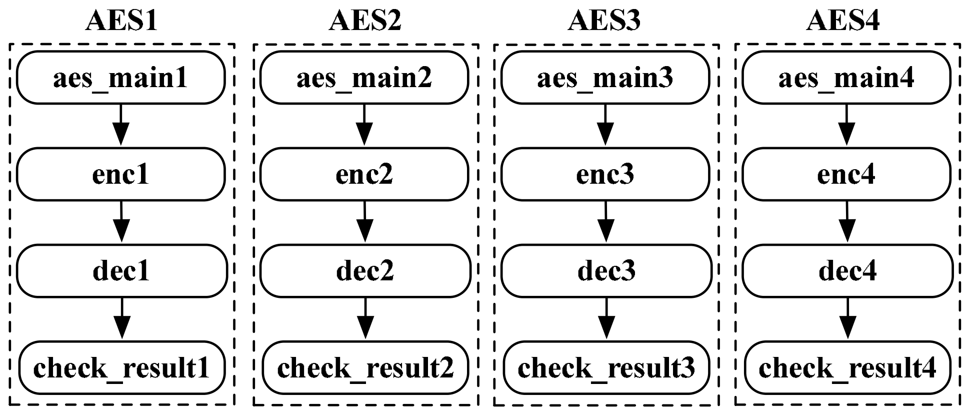


Figure 4.4: Construction of Quad-AES.

- encX: SW, decX: SW (mapping No.1 in Tables 4.1, 4.2, and 4.3)
- encX: HW, decX: SW (mapping No.2 in Tables 4.1, 4.2, and 4.3)
- encX: SW, decX: HW (mapping No.5 in Tables 4.1, 4.2, and 4.3)
- encX: HW, decX: HW (mapping No.8 in Tables 4.1, 4.2, and 4.3)

If the processes encX and decX are mapped to HW, they are also mapped to shared hardware with priority-based polling (SH-HW (Priority)) and round-robin polling (SH-HW (Round)). For Triple-AES and Quad-AES, the mappings from No.17 to No.20 were added in order to evaluate partially shared hardware. In these mappings, two encX processes and/or two decX processes are mapped to SH-HW (Round). Note that, through the design space exploration, only the mapping is changed. According to the mapping, Extended SystemBuilder automatically synthesizes the implementation which is executable on FPGA. On an average, Extended SystemBuilder took about an hour to synthesize an implementation. To complete the exploration of an AES system, it took less than 24 hours by a single designer.

In this work, Altera StratixII FPGA board with four Nios II soft-core processors [29] was used as the target architecture. The processes mapped to SW were cross-compiled and

Table 4.1: Mappings and the polling policies of Dual-AES.

No.	enc1	enc2	dec1	dec2
1	SW	SW	SW	SW
2	HW	HW	SW	SW
3	SH-HW (Priority)		SW	SW
4	SH-HW (Round)		SW	SW
5	SW	SW	HW	HW
6	SW	SW	SH-HW (Priority)	
7	SW	SW	SH-HW (Round)	
8	HW	HW	HW	HW
9	SH-HW (Priority)		HW	HW
10	SH-HW (Round)		HW	HW
11	HW	HW	SH-HW (Priority)	
12	HW	HW	SH-HW (Round)	
13	SH-HW (Priority)		SH-HW (Priority)	
14	SH-HW (Round)		SH-HW (Round)	
15	SH-HW (Round)		SH-HW (Priority)	
16	SH-HW (Priority)		SH-HW (Round)	

linked with the TOPPERS/FDMP kernel [35], which is a RTOS for multi processors, to be executed on the Nios II soft-core processors. The processes mapped to HW and shared HW were synthesized with a commercial behavioral synthesis tool eXCite [20]. These compilation and synthesis tasks were automatically done by Extended SystemBuilder.

4.3.2 Design Space of Hardward Sharing

In terms of hardware area and execution time on the FPGA, the 16 mappings were evaluated on Dual-AES, and 20 mappings were evaluated on Triple-AES and Quad-AES as shown in Tables 4.1, 4.2, and 4.3, respectively. Figures 4.5, 4.6, and 4.7 show the hard-

Table 4.2: Mappings and the polling policies of Triple-AES.

No.	enc1	enc2	enc3	dec1	dec2	dec3
1	SW	SW	SW	SW	SW	SW
2	HW	HW	HW	SW	SW	SW
3	SH-HW (Priority)			SW	SW	SW
4	SH-HW (Round)			SW	SW	SW
5	SW	SW	SW	HW	HW	HW
6	SW	SW	SW	SH-HW (Priority)		
7	SW	SW	SW	SH-HW (Round)		
8	HW	HW	HW	HW	HW	HW
9	SH-HW (Priority)			HW	HW	HW
10	SH-HW (Round)			HW	HW	HW
11	HW	HW	HW	SH-HW (Priority)		
12	HW	HW	HW	SH-HW (Round)		
13	SH-HW (Priority)			SH-HW (Priority)		
14	SH-HW (Round)			SH-HW (Round)		
15	SH-HW (Round)			SH-HW (Priority)		
16	SH-HW (Priority)			SH-HW (Round)		
17	SH-HW (Round)		HW	SH-HW (Round)		HW
18	SH-HW (Round)		HW	HW	SH-HW (Round)	
19	SH-HW (Round)		HW	SH-HW (Round)		
20	SH-HW (Round)			SH-HW (Round)		HW

Table 4.3: Mappings and the polling policies of Quad-AES.

No.	enc1	enc2	enc3	enc4	dec1	dec2	dec3	dec4
1	SW	SW	SW	SW	SW	SW	SW	SW
2	HW	HW	HW	HW	SW	SW	SW	SW
3	SH-HW (Priority)				SW	SW	SW	SW
4	SH-HW (Round)				SW	SW	SW	SW
5	SW	SW	SW	SW	HW	HW	HW	HW
6	SW	SW	SW	SW	SH-HW (Priority)			
7	SW	SW	SW	SW	SH-HW (Round)			
8	HW	HW	HW	HW	HW	HW	HW	HW
9	SH-HW (Priority)				HW	HW	HW	HW
10	SH-HW (Round)				HW	HW	HW	HW
11	HW	HW	HW	HW	SH-HW (Priority)			
12	HW	HW	HW	HW	SH-HW (Round)			
13	SH-HW (Priority)				SH-HW (Priority)			
14	SH-HW (Round)				SH-HW (Round)			
15	SH-HW (Round)				SH-HW (Priority)			
16	SH-HW (Priority)				SH-HW (Round)			
17	SH-HW (Round)		SH-HW (Round)		SH-HW (Round)		SH-HW (Round)	
18	SH-HW (Round)		SH-HW (Round)		SH-HW (1&4) (Round)	SH-HW (Round)		SH-HW (1&4) (Round)
19	SH-HW (Round)		SH-HW (Round)		SH-HW (Round)			
20	SH-HW (Round)				SH-HW (Round)		SH-HW (Round)	

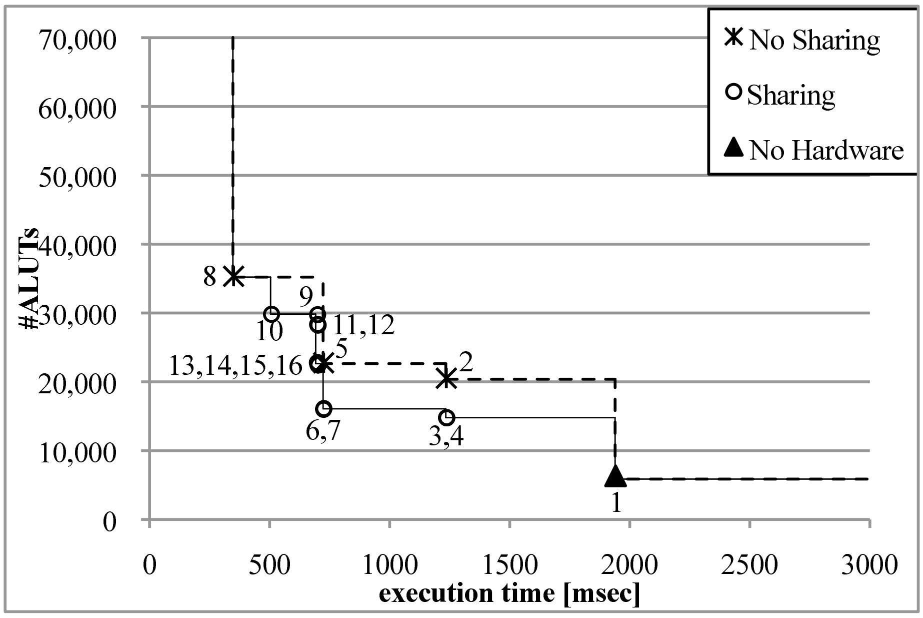


Figure 4.5: Trade-offs between performance and hardware area: Dual-AES.

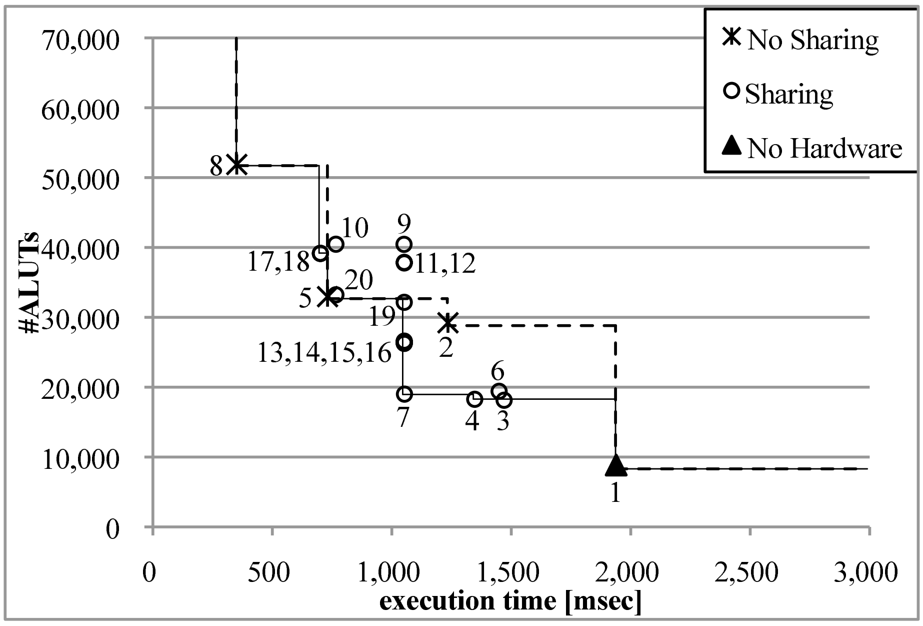


Figure 4.6: Trade-offs between performance and hardware area: Triple-AES.

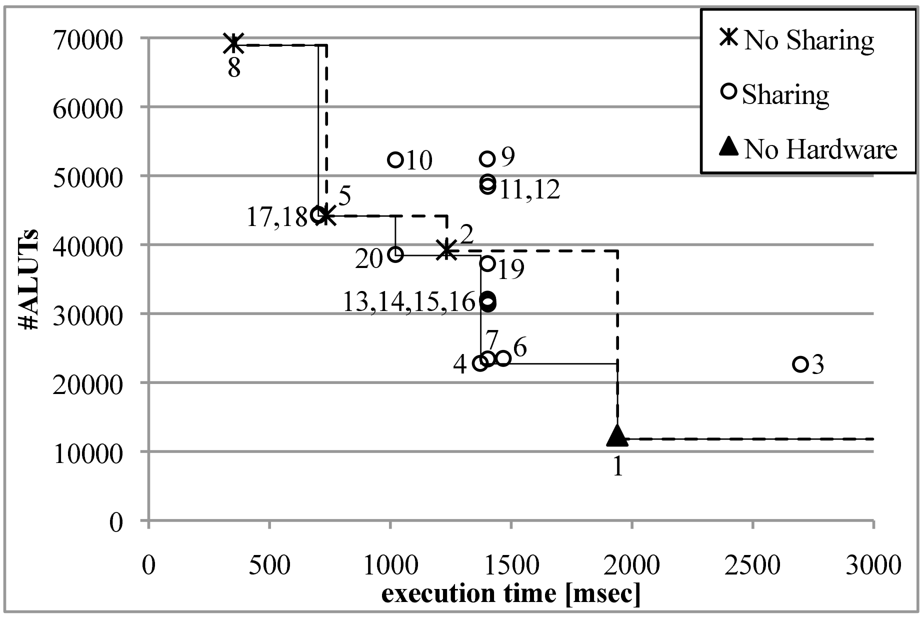


Figure 4.7: Trade-offs between performance and hardware area: Quad-AES.

ware area (in #ALUTs) and the execution time (in milli-second) of each mapping on the three systems. #ALUTs shows the hardware area of processors, peripherals, and processes mapped to HW and shared HW. In the figures, the solid lines and the broken lines represent the trade-offs of all mappings and those without hardware sharing, respectively. It is easily observed that the solid lines (with hardware sharing) represent better trade-offs than the broken lines (without hardware sharing) on the three AES systems. In Figure 4.6, the mapping No.7 (with hardware sharing), has less hardware area and better performance than the mapping No.2 (without hardware sharing). Hardware sharing, therefore, can bring better area-performance trade-offs.

As mentioned in section 4.2.3, the designers can explore the number of processes to be shared. In Figure 4.6, the mapping No.20 has two shared HW each of which are shared by two processes. Since the mapping No.20 is on the solid line, it is a candidate of an optimized solution. This result indicates that it is important to explore the number of processes to be shared, which is supported by the proposed hardware sharing method.

4.3.3 The Reduction of Hardware Area

In Figure 4.5, if the mappings No.2, No.3, and No.4 are focused on, the hardware area was reduced by 28% thanks to hardware sharing. Their only difference of the mapping is shared or not shared. The same can be said for the mappings No.5, No.6, and No.7. Also, if the mappings No.2, No.3, and No.4 in Figures 4.6 and 4.7 are focused on, the hardware area was reduced by 37% and 42%, respectively. In the figures, the mappings marked with a circle which has shared HW have the least hardware area among the mappings except the mapping No.1 marked with black triangle which has no hardware. Thus, the proposed hardware sharing method is effective to reduce the hardware area.

In the three systems, the ratio of the hardware area on the mappings No.3 and No.4 is almost the same, and the only difference between the mappings No.3 and No.4 is the polling policy. The same can be said for the mappings No.6 and No.7 in the three systems. In other word, the polling policy did not influence reduction of the hardware area if the processes have the same mapping. The proposed hardware sharing method, therefore, can reduce the hardware area with both priority-based polling and round-robin one.

4.3.4 The Execution Time with Polling Policy

In order to make clear the relation between the polling policy and the execution time of each application, the execution time of each application is measured. On the three AES systems, Figures 4.8, 4.9 and 4.10 show the execution time of each AES application in the mappings as shown in Tables 4.1, 4.2, and 4.3, respectively.

The polling policy of the mappings No.4, No.7, No.10, No.12, and No.14 on the three systems is only round-robin polling. Also, the mappings No.18, No.19, and No.20 on Triple-AES and Quad-AES use only round-robin polling. In these mappings, the execution time of each application was averaged. This result indicates that the wrapper with round-

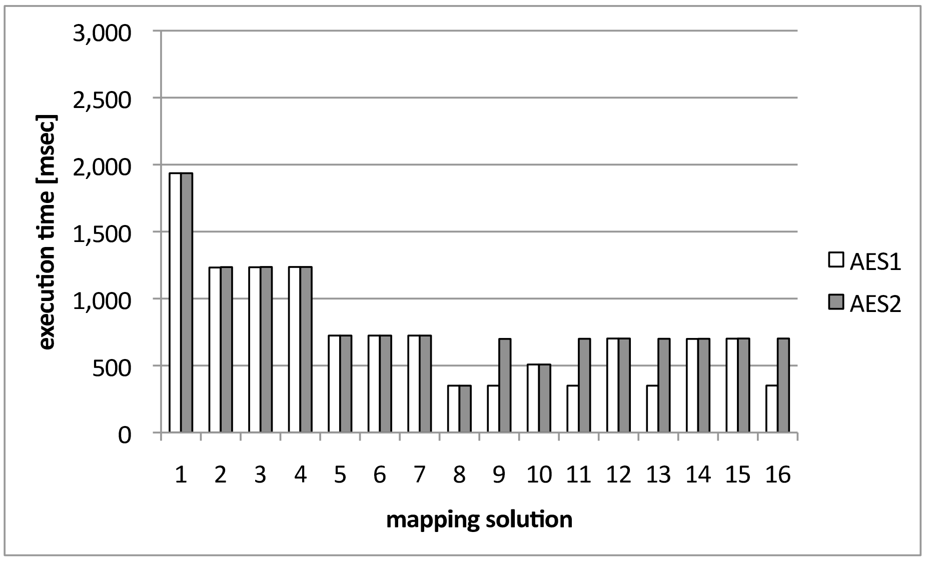


Figure 4.8: Execution time of AES applications: Dual-AES.

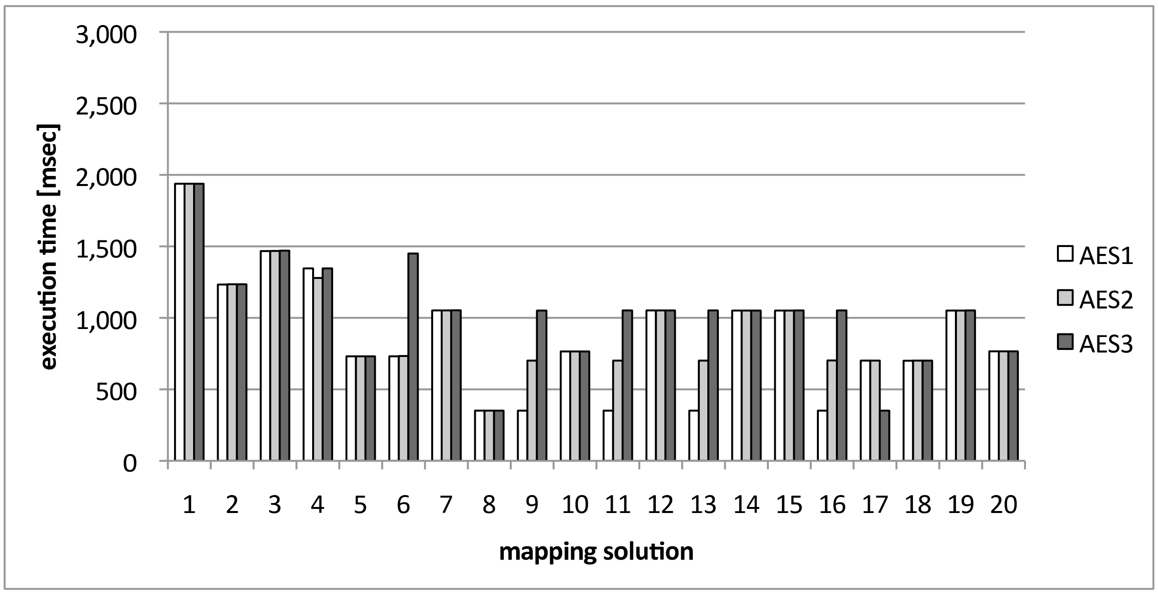


Figure 4.9: Execution time of AES applications: Triple-AES.

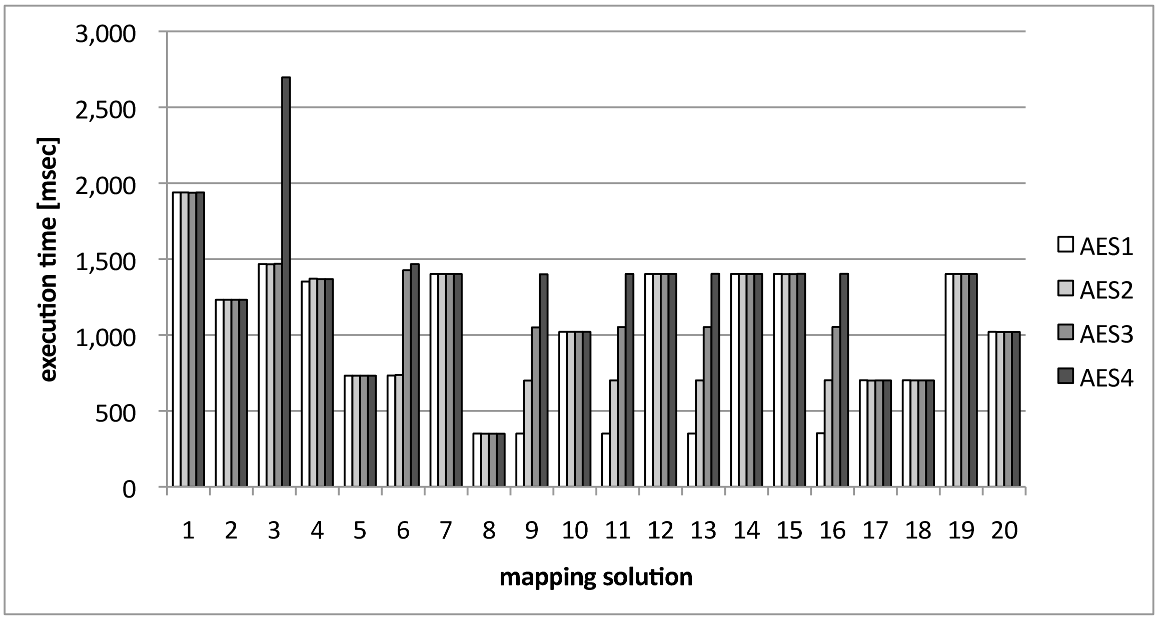


Figure 4.10: Execution time of AES applications: Quad-AES.

robin polling equally selected the application running on the shared process.

The mappings No.9, No.11, and No.13 on the three systems shows typical results of priority-based polling. In these mappings, AES1 which has the highest priority is completed at first. Then AES2, AES3, and AES4 which have the second, the third, and the lowest priority, respectively, were completed in the order of the priorities.

In the mapping No.6 with priority-based polling, however, AES1 and AES2 on the three systems were completed at almost the same time. Since encX processes in the mapping No.6 were mapped to software, the execution of encX processes was not faster than that of the shared process. In particular, when the wrapper started the polling, enc1 process was running while enc2 process had written the data to the channel. Thus, the wrapper selected AES2 instead of AES1 which the previous process of the shared process was running. As a result, AES1 and AES2 were selected by round-robin manner, and they were completed at the same time. Then, AES3 and AES4 in the mapping No.6 were completed after the completion of AES1 and AES2 in the systems.

Comparing the mapping No.9 with the mapping No.10 in the three systems, the mapping No.9 with priority-based polling took longer than the mapping No.10 with round-robin one in the total execution time.

In the case of that the deadline of a particular application is very strict, priority-based polling is more suitable than round-robin one. On the other hand, in the case of that the execution times of all application should be averaged, round-robin polling is more suitable.

4.4 Conclusion

In this chapter, an automatic communication synthesis method for hardware sharing is proposed. In addition, the method is implemented on system-level design tool named Extended SystemBuilder. With Extended SystemBuilder, the designers can explore wider design space including shared hardware in short time since Extended SystemBuilder automatically synthesizes communication for hardware sharing by only changing the mapping. A case study for shared hardware has been conducted on AES applications. The case study demonstrated that hardware sharing brought better area-performance trade-offs and a wider design space. The case study also demonstrated that the hardware sharing reduced hardware area while it kept the performance.

Chapter 5

Mapping Explorer: A Fast Design Space Exploration by Pareto-Update Search

5.1 Introduction

System-level design tools use models which is described at a high level of abstraction in order to generate an implementation of the system according to its mapping. Designers can evaluate the system performances (e.g., execution time) of the mapping by executing the generated implementation. Because the implementations are automatically generated by the tools according to the mapping, designers can evaluate the system performances of different mappings by simply changing the mapping. However, it takes a long time to generate the implementations for design space exploration such as mapping exploration described above. For example, it took a day to synthesize and evaluate 12 implementations of different mappings on both a single-core processor and a dual-core processor platforms [31].

In addition, Extended SystemBuilder supports process-level hardware sharing as described in Chapter 4. This extension brought more and more mappings. As the number of mappings increases, the time to generate implementations also increases. Therefore, the

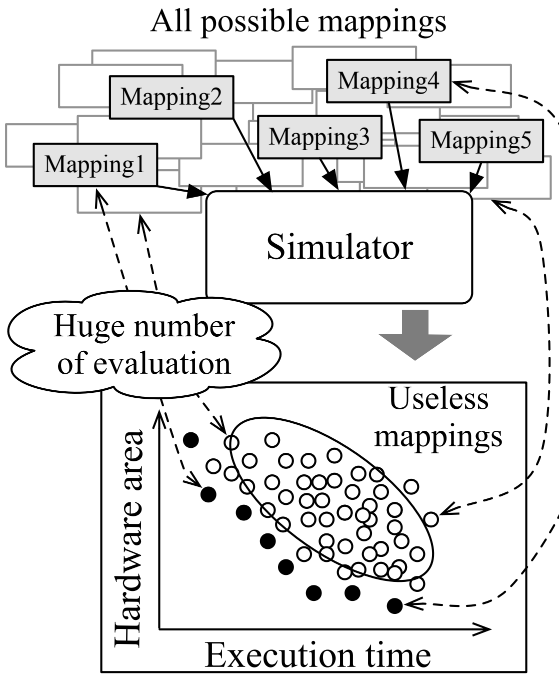


Figure 5.1: Simulation-based design space exploration methods: exhaustive search.

only use of basic system synthesis tools is no longer suitable for design space explorations of current, complex embedded systems.

To decrease the time needed to evaluate the system performances of a mapping, system-level performance estimation tools [9, 10, 11] have been introduced. Since these tools use profiles recorded by system-level design tools and FPGAs, the simulation accuracy is sufficiently high, and the exploration can be accelerated. For this reason, such tools allow for the possibility of finding the best mapping by an exhaustive search (shown in Figure 5.1). However, despite the simulation speed, there is still a problem with exhaustive searches. Assuming the numbers of functions and PEs are 10 and five, respectively, there are 5^{10}, or almost 10 million, mappings. If the simulator needs one second to simulate each mapping, it will require more than two weeks to complete the search and find the best mapping. Such a big number of mappings do not seem to be reasonable for design space exploration. Therefore, even though the simulators can accelerate the exploration, an

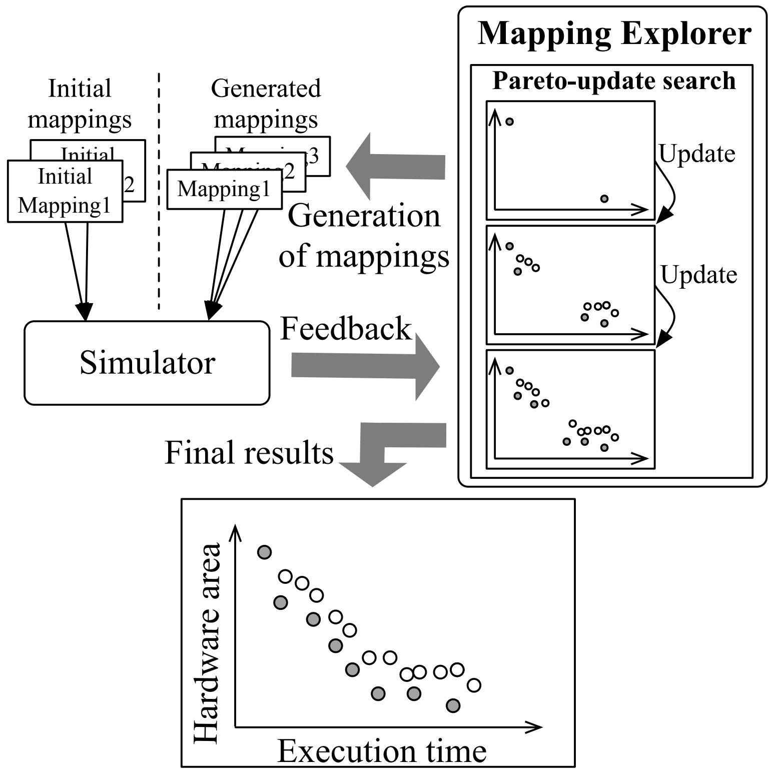


Figure 5.2: Simulation-based design space exploration methods: pareto-update search.

efficient exploration method is still necessary. In order to reduce the design time, useless mappings in Figure 5.1 should not be explored.

In order to find the best mapping during the design space exploration, the concept of the pareto solution is used, as this indicates the candidates for the best mapping [15]. An example of a pareto solution in relation to execution time and hardware area is illustrated by the set of black circles in Figure 5.1. Because the pareto solution contains all candidates for the best mapping, designers can select mappings which satisfy the requirements of the system performances from the pareto solution. This has the potential to shorten the design time.

There are some strategies for generating mappings in order to quickly find the pareto solution, as shown in Figure 5.2. Using these strategies, the useless mappings shown in Figure 5.1 will not be simulated. One famous strategy involves genetic algorithms (GAs, hereafter). The number of simulations is drastically reduced using a GA instead of an exhaustive search. Reducing the number of simulations shortens the exploration time. However, multiple executions of the exploration that increase the number of simulations may obtain better pareto solutions, because GAs are heuristic methods. In addition, GAs require parameters, such as the mutant probability and crossover probability, to be calibrated for each target before starting the exploration.

The author proposes an exploration algorithm named pareto-update search for finding pareto solutions between execution time and hardware area. The author also developed Mapping Explorer which is a tool to find pareto solutions by pareto-update search as shown in Figure 5.2. Pareto-update search starts with two mappings as initial pareto solution. Then it repeatedly updates the pareto solution with the generation and simulation of the mappings. In order to explore efficiently, pareto-update search tries not to generate and simulate useless mappings. In other words, pareto-update search generates only mappings whose probability of being in the pareto solution is high. Therefore, the number of simulated mappings is reduced. On the other hand, the pareto solution found by pareto-update search might have errors compared to an ideal pareto solution. Since the simulator contains an error of up to 10%, a pareto solution found by exhaustive search with simulators also includes errors compared to the measures from a real implementation. Hence, a pareto solution with a small error is acceptable for designers to decide a mapping satisfying the requirements of system performances. The goal of pareto-update search is to find a pareto solution efficiently with a small error.

The rest of this chapter is organized as follows. The author considers some related work in section 5.2. In section 5.3, the author defines pareto solution. Section 5.4 describes details of proposed exploration algorithm which is used in Mapping Explorer, and section 5.5 presents a case study that demonstrate the effectiveness of proposed exploration method. Finally, section 5.6 give conclusions.

5.2 Related Works

The problem of finding a pareto solution is known as a multi-objective optimization problem [36]. In the past, many studies have tried to solve the mapping exploration problem and find a pareto solution using heuristic algorithms [37]. In general, a multi-objective optimization cannot be solved by dividing the optimizations. This is because the objectives to be optimized are often in conflict with each other, making it almost impossible to find a solution that optimizes all the objective processes at the same time. Many traditional optimization techniques such as simulated annealing [38] and Tabu searches [39] are mono-objective, and they are difficult to extend to multi-objective cases, as they were not developed to find multiple solutions in a problem.

GAs have been considered as efficient approaches for multi-objective search and optimization [40]. There are different approaches to GA-based multi-objective optimization [41], divided into three types:

Type1 Approaches employing aggregation processes

Type2 Approaches without the notion of a pareto solution

Type3 Approaches based on pareto

Type1 reduces the problem of multi-objective optimization to one of single-objective optimization by aggregating the objective processes. Type2 solves some of the complexities encountered with Type1. However, several problems found in Type1 approaches persist, i.e., missing certain points in concave regions. Currently, Type3 is the most successful approach to solving multi-objective optimizations. The basic algorithm consists of selecting individuals in a pareto solution from the rest of the population. These individuals are then allocated to the highest rank and eliminated from further contention. Another set of individuals in the pareto solution is allocated to the next highest rank by being determined from the remaining population. This procedure is iterated until the whole population is suitably ranked. Multi-objective genetic algorithm uses this approaches [42] In addition, the most extensively used approaches belonging to this class are NSGA-II [43] and SPEA2 [44]. These two approaches are based on multi-objective genetic algorithms. The differences between NSGA-II and SPEA2 concern their ranking of individuals and elimination from further contention.

A considerable amount of research has applied these two algorithms to design space exploration. Hidalgo et al. applied both NSGA-II and SPEA2 to the large design space of possible dynamically allocated data types [45]. Their results show that NSGA-II and SPEA2 could explore the entire pareto solution faster than other heuristic algorithms. Silvano et al. applied NSGA-II to their design space exploration environment to find the pareto solution between performance and power consumption [46]. Their targets were multi-core processors architectures with a combination of 11 configuration parameters. Ascia et al. used SPEA2 to obtain pareto solutions between the execution time and power consumption of a parameterized system-on-a-chip [47]. Their approach was validated on two different parameterized architectures, one based on a RISC processor and the other based on a parameterized VLIW architecture. Compared with an exact approach, their method simulated up to 94% fewer configurations. The pareto solutions found by their approach differed by an average of 0.26% from the ideal pareto solution obtained by the exact approach.

There are other methods for solving multi-objective optimizations. Tiwary et al. proposed a hybrid of a GA and simulated annealing to find *yield-aware* pareto solutions [48]. During the exploration, the GA and simulated annealing are sequentially used. The advantage of this algorithm is that the GA explores the wider design space, while the simulated annealing ensures a better stochastic search in the entire design space. As the GA and simulated annealing are combined, this algorithm is somewhat complicated, making it difficult to apply to other exploration problems.

Wang et.al. applied the algorithm named *ants colony optimization* to solve the problem of function allocation [49]. They have tried to find an allocation which minimizes the execution time of the system under given architecture resources. Compared to the exhaustive search, their method required the small number of simulations to obtain near optimal results.

5.3 Definition of a Pareto Solution

Design space exploration is considered to be a multi-objective optimization problem, as the elements in the objective function $\mathbf{f}(\mathbf{x})$ often compete against each other. However, maximizing the performance of every objective function is not feasible for a multi-objective optimization problem; therefore, *trade-offs* among the different parameters should be considered. Improving the performance in one area might cause a decrease in another competing objective function. In such cases, the concept of pareto efficiency is often used.

In order to define the pareto efficiency, the author says that a set of system performances $\mathbf{f_a}$ is considered more efficient than $\mathbf{f_b}$ if $\mathbf{f_a}$ *dominates* $\mathbf{f_b}$:

$$\mathbf{f_a} \succ \mathbf{f_b} \Longleftrightarrow \forall(f_{a_i} \geq f_{b_i}) \wedge \exists(f_{a_i} > f_{b_i})\ , i = \{1, ..., n\}. \tag{5.1}$$

A set $\mathbf{f}^*$ is considered pareto efficient if it is not dominated by any other set of system performances $\mathbf{f}$. In this work, the surface generated in the given performance space by the complete set of pareto efficient points is called the pareto solution.

5.4 Pareto-Update Search

In this section, the detail of pareto-update search is described. Pareto-update search is used in Mapping Explorer.

5.4.1 Assumptions and Singular Mappings

For process p in the set of processes $\mathbf{P}$, $Exe_{sw}(p)$ and $Exe_{hw}(p)$ give the execution times of software implementation and hardware implementation, respectively. The execution time includes both the computation time and the communication time. The author assumes that:

$$Exe_{sw}(p) > Exe_{hw}(p), \forall p \in \mathbf{P}. \tag{5.2}$$

In general, the execution time of the software implementation is longer than that of the hardware implementation for the same process, so the author considers this assumption to be appropriate.

$Exe_{hw}(p)$ and $Exe_{sw}(p)$ are assumed to be static. They are not affected by the allocation of the other processes. Thus, changes in the communication time by the change of a mapping (e.g., bus arbitration time) are not considered in this study. This assumption is appropriate in the case that the relative error is more important than the absolute error, as the change in the communication time is generally small compared to the entire execution time.

The hardware interconnection area $Area_{com}$ is assumed to be static, even if the number of processes that are implemented on the hardware changes. The hardware area of processors is given by the number of processors num_{cpu} and the hardware area of a single processor $Area_{cpu}$. It is assumed that the hardware area of the entire system is given by:

$$\left(\sum_{p \in \mathbf{P_{HW}}} G_{area}(p) \right) + num_{cpu} \times Area_{cpu} + Area_{com} \tag{5.3}$$

where $\mathbf{P_{HW}}$ is the set of processes allocated to hardware, and $G_{area}(p)$ is the hardware area for process p in the set of processes $\mathbf{P_{HW}}$. It is assumed that any change in the hardware interconnection area will be small compared to the entire hardware area, and so these assumptions are also appropriate.

With the above assumptions, there are two singular mappings: ALLSW and ALLHW. ALLSW allocates all processes to a single processor, whereas ALLHW allocates all processes to dedicated hardware whenever possible. Thus, ALLSW has the smallest hardware area and the longest execution time, and ALLHW has the largest hardware area and the shortest execution time.

As there is no mapping that has a longer execution time or smaller hardware area than ALLSW, this is surely a pareto efficient mapping; in other words, ALLSW must be in the pareto solution. ALLHW is also considered as a pareto efficient mapping, although there is a case in which ALLHW is not pareto efficient. This is because a mapping with the same execution time as ALLHW but less hardware area might exist, as some processes can be allocated to a single processor on the target. Such mappings exist if the total execution time of processes on a processor is shorter than that of processes on dedicated hardware. Thus, it is not always true that ALLHW is a pareto efficient mapping. However, as this situation does not occur often, the author assumes ALLHW to be a pareto efficient mapping in this work.

5.4.2 Generation of New Mappings

First of all, the generation of new mappings is defined. Pareto-update search uses two types of mapping generation: *mapping shift* and *mapping swap*.

Mapping Shift

Mapping shift is a method that generates a set of new mappings from a given input mapping. The basic idea is to change the allocation of one process to other PEs while the allocation of the other processes remains the same. Assuming n as the number of processes and m as the number of PEs, a mapping shift generates $n \times (m-1)$ mappings from an input mapping.

For example, assuming three processes (P1, P2, P3), three PEs (CPU1, CPU2, HW) and an input mapping (P1 = CPU1, P2 = CPU2, P3 = HW), there are a total of six shifted mappings, as shown below:

No.1: (P1 = CPU2, P2 = CPU2, P3 = HW)

No.2: (P1 = HW, P2 = CPU2, P3 = HW)

No.3: (P1 = CPU1, P2 = CPU1, P3 = HW) *duplication of No.1

No.4: (P1 = CPU1, P2 = HW, P3 = HW)

No.5: (P1 = CPU1, P2 = CPU2, P3 = CPU1)

No.6: (P1 = CPU1, P2 = CPU2, P3 = CPU2)

As the target architecture uses homogeneous processors, mappings No.1 and No.3 are the same. Duplicated mappings are not generated in order to reduce the number of simulation.

Mapping Swap

Mapping swap also generates a set of new mappings from an input mapping. The basic idea is to swap the allocation of two processes while those of the other processes remain untouched. As there are several processes, new swapped mappings are generated by swapping the allocation of all combinations of two processes.

Assuming three processes (P1, P2, P3), three PEs (CPU1, CPU2, HW), and an input mapping (P1 = CPU1, P2 = CPU2, P3 = HW), there are a total of three swapped mappings:

No.1: (P1 = CPU2, P2 = CPU1, P3 = HW)

No.2: (P1 = HW, P2 = CPU2, P3 = CPU1)

No.3: (P1 = CPU1, P2 = HW, P3 = CPU2)

Table 5.1: An example set of processes with their execution time (Exe) and hardware area.

Process name	Exe on SW [time units]	Exe on HW [time units]	Hardware area [area units]
A	10	4	10
B	7	5	6
C	6	3	7
D	5	4	8

As for the mapping shift method, any duplicated mappings are not actually generated by a mapping swap.

5.4.3 Overview of the Pareto-Update Search

In this section, the author discusses how to efficiently find a pareto solution. In Table 5.1, an example set of processes (named A, B, C, and D) is shown, together with their execution times for a software implementation and a hardware implementation. The hardware area of each process is also shown, but the hardware area of the processor is not included in the table.

There are data dependencies between A and B, B and C, and C and D. For this reason, all processes are assumed to run in a pipeline. Thus, depending on the process allocation, the entire execution time is calculated as the sum of the execution times of each process. In the following, two cases are discussed separately.

Target I: Single Processor with Dedicated Hardware

In this section, the author considers the use of a mapping shift for a target with a single processor and dedicated hardware. It is assumed that ALLSW is in an ideal pareto solution (see Section 5.4.1). In Figure 5.3, each point indicates a mapping. There are four mappings generated from ALLSW through the mapping shift, labeled 1A, 1B, 1C, and 1D. Shifting

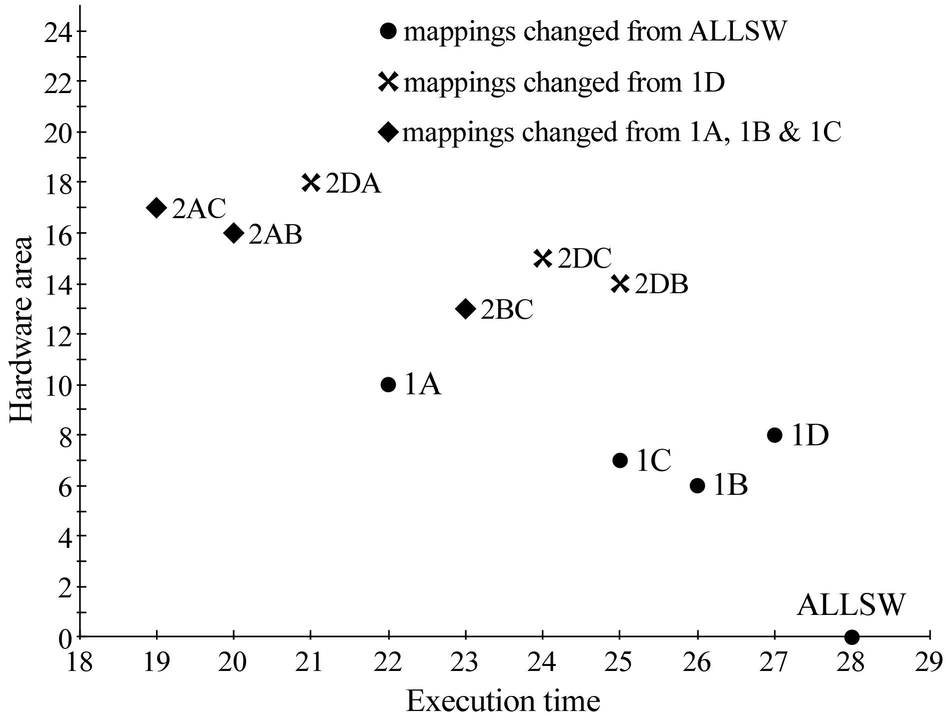


Figure 5.3: Example of exploration from ALLSW.

the allocation of a process from software to hardware reduces the execution time and increases the hardware area for the target. For that reason, all four shifted mappings have a shorter execution time and larger hardware area than ALLSW. Details of their execution time and hardware area are shown in Table 5.2. Among the four mappings, 1A, 1B, and 1C are pareto efficient, whereas 1D is not.

The mappings labeled 2AB, 2AC, and 2BC are generated from 1A, 1B, and 1C. The allocation of the processes A, B, or C is changed to HW from 1A, 1B, and 1C using a mapping shift. Duplicated mappings are not shown in the figure or the table. The mappings labeled 2DA, 2DB, and 2DC are generated from 1D. The allocation of the processes A, B, or C is changed to HW from 1D using a mapping shift. Under the assumptions, shifting the allocation of a process in the second mapping shift will give the same reduction in the execution time and increase in the hardware area as in the first mapping shift. Hence,

Table 5.2: Execution time and hardware area of the entire system for the example set of processes.

ID	AS*	1A	1B	1C	1D	2AB	2AC	2BC	2DA	2DB	2DC
Map A	SW	HW	SW	SW	SW	HW	HW	SW	HW	SW	SW
Map B	SW	SW	HW	SW	SW	HW	SW	HW	SW	HW	SW
Map C	SW	SW	SW	HW	SW	SW	HW	HW	SW	SW	HW
Map D	SW	SW	SW	SW	HW	SW	SW	SW	HW	HW	HW
Exe	28	22	26	25	27	20	19	23	21	25	24
Area	0	10	6	7	8	16	17	13	18	14	15

* AS : ALLSW

there is a high probability that mappings 2AB, 2AC, and 2BC will become pareto efficient because they are generated from pareto efficient mappings. In contrast, mappings 2DA, 2DB, and 2DC cannot become pareto efficient mappings as they are generated from 1D, which is not pareto efficient. Among these six mappings, it is clear from Figure 5.3 that 2AC, 2AB, and 2BC exhibit more pareto efficiency than 2DA, 2DC, and 2DB.

A similar argument can be made for the mappings generated from 2AC, 2AB, 2BC, 2DA, 2DC, and 2DB. Mapping 2BC is not pareto efficient at this time, as it is less efficient than 1A. The mappings generated from 2AC and 2AB have a high probability of being pareto efficient mappings, whereas those generated from other mappings cannot be pareto efficient. As a result, it is not necessary to explore mappings generated from those that are not themselves pareto efficient.

Thus, the author explores an entire pareto solution by repeated generation from pareto efficient mappings. This reduces the number of simulations by ignoring mappings that are not pareto efficient during the exploration. As the exploration starts with ALLSW, the outcome can be an ideal pareto solution.

The same exploration method can be adapted to ALLHW. From ALLHW, the execution time is increased and the hardware area is reduced step-by-step. As mentioned previously, it is not always true that ALLHW is a pareto efficient mapping. In such a case, more pareto efficient mappings are included in those generated from ALLHW. The generated mappings then take the place of ALLHW as the pareto efficient mapping. Thus, the exploration can be started from ALLHW as well as ALLSW.

Target II: Multiple Processors with Dedicated Hardware

The hardware area is certainly reduced when the allocation of processes is changed from HW to SW on a target with a single processor and dedicated hardware. This does not necessarily happen on a target with multiple processors and dedicated hardware. For example, when the allocation of a process in a pareto efficient mapping is changed from CPU1 to CPU2 using a mapping shift, the hardware area remains the same but the execution time of the entire system changes. The same case happens when the allocation of two processes on different processors is changed by a mapping swap. If the execution time decreases, the new mapping becomes the pareto efficient mapping. In this case, the longest execution time among the processors decreases, and the execution time for one of the other processors increases. Thus, iterative mapping generation from the pareto efficient mapping averages the execution times for processors, and that can shorten the entire execution time.

However, there is a possibility that the mapping will become locally optimal, because it may require several process allocations to be changed at the same time in order to change the pareto efficient mapping. The exploration may stop at a locally optimal solution when the allocation of a process is shifted or that of processes is swapped among the processors. Even though the exploration stops at a locally optimal solution, the execution times for processors are averaged. It is ideal that the execution times for the processors are equal. Thus, the mapping that the execution times for the processors are averaged can be a near optimal solution.

When the allocation of a process is changed from SW to HW or vice versa, pareto

```
1: $Pareto \Leftarrow \{ALLSW, ALLHW\}$
2: $BaseMaps \Leftarrow \{ALLSW, ALLHW\}$
3: while $BaseMaps \neq \phi$ do
4:    $GeneratedMaps \Leftarrow LocalSearch(BaseMaps)$
5:    $Pareto \Leftarrow Update(Pareto, GeneratedMaps)$
6:    $BaseMaps \Leftarrow SelectBaseMaps(Pareto)$
7: end while
8: Output $Pareto$
```

Figure 5.4: The pareto-update algorithm.

efficient mappings can be found as described in Section 5.4.3. Therefore, repeating the generation of new mappings from pareto efficient mappings effectively explores a near ideal pareto solution for a target that has multiple processors and dedicated hardware.

5.4.4 The Algorithm of Pareto-Update Search

The algorithm of pareto-update search is depicted in Figure 5.4. In the figure, left arrows indicate an assignment of the set. On lines 1 and 2, *Pareto* and *BaseMaps* are initialized by the set of ALLSW and ALLHW.

Lines 4 to 6 are repeated until *Pareto* reaches a steady state. The local search at line 4 consists of two steps: the generation of new mappings, and the simulation of generated mappings. New mappings are generated from those in *BaseMaps* through mapping shifts and mapping swaps. Note that the same mapping is only generated once during the exploration. After the generation of new mappings, the execution time and hardware area of each generated mapping is estimated by the simulator, and mappings that are not pareto efficient are discarded. The remaining mappings are used to update *Pareto* on line 5.

On line 6, two mappings are selected from *Pareto* as the input to a local search. Starting from ALLSW, the pareto efficient mappings should increase in the hardware area, as the initial value of the hardware area is minimized. Similarly, pareto efficient mappings should move from larger to smaller hardware areas when starting from ALLHW. Thus, one of

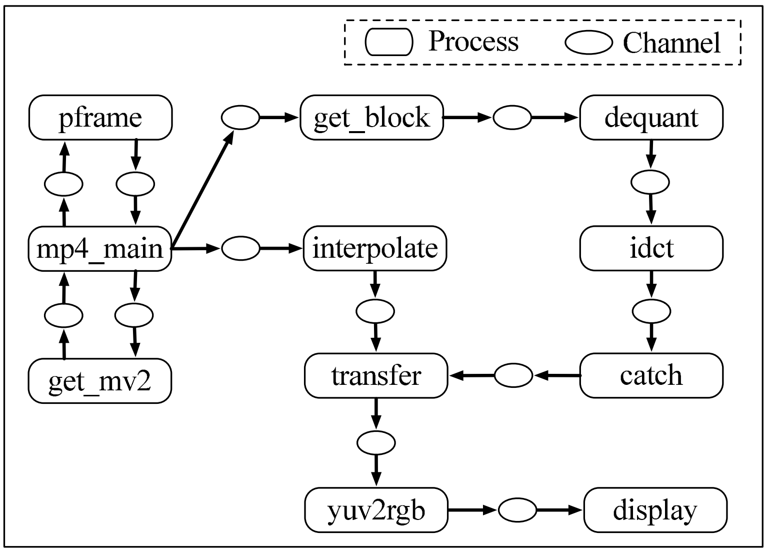


Figure 5.5: Processes in MPEG-4 decoder application.

the mappings has the largest hardware area and the other has the smallest hardware area among the mappings that are not selected in *Pareto*. Under this selection rule, one mapping evolves step-by-step from the largest hardware area to a smaller hardware area, and vice versa for the other mapping. As the exploration proceeds, there may come a point where only one mapping can be selected. In such a case, this mapping is added to *BaseMaps*. There may also be a point where no mapping can be selected. This is the end of the exploration, as all possible mappings have already been generated by mapping shift and mapping swap from the current pareto efficient mappings. Through these three steps, the entire set of pareto efficient mappings and the pareto solution can be found.

5.5 A Case Study on MPEG-4 Decoder Application

5.5.1 Target Systems and Evaluation of Pareto Solutions

The author explored pareto solutions between execution time and hardware area with Mapping Explorer. A target system is MPEG-4 decoder application [50] which consists of 11

Table 5.3: Number of ALUTs.

CPU & com	# ALUTs	Process	# ALUTs	Process	# ALUTs
Nios II	2,062	pframe	2,839	get_mv2	4,281
Interconnect	280	get_block	2,601	dequant	1,188
—	—	idct	3,199	catch	748
—	—	interpolate	6,613	transfer	3,970
—	—	yuv2rgb	1,933	display	1,080

processes as shown in Figure 5.5. The arrows in the figure indicate the data dependencies of blocking channels among the processes. The process named mp4_main can only be allocated to processors, and other processes can be allocated to processors or dedicated hardware.

The author explored four architectures named “Single-core-HW”, “Dual-core-HW”, “Triple-core-HW”, and “Quad-core-HW” which have one, two, three, and four Nios II processors [29], respectively, and all these architectures have dedicated hardware as PEs. For example, Dual-core-HW has two processors and dedicated hardware as PEs. There are three PEs in total onto which processes are mapped.

In order to estimate the execution time, the author used a simulation tool developed by Shibata et al. [11]. In order to estimate the hardware area, the author first measured the hardware area of all processes and processors on Altera Stratix II FPGA board. Depending on the mapping, the hardware area is calculated by formula 5.3 with measured number of ALUTs shown in Table 5.3. Note that the hardware area of communications (e.g., bus) are included in Interconnect.

The author used two algorithms to find pareto solutions between execution time and hardware area on all target architectures. The algorithms are exhaustive search and pareto-update search. The author compared pareto solutions found by pareto-update search and exhaustive search in terms of two items, the number of simulated mappings and the error of

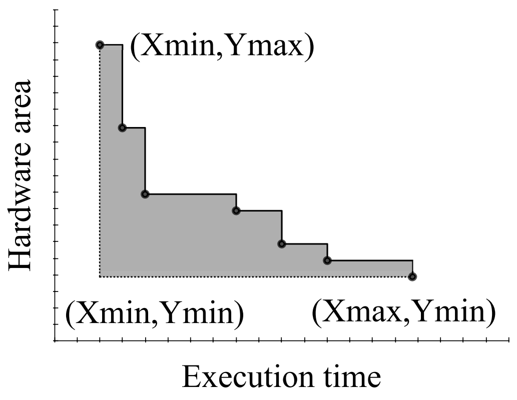


Figure 5.6: Calculation on the area of a pareto solution.

pareto solution. The number of simulated mappings indicates how efficient the algorithm is. The error is evaluated by the area of pareto solution [51]. Figure 5.6 shows how to calculate the area of a pareto solution. Two points, (Xmin, Ymax) and (Xmax, Ymin) in the figure, are brought from the pareto solution of exhaustive search. The area of a pareto solution is painted in gray, and it is enclosed with dotted lines and solid lines. Two dotted lines are drawn from (Xmin, Ymin) to (Xmin, Ymax) and from (Xmin, Ymin) to (Xmax, Ymin). The error is a ratio of the areas of pareto solutions explored by pareto-update search and exhaustive search.

5.5.2 Exploration Results

Figure 5.7 shows the pareto solutions between execution time and hardware area on Quad-core-HW. In the figure, dots with circle and dots with cross are mappings in pareto solutions of exhaustive search and that of pareto-update search, respectively. The figure indicates that pareto-update search found almost the same pareto solution as that exhaustive search found. Hence, it can be said that pareto-update search could find a near ideal pareto solution.

The details of the exploration results are shown in Table 5.4. The results indicate that pareto-update search is very efficient and accurate enough to find a pareto solution between

Table 5.4: Exploration results on MPEG-4 decoder application.

architecture	algorithm	# simulated mappings	error [%]
Single-core-HW	Exhaustive search	1,024	—
	Pareto-update search	346 (33.789%)	0.000
Dual-core-HW	Exhaustive search	59,049	—
	Pareto-update search	1,485 (2.515%)	0.066
Triple-core-HW	Exhaustive search	524,800	—
	Pareto-update search	2,607 (0.497%)	0.091
Quad-core-HW	Exhaustive search	1,657,470	—
	Pareto-update search	2,899 (0.175%)	0.092

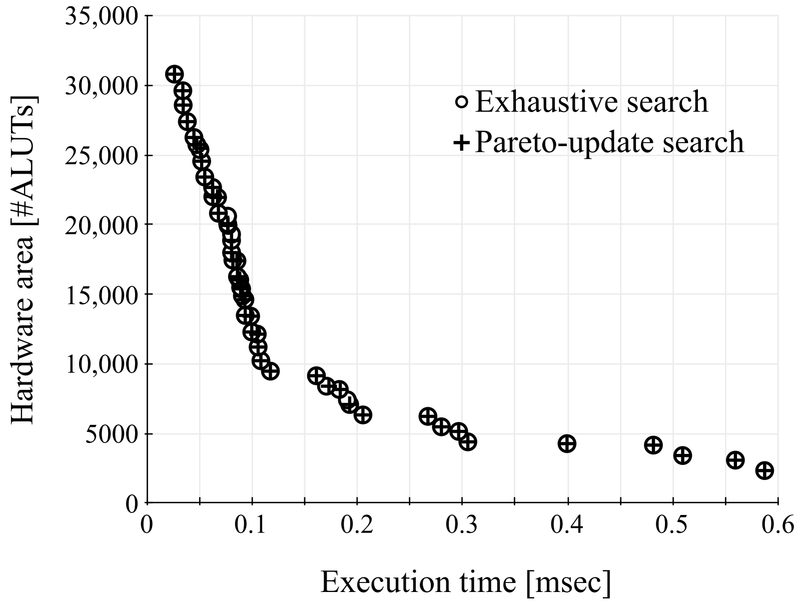


Figure 5.7: Pareto solutions of MPEG-4 on Quad-core-HW architecture.

execution time and hardware area. The number of simulated mappings with pareto-update search is very small. Compared to exhaustive search, the numbers of simulated mappings are reduced to 33.8% and 0.18% which are the worst and the best reduction, respectively. Since exploration time is proportional to the number of simulated mappings, pareto-update search is very fast to find a pareto solution. Furthermore, applying pareto-update search to wider design space is more effective since the ratio of the number of simulated mappings decreases on wider design space. Therefore, pareto-update search seems to be useful to explore the design space of complex systems.

As mentioned before, the goal is to efficiently find a pareto solution whose error is less than the error of simulators themselves. The errors of pareto solutions found by pareto-update search are very low, even the worst error is less than 0.1% in this case study. In particularly, the error is 0.0% on Single-core-HW. Since the allocation of each process is either SW or HW on Single-core-HW, mappings generated by mapping shift covers the all mappings which are in the pareto solution. Hence, mappings in the pareto solution of exhaustive search are generated during the local search on Single-core-HW. Even though pareto-update search only generates shifted and swapped mappings, the mappings found by pareto-update search can be the same as the ideal pareto solution on Single-core-HW.

Additionally, an advantage of pareto-update search is that the error does not tend to increase exponentially even when the number of simulated mappings of exhaustive search increases exponentially. This is important to explore more complex systems. Pareto-update search not only reduces the number of simulated mappings but also finds a pareto solution with very small error. Therefore, pareto-update search is notably effective to find a pareto solution between execution time and hardware area.

5.6 Conclusion

The time to design embedded systems increases along with their complexity. The increase of complexity has brought unacceptable long times to design space exploration. In this chapter, the author proposed pareto-update search which is a fast exploration method to find a pareto solution between execution time and hardware area. In addition, the author developed a tool, Mapping Explorer, which uses pareto-update search in order to realize efficient system-level design. Pareto-update search starts the exploration with two mappings whose probability of being in the pareto solution is high. The exploration repeats both local search for mappings in the pareto solution and the update of the pareto solution with the result of local search. The proposed method reduced the number of simulated mappings to 0.18% compared to that of exhaustive search on a MPEG-4 decoder application. Moreover, the error of the pareto solutions is less than 0.1%. Therefore, pareto-update search is notably effective to find a pareto solution between execution time and hardware area.

Chapter 6

Improvement Analyzer: Performance Improvement with Bottleneck Analysis

6.1 Introduction

For embedded systems, it is important to satisfy the requirements of system performances such as execution time and hardware area. In Chapter 5, the author proposed pareto-update search to find mappings on a pareto solution between execution time and hardware area. However, mappings on a pareto solution may not satisfy the requirements of the system performances. In this manner, system designers are facing a problem that they must efficiently design a system satisfying the requirements of the system performances.

Figure 6.1 shows an example of design flow for embedded systems with dedicated hardware. It starts from changing software description to system-level description. Then, designers conduct exploration of software/hardware partitioning by Mapping Explorer. During the exploration of software/hardware partitioning, system designers try to find a mapping that satisfies the requirements of the system performances by changing the allocation of processes. This process is accelerated by Mapping Explorer as described in Chapter 5.

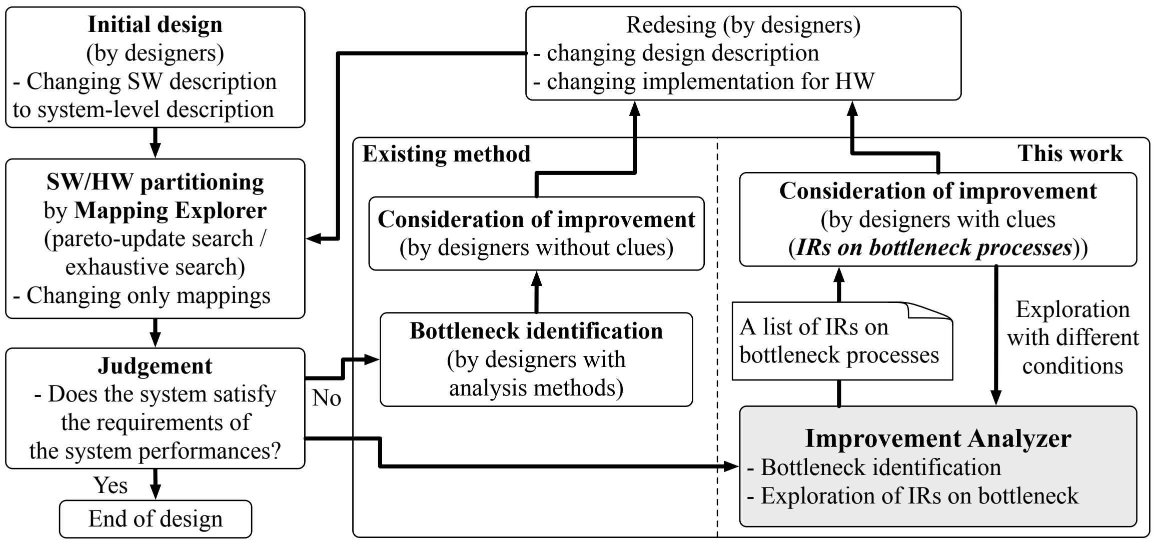

Initial design
(by designers)
- Changing SW description to system-level description
SW/HW partitioning by Mapping Explorer
(pareto-update search / exhaustive search)
- Changing only mappings
Judgement
- Does the system satisfy the requirements of the system performances?
Yes
No
End of design
Redesing (by designers)
- changing design description
- changing implementation for HW
Existing method
This work
Consideration of improvement
(by designers without clues)
Bottleneck identification
(by designers with analysis methods)
Consideration of improvement
(by designers with clues (IRs on bottleneck processes))
A list of IRs on bottleneck processes
Exploration with different conditions
Improvement Analyzer
- Bottleneck identification
- Exploration of IRs on bottleneck

Figure 6.1: Entire design flow.

After the exploration of software/hardware partitioning, the designers must check whether a mapping satisfies all requirements of the system performances because exploration of software/hardware partitioning may not find a mapping that satisfies them. For example, a mapping satisfies the requirements of hardware area but may not satisfy that of execution time. From this mapping, changing an allocation of a process from software to hardware makes execution time faster. However, it brings bigger hardware area. As the result, new mapping satisfies the requirements of execution time but may not satisfy that of hardware area. Thus, exploration of software/hardware partitioning does not always find a mapping that satisfies all requirements of the system performances. In such case, designers need improve the system-level description.

There are two big problems to improve the design description. The first problem is identification of bottlenecks on the system. In the existing design method, designers identify the bottlenecks using analysis tools. Fei et al. divided execution logs into particular behavior groups and analyze the behavior groups [12]. Valle et al. proposed a method to make logging of the system performances easy [13]. The second problem is identification of improvement rates (IRs, hereafter) of the bottlenecks. In the existing design methods, designers must identify how much they have to improve the bottlenecks to satisfy the requirements of the system performances. Then, they consider how to change the system-level description to improve the system performances. The existing design methods take more time of designers to identify how much they must improve the bottlenecks than to consider how to change the system-level description.

In this chapter, the author proposes Improvement Analyzer which is a tool to automatically identify not only the bottlenecks but also a list of IRs of the bottlenecks that are necessary to satisfy the requirements of the system performances. With Improvement Analyzer, designers no longer identify how much they must improve the bottlenecks because Improvement Analyzer automatically identifies that. It is ideal for designers to know the essential IRs of the bottlenecks. In addition, Improvement Analyzer lists up several candidates to improve the system performances. Thus, Improvement Analyzer brings shorter time to identify the IRs of the bottlenecks, and designers can take more time to consider various ways to improve the system-level description.

The main contribution is to propose a method to explore the IRs of the bottlenecks. In addition, Improvement Analyzer explores IRs of not only execution time but also hardware area. A case study on AES shows the effectiveness of Improvement Analyzer.

The rest of this chapter is organized as follows. Section 6.2 presents entire design flow. Section 6.3 presents the detail of Improvement Analyzer. Section 6.4 shows the effectiveness of Improvement Analyzer through the case studies, and section 6.5 concludes this chapter.

6.2 Overview of the Design Flow

Figure 6.1 shows the entire design flow. The beginning of design flow is initial design of the system. At this step, designers make a system-level description. In detail, designers are assumed to make a system-level description of Extended SystemBuilder. Extended SystemBuilder takes three inputs, a target architecture template, a mapping, and a system-level description that consists of processes written in C and channels among the processes. Depending on the mapping and the target architecture template, Extended SystemBuilder automatically generates the implementation of processes and channels.

After initial design step, designers explore software/hardware partitioning by Mapping Explorer. With Mapping Explorer, designers can easily get different implementations by only changing mappings. Thus, it can accelerate the exploration of software/hardware partitioning. At this step, designers evaluate the execution time and hardware area of various system implementations by only changing mappings. Note that designers do not change the system-level description at this step. If a mapping that satisfies the requirements of the execution time and hardware area is found during the exploration of software/hardware partitioning, the exploration ends. On the other hand, there may be no mapping that satisfies the requirements of the system performances in spite of evaluating all possible mappings. In such case, improvement of system-level description is the only way to get better system performances.

In order to efficiently improve the system-level description, Improvement Analyzer is used at next step. Improvement Analyzer identifies the bottlenecks of the system performances at first. Then, IRs of the bottlenecks are explored. IRs are ratio of improvement on the bottleneck that are necessary to satisfy the requirements of the system performances. After the exploration of IRs, designers get a list of candidates showing IRs. With the list, designers consider how to change the system-level description to improve the system performances. If the list is not helpful to improve the system performances, designers can try to explore IRs of the bottlenecks with different conditions of the exploration.

After the consideration of improvement, designers change the system-level description at redesign step. Then, they again explore software/hardware partitioning with improved system-level description. The design flow repeats the exploration of software/hardware partitioning, exploration of IRs of the bottlenecks, and changing system-level description with explored IRs until designers get a mapping that satisfies the requirements of the system performances.

6.3 Exploration of Improvement Rate of the Bottleneck

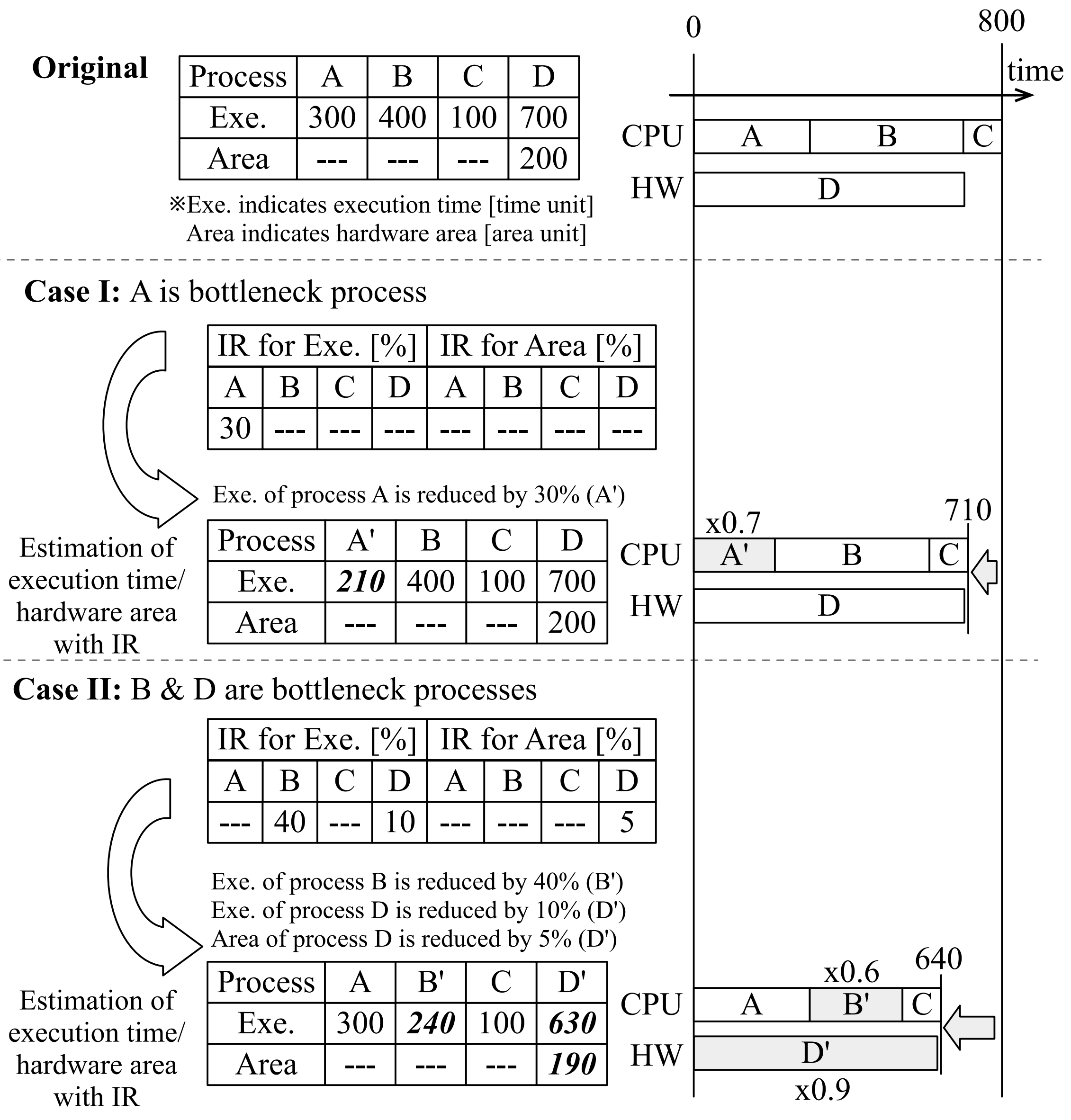
This section describes how Improvement Analyzer identifies bottlenecks and a list of IRs.

6.3.1 Definition of Bottleneck Process

A process X is defined as a bottleneck process if reduction of the execution time of process X shortens the entire execution time of the system without any change of the mapping.

Figure 6.2 shows an example of bottleneck processes. The example has four processes. The original execution time of processes A, B, C, and D are 300, 400, 100, and 700, respectively. Processes A, B, and C are mapped to a processor (CPU), and process D is mapped to dedicated hardware (HW). The original entire execution time is 800 as shown on the right side of the figure.

Case I shows that process A is a bottleneck process. The execution time of process A is assumed to be 210 (A'), that is 70% of process A. The entire execution time of Case I is reduced to 710 because the execution time of process A' is applied. Reducing the execution time of process A causes to shorten the entire execution time. Therefore, process A is a bottleneck process under the definition.



Original

Process	A	B	C	D
Exe.	300	400	100	700
Area	---	---	---	200

※Exe. indicates execution time [time unit]
Area indicates hardware area [area unit]

Case I: A is bottleneck process

IR for Exe. [%]				IR for Area [%]			
A	B	C	D	A	B	C	D
30	---	---	---	---	---	---	---

Exe. of process A is reduced by 30% (A')

Process	A'	B	C	D
Exe.	***210***	400	100	700
Area	---	---	---	200

Case II: B & D are bottleneck processes

IR for Exe. [%]				IR for Area [%]			
A	B	C	D	A	B	C	D
---	40	---	10	---	---	---	5

Exe. of process B is reduced by 40% (B')
Exe. of process D is reduced by 10% (D')
Area of process D is reduced by 5% (D')

Process	A	B'	C	D'
Exe.	300	***240***	100	***630***
Area	---	---	---	***190***

0
800
time
CPU
A
B
C
HW
D
Estimation of execution time/ hardware area with IR
x0.7
710
CPU
A'
B
C
HW
D
Estimation of execution time/ hardware area with IR
x0.6
640
CPU
A
B'
C
HW
D'
x0.9

Figure 6.2: An example of IRs and bottleneck processes.

With the definition, several processes may become bottleneck processes at the same time as shown in Case II. The entire execution time also becomes shorter than original one when the execution time of processes B and D are assumed to be 240 (B') and 630 (D'), respectively. Thus, processes B and D are bottleneck processes.

6.3.2 Definition of Improvement Rate

IR indicates the ratio to shorten the execution time or to reduce the hardware area of a process compared to original one. Examples of IR are also shown in Figure 6.2. Each process has two types of IR, one for execution time and the other for hardware area. For example, the original execution time and hardware area of process D are 700 and 200, respectively. In Case II, it has 10% of IR of execution time and 5% of IR of hardware area. Thus, the execution time and hardware area of process D are assumed to be 630 and 190, respectively.

6.3.3 Exploration of the Improvement Rates on Bottleneck Processes

Under the definition of a bottleneck process and that of IR, estimating the entire execution time with IRs identifies bottleneck processes. If the entire execution time is reduced, processes that have IRs are bottlenecks. Thus, increasing the value of IRs clarifies whether the processes are bottlenecks or not.

Figure 6.3 shows an exploration flow of Improvement Analyzer to identify IRs of bottleneck processes. The inputs are a mapping which is used to explore the IRs, and the requirements of execution time and hardware area. The output is a set of IRs that is needed to satisfy the requirements of the system performances.

At the beginning, all IRs in a set of IRs (base_set_IRs) are initialized to 0% (Initialize). At the same time, evaluation value of base_set_IRs (base_set_IRs.eval) is initialized to max-

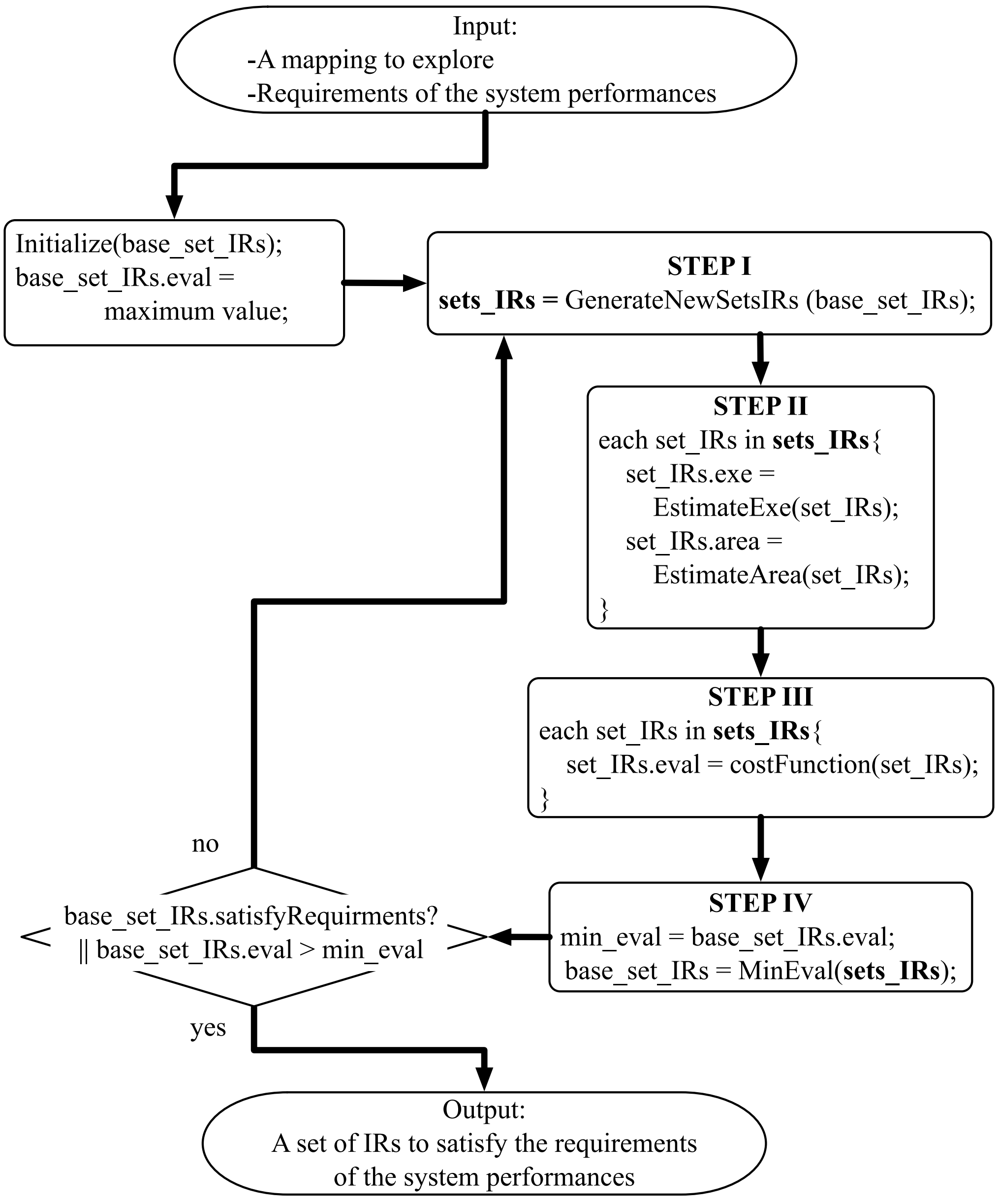

Input:
-A mapping to explore
-Requirements of the system performances
Initialize(base_set_IRs);
base_set_IRs.eval = maximum value;
STEP I
sets_IRs = GenerateNewSetsIRs (base_set_IRs);
STEP II
each set_IRs in sets_IRs{
set_IRs.exe = EstimateExe(set_IRs);
set_IRs.area = EstimateArea(set_IRs);
}
STEP III
each set_IRs in sets_IRs{
set_IRs.eval = costFunction(set_IRs);
}
STEP IV
min_eval = base_set_IRs.eval;
base_set_IRs = MinEval(sets_IRs);
base_set_IRs.satisfyRequirments?
|| base_set_IRs.eval > min_eval
no
yes
Output:
A set of IRs to satisfy the requirements
of the system performances

Figure 6.3: Exploration flow of Improvement Analyzer to identify IRs of bottleneck processes.

imum value. After the initialization, four steps are repeated. At step I, new sets of IRs are generated from A to increase values of IRs. At step II, execution time and hardware area are estimated with generated sets of IRs. At step III, all generated sets of IRs are evaluated by the cost function because increasing the same value of IR causes to produce an unrealizable IR such as 100%. At step IV, the best set of IRs is selected for further exploration. After step IV, the exploration ends if a set of IRs satisfies the requirements of the system performances or there is no better set of IRs.

After an exploration, designers get a mapping and its best set of IRs of bottleneck processes. The best set of IRs indicates how much bottleneck processes should be improved to satisfy the requirements of the system performances on the mapping. In addition, exploration on different mappings may bring better ones. Thus, designers can easily find the best mapping and its set of IRs of bottleneck processes among several pairs of them.

The detail of each step is described in following.

STEP I

From base_set_IRs, new sets of IRs are generated by a function GenerateNewSetsIRs. Only an IR in base_set_IRs is increased at once. Since a new sets of IRs are generated from each process's IR of execution time and hardware area, the number of new sets is twice the number of processes in maximum. Before the exploration, designers must define static increasing value of IR.

STEP II

Execution time and hardware area are estimated for all sets of IRs in sets_IRs. A trace-based estimation tool [11] is assumed to be used to estimate the entire execution time with IRs. The tool usually takes profiles of execution time of processes as input. For that, the tools can estimate the entire execution time which is applied IRs by reconfiguring the profiles of execution time depending on IRs. Hardware area is estimated by summation of the area of hardware modules. Area of hardware module is reduced when the hardware area is estimated by applying IRs.

STEP III

Sets of IRs are evaluated by the cost function (costFunction) to determine the best set of IRs. Better set of IR is assumed to have smaller value. Without this evaluation, only an IR of a process may increase. This causes to produce an unrealizable value of IR such as 100%. The definition of the cost function is described in section 6.3.4.

STEP IV

A function, MinEval, returns a set of IRs that has minimum evaluated value. Because better set of IRs has smaller value, this step selects the best set of IRs among the generated sets of IRs. The selected set of IRs become base_set_IRs for further exploration.

6.3.4 Detail of Cost Function

Increase of IRs certainly satisfies the requirements of the system performances. An example is that IRs of 100% for execution time on all processes bring 0 second of the entire execution time. It is impossible to realize such IRs. From this point, a set of minimum IRs that satisfies the requirements of the system performances should be explored.

In order to determine a better set of IRs, the author proposes a cost function. It is assumed that smaller value of cost function is better. The main purpose of cost function is to find a set of minimum IRs that satisfies the requirements of the system performances. For that reason, the value of cost function must decreases if the estimated system performances which are applied IRs gets close to the requirements of the system performances. On the other hand, the value must increase in order to avoid impossible rates if IRs get bigger.

In addition, some processes may be no longer improved. Such processes should not be listed to be improved more. Thus, the easiness of improvement on all processes should be user-settable. For that reason, cost function should have three features below:

1. The value gets smaller (better) if the estimated performances with a set of IRs get close to the requirements.
2. The value should be bigger on large IRs.
3. Designers can set easiness of improvement on all processes separately.

In the following, detail of cost function is described. For the first feature, distance between estimated values and the requirements of the system performances is used. For the second feature, penalty depending on IRs is added to cost function. The third feature is handled by introducing value of easiness to improve process.

There are three kind of parameters determined by designers before the exploration starts. Note that *xxx* should be either "exe" or "area" indicating execution time or hardware area, respectively.

- $target_{xxx}$: value of the requirements of exe/area
- $ease_{p\_xxx}$: value of easiness of exe/area to improve process p
- max_{xxx} : maximum value of exe/area

The inputs of the cost function are a set of IRs ($rate_{p\_xxx}$) and estimated values of execution time (est_{exe}) and hardware area (est_{area}). The cost function consists of penalty ($penal$) of IRs and distance (dis) between the estimated values and the requirements of the system performances. From formula (6.1) to formula (6.3) show calculation of dis. Because the proposed method deals with two requirements, execution time and hardware area, distances for execution time (dis_{exe}) and hardware area (dis_{area}) are calculated as shown in formula (6.2) before calculating dis.

$$dif_{xxx} = est_{xxx} - target_{xxx} \tag{6.1}$$

$$dis_{xxx} = \begin{cases} 3 \times dif_{xxx} + 0.1 & (dif_{xxx} > 0) \\ 0.001 \times dif_{xxx} & (others) \end{cases} \tag{6.2}$$

With dis_{exe} and dis_{area}, distance dis is calculated by formula (6.3).

$$dis = \frac{dis_{exe} + dis_{area}}{2} \tag{6.3}$$

The calculation of penalty (*penal*) is shown below. In the formula, $rate_{p\_exe}$ and $rate_{p\_area}$ indicate the IRs of execution time and hardware area of process $p \in P$, respectively. Note that P is a set of processes in the system-level description. Standard value of easiness to improve process p ($ease_{p\_xxx}$) is one. If it is bigger than one, it means that the process p is hard to improve. If it is smaller than one, it means that the process p is easy to improve. Values of easiness to improve process must be determined by designers before the exploration starts. The penalty of execution time ($penal_{exe}$) and hardware area ($penal_{area}$) are calculated by formula (6.4).

$$penal_{xxx} = \frac{\sum_{p \in P} (rate_{p\_xxx} * ease_{p\_xxx})^3}{|P|} \tag{6.4}$$

The total penalty (*penal*) is given by formula (6.5). It is an average of penalties of execution time ($penal_{exe}$) and hardware area ($penal_{area}$) given by (6.4).

$$penal = \frac{penal_{exe} + penal_{area}}{2} \tag{6.5}$$

Finally, the cost function returns its value (*eval*) by formula (6.6).

$$eval = dis + penal \tag{6.6}$$

6.4 A Case Study

This section shows a case study on AES encryption and decryption application (hereafter, AES application) in CHStone [34]. The author improved the system performances of AES application twice along with the design flow shown in Figure 6.1.

The target architecture is Altera Stratix II FPGA board which has a single Nios II soft-core processor and dedicated hardware [29]. The system performances are evaluated in terms of execution time and hardware area. The author used the number of ALUTs in FPGA as hardware area.

6.4.1 Calibration of Coefficients in the Cost Function

Before the author improved AES application, the author had calibrated the coefficients in cost function by MPEG-4 decoder application that consists of 11 processes. There are three target architectures. The author used five combinations of the requirements, and the author explored five mappings on each combination of requirements on target architecture. In total, the author explored 75 patterns with various coefficients. Note that the easiness of improvement was set to one on all of them. From the various tries, the author found out the value of coefficients used in formula (6.2).

6.4.2 Initial Design of AES Encryption and Decryption

AES application is written in C language, and the number of lines is 716. In addition, the number of “if”, “switch”, and “for” sentences are 26, 10, and 24, respectively.

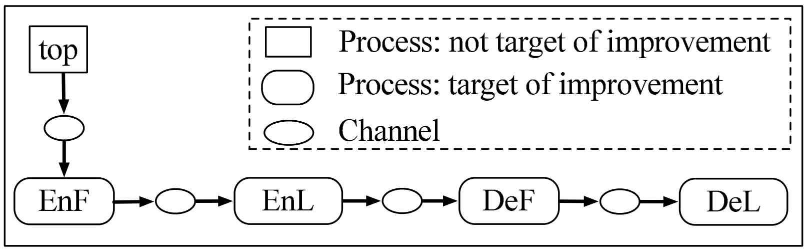

top
Process: not target of improvement
Process: target of improvement
Channel
EnF
EnL
DeF
DeL

Figure 6.4: The structure of AES encryption and decryption application.

Table 6.1: Summary of improvement on AES application. (Exe. and Area indicate execution time and hardware area, respectively)

	Allocation of process / Difference from Initial †				Measured	
Design	EnF	EnL	DeF	DeL	Exe. [sec]	Area [#ALUTs]
Initial	HW	HW	HW	HW	1.31	19,244
Imp1	SW	HW	HW	HW / M, L	0.83	16,573
Imp2	HW / S	HW / S	HW	SW / M	1.29	12,501

†M: reducing memory accesses, L: loops are unrolled,
S: Option of high-level synthesis tool is set to use a single ALU

At first, the author divided the AES application into five processes as shown in Figure 6.4 so that the author can use Extended SystemBuilder and explore software/hardware partitioning. The global arrays in C description are changed to shared-memory communication. Because the process named "top" is the sequencer of the application, it is not a target of improvement and exploration of software/hardware partitioning. EnF and EnL are the first half and last half of encryption, respectively. Also, DeF and DeL are the first half and last half of decryption, respectively. AES application repeats the encryption and decryption for 100 times. On each time, it encrypts and decrypts 10 blocks of data consisting of 16 integers. Note that the author did not optimize the system-level description at this step. The system-level description without any optimization is shown as Initial.

6.4.3 Improvement of System-Level Description

The aim to improve AES application is that the hardware area reduces while the execution time remains less than Initial design.

In this case study, the author improved system-level description twice along with the proposed design flow as shown in Figure 6.1. Table 6.1 shows a summary of improvements on AES application. Initial, Imp1, and Imp2 indicate initial design, first improved design, and second improved design, respectively. The first round of improvements indicates getting Imp1 from Initial. The second round of improvements indicates getting Imp2 from Imp1.

After two rounds of improvements, the author finally got a design named Imp2 (1.29 seconds of the execution time and 12,501 in #ALUTs) that has shorter execution time than Initial design (1.31 seconds of the execution time and 19,244 in #ALUTs). Imp2 design also has the hardware area which is reduced by 35% from Initial design. The following presents the details of two rounds of improvements.

First Round of Improvement

At first, the author explored software/hardware partitioning of Initial design. As AES application has four processes that can be allocated to software and hardware, so there are 16 mappings in total. The mappings were generated by Mapping Explorer with exhaustive search. The author evaluated the system performances by implementing all generated mappings onto the FPGA board using Extended SystemBuilder. The author found that the shortest execution time and the number of ALUTs were 1.31 seconds and 19,244, respectively, as shown Initial in Table 6.1. The author decided that the aim to improve AES application is that the hardware area reduces while the execution time keeps faster than 1.31 seconds.

Table 6.2: A list of IRs for Initial design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 18,000)

	Mapping				IR of Exe. [%]]				IR of Area [%]			
ID	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL
1	SW	HW	HW	HW	—	—	—	5	—	5	—	5
2	SW	HW	SW	HW	25	—	15	5	—	—	—	—
3	HW	SW	SW	HW	—	40	15	5	—	—	—	—

In order to improve the system performances, the author decided the requirements of 1.3 seconds for the execution time and 18,000 for the hardware area. For all processes, the easiness to improve process was set to one. From the exploration of software/hardware partitioning of Initial design, the author found 11 mappings in the pareto solution between execution time and hardware area. The author explored IRs for those 11 mappings under the requirements of the system performances described above by Improvement Analyzer.

After the exploration, the author had 11 sets of IRs. The best three results are shown in Table 6.2. In the table, "Mapping", "IR of Exe.", and "IR of Area" indicate the allocation of processes, IRs of execution time, and IRs of hardware area, respectively.

From the result of mapping ID1, DeL is identified as a bottleneck for execution time. Its execution time must be reduced 5% in order to satisfy the requirements of the system performances. By comparing three results, ID1, ID2, and ID3, mapping ID1 had least IRs in total. From that reason, the author selected the mapping ID1. Then, the author tuned the source code in order to reduce the number of memory accesses in DeL. The author also unrolled the loop instructions in DeL in order to improve the performance because it is implemented on dedicated hardware. Then the author implemented the tuned design of mapping ID1 shown in Table 6.2 onto the FPGA board. This design is called Imp1.

Table 6.3: A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 14,000)

	Mapping				IR of Exe. [%]]				IR of Area [%]			
ID	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL
4	HW	HW	HW	SW	—	—	—	—	5	35	—	—
2	SW	HW	SW	HW	20	—	15	—	—	—	—	10
5	SW	HW	HW	SW	5	—	—	50	—	—	—	—

Second Round of Improvement

The author explored software/hardware partitioning in the system-level description of Imp1. The mappings were generated by Mapping Explorer with exhaustive search. Then they were implemented onto the FPGA board by Extended SystemBuilder. After the exploration, the pareto solution consists of 10 mappings. Among the 10 mappings, the mapping of Imp1 in Table 6.1 had the execution time of 0.83 seconds and hardware area of 16,573 in #ALUTs. This design actually satisfied the requirements of the system performances. However, the execution time of Imp1 (0.83 seconds) was shorter than the requirement of the execution time (1.3 seconds). In general, it is possible to reduce hardware area by lowering the performance of execution time. For that reason, the author decided to lower the performance of execution time in order to reduce the hardware area. Since the hardware area of Imp1 was decreased 2,500 from that of Initial, the author decided the requirement of 1.3 seconds for the execution time and the requirement of 14,000 for the hardware area which was decreased 2,500 from that of Imp1 (16,573).

The author explored IRs of those 10 mappings in pareto solution. The best three results are shown in Table 6.3. From the result, mapping ID4 had the least IRs in total. In addition, mapping ID4 only required to reduce hardware area. In general, a easy way to reduce hardware area is to change the option of high-level synthesis tool. In this way, the designer does not have to change the system-level description. For that reason, the author decided to change mapping ID4 and the option of high-level synthesis tool of processes EnF and

Table 6.4: A list of IRs for Initial design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 17,000)

	Mapping				IR of Exe. [%]]				IR of Area [%]			
ID	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL
1	SW	HW	HW	HW	—	—	—	5	—	5	—	20
2	SW	HW	SW	HW	25	—	15	5	—	—	—	—
3	HW	SW	SW	HW	—	40	15	5	—	—	—	—

EnL. In detail, the option was changed from “a use of several ALUs” into “a use of a single ALU” in order to reduce hardware area. Note that only the option of high-level synthesis tool was changed. This design is called Imp2. Then, the author explored software/hardware partitioning of Imp2, and the author got a mapping whose execution time and hardware area were 1.29 seconds and 12,501 in #ALUTs, respectively, as shown Imp2 in Table 6.1. The mapping satisfied the requirements, and the system design ended.

6.4.4 Improvement of System-Level Description with Different Requirements

In previous section, the requirements of the system performances were decided by the author’s own. In order to show an efficiency of Improvement Analyzer, this section indicates the possibility to get almost the same design as Imp2 by selecting different requirements of the system performances. In detail, the author decided hardware requirement of 17,000 for the first round of improvement. The author also decided hardware requirements of 15,000 and 13,000 for the second round of improvement.

On the first round of improvement, Table 6.4 shows the result of exploration of IRs under the requirements of 1.3 seconds for the execution time and 17,000 for the hardware area. From the result, mapping ID1 had the least IRs in total. In addition, DeL was identified as a bottleneck of execution time on mapping ID1, and EnL and DeL were identified as a bottleneck for hardware area on mapping ID1. Although the value of IR is different,

Table 6.5: A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 15,000)

	Mapping				IR of Exe. [%]]				IR of Area [%]			
ID	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL
4	HW	HW	HW	SW	—	—	—	—	—	15	—	—
2	SW	HW	SW	HW	20	—	15	—	—	—	—	—
5	SW	HW	HW	SW	5	—	—	50	—	—	—	—

Table 6.6: A list of IRs for Imp1 design. (requirement of execution time: 1.3 seconds, requirement of hardware area: 13,000)

	Mapping				IR of Exe. [%]]				IR of Area [%]			
ID	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL	EnF	EnL	DeF	DeL
2	SW	HW	SW	HW	20	—	15	—	—	5	—	20
5	SW	HW	HW	SW	5	—	—	50	—	—	—	—
6	HW	SW	HW	SW	—	55	—	—	—	—	—	—

identified bottleneck processes are the same as Table 6.2. Thus, it is possible to get the same design with Imp1 shown in Table 6.1.

On second round of improvement, Table 6.5 shows the result of exploration of IRs under the requirements of 1.3 seconds for the execution time and 15,000 for the hardware area. From the result, mapping ID4 had the least IRs in total. In addition, mapping ID4 was only required to reduce hardware area of process EnL. In order to reduce hardware area of process EnL, the option of high-level synthesis tool was changed from "a use of several ALUs" into "a use of a single ALU". Note that only the option of high-level synthesis tool was changed. In the result, the execution time and the hardware area of mapping ID4 were 1.29 seconds and 13,037 in #ALUTs, respectively. The result is almost the same with Imp2 shown in Table 6.1 which is the result under the requirements of 1.3 seconds for the execution time and 14,000 for the hardware area.

Moreover, on the second round of improvement, Table 6.6 shows the result of exploration of IRs under the requirements of 1.3 seconds for the execution time and 13,000 for the hardware area. From the result, mapping ID2 had a lot of processes to be improved.

Table 6.7: The quantity of work spent by designers.

Method	Time to analyze bottleneck	Action items for improvement
Existing method	7 hours	Consideration of how to improve, Exploration of IRs
Improvement Analyzer	0 hour	Consideration of how to improve

In addition, mappings ID5 and ID6 have a few processes to be improved, and they need more than 50% of improvement in total. Thus, the author judged that it was very difficult to improve the system-level description. In such case, the designers should relax the requirements of the system performances and explore IRs. In other word, it is possible to explore IRs under the requirements of 14,000 and 15,000 for the hardware area by relaxing the requirement of the hardware area. As a result, the designers could get the same design with Imp2 shown in Table 6.1.

Therefore, even if the designers select different requirements from the two rounds of improvement in section 6.4.4, the designers could improve the system-level description as same as Imp2 shown in Table 6.1.

6.4.5 Design Efficiency

This section compares design efficiency between existing method and Improvement Analyzer. As shown in Figure 6.1, the way to analyze bottleneck processes is different between existing method and Improvement Analyzer. With existing method, the designers analyze bottleneck processes by visualizing the execution logs. By this way, IRs cannot be explored. Thus, designers must explore IRs by themselves. On the other hand, Improvement Analyzer automatically explores IRs.

Table 6.7 shows time to analyze bottleneck processes. With existing method, the author needed to analyze the execution logs of 21 mappings in total on two rounds of improvement. Since it is assumed to take 20 minutes in average to analyze a execution log, it is estimated to take 420 minutes (7 hours) to analyze the execution logs in total. On the other hand, Improvement Analyzer used an algorithm to explore IRs. The author did not take time to analyze bottleneck processes with Improvement Analyzer. Thus, Improvement Analyzer shortened the time to analyze bottleneck processes.

Table 6.7 also shows action items for the improvement of the system-level description which were done by the author. Since existing method did not provide specific IRs, the author had to explore IRs by himself. On the other hand, Improvement Analyzer explored IRs and analyzed bottleneck processes at the same time. In addition, the author had to consider how to improve the system in both existing method and Improvement Analyzer. Thus, Improvement Analyzer has less action items for improvement, and it shortened the time to improve the system-level description. Therefore, Improvement Analyzer is effective to efficient the system design.

6.5 Conclusion

The author proposed Improvement Analyzer which is a tool to identify system bottlenecks and explore improvement rates of them for embedded systems. Because Improvement Analyzer automatically identifies not only bottlenecks but also a list of improvement rates that is necessary to satisfy the requirements of the system performances, Improvement Analyzer helps designers to improve the system-level description without a time-consuming analysis. The case study on AES encryption and decryption application showed that Improvement Analyzer surely identified system bottlenecks automatically. It also showed that the entire design time of AES encryption and decryption system is shortened by Improvement Analyzer. In addition, by using Improvement Analyzer, designers can focus on to consider how to improve the system-level description with the list of improvement rates. Therefore, Improvement Analyzer is effective to improve the embedded systems efficiently.

Chapter 7

Conclusions

7.1 Summary of Contributions

This dissertation presented efficient system-level design space exploration method for embedded systems. The proposed method consists of three tools: Extended SystemBuilder, Mapping Explorer, and Improvement Analyzer.

In Chapter 2, the author explained a system-level design methodology and SystemBuilder. SystemBuilder is a tool to realize system-level design. SystemBuilder automatically synthesizes a target implementation with multi-core processors and dedicated hardware from system-level description of the target system. Since SystemBuilder automatically generates the target implementations from system-level description, it is easy for designers to design coarse-grain pipelined system.

In Chapter 3, the author presented a case study on an AES encryption system in order to evaluate the basic design efficiency of SystemBuilder. In the case study, sequential description of AES encryption system was divided into five processes with pipelined parallel structure. This case study clarified that SystemBuilder has high effectiveness of designing systems, especially designing pipelined system. During the case study, the author found three problems that current system-level design tools have. For that reason, the author proposed the solutions of these three problems, which are described in Chapter 4, 5, and 6.

In Chapter 4, the author proposed automatic communication synthesis for hardware sharing. The proposed method realizes automatic synthesis of communications for hardware modules which are shared by multiple applications. SystemBuilder has been extended so that it generates interface circuitry for the shared hardware module, which is named Extended SystemBuilder. Since the applications may run concurrently, the interface circuitry generated by Extended SystemBuilder realizes mutually exclusive accesses to the shared hardware module. The case study demonstrated that hardware sharing brought better area-performance trade-offs and a wider design space.

As the design space becomes wider, the number of possible mappings increases exponentially. Even if Extended SystemBuilder makes evaluation of the system performances easy by automatic generation of target implementation, it is hard to evaluate an enormous number of mappings. In Chapter 5, the author proposes an efficient exploration algorithm named pareto-update search. The author also developed a tool, Mapping Explorer, which uses pareto-update search in order to accelerates the exploration of software/hardware partitioning for finding appropriate mappings. Pareto-update search focuses on the trade-off relationship between execution time and hardware area. By using this relationship, pareto-update search can reduce the number of mappings which should be evaluated the system performances compared to exhaustive search. In a case study of MPEG-4 decoder application, pareto-update search reduced the number of simulated mappings to 0.18% compared to that of exhaustive search.

In Chapter 6, the author proposed Improvement Analyzer which is a tool to analyze bottlenecks of the system by the exploration of improvement rates. Improvement rates indicate the ratios to shorten the execution time and reduce the hardware area of a process compared to original ones. Improvement Analyzer automatically identifies not only bottlenecks but also a list of improvement rates that is necessary to satisfy the requirements of the system performances. In addition, Improvement Analyzer lists up several candidates to improve the system-level description. Thus, Improvement Analyzer is useful for designers in case that they cannot find an appropriate mapping during the exploration of software/hardware partitioning. By using the results of Improvement Analyzer, designers easily improve the system-level description to get better system performance such as execution time and hardware area. The case study on AES encryption and decryption application surely showed that the bottlenecks of the system are automatically identified by Improvement Analyzer.

As shown in the case studies on AES application and MPEG-4 decoder application in the chapters, system designers can explore design space at system-level efficiently with three tools above .

7.2 Future Directions

As mentioned in Chapter 3, Extended SystemBuilder is suitable to design pipelined system such as AES application and MPEG-4 decoder application. Because the proposed method is based on Extended SystemBuilder, the proposed method is effective to design pipelined systems. However, embedded systems are not only pipelined systems. They are variety. An example of an embedded system that is not pipelined is a control system.

The control systems are now getting more and more complex, and designers are requiring efficient design methodology. As presented in this dissertation, system-level design has potential to be a solution. In order to apply system-level design tools to design of control systems, the tools must be able to handle communication with sensors. There are two types of communication between sensors and PEs. One is data transfer and the other is interrupt signal. Because Extended SystemBuilder only supports communication among the processes, it cannot handle communication with sensors. Not only Extended SystemBuilder but also other system-level design tools have the same issue.

One of the promising solutions to the above problem is to extend system-level design tools. The point of extension is how to handle the interrupt signals and the processes activated by interrupt. Because the behavioral model does not consider asynchronous event, interrupt signals and processes activated by interrupt are a big problem to extend the model. By extending system-level design tools, the problems can be solved, and design efficiency of control systems can be increased. The author expects that further advance of these technologies will be helpful in the future complex embedded system design.

ACKNOWLEDGEMENT

I am profoundly indebted to my adviser, Associate Professor Shinya Honda, for his guidance, understanding, generosity, and most importantly, his sincerity during my studies at Nagoya University.

I would like to deeply express my respect and gratitude to Professor Hiroaki Takada and Professor Hiroyuki Tomiyama (currently with Ritsumeikan University) for giving invaluable suggestions and directions throughout the work. Their insightful indications and comments on the work led me to foster an ability of seeing things from different standpoints. It was my great pleasure to be given numerous opportunities to work in the fields of system-level design of embedded systems and to have studied in Takada Laboratory for six years.

This research was in part supported by STARC (Semiconductor Technology Academic Research Center) as collaborative research projects from 2006 to 2008 and from 2009 to 2011. I would like to thank researchers and guest researchers of STARC for helpful discussions and advices.

I am very grateful to Professor Masato Edahiro for serving on a committee member of this dissertation and providing invaluable suggestions and comments.

I would like to express my special thank to Takuya Ishikawa, Seiya Shibata, and the past and present members of Takada Laboratory and Edahiro Laboratory. Their helpful comments and cooperation were encouraging and stimulating to me, and fulfilling discussions with them led me to have new ideas on my studies. Also, thanks to them, my academic life was very enjoyable.

I would like to thank YXI and Soliton for providing a high-level synthesis tool and technical supports.

Last but not least, I thank my family for their never-ending support, encouragements, and their belief in me.

My financial support for this work has been in part provided by the Research Fellowship of the Japan Society for the Promotion of Science for Young Scientists.

Bibliography

[1] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, “Electronic system-level synthesis methodologies,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.

[2] S. Shibata, S. Honda, H. Tomiyama, and H. Takada, “Advanced systembuilder: A tool set for multiprocessor design space exploration,” in *Proceeding of International SoC Design Conference (ISOCC)*, vol. 2010, Incheon, Korea, Nov 2010, pp. 79–82.

[3] S. Shibata, Y. Ando, S. Honda, H. Tomiyama, and H. Takada, “Efficient design space exploration at system level with automatic profiler instrumentation,” *IPSJ Transactions on System LSI Design Methodology*, vol. 3, pp. 179–193, Aug 2010.

[4] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, “System-on-chip environment: a specc-based framework for heterogeneous mpsoc design,” *EURASIP J. Embedded Syst.*, vol. 2008, pp. 1–13, Jan. 2008.

[5] A. D. Pimentel, “The artemis workbench for system-level performance evaluation of embedded systems,” *International Journal of Embedded Systems*, vol. 3, pp. 181–196, Sept. 2008.

[6] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Peace: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–25, Aug. 2007.

[7] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, April 2003.

[8] S. Mahadevan, K. Virk, and J. Madsen, "Arts: A systemc-based framework for multiprocessor systems-on-chip modelling," *Design Automation for Embedded Systems*, vol. 11, no. 4, pp. 285–311, Dec. 2007.

[9] K. Ueda, K. Sakanushi, K. Takeuchi, and M. Imai, "Architecture-level performance estimation method based on system-level profiling," in *Proc. Computers & Digital Techniques*, vol. 152, no. 1, 2005, pp. 12–19.

[10] T. Wild, A. Herkersdorf, and G.-Y. Lee, "Tapes - trace-based architecture performance evaluation with systemc," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10617-006-9589-4

[11] S. Shibata, Y. Ando, S. Honda, H. Tomiyama, and H. Takada, "A fast performance estimation framework for system-level design space exploration," *IPSJ Transactions on System LSI Design Methodology*, vol. 5, pp. 44–54, 2012.

[12] F. Gao and S. Sair, "Long-term performance bottleneck analysis and prediction," in *Computer Design, 2006. ICCD 2006. International Conference on*, 2006, pp. 3–9.

[13] P. Del Valle, D. Atienza, I. Magan, J. Flores, E. Perez, J. Mendias, L. Benini, and G. De Micheli, "A complete multi-processor system-on-chip fpga-based emulation

framework," in *Very Large Scale Integration, 2006 IFIP International Conference on*, 2006, pp. 140–145.

[14] D. Nunes, M. Saldana, and P. Chow, "A profiler for a heterogeneous multi-core multi-fpga system," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 113–120.

[15] G. Palermo, C. Silvano, and V. Zaccaria, "Multi-objective design space exploration of embedded systems," *Journal of Embedded Computing*, vol. 1, no. 3, pp. 305–316, March 2005.

[16] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of IFIP Congress 74*, 1974.

[17] SystemC Open Initiative, http://www.systemc.org/.

[18] D. Gajski, N. Dutt, A.-H. Wu, and S.-L. Lin, *High-Level Synthesis*. Kluwer Academic Publishers, 1992.

[19] K. Wakabayashi, "Cyberworkbench: integrated design environment based on c-based behavior synthesis and verification," in *VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, Apr. 2005, pp. 173 – 176.

[20] Y Explorations Inc., http://www.yxi.com/.

[21] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[22] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, “Autopilot: A platform-based esl synthesis system,” in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 99–112. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8588-8_6

[23] U. Holtmann and R. Ernst, “Combining mbp-speculative computation and loop pipelining in high-level synthesis,” in *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, Mar. 1995, pp. 550 –556.

[24] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Spark: A high-level synthesis framework for applying parallelizing compiler transformations,” in *Proceedings of the 16th International Conference on VLSI Design*, ser. VLSID ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 461–. [Online]. Available: http://portal.acm.org/citation.cfm?id=832285.835535

[25] R. C. Cofer and B. F. Harding, *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Newnes, 2005.

[26] D. D. Gajski, J. Zhu, R. Dömer, and A. Gerstlauer, *SpecC: specification language and design methodology*. Kluwer Academic Publishers, 2000.

[27] TOPPERS/ASP kernel, http://www.toppers.jp/en/asp-kernel.html.

[28] TOPPERS/FMP kernel, http://www.toppers.jp/en/fmp-kernel.html.

[29] Altera Corporation, http://www.altera.com/.

[30] Xilinx, http://www.xilinx.com.

[31] S. Honda, H. Tomiyama, and H. Takada, “Rtos and codesign toolkit for multiprocessor systems-on-chip,” in *Design Automation Conference, 2007. ASP-DAC ’07. Asia and South Pacific*, Jan. 2007, pp. 336 –341.

[32] GTKWave, http://gtkwave.sourceforge.net.

[33] S. Mangard, M. Aigner, and S. Dominikus, “A highly regular and scalable aes hardware architecture,” *Computers, IEEE Transactions on*, vol. 52, no. 4, pp. 483–491, 2003.

[34] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis,” *Journal of Information Processing*, vol. 17, pp. 242–254, Oct 2009.

[35] TOPPERS Project, http://www.toppers.jp/en/index.html.

[36] G. Stehr, H. Graeb, and K. Antreich, “Performance trade-off analysis of analog circuits by normal-boundary intersection,” in *Proceedings of the 40th annual Design Automation Conference*, ser. DAC ’03. New York, NY, USA: ACM, 2003, pp. 958–963. [Online]. Available: http://doi.acm.org/10.1145/775832.776073

[37] T. Wiangtong, P. Y. K. Cheung, and W. Luk, “Comparing three heuristic search methods for functional partitioning in hardware-software codesign,” *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–449, July 2002.

[38] S. Kirkpatrick, “Optimization by simulated annealing: Quantitative studies,” *Journal of Statistical Physics*, vol. 34, no. 5-6, pp. 975–986, 1984. [Online]. Available: http://dx.doi.org/10.1007/BF01009452

[39] F. Glover and M. Laguna, *Tabu Search.* Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[40] C. M. Fonseca and P. J. Fleming, “An overview of evolutionary algorithms in multiobjective optimization,” *Evol. Comput.*, vol. 3, no. 1, pp. 1–16, Mar. 1995. [Online]. Available: http://dx.doi.org/10.1162/evco.1995.3.1.1

[41] C. A. C. Coello, “A comprehensive survey of evolutionary-based multiobjective optimization techniques,” *Knowledge and Information Systems*, vol. 1, pp. 269–308, 1998.

[42] M. Palesi and T. Givargis, “Multi-objective design space exploration using genetic algorithms,” in *Proceedings of the tenth international symposium on Hardware/software codesign*, ser. CODES ’02. New York, NY, USA: ACM, 2002, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/774789.774804

[43] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[44] E. Zitzler, M. Laumanns, and L. Thiele, “SPEA2: Improving the strength pareto evolutionary algorithm,” Gloriastrasse 35, CH-8092 Zurich, Switzerland, Tech. Rep. 103, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.2440

[45] J. I. Hidalgo, J. L. Risco-Martín, D. Atienza, and J. Lanchares, “Analysis of multi-objective evolutionary algorithms to optimize dynamic data types in embedded systems,” in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, ser. GECCO ’08. New York, NY, USA: ACM, July 2008, pp. 1515–1522, nSGA2,SPEA2. [Online]. Available: http://doi.acm.org/10.1145/1389095.1389388

[46] C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck *et al.*, “Multicube: Multi-objective design space exploration of multi-core architectures,” in *Proceedings of the 2010 IEEE Annual Symposium on VLSI*. IEEE Computer Society, 2010, pp. 488–493.

[47] G. Ascia, V. Catania, and M. Palesi, “A ga-based design space exploration framework for parameterized system-on-a-chip platforms,” *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 4, pp. 329–346, aug. 2004.

[48] S. Tiwary, P. Tiwary, and R. Rutenbar, “Generation of yield-aware pareto surfaces for hierarchical circuit design space exploration,” in *Design Automation Conference, 2006 43rd ACM/IEEE*, 0-0 2006, pp. 31 –36.

[49] G. Wang, W. Gong, and R. Kastner, “Application partitioning on programmable platforms using the ant colony optimization,” *Journal of Embedded Computing*, vol. 2, no. 1, pp. 119–136, September 2006.

[50] Embedded Microprocessor Benchmark Consortium (EEMBC), http://www.eembc.org/home.php.

[51] H. Kawashima, G. Zeng, N. Atsumi, T. Tatematsu, H. Takase, and H. Takada, “An evaluation method for deps profile and its application to checkpoint selectio,” in *IPSJ SIG Technical Report*, vol. 2011-EMB-20, no. 4, Mar 2011, pp. 1–6, (in Japanese).

List of Publications by the Author

Research Achievements Related to the Dissertation

Journal Papers

1. Yuki ANDO, Seiya SHIBATA, Shinya HONDA, Hiroyuki TOMIYAMA and Hiroaki TAKADA, "Automatic Communication Synthesis with Hardware Sharing for Multi-Processor SoC Design", IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E93-A, No. 12, pp. 2509-2516, Dec 2010.

2. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, "SEEDS: 組込みシステム向けシステムレベル設計環境", 電子情報通信学会論文誌 D, Vol. J97-D, No. 3, Mar 2014. (to appear)

International Conferences

1. Yuki ANDO, Seiya SHIBATA, Shinya HONDA, Hiroyuki TOMIYAMA and Hiroaki TAKADA, "A Case Study on AES Encryption System Design with SystemBuilder", Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI) 2009, pp. 283-288, Mar 2009.

2. Yuki Ando, Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama and Hiroaki Takada, "Automatic Communication Synthesis with Hardware Sharing for Design Space Exploration", IEEE International Symposium on Circuits and Systems, pp. 1863-1866, May 2010.

3. Yuki Ando, Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, Hiroaki Takada, "Fast Design Space Exploration for Mixed Hardware-Software Embedded Systems", In Proc. of International SoC Design Conference (ISOCC), pp. 92-95, Nov 2011.

4. Yuki Ando, Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, Hiroaki Takada, "Automated Identification of Performance Bottleneck on Embedded Systems for Design Space Exploration", In Proc. of International Embedded Systems Symposium (IESS), Springer IFIP 403, pp. 171-180, Jun 2013.

Domestic Symposiums

1. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, "システムレベル設計環境 SystemBuilder を用いた AES 暗号化システムの設計事例", 電子情報通信学会大会講演論文集, Vol. 2008, pp. S5-S6, Sep 2008.

2. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, "設計空間探索におけるハードウェア共有用通信の自動合成", 情報処理学会研究報告, 2010-EMB-15, pp. 1-6, Jan 2010.

3. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, "ソフトウェア・ハードウェア混在システム向けの効率的な設計空間探索手法", 情報処理学会シンポジウム論文集, Vol. 2011, No. 5, pp. 3-8, Aug 2011.

4. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, "組込みシステムのアーキテクチャ探索における性能ボトルネック解析", 信学技報, Vol. 112, No. 320, VLD2012,

pp. 19-24, Nov 2012.

Other Research Achievements

Journal Papers

1. Seiya Shibata, Yuki Ando, Shinya Honda, Hiroyuki Tomiyama and Hiroaki Takada, “Efficient Design Space Exploration at System Level with Automatic Profiler Instrumentation”, IPSJ Transactions on System LSI Design Methodology, Vol. 3, pp. 179-193, Aug 2010.

2. 谷口一徹，安藤友樹，高瀬英希，安積卓也，松原豊，村上靖明，菅谷みどり，“学生および若手技術者による組込みシステム技術に関するサマースクールの実践”, 情報処理学会論文誌, Vol. 52, No. 12, pp. 3221-3237, Dec 2011.

3. Seiya Shibata, Yuki Ando, Shinya Honda, Hiroyuki Tomiyama and Hiroaki Takada, “A Fast Performance Estimation Framework for System-Level Design Space Exploration”, IPSJ Transactions on System LSI Design Methodology, Vol. 5, pp. 44-54, Feb 2012.

International Conferences

1. Seiya Shibata, Yuki Ando, Shinya Honda, Hiroyuki Tomiyama and Hiroaki Takada, “Automatic Instrumentation of Profilers for FPGA-based Design Space Exploration” , 2009 International Conference on Field-Programmable Technology (FPT’ 09), pp. 292-295, Dec. 2009.

Domestic Symposiums

1. 柴田誠也, 安藤友樹, 本田晋也, 冨山宏之, 高田広章, “マルチプロセッサ対応システムレベル設計環境 SystemBuilder を用いた FPGA 向け設計事例”, 信学技報, Vol.111, No.218, RECONF2011, pp. 57-62, Sep 2011.

2. 湊雅登, 安藤友樹, 柴田誠也, 本田晋也, 高田広章, “システムレベル通信モデルにおける FIFO ベース通信チャネルの効率化機構と自動合成”, 信学技報, Vol. 111, No. 324, VLD2011, pp. 91-96, Nov 2011.

3. 石田薫史, 柴田誠也, 安藤友樹, 本田晋也, 高田広章, 枝廣正人, “高位合成による STP エンジン及び FPGA への AES/ADPCM の実装と評価”, 信学技報, Vol. 112, No. 70, RECONF2012, pp. 77-82, May 2012.

4. 安藤友樹, 石田薫史, 本田晋也, 高田広章, 枝廣正人, “割込み処理を考慮したシステムレベル設計手法”, 信学技報, Vol. 113, No. 320, VLD2013, pp. 119-124, Nov 2013.

5. 石田薫史, 安藤友樹, 本田晋也, 高田広章, 枝廣正人, “ヘテロマルチプロセッサシステム向けプロセッサ間通信の自動合成”, 信学技報, Vol. 113, No. 325, RECONF2013, pp. 63-68, Nov 2013.

Poster Presentations

1. 安藤友樹, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, “組込みマルチプロセッサシステムの設計改善支援”, 第 12 回組込みシステム技術に関するサマーワークショップ（SWEST12), pp. 54-56, Sep 2010.

2. 安藤友樹, 高田広章, “高速性能見積もりモデルによる組込みマルチプロセッサの設計探索支援”, STARC シンポジウム FY2011, Feb 2012.

3. 安藤友樹, 高田広章, “高速性能見積もりモデルによる組込みマルチプロセッサの設計探索支援”, STARC シンポジウム 2013, Jan 2013.

Awards and Honors

1. STRAC シンポジウム FY2011 学生ポスター発表　優秀発表賞
2. 第 163 回システム LSI 設計技術研究会　優秀発表学生賞
3. Grant-in-Aid for JSPS Fellows (DC2), from Apr., 2012 to 2013.