An Architecture Design Space Exploration Method of System-on-a-Chip for CNN-based Artificial Intelligence Platform

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2019

Salita SOMBATSIRI

Publications

Journal Article (Refereed)

[J1] Salita Sombatsiri, Yoshinori Takeuchi, and Masaharu Imai: An Efficient Performance Estimation Method for Configurable Multi-Layer Bus-based SoC, IPSJ Transaction on System LSI Design Methodology, vol.8, pp.26–37, 2015.

[J2] Salita Sombatsiri, Seiya Shibata, Yuki Kobayashi, Hiroaki Inoue, Takashi Takenaka, Takeo Hosomi, Yu Jaehoon, and Yoshinori Takeuchi: Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks on FPGA, IPSJ Transaction on System LSI Design Methodology, 2019. (To appear)

International Conference Papers (With review)

[I1] Napat Luevisadpaibul, Salita Sombatsiri, and Krerk Piromsopa, "An FPGA Implementation of ATA Host Controller toward Scalable iATA NAS," In proceedings of the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE 2011), pp. 229–233, Bangkok, Thailand, 2011.

[I2] Salita Sombatsiri, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai, "On-chip Communication Buffer Architecture Optimization Considering Bus Width," Proceedings of IEEE 6th International Symposium on Multicore SoCs (MCSoC2012), pp. 29–36, Aizu-Wakamatsu, Japan, 2012.

[I3] Salita Sombatsiri, Kazuhiro Kobashi, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai, "An AMBA Hierarchical Shared Bus Architecture Design Space Exploration Method considering Pipeline, Burst and Split Transaction", Proceedings of IEEE 10th International Conference in Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON2013), pp. 1-6, Krabi, Thailand, 2013.

[I4] Salita Sombatsiri, Seiya Shibata, Yuki Kobayashi, Hiroaki Inoue, Takashi Takenaka, and Takeo Hosomi: "Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks," Proceedings of 21th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2018), pp. 188-193, Matsue, Japan, 2018.

[I5] Salita Sombatsiri, Yu Jaehoon, Yoshinori Takeuchi, and Masaharu Imai: "EMG-based Thai Tone Recognition using Convolution Neural Networks and Spectrograms," BHI-2017 International Conference on Biomedical and Health Informatics, FrRAF.16, Feb. 16-19, 2017, Orlando, USA. [Short Paper]

Abstract

Recent advancement of artificial intelligence (AI) algorithms and computing platforms enables various cutting-edge applications, such as image/video analytics, speech recognition, and autonomous driving. In order to achieve a real-time response for these applications, the edge computing, which locates between the endpoint devices and the cloud, has become more compelling in the paradigm of an AI platform. Designing an AI-based edge computing device is very complicated and time-consuming since the edge computing devices have strict constraints in high-performance, yet compact and low-power. Therefore, the following three requirements are involved in designing optimal architectures for edge computing: (1) quickly evaluate the design quality of an architecture; (2) accelerate deep learning algorithms; (3) efficiently explore the design space to find optimal architectures. This thesis proposes an efficient method in designing edge system-on-a-chip (SoC) architecture for AI applications to fulfill the above-mentioned three requirements.

First, to quickly estimate the execution time of an application on each architecture, this thesis proposes an efficient performance estimation method for configurable multi-layer bus-based SoC. Design quality estimation, specifically, performance estimation, is time-consuming since it analyzes the behavior of the architecture. The speed of the estimation method is often more critical than the estimation accuracy in the early design stage because there is a massive amount of architectures to evaluate during the architecture exploration process. The proposed performance estimation method provides a fast and accurate method to evaluate the execution time of each architecture. It analyzes system behavior based on system-level profiling, speculates dynamic bus contention and predicts bus behavior with graph analysis. The experimental results show that the proposed method has achieved 25.6x speedup over the register-transfer level (RTL) simulation in evaluating the execution time of eight architectures. The error of the estimation results is within 8% compared to the conventional RTL simulation. Hence, the proposed method is efficient and suitable for architecture exploration process.

Second, a parallelism-flexible convolution core for sparse Convolutional Neural Network (CNN) is proposed in order to accelerate the deep learning algorithm, specifically a CNN, as a high-performance intellectual property (IP). A

computation-intensive CNN becomes critical for real-time inference processing on edge devices for many applications. The proposed parallelism-flexible convolution core achieves high-performance by maximizing calculation-skip and parallel calculation in all convolutional layers of a CNN. It skips multiply-accumulates (MACCs) related to zero-valued weights efficiently with the use of the compressed CNN model together with the output-stationary scheme. It alternates dataflow and schedules MACCs flexibly according to the specification of each convolutional layer to improve multiplier utilization. The results have shown that the integration of both techniques improves performance by 4x speedup over the baseline architecture and 3x in effective GMACS over prior arts of CNN accelerator.

Third, an architecture exploration of SoCs for CNN-based AI platform is proposed to efficiently explore the design space with IP-based design and system-level design. The complexity of finding optimal architectures in the early design stage lies in IP selection and bus selection because there is a vast amount of IPs, bus architectures, and their parameters. In the proposed architecture exploration method, the IPs and bus architecture are parameterized and explored using a parameter set search tree. The proposed method consists of process mapping, channel mapping, bus protocol mapping, functional block's parameter mapping, functional block's and bus' execution frequency mapping, bus width mapping, and the number of buffer mapping. In the process mapping, a process is mapped to an IP. In the channel mapping, the data transfers are mapped onto either a hierarchical shared bus or configurable multi-layer bus by mapping the data transfers into clusters, each of which is connected to a bus matrix. Then, the functional block's parameters, i.e. the number of instances of each functional block and the number of processing elements (PEs) within each functional block, are selected considering data tiling in order to distribute and parallelize the workload of computation-intensive processes. This parameterization allows the MACCs to be parallelized on multiple instances of functional blocks. The results show that the proposed method discovers varieties of architecture having various functional blocks and multi-layer bus configurations.

This thesis contributes to designing SoCs for CNN-based AI platform at the edge, especially architecture design and optimization in the early design stage. It provides an efficient method to explore the architecture candidates, including a parameterized IPs and multi-layer bus, and evaluate their design qualities. The method can find architectures with superior design qualities within a short time. Hence, it is suitable for discovering good architecture candidates in the early stage of designing an SoC.

Acknowledgment

This thesis is possible with the valuable insights, supports, and efforts of all the persons and institutions who have supported and encouraged me not only in this research, but also in living in Japan.

First and foremost, I would like to express my heartfelt gratitude and sincere appreciation to my advisors, Professor Masanori Hashimoto, Osaka University, Professor Yoshinori Takeuchi, Kindai University, and Emeritus Professor Masaharu Imai, for not only invaluable guidance in research, but also sincere support in living in Japan. Thank you for giving me countless opportunities since my first day in Integrated System Design Laboratory and encouraging me to grow as a good researcher. I am honored to be their student. This thesis would not have been possible without their kind supports and efforts.

I am grateful to the committee members of my thesis, Professor Takao Onoye, Osaka University, and Associate Professor Ittetsu Taniguchi, Osaka University, for investing their invaluable time giving useful comments towards improving this thesis.

I am thankful for my supervisors at NEC Corporation, Dr. Yuichi Nakamura, Mr. Takeo Hosomi, Dr. Hiroaki Inoue, Dr. Takashi Takenaka, Dr. Yuki Kobayashi, and Dr. Seiya Shibata for their support and understanding in my pursue of PhD. Thank you for their invaluable advice towards a part of this thesis.

I am grateful for Assistant Professor Jaehoon Yu, Osaka University, and Dr. Keishi Sakanushi (my former Assistant Professor), for their valuable guidance in the experiments and research.

I would like to express my appreciation to the current and former members of Integrated System Design Laboratory for their insights. I am especially grateful to Mr. Kazuki Ohya and Mr. Yuji Kamata, my tutors who have made their support in a number of ways.

Last but not least, I would like to express my special thanks to my family, especially my father, my mother, my sister and my brother, who have always been supportive of me in both good times and hard times, and embraced me whenever I fall. I am grateful for Dr. Pavadee Saisuwan and Dr. Songpol Chaunchaiyakul for her advice in English. I would like to thank Mr. Nuttee Woramongkol, Dr. Kamalas Udomlamlert, and Dr. Nattapong Thammasan for helping me get back on my feet during challenging moments in the writing of this thesis. Also, I would like to express my thanks to my friends from my home country, who are dear to

me since before I came to Japan, and also those who I have met in Osaka and Tokyo, for being more than ready to comfort me when I feel down, even from a distance.

It might be a simple word, but I sincerely want to say "Thank you."

Contents

List of Figures

List of Tables

Abbreviations

AHB	advanced high-performance bus
AI	artificial intelligence
AL-EDG	architecture-level execution dependency graph
ALM	architecture-level model
AMBA	advanced microcontroller bus architecture
APB	advanced peripheral bus
ASIC	application-specific integrated circuit
BCA	bus cycle accurate
CA	cycle accurate
CNN	Convolutional Neural Network
DCNN	deep convolutional neural network
DMAC	direct memory access controller
ESL	electronic system level
FIFO	first in first out
FPGA	field programmable gate array
GPU	graphic processing unit
HDL	hardware description language
IFM	input feature map
IoT	internet of things
IP	intellectual property
MACC	multiply-accumulate
MoC	model-of-computation
NN	neural network
OFM	output feature map
PE	processing element
RTL	register-transfer level
SLDL	system-level design language
SL-EDG	system-level execution dependency graph
SLM	system-level model
SoC	system-on-a-chip
SPL	software programming language
TLM	transaction-level modeling

Chapter 1

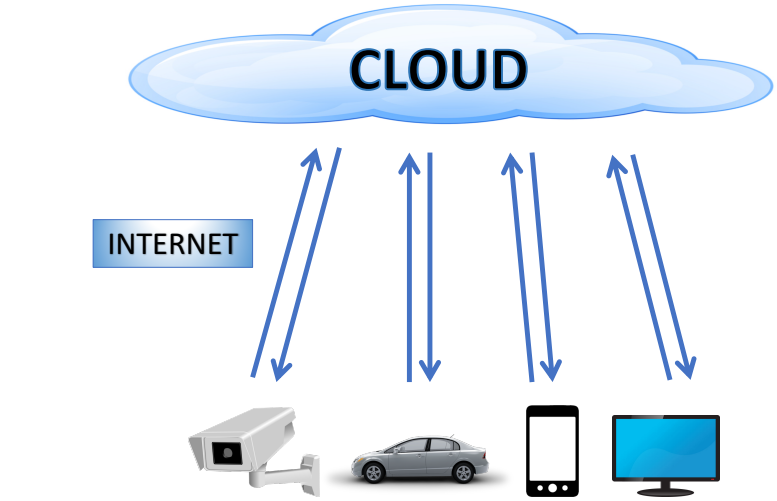
Introduction

First, this chapter introduces the current Artificial Intelligence (AI) applications and its computing paradigm. The paradigm is shifting from the cloud platform to cutting-edge Internet of Things (IoT) and AI platform in order to provide a real-time response. Then, this chapter overviews the design flow of AI-based edge computing devices. Next, the requirements for designing the edge computing devices are described. Finally, the contribution and organization of this thesis are described.

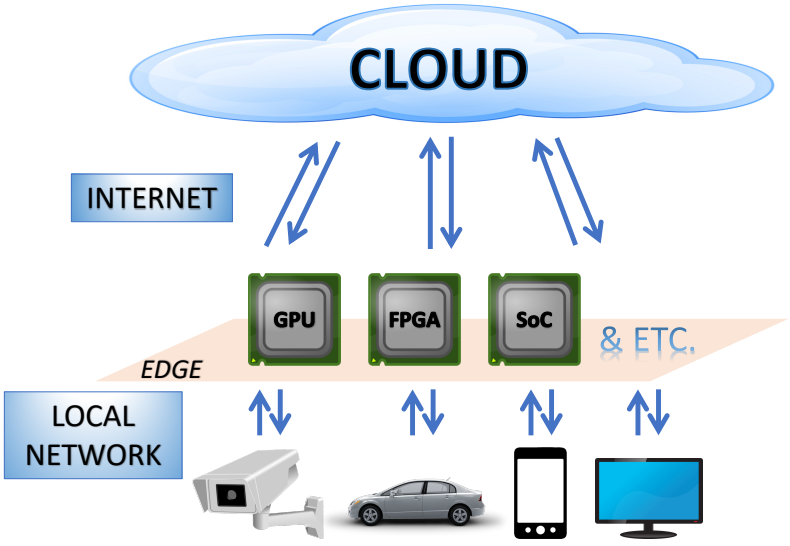
1.1 Background

Over the past decades, AI has gained an extensive academic and industrial popularity in various fields of applications. It has been integrated into linguistics, healthcare, image/video analytics, and etc., comprising interdisciplinary practical applications. In linguistics, AI is widely applied to language modeling tasks like speech recognition [1–3], speech synthesis [4, 5], and natural language processing [6, 7], which enable automatic translation, chat bot, and so on. AI is transforming healthcare in many ways, such as detecting disease early with a fast and precise diagnosis [8–10] and supporting clinical decision making and treatment [11]. In image and video analytics, AI involves object detection and classification [12–15] that analyze the environment for surveillance systems and autonomous driving applications. The usage of AI is increasing dramatically every year since it shows a remarkable outcome, e.g. recognition accuracy, in various tasks that enable applications in broader fields.

The popularity of AI comes from three key factors: the emerging of big data, advanced AI algorithms, and high-performance computing platform. First, the emerging of big data provides a vast amount of data for AI to learn useful information effectively. Second, the advanced AI algorithms in recent years exhibit practical usability with remarkable improved accuracy. For example, the accuracy of image recognition task has risen dramatically with the use of deep convolutional neural network (DCNN) [16–18]. Third, the rapid improvement of computing



(a)



(b)

Figure 1.1: Paradigm of AI platform: (a) conventional IoT and AI platform; (b) cutting-edge IoT and AI platform.

platform, such as graphic processing unit (GPU) and field programmable gate array (FPGA), enables the massive computation of AI algorithms especially deep learning [19]. These factors empower AI so that AI is effective in terms of both accuracy and reasonable computation time.

A cutting-edge paradigm of an AI platform is evolving from the conventional IoT platform paradigm with the inclusion of edge computing layer between the cloud and endpoint devices. Figure 1.1(a) shows the conventional IoT platform where the endpoint devices, such as mobile devices and cameras, collect and upload data to the cloud directly via internet. In cutting-edge IoT and AI platform, the edge computing is introduced to perform data computation near the devices where the data originates as shown in Fig. 1.1(b). The edge is connected to the endpoint devices through the local network and to the cloud via the internet connection. Data processing at the edge usually includes, but not limited to, data analysis and actuation of AI applications.

IoT and AI platform requires computation at the edge for three main reasons. First, the concern for users' privacy is rising because more and more data that could identify users are transmitted over the internet. Computation at the edge analyzes user's data collected via local network and filters-out identity-related data before uploading extracted knowledge to the internet. Second, it is crucial to conserve internet bandwidth because the amount of data is increasing. Edge computing devices send only a small amount of extracted knowledge to save the bandwidth. Third, many applications, such as surveillance systems and autonomous driving, require real-time response. Edge computing reduces latency in responding to endpoint devices since data processing takes place near the data origins. Therefore, computing at the edge is crucial to IoT and AI platform.

Requirements for edge computation of AI platform are as follows.

- High-performance computing devices and accelerators that can process inference phase of AI algorithms, especially deep learning, in real time.
- Compact computing devices since edge computing is located near endpoint devices, such as surveillance cameras, where device installation space is limited.
- Low-power computing devices because of the limited power supplies at the edge.

Such computing devices can be FPGAs, GPUs for edge, system-on-a-chips (SoCs), embedded systems and etc.

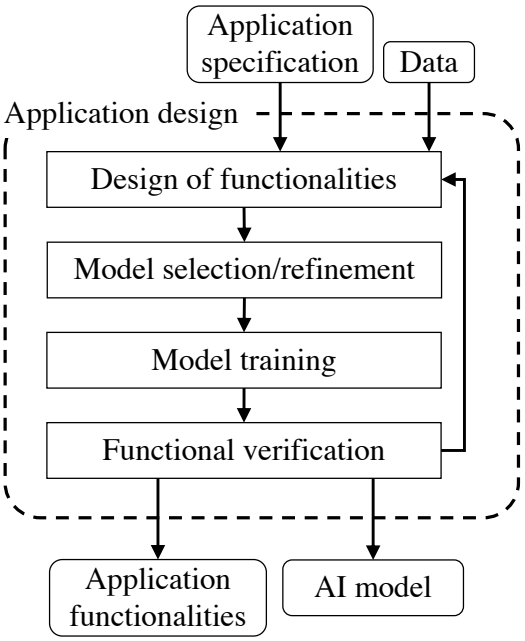


Figure 1.2: Application design flow of AI-based applications.

1.2 Design Flow of AI-based Edge Computing Devices

Conventionally, the design of AI-based edge computing devices is divided into two major phases: AI application design and hardware design. Given an application that includes analysis or prediction tasks, software engineer designs the functionalities of the application, selects and refines a model, such as deep learning model, Support Vector Machine (SVM), and etc., that represents the given data in the AI algorithm design phase, and then, hardware engineer optimizes the model and develops the hardware that is the most suitable for the application in the hardware design phase. This thesis focuses on algorithm and hardware design of deep learning-based applications since a vast amount of recent AI-based applications employ deep learning models [1–7, 9, 10, 12, 14, 15].

1.2.1 AI Application Design

Figure 1.2 shows the design flow of AI-based applications. Application design is an iterative process. Given an application specification and a set of data, first, design of functionalities determines processing procedures, including pre-processing, processing for recognition or detection tasks, and post-processing of the application. Then, for the recognition or detection tasks, a model is selected from a wide variety of AI algorithms, as well as model's hyperparameters (configuration of the model). Major deep learning models include neural networks (NNs),

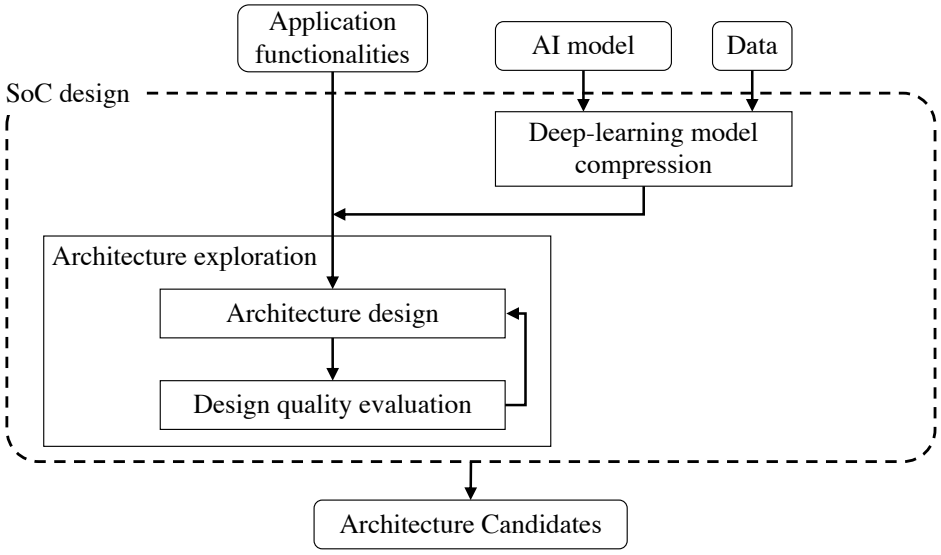


Figure 1.3: SoC design flow of deep learning-based applications.

Convolutional Neural Networks (CNNs), a variety of Recurrent Neural Networks (RNNs), and etc. Hyperparameters include, but not limited to, the number of layers, the number of hidden nodes within a layer and learning rate for training. Next, model training adjusts the model with the given set of data, so that it can represent and analyze the characteristics of the data, and make accurate predictions of the incoming unseen data after the model deployment. Finally, the functionalities are verified, which also includes a model evaluation using metrics, such as prediction accuracy. If the model evaluation result satisfies the application specification, the AI application design yields application functionalities and AI model in software programming language (SPL), and proceeds to the hardware design phase. Otherwise, the flow repeats design of functionalities, model selection or hyperparameters refinement.

1.2.2 Hardware Design

AI-based applications are accelerated on heterogeneous computing platforms, including GPUs, FPGAs, SoCs, and embedded systems. Hardware designer determines the suitable hardware based on application constraints, such as response time, installation space, and power supply. GPUs provide easy programming interface, but the GPU servers are bulky and power-hungry. FPGAs provide another option for low-power platform, but their performance can be limited by their available resources. This thesis mainly focuses on an SoC since it offers the highest power efficiency. Here, single-chip heterogeneous computing platform consisting of CPU, GPU and FPGA are gaining popularity since various computations can be allocated to each component depending on the compatibility between individ-

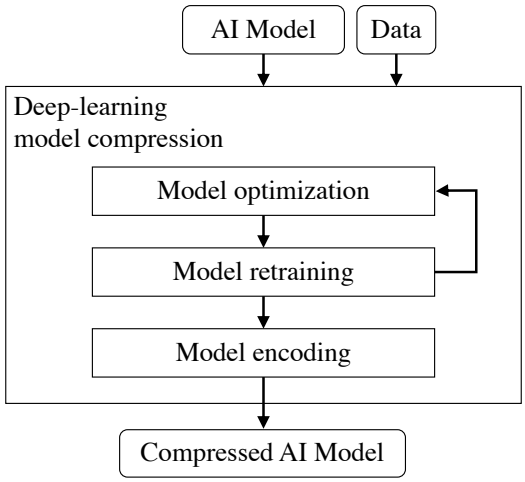


Figure 1.4: Flow of deep learning model compression.

ual computations and hardware components. Such platforms might be included in our target since they could provide power-efficient computing exploiting the heterogeneousness. This thesis addresses the design flow of SoC in details since it is subjected to a wide variety of customization and consumes long period for hardware design.

A flow of SoC design for AI-based applications, specifically, deep learning-based applications, is comprised of deep learning model optimization procedure and architecture exploration procedure as shown in Fig. 1.3. Deep learning model compression analyzes and compresses the deep learning model in order to fully utilize hardware for both data processing and communication. For example, a VGG-16 [17] contains as much as 154.7G multiply-accumulates (MACCs) and 138.36M model parameters (weights) in total to be transferred and stored on the chip. Many studies have shown that through weight pruning and quantization techniques to compress the model, the VGG-16 requires only 32.5% of MACCs to achieve the same results and the compressed model reduces 92% of the weight's off-chip communication and 95% of on-chip memory size related to weights [20]. Architecture exploration searches the design space for optimal architectures that satisfy all the design constraints in an early design stage.

Deep Learning Model Compression

Two objectives of deep learning model compression are to reduce the number of computations and the total size of a model. This is possible because many state-of-the-art studies have shown that in model deployment, aka inference phase, many MACCs of a dense and full-precision deep learning model are redundant [20–23]. Generally, the model compression consists of model optimization, retraining, and encoding as shown in Fig. 1.4. It iterates model optimization and model retraining

process to obtain an optimal model while preserving the equivalent accuracy.

Two main approaches to optimize deep learning model are quantizing arithmetic precision and pruning weights. Quantizing arithmetic precision of kernels, input feature maps (IFMs), and output feature maps (OFMs) from floating point to a few bits of fixed-point precision [21, 22, 24, 25] saves hardware resources for computing one MACC. Pruning weight process eliminates redundant MACCs by zeroing out some weight values within the deep learning model [20, 23, 26, 27], which results in a sparse model. Skipping these zero-operand MACCs reduces computation time by the degree of sparsity. Model optimization can reduce the number of MACCs.

Model retraining is the key in maintaining equivalent accuracy after the model is optimized. It is natural that the model loses accuracy due to the reduced precision and zeroed-out weights. Retraining the optimized model benefits in recovering the accuracy.

Model encoding aims to reduce the total number of bits for representing the model. Many conventional compression techniques are applied to compress the optimized model, such as entropy coding [20] and lossless source coding [25]. This can reduce the total size of the model, and hence save off-chip bandwidth in transferring the model from external memory onto SoCs.

Architecture Exploration

The advancement of semiconductor process technology has made it feasible to fabricate a large scale integrated (LSI) circuit on an SoC. A high-performance requirement has never been more compelled for operations of multifunction devices in an AI platform. While the multifunction dilates the complexity of the systems, strict constraints of design qualities, including high-performance, small area, and low energy consumption, are raised at the same time to further complicate the SoC design with trade-off relations between design qualities.

An architecture exploration methodology is introduced to search the combination of components comprising an SoC in the design space. The design space includes all possible combinations of the components, organization, as well as bus architectures. The architecture exploration determines the components for each functionality of an application and evaluates architecture's design quality. Typically, it is an iterative process between determining the components and evaluating design qualities to explore a vast amount of architectures. The architecture exploration finds optimal architectures considering the constraints of design qualities.

As illustrated in Fig. 1.3, the flow of architecture exploration in designing SoC for AI-based application includes architecture design and design quality evaluation steps. First, architecture design step decides hardware for data processing and communication architecture, aka bus architecture. Next, the design qualities, such as performance, area and energy consumption, of each decided architecture

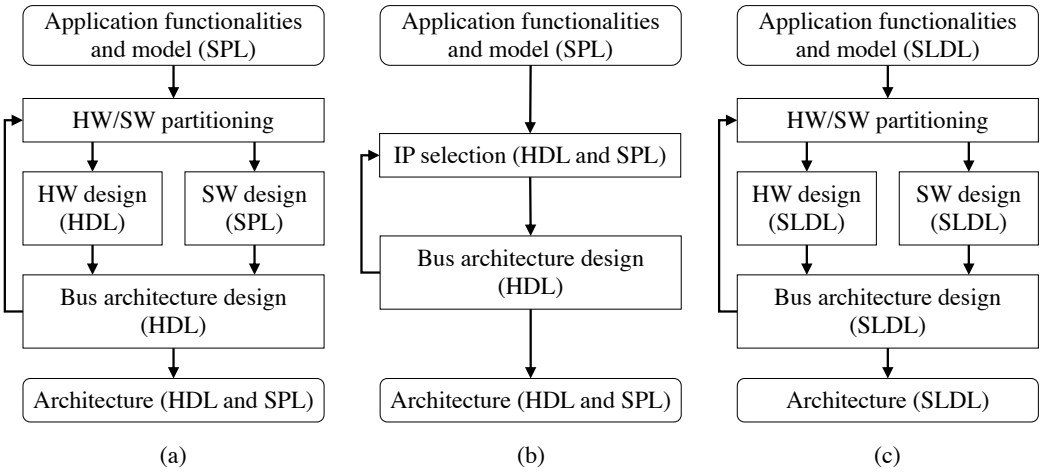


Figure 1.5: Flow of architecture design: (a) conventional approach; (b) IP-based approach; (c) system-level approach.

are evaluated. Both steps are performed repeatedly in order to find optimal architecture candidates, aka Pareto solutions.

A conventional approach designs architecture and evaluates design qualities using hardware description language (HDL). Architecture design step of the conventional method is shown in Fig. 1.5(a). It starts with hardware/software (HW/SW) partitioning, in which application functionalities described in SPL are mapped onto either hardware or software. Then, the hardware parts are designed as functional blocks using HDL, while the software parts are designed using SPL. Next, bus architecture, including bus topology, protocol, and connection among functional blocks and software parts, are designed in HDL. Finally, design quality evaluation is performed using hardware-software co-simulation. Designer should repeat these steps for various architectures to find the optimal architecture. However, designing and evaluating architectures with the conventional approach are costly in the early design stage because it takes long time and large human resources.

To reduce the period for architectural design, IP-based approach is introduced as shown in Fig. 1.5(b). It selects previously designed functional blocks, called intellectual property (IP), from IP database for each partitioned hardware and software parts, as well as bus architecture. This approach shortens the design time of an architecture by reusing existing IPs since the designers do not have to design the architecture from scratch. However, HDL-based hardware-software co-simulation remains as a time-consuming task.

Figure 1.5(c) shows the flow of system-level approach for architecture design. In this approach, application functionalities are described with an architecture-independent system-level model using system-level design language (SLDL). The SLDL can describe both parallel data processing and data transfer, so it is ca-

pable of simulating application functionalities in both system-level model and architecture-level model. After HW/SW partitioning, this approach implements a system-level model and architecture-level model with the same SLDL-based application functionality description, which saves the design time. Design qualities are evaluated based on the simulation of an architecture-level model in SLDL, which is faster than the HDL-based hardware-software co-simulation. Furthermore, the architecture design space is explored before the HDL/SPL implementation. The system-level approach shortens both design and evaluation time. Nevertheless, the cost for preparing and simulating SLDL model repeatedly for each architecture is still large.

1.3 Requirements in Designing AI-based Edge Computing Devices

Designing an AI-based edge computing devices has several design constraints. Hence, designing optimal architecture candidates becomes very complicated and time-consuming.

1.3.1 Quickly Evaluate the Design Quality of an Architecture

Design quality estimation, specifically, performance estimation, is time-consuming since it analyzes the behavior of the architecture. Furthermore, in the architecture exploration process, there are a massive amount of architectures to evaluate, and hence the speed of design quality estimation method is critical. To empower architecture exploration and shorten the design time, a fast and accurate design quality estimation method is required.

1.3.2 Accelerate Deep Learning Algorithms

Processing deep learning algorithms, even in inference phase, is usually critical in terms of performance since they are either computation-intensive (CNNs) or memory-intensive (Deep Neural Networks, Long Short-term Memory RNNs). To efficiently accelerate the deep learning, high-performance and specific accelerators are needed.

1.3.3 Efficiently Explore the Design Space to Find Optimal Architectures

To find optimal architectures, it is crucial to explore among an enormous design space. That includes HW/SW partitioning and IP selection, bus architecture selection, and parameters of the architecture, such as execution frequency and bus

width. All of the selections affect design qualities. Hence, the architecture exploration must be able to find and evaluate optimal architectures quickly in order to shorten the design time.

1.4 Objective of this Thesis

In designing an optimal edge SoC architecture for AI applications, there are two major concerns. First, the processing of deep learning model usually involves a massive amount of data transferring between functional blocks on the chip. Second, deep learning algorithms include billions of MACCs that might become the bottleneck of the system. IPs and bus architecture comprising the optimal architecture must be selected carefully from a variety of candidates in order to achieve real-time performance in the model deployment phase.

The objective of this thesis is to provide an efficient method in designing edge SoC architecture for AI applications. This thesis copes with the requirement of quickly estimating design qualities, accelerating deep learning algorithms, and efficiently exploring architecture design space since they contribute to achieving real-time performance in the model deployment phase. Figure 1.6 shows the contributions of this thesis.

This thesis first deals with the requirement of quickly estimating the design qualities with an efficient performance estimation method based on the analysis of bus behavior and system-level profiling. Even though an accurate simulation-based performance evaluation is necessary in the final design procedure, but in the early design stage which explores potential architectures, a quick estimation method is more crucial than the highly-accurate but slow ones in order to evaluate a vast number of architectures. However, the quick estimation methods, specifically, static performance estimation methods [28–33], suffer low accuracy problem. One of the causes is that they fail to capture the dynamic bus contention during system execution. This thesis proposes an efficient performance estimation method for configurable multi-layer bus-based SoCs. It analyzes system behavior based on system-level profiling, speculates dynamic bus contention and predicts bus behavior with graph analysis, called the architecture-level execution dependency graph (AL-EDG) analysis. The experimental results show that it can estimate execution time of multiple bus architecture accurately, even while it gains speedup over the simulation-based performance evaluation. Hence, it is suitable as a part of the architecture exploration method in the early stage of SoC design.

Next, this work proposes a parallelism-flexible convolution core for sparse CNN in order to fulfill the requirement of accelerating deep learning algorithms with a high-performance IP. CNN is one of the most vigorous AI algorithms especially in image and video analytic domains, such as surveillance systems and autonomous driving. CNN's processing usually takes place at the edge in order to achieve real-time response. Unfortunately, CNN involves an excessive computation that becomes critical for real-time inference processing on both edge devices.

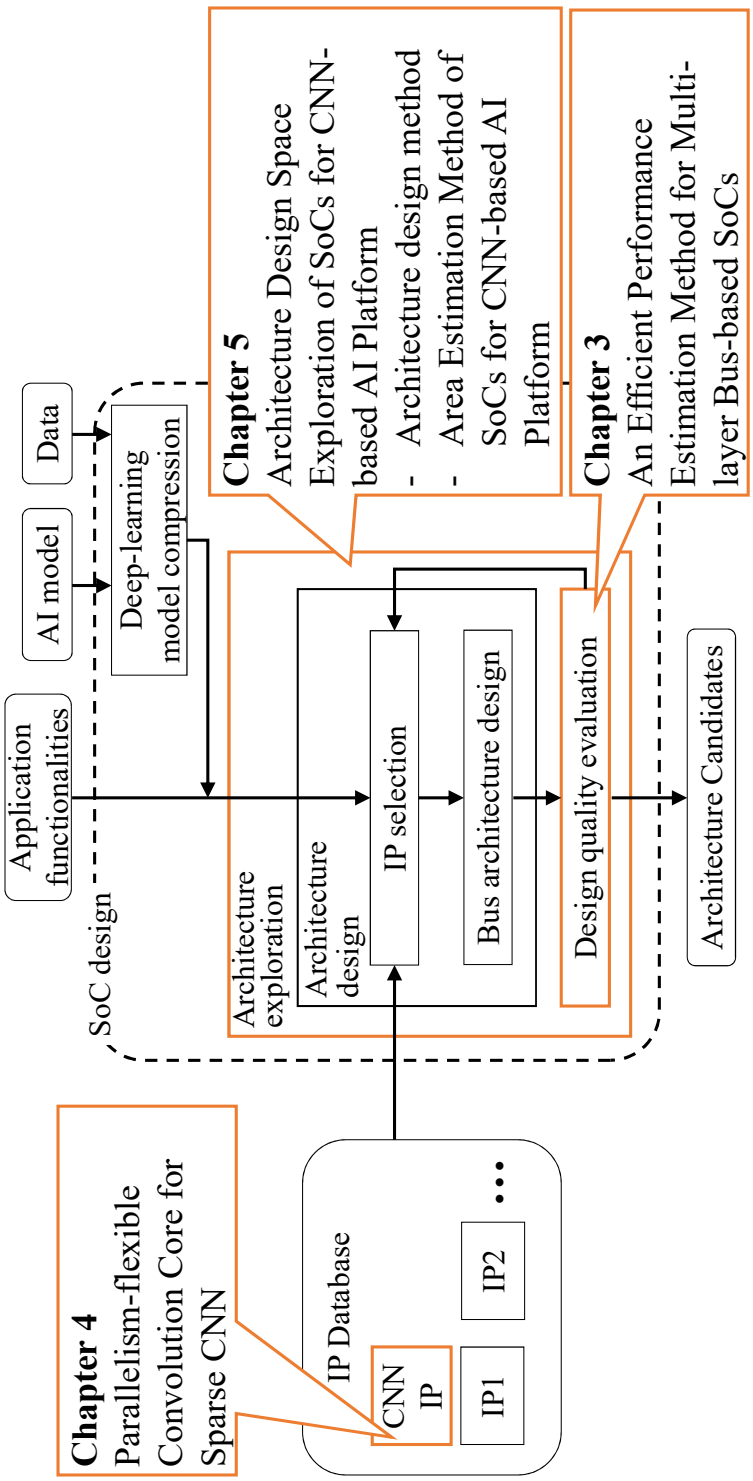


Figure 1.6: Contribution of this thesis to hardware design of SoCs for CNN-based AI platform.

Most CNN accelerators [34–36] fall behind their peak performance because they fail to maximize parallel calculation in some of the convolutional layers due to the fixed dataflow and computation scheduling. The proposed parallelism-flexible convolution core alternates dataflow and schedules MACCs flexibly according to the specification of each convolutional layer to improve multiplier utilization. Furthermore, it efficiently leverages sparsity by skipping MACCs related to zero-valued weight easily with the use of the compressed CNN model. The result shows that it outperforms the prior arts of CNN accelerator in total performance.

Finally, this thesis resolves the requirement to efficiently explore the design space with IP-based design and system-level design. Finding optimal architectures in the early design stage can shorten the design time. The complexity lies in IP selection and bus selection. First, the modeling granularity (level in modeling MACCs of deep learning algorithms) regulates IP selection. Typically, a process is mapped to an IP, which may introduce imbalance workload in deep learning application because the process of the deep learning algorithm is usually heavier than other processes. Second, there are a vast amount of bus architectures, including hierarchical shared bus and various configurations of a multi-layer bus. In the proposed exploration method, the IPs and bus architecture are parameterized. After the designer models a deep learning application with SystemC [37], the proposed architecture exploration method selects IPs via process mapping. It explores hierarchical shared bus and configurable multi-layer bus architecture via channel mapping and mapping the clusters, which is a groups of hardware components, onto the bus matrix. Then, it selects other specifications such as execution frequency and bus width via parameter mapping. The number of instances of process's functional block and the number of processing elements (PEs) within each functional block is also parameterized in order to allow the MACCs of coarse-grained modeling to be parallelized on multiple instances. Furthermore, data tiling is handled as a part of coping with granularity. The proposed method can discover architectures with varieties of functional block parameters and multi-layer bus configurations. Hence, the proposed architecture exploration method can find optimal architectures quickly in the early design stage.

The rest of this thesis is organized as follows. Chapter 2 presents related work. Chapter 3 proposes an efficient performance estimation method for configurable multi-layer bus-based SoCs, which considers the behavior of standard bus protocols and dynamic bus contention. In chapter 4, a parallelism-flexible convolution core for sparse CNN that leverages multiple types of parallelism and weight sparsity is proposed. Chapter 5 proposes an architecture exploration of SoCs for CNN-based AI Platform based on the performance estimation method in chapter 3 and the parallelism-flexible convolution core in chapter 4. Finally, chapter 6 concludes this thesis and describes future work.

Chapter 2

Related Work

This chapter reviews related studies. First, it explains three approaches for performance estimation and their trade-offs. Second, it reviews prior arts of CNN accelerators in terms of techniques for acceleration. Third, the chapter describes an architecture design space exploration methodology, including communication architecture exploration and design space exploration for CNN-based platform.

2.1 Performance Estimation

There are three main approaches in performance estimation: simulation-based, static and hybrid performance estimation. These approaches have a trade-off between speed and estimation accuracy. Simulation-based approach achieves high accuracy, but usually consumes a lot of time for analyzing the behavior of each architecture. Furthermore, since the performance obtained from simulation depends on and varies by the input data, the selection of input data for simulation affects the accuracy of performance estimation. On the other hand, a static approach is fast and independent of input data, but less accurate than the former one. The hybrid approach is the compromise between the two approaches. It collects execution traces from simulation to enhance the accuracy of the fast static approach. This section describes each approach in details.

2.1.1 Simulation-based Performance Estimation

Hardware-software Co-simulation

Traditionally, hardware-software co-simulation, e.g. register-transfer level (RTL)-based simulation method, is the most common way to evaluate the performance of an architecture. It simulates the behavior of the target architecture with co-simulation environment [38–43]. These co-simulators provide two major functionalities: (1) multiple simulators for various levels of abstraction, including an HDL simulator for hardware model evaluation, an instruction set simulator (ISS)

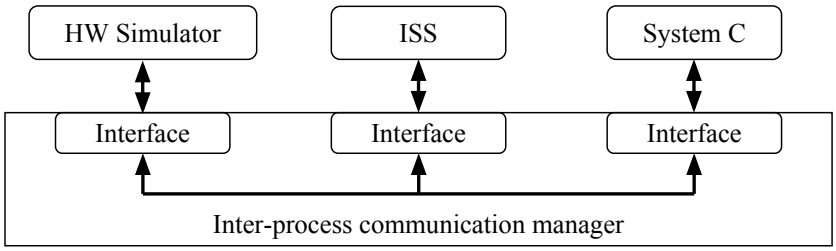


Figure 2.1: A basic structure of hardware-software co-simulator.

for software model evaluation, and simulators for other modeling languages, such as SystemC [37]; (2) interfaces and manager for inter-process (inter-simulator) communications. Figure 2.1 shows the basic structure of these co-simulators. Despite the fact that there are several studies that accelerate these simulators [44, 45] and these co-simulators can evaluate every architectural platform, including various bus architectures, it takes an unacceptably long time because a complete behavior of the architecture is simulated. Furthermore, the detailed of all components and signals within the architecture must be implemented in the RTL model, which requires not only a long time but also a lot of human resources. Therefore, hardware-software co-simulation approach is not suitable for performance estimation in the early stage of the design, where a vast amount of architecture must be evaluated.

Architecture-level Simulation

Architecture-level simulation approach provides a more efficient way to evaluate performance than the conventional co-simulation approach in terms of modeling and time for simulation. In terms of modeling, each architecture is abstracted with a model of its architectural elements and behavior. A model is composed of functionality description including data processing and data transfer, which are synchronized from time to time according to the abstraction level. Hence, it is easier to describe a model for architecture-level simulation than co-simulation. In terms of time for simulation, architecture-level simulation is faster than the co-simulation since architectural details, e.g. synchronize signals, are ignored.

Describing an architecture with a model for architecture-level simulation enables modeling and simulation in several abstraction levels. In Metropolis simulation environment, metamodel describes an architecture as processes, which represent sequential data processing, and media, which represents data transfer path [46]. A model in Ptolemy II simulation environment describes an architecture with networks of actors, which are components executing concurrent data processing and sharing data via message passing, and directors, which specify model-of-computations (MoCs), such as process networks, discrete-events, synchronous dataflow models, of the actors [47, 48]. In addition, fast co-simulation

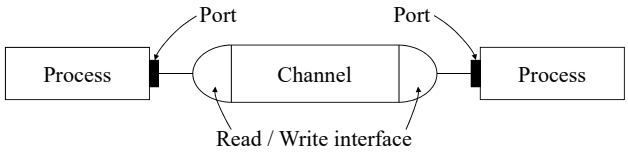


Figure 2.2: An example of model for system-level simulation.

models describe HW/SW platform at an arbitrary abstraction level and provide HW/SW co-simulation interface [49–51].

These models are efficient for performance estimation of various architectures before detailed design. They can accelerate the simulation by the orders of magnitude. Even though the estimation is less accurate than the co-simulation approach, the obtained estimation accuracy remains acceptable in most cases. However, employing architecture-level simulation in the architecture exploration suffers from two problems. First, the models must be rebuilt and simulated for each of the architectures. Second, the simulation speed is still slow. The simulation of an architecture-level model is approximately 1.5 times slower than the system-level simulation [52]. For that reason, the architecture-level simulation is too slow to evaluate a large number of architecture candidates in the design space.

System-level Simulation

System-level simulation achieves faster performance estimation of various hardware-software systems than RTL simulation by abstracting data transfers, aka transactions, at several abstraction levels. Typically, functionality of architectural elements, including functional blocks and bus architecture, are modeled as modules and data transfers are modeled as channels using high-level languages, such as SpecC [53] and SystemC [37], as shown in Fig. 2.2. The detailed signals of data transfers are abstracted in various levels of abstraction, i.e. cycle accurate (CA), bus cycle accurate (BCA), and transaction level model (TLM) [37]. The higher abstraction provides faster simulation trading estimation accuracy.

CA- and BCA-level simulation approach achieves an extremely accurate estimation at the cost of simulation speed and modeling effort. Several studies provide fast simulation platforms that model the multi-processor systems using SystemC at the cycle-accurate and signal accurate level of simulation timing [54–57]. With these methods, standard bus protocols of on-chip communication architecture can be modeled in terms of arbiter, decoder and protocol's cycles so that dynamic behavior of bus according to bus contention and bus protocol is captured precisely, which results in an accurate performance estimation [58–60]. The speed of CA- and BCA-level simulation is likely to be 10-100 times faster than the speed of RTL simulation and requires 3-10 times less modeling effort [61, 62].

TLM-based approach accelerates system-level simulation by using read and write function calls to capture the beginning and the end of transactions. The ar-

chitectures are modeled at timed- and untimed-transaction level to enable both fast functional verification and performance evaluation [63–65]. TLM-based methods enable a fast evaluation of several bus architectures, i.e. topology and protocols, because they abstract the details of transactions with read and write interface of channels [66, 67]. Additionally, an AMBA standard bus protocol modeled in [68] is capable of capturing arbitration behavior and bus contention. Pasricha *et al.* propose a novel cycle count accurate method at transaction boundaries (CCATB) in evaluating AMBA shared bus protocols [69, 70] and multi-layer bus protocols [71] . The CCATB maintains cycle-count accuracy at the beginning and the end of transactions without updating system's state at every cycle. Similarly, another cycle count accurate model is proposed for modeling bus components, such as arbiter and decoder, and approximating bus' arbitration and decoding cycles in order to obtain the accuracy close to the cycle-accurate simulation [72, 73]. The timed TLM simulation based on the system modeling in SystemC language, is about 10-20 times faster and requires approximately 2 times less than the BCA's [61, 70].

System-level simulation is further improved with several techniques. In terms of simulation capability, since simulators of system-level languages provide simulation environment at multiple abstraction levels, CA, BCA, and TLM model, which are described in SystemC are leveraged simultaneously to compromise simulation speed and estimation accuracy [74, 75]. In terms of simulation speed, threads are exploited to accelerate the simulation. Time-decoupling technique is deployed to reduce synchronization between modules so that SystemC kernels can simulate the behavior of components in the model in parallel with multiple threads [76, 77]. Parallel SystemC kernel technique analyzes SystemC code to find available independent threads in order to leverage multi-core host machine [78–81].

System-level simulation method is accurate enough for evaluating the performance of various bus architecture at an early design stage because it can capture dynamic behavior of bus. However, there exist two problems in deploying SystemC-based simulation into the architecture exploration. First, the speed of high-level simulations, e.g. simulating timed-model, BCA model, or CA model using system-level languages, is still slow to evaluate a vast amount of architectures. Second, the performance estimation of each architecture requires an individual high-level abstraction model, which takes up to 3 and 4 days of modeling effort to create timed- and BCA model for each architecture, respectively [69].

2.1.2 Static Performance Estimation

Most fast estimation methods employ a static MoC to describe data processing and data transfers. They analyze system execution using a graph-like MoC, such as Petri Nets, synchronous data flow (SDF), and time marked graph. In addition, static approach is capable of verifying deadlock [28, 29] and modeling pipeline

behavior [30, 82, 83]. For a more accurate estimation of time-varying application, some static performance estimation methods apply statistical concept to obtain a performance estimate based on statistical distribution rather than a worst-case timing [31, 84, 85].

In evaluating communication architecture, static performance analysis also offers analytic properties in modeling latency and detecting deadlock. The work in [86] aims to analyze and reduce the number of on-chip sharing buffer with integer linear programming. In [28], a formal model is used for approximating the performance of AMBA shared bus and detecting a deadlock. In [87], the architecture-specific overhead and latency of hierarchical shared bus are analyzed using SDF. The work in [88] predicts bus arbitration's stall cycle due to bus contention using a statistical model. Cho *et al.* estimate system bus latency for both shared bus and multi-layer shared bus [32]. While these studies focus on hierarchical shared bus, the work in [33, 89] study multi-layer bus, aka crossbar bus. The work [89] proposes an analytical model and predicts performance distribution using tasks' time variation for crossbar bus-based multi-processor SoC (MPSoC). The method in [33] analyzes performance of multi-layer bus considering single-master clusters and slave clusters.

Even though performance estimation methods using static MoC provide a very fast estimation, they suffer from low estimation accuracy. That is because they fail to capture and predict dynamic bus behavior due to bus contention and bus protocols. Furthermore, most studies do not consider several popular configurations of multi-layer bus, such as multiple-master cluster or hierarchical shared bus subsystem cluster, which attaches to a port of multi-layer bus.

2.1.3 Hybrid Performance Estimation

Hybrid performance estimation approach between simulation and static estimation is accurate and fast. It is divided into two categories: (1) static trace-based simulation and (2) dynamic trace-based performance analysis.

Static trace-based simulation collects events from static functionality analysis. TAPES simulation framework generates resource traces from system's functionality profiling and translates them as transactions on each architecture during simulation [90]. A dynamic trace-based method constructs temporal order of execution as concurrent execution traces on given resources, and then, emulates the traces on a hierarchical shared bus-based architecture. In [91], a simulation leverages dynamic cost of shared bus arbitration predicted by a statistical model using workload trace in order to save time from simulating arbitration on every bus access. Nevertheless, these methods require lengthy simulation for every architecture.

Dynamic trace-based performance analysis consists of two steps: data processing and data transfer event trace extraction from simulation and static performance estimation considering traces. SystemBuilder generates event traces based on

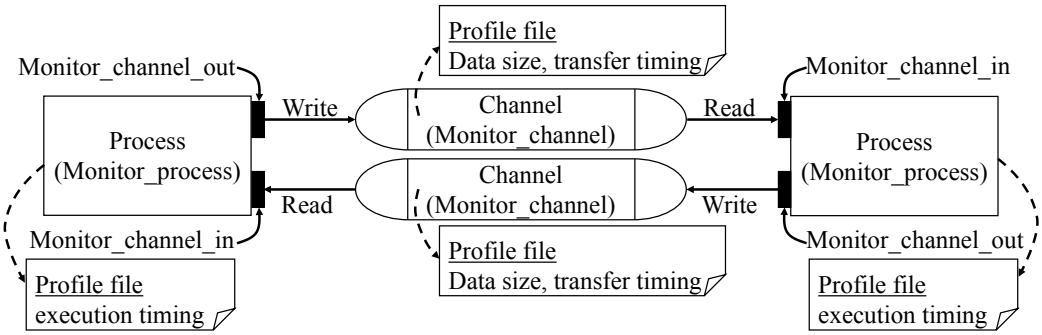


Figure 2.3: System implementation for system-level profiling in SystemC.

FPGA simulation and analyzes the traces according to each architecture template to estimate execution time [92]. The hybrid analysis of SystemC model collects execution traces of an application based on dynamic simulation and analyzes the trace for race condition and rescheduling [93]. Nevertheless, they consider neither bus topology nor bus protocols. Takahashi *et al.* propose a bus architecture independent simulation model that collects communication traces including transactions' start time, quantity of data and destination [94]. Lahiri *et al.* propose a method that collects data processing and transfers event by cosimulation and constructs a communication analysis graph (CAG), which describes data processing and transfers as nodes and execution dependencies as edges [95]. Then, it analyzes the CAG according to bus architectures to obtain execution time. The method proposed in [96] performs timed-functional simulation to gather the order of processing and communication events as traces, and repeatedly analyzes bus arbitrations, preemption and overhead based on bus protocol for various bus architectures using the traces. The latter three methods are sufficient for timing analysis of various bus architectures because the lengthy simulation take place only once for the same set of functional blocks and various bus architectures. However, remodeling and simulation for trace extraction are still needed to collect the traces for architectures that contain different functional block set and they focus only on hierarchical shared bus platform. Yet, the overall estimation for architecture exploration takes too long.

A performance estimation method based on system-level profiling [52], our preceding study, consists of four steps. First, a system-level profiling is performed by a SystemC simulation of a loosely-timed system-level model, called system-level model (SLM), where data processing and transfers are described as processes and channels, respectively. To profile the execution timing, processes and channels are encapsulated with monitor_process and monitor_channel class, respectively, as shown in Fig. 2.3. The system-level profiling then generates profiling information including traces of data processing timings from monitor_process class, and traces of the amount of transferred data and transfer timings from monitor_channel class. It is at least 20 times faster than BCA model's simulation.

Second, the method constructs System-Level Execution Dependency Graph (SL-EDG) according to the profiling information, in which vertices represent data processing and transfers, and edges represent execution dependencies. The system-level profiling and SL-EDG construction take place only once because they are architecture independent. Third, Architecture-Level Execution Dependency Graph (AL-EDG) is constructed by adding edges representing dependencies raised by the availability of buffer resources. Finally, the AL-EDG is analyzed to estimate system execution time. Although the method is effective in terms of time spent for the estimation process, it has three main limitations. First, the analyzable bus model is limited to shared bus and a data transfer must be conducted by only one bus. Consequently, each system-level channel must occupy a dedicated functional block's port that is connected to a bus. Second, the assumption regarding data communication does not satisfy master-slave communication concept, which exists in most high-speed bus protocols. Third, it does not consider bus protocol. The performance analysis models neither dynamic bus behavior nor deadlock state. For that reason, actual bus operations are ignored and deadlock cannot be detected.

In this thesis, the multi-layer bus architecture and bus protocols are studied and modeled in order to efficiently estimate performance of SoC architecture. The architectural model is extended so that it can also represent configurable multi-layer bus, memory, and direct memory access controller (DMAC) engaged in data communication. Communication port model is also improved to indicate master-slave roles of ports on the connecting bus and specify port sharing among multiple channels. The AL-EDG is constructed according to data communication path including memories and DMAC specified by the architectural model. In the analysis procedure, master or slave roles of communication ports and buffer status are also considered when analyzing bus requests. Furthermore, bus contention is recognized in order to predict probable dynamic bus behavior, i.e. split, retry and preemption operation. With the proposed method, the performance of SoC can be evaluated quickly and accurately, which will be discussed in Chapter 3.

2.2 CNN Accelerators

This section explains the prior-art CNN accelerators in terms of four techniques: data-reuse maximization, data precision minimization, calculation-skip maximization and parallel computation maximization. The CNN accelerators exploit one or more of these techniques to achieve real-time performance.

2.2.1 Data-reuse Maximization

Recent CNN accelerators exploit the weight sharing property and data locality within a convolutional layer to maximize data-reuse. They reuse input feature

maps (IFMs), kernels, and output feature maps (OFMs) in on-chip memory to reduce high-latency and energy-consuming external memory access through dataflow pattern and data tiling. Hence, data reuse improves performance and reduces power consumption.

Efficient dataflow promotes data reuse in four major patterns. First, weight-stationary dataflow pattern maximizes weight reuse in the processing elements (PEs), and shifting IFMs and OFMs to the neighboring PEs [97–100]. Second, the output-stationary dataflow pattern maximizes output data reuse by accumulating the OFMs locally in the PEs, while circulating the weights and/or IFMs during the computation [101–104]. Third, global reuse dataflow pattern reuses both weights and IFMs from the global on-chip memory [34, 105–107]. Fourth, row-stationary dataflow maximally reuses weights, IFMs, and OFMs locally in a row unit [108].

Data tiling partitions and processes IFMs in small tiles [34, 105, 109] to reuse IFMs with all the kernels. The SCNN [110] maps data tiles onto its PEs in order to reuse both IFMs and OFMs locally without inter-layer external memory access.

The proposed parallelism-flexible convolution core in chapter 4 exploits data tiling and output-stationary dataflow pattern to distribute kernels and reuse IFMs and OFMs locally. Both techniques enable calculation-skip without complex dataflow control to access IFMs or sparse weights, where the execution time is reduced by the degree of sparsity.

2.2.2 Data Precision Minimization

Data precision minimization is achieved through a quantization method that reduces the number of required bits for CNN computation without the loss of accuracy. Several techniques quantize arithmetic precision of kernels, IFMs, and OFMs from floating point to a few bits of fixed-point precision. Several studies quantize the values of kernels, IFMs, and OFMs into a dynamic fixed-point precision, in which each layer of the CNN uses its layer-wise precision for the convolution [22, 36, 111]. The method in [22] analyzes the activation values from inferencing a large set of input images with a floating-point CNN and determines a fixed-point precision for the kernels, IFMs, and OFM layer by layer. Similarly, the method in [36] analyzes the values and quantizes the CNN with singular value decomposition technique. The method in [111] employs approximation approach to reduce bit width layer by layer. The study in [20] encodes the kernels into a codeword that represents a value by weight sharing technique to further reduce the total size of the kernels.

Another key success in quantization is fine-tuning. The study in [111] proposes a framework that reduces the bit width of kernels, IFMs, and OFMs layer by layer, and fine-tunes the CNN to assure the accuracy. The quantized CNN method quantizes the weights of the CNN using k-mean clustering to minimize the estimation error of each layer's response and proposes a training scheme to suppress the accumulative quantization error by paying a layer-wise training cost

for correcting errors in each layer [21].

This optimization lowers both computational resource per one MACC and storage requirement of the customized hardware. The study in [112] has shown that 16-bit fixed point precision is required to preserve the accuracy of ImageNet classification [113]. Furthermore, the SCNN computes a sparse CNN with 16 bits for multiplication and 24 bits for accumulation [110].

The above-mentioned studies have shown that 16-bit fixed-point precision for multiplication and 32-bit fixed-point precision for accumulation are sufficient for image recognition without sacrificing recognition accuracy. Therefore, the proposed parallelism-flexible convolution core performs the computation of CNN using 16 bits for multiplication and 32 bits for accumulation. The 32-bit accumulation is implemented instead of 24 bits like the SCNN for preventing the accumulation overflow.

2.2.3 Calculation-skip Maximization

Calculation-skip maximization omits zero-operand MACC from the sparsity in IFMs and weights of the kernels. It reduces the number of MACCs involved with non-zero weights and can accelerate CNN inference computation by the degree of sparsity. Sparsity in IFMs comes from activation functions such as Rectified Linear Units (RELU), which rectifies the values less than zero into zero. Weight pruning process introduces sparsity in weights by zeroing out weight values with the trade-off between the number of remaining weights and recognition accuracy. Many state-of-the-art studies have shown that more than 80% of weight sparsity is possible without jeopardizing the accuracy [20, 23].

Unlike dense CNN, accessing weights and IFMs of sparse CNN has irregular patterns that may incur complex control. Recent accelerators exploit weight sparsity or activation sparsity or both. The ones that exploit weight sparsity usually use kernels in sparse format to access non-zero weights and skip MACCs having zero-valued weights efficiently with the output-stationary dataflow pattern [109, 114]. The architectures that leverage IFM sparsity usually include a zero-detection mechanism to dynamically skip zero-operand multiplication [110, 112]. As a result, these architectures achieve performance improvement over the dense CNN accelerator.

The proposed parallelism-flexible convolution core computes the sparse CNN because the above-mentioned studies have shown that the weight sparsity at a certain level does not degrade the recognition accuracy while it is capable of reducing the number of calculation. The proposed convolution core leverages weight sparsity from a compressed CNN model, which is a format of representing a sparse model, in a straightforward way. Hence, the proposed convolution core skips the computation related to the zero-valued weights efficiently.

2.2.4 Parallel Calculation Maximization

To maximize parallel calculation, CNN accelerators schedule MACCs exploiting various types of parallelism onto their vast amount of multipliers. The reconfigurable processor array maps intra-output parallelism onto its PEs [101]. Many high-performance architectures schedule multiple types of parallelism onto multipliers by rows, columns, or groups of PEs [34–36]. The types of parallelism are (1) intra-output parallelism, which is the parallelism in computing the output pixels of the same OFM; (2) inter-output parallelism, which is the parallelism in computing several OFMs; (3) operation-level parallelism, which is the fine-grained parallelism in computing multiple multiplications of one output pixel. However, they cannot achieve high performance in terms of giga operations per second (GOPS) in all the layers because the scheduling is fixed, while the dominant parallelism of in each layer usually varies across the CNN with the different layer specification, such as the size and number of OFMs.

To further increase parallel calculation in every layer, the architecture should flexibly schedule MACCs onto the multipliers according to the dominant parallelism of each layer. The FlexFlow architecture [115] adjusts its scheduling of multiple types of parallelism to improve multiplier utilization layer by layer. Even though it achieves near peak performance, it neither supports the compressed CNN model nor exploits sparsity efficiently because it exploits operation-level parallelism. It is difficult to skip zero-operand MACCs while exploiting multiple types of parallelism, especially operation-level parallelism, without either complex dataflow control mechanism or wasting a vast number of multiplier cycles. That is because irregular sparsity pattern disarranges non-deterministic weight, IFM, and OFM access.

The proposed parallelism-flexible convolution core in chapter 4 integrates the above-mentioned acceleration techniques. Namely, it employs output stationary dataflow in order to reuse output OFMs locally. At the same time, output stationary dataflow benefits in leveraging weight sparsity by broadcasting only non-zero weights to the PEs. The sparse memory access to the IFMs is simplified by using the indices of the non-zero weights as an address to access the consecutive IFM pixels located in the PE's local buffer. Likewise, the indices are used to access the OFMs in the partial sum buffer. With our partial sum buffer layout, all or groups of the PEs can access to the same address, which reduces the irregular memory access patterns. The proposed convolution core alternates dataflow to exploit multiple parallelisms, i.e. intra- and inter-output parallelism, according to convolutional layers' specification. It leverages all the techniques to accelerate the computation of convolutional layers effectively.

2.3 Architecture Design Space Exploration

This section explains related studies regarding architecture design space exploration. First, it overviews architecture exploration frameworks in general. Then, the studies about the communication architecture exploration are reviewed. The communication architecture includes bus architecture, memories, and DMACs. Finally, this section describes the studies that propose architecture exploration frameworks for CNN.

2.3.1 Architecture Exploration

Heuristic architecture exploration methods are proposed to search optimal SoC architectures within the design space. Since the design space becomes extremely huge as the choices of architecture, such as IPs and communication architecture, grow, exhaustive search takes unacceptably long time. Therefore, heuristic architecture exploration methods employ algorithms to efficiently eliminate architecture candidates in the design space, aka design space pruning. These heuristic architecture exploration methods and their algorithms are divided into two categories: structured architecture exploration methods and randomized architecture exploration methods.

The structured architecture exploration methods refer to the methods that explore and prune the design space systematically. They traverse the design space through a data structure, such as search tree [116], using search algorithms like tabu search [117]. To fasten the exploration, the design space is pruned by constraints and results of previous search [118–120]. Some methods additionally employ machine learning algorithm in assisting design space pruning. The machine learning model is trained with the past architecture configuration and design quality pairs, and either predicts non-pareto elimination for pruning [121] or samples high quality architectures [122, 123]. However, these methods do not focus on communication architecture. They either assume virtual bus architecture or hierarchical shared bus. Furthermore, the prediction of machine learning model is not accurate enough to guarantee Pareto solutions.

On the other hand, the randomized architecture exploration methods refer to the methods that search the design space through architecture transformation. These methods employ optimization algorithms such as genetic algorithm (GA), simulated annealing algorithm (SA) and evolutionary algorithm. GA is often used in improving search behavior to find Pareto-optimal architectures through crossover or mutation operations on the genes to generate architecture candidates [124, 125]. SA is employed for hardware/software partitioning because of its ability in escaping local minima [126, 127]. The work in [128] evaluates multi-objective evolutionary algorithm (MOEA) in solving mapping problem and shows that MOEA finds optimal architectures within a reasonable amount of time. However, these randomized architecture exploration methods do not guarantee optimal

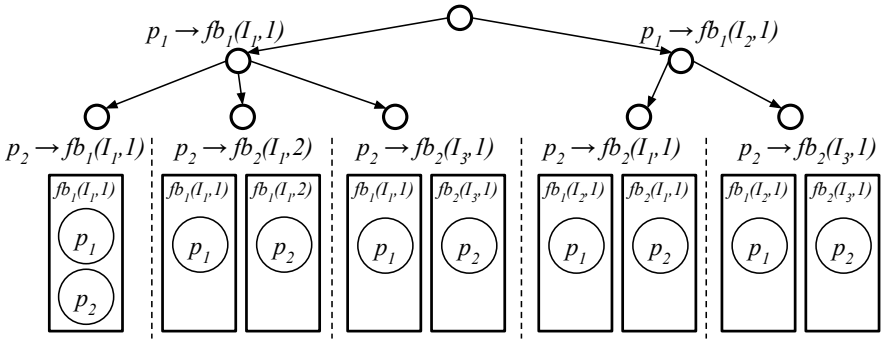


Figure 2.4: An example of process mapping.

solutions and are sensitive to initial solution.

In terms of finding optimal parameters, the architecture exploration method parameterizes SoC architecture and explores parameter values of functional blocks and communication architecture. Givargis *et al.* propose a parameter exploration method for an SoC architecture [129]. It is very useful for exploring parameters for a specific functional block set and a bus architecture, e.g. the cache size and the bus width. However, it is not suitable for exploring the functional blocks and communication architecture themselves. Matai *et al.* provide a method that composes previously designed components in SoC design for a new application and optimizes the design by selecting parameters of the components [130]. It resembles the proposed method in this thesis in terms of IP (component) mapping and IPs' parameter mapping. However, they do not consider communication mapping to multi-layer bus.

Ueda *et al.* propose an architecture exploration framework based on system-level modeling [131], which is our preceding study. From an SLM which describes data processing and data transfers in an application, it explores the design space of the target architecture to find Pareto-optimal solutions in terms of performance, area, and power consumption. The target architecture includes functional blocks, hierarchical shared bus, and buffers. There are five inputs to the exploration systems, which are IP database, bus database, design constraints, SLM, and profiling information.

Given design constraints and parameter candidates of the architecture to be explored with the user-defined values, the method explores the design space by traversing through the parameter set search tree to construct an architecture candidate and explore its parameters. The procedures of the method are as follows.

1. **Process mapping** selects a functional block for the data processing of each process as shown in Fig. 2.4. In the figure, $p_i \rightarrow fb_j = (I_k, l)$ denotes that the process p_i is mapped to functional block fb_j, which is the l^{th} instance of IP I_k. It is assumed that p_1 can be executed on I_1 and I_2, and p_2 can be executed on I_1 and I_3. First, p_1 is mapped on to the first instance of either I_1

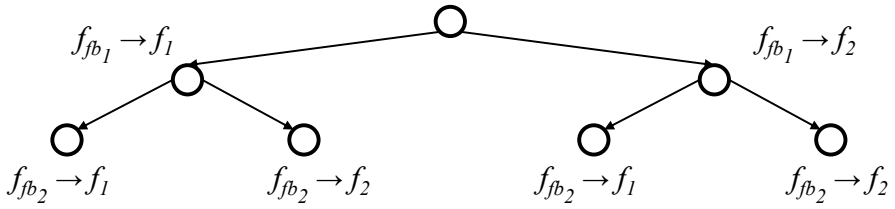


Figure 2.5: An example of functional block's execution frequency mapping.

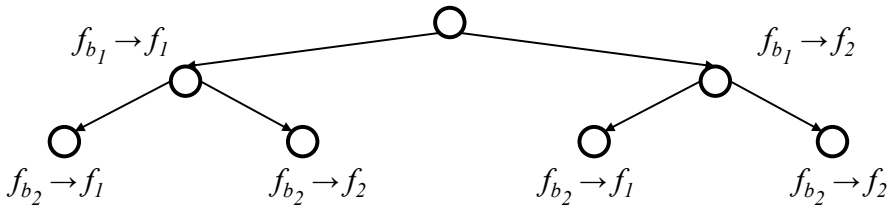


Figure 2.6: An example of bus' execution frequency mapping.

or I_2, constructing the first functional block fb_1 to the architecture. Then, p_2 can be mapped on to a new instance of I_1 or a new instance of I_3. It can also be mapped to the first functional block fb_1 if fb_1 is an instance of I_1. In the case that a new instance is constructed, fb_2 is added to the architecture.

2. **Channel mapping** selects a bus for each channel.

3. **Functional block's execution frequency mapping** determines the execution frequency of each functional block. The execution frequency candidates of each functional block are registered in the IP database. Figure 2.5 is an example of functional block's execution frequency mapping when there are two functional blocks, fb_1 and fb_2, and there are two execution frequency candidates, f_1 and f_2.

4. **Bus' execution frequency mapping** determines the execution frequency of each bus. The bus' execution frequency is selected from a set of user-defined bus' execution frequency candidates. Figure 2.6 is an example of bus frequency mapping when there are two buses, b_1 and b_2, and there are two execution frequency candidates, f_1 and f_2.

5. **Bus width mapping** determines the width of each bus in the architecture. The parameter value is selected from a user-defined candidate set.

6. **Number of buffer mapping** determines the number of buffers for storing data in each channel. The candidates of number of buffers are user-defined.

To quickly search the design space, Ueda *et al.* prune the parameter set search tree. The descendants of the parameter set search tree are pruned when one of the following conditions is met.

- One or both lower bounds of the execution time and the hardware area of the current search node exceed the user-defined design constraints.
- The lower bound and upper bound of the execution time are equal.
- Both lower bound of the execution time and the hardware area of the current search node exceed those of the explored optimal architecture at that moment.

2.3.2 Communication Architecture Exploration

The design space of on-chip communication architecture broadens with the evolving topologies of bus architecture. Several high-performance and low-power bus topologies, such as hierarchical shared bus, multi-layer bus (aka crossbar bus), and network-on-chip (NoC), have been developed in order to support the communication between the increasing number of functional blocks within an SoC. Figure 2.7 illustrates three types of bus architecture: a hierarchical shared bus, multi-layer bus, and cascaded multi-layer bus. The hierarchical shared bus includes multiple shared buses which are connected with bus bridges. The multi-layer bus includes a bus matrix fabric, aka crossbar, which contains multiple buses to parallelize the data communication from different master-slave layer. The cascaded multi-layer bus is a hierarchy of multiple bus matrix fabrics which are connected with bus bridges.

Communication architecture exploration finds optimal bus architecture and components, i.e. DMAC and memories, associated with the data transfer between functional blocks. To efficiently explore communication architecture, including buses, memories, and DMAC, fast communication architecture exploration methods have been proposed.

Hierarchical Shared Bus-based Communication Architecture Exploration

The communication architecture exploration based on bus templates provides fast and systematic way to explore the communication architecture design space. Gong *et al.* develop a method that transforms functional specification to an implementation model and explore communication architecture using four fixed templates [132]. Lahiri *et al.* automate a bus architecture exploration by mapping the data transfer to the selected bus template [133]. It determines an initial bus architecture and iteratively optimizes the bus architecture protocols to maximize system performance. However, it is not suitable for exploring various bus architectures because a large amount of bus templates must be prepared for various bus architectures, and hence, the available bus templates limit the design space.

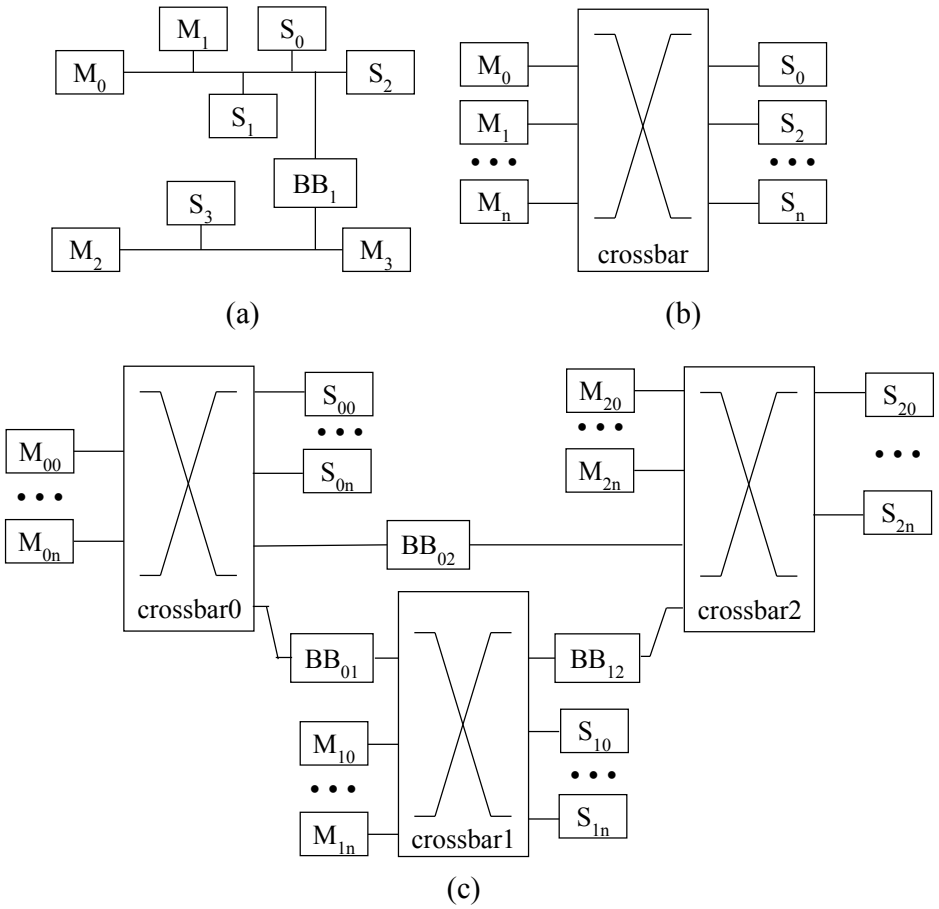


Figure 2.7: Three types of bus architecture (*M* refers to master, *S* refers to slave, *BB* refers to bus bridge): (a) hierarchical shared bus; (b) multi-layer bus; (c) cascaded multi-layer bus.

Bus synthesis methods generate both bus topologies and parameters. Parischa *et al.* propose a bus architecture synthesis method aiming to find the optimal-area design under performance constraints [69–71]. This method can effectively generate parameters of a bus architecture, which include the number of buses, bus width, bus frequency, etc. Pandey *et al.* synthesize the communication architecture topology and optimize the architecture in terms of bus width and number of buses [134]. These methods widen the design space in terms of bus architecture, but they are not suitable for the functional block set exploration.

Exploring communication also includes other components along the communication path of a data transfer, e.g. bus bridges and memories. Kim *et al.* develop a method that explores bus bridge and memory allocation in addition to bus topologies and other parameters, such as bus arbitration scheme, frequency, and bus width [135]. It starts with one bus and then, scatters communication traffics into multiple buses to reduce bus contention and maximizes concurrency. The method in [136] considers distributed memory system and explores bus architectures.

Bus architecture is optimized through three major techniques based on the mapping of data transfer to communication architecture. The first technique is bus merging to minimize the data transfer between multiple buses because more buses cause more power and performance penalties [137]. The second technique is bus splitting or partitioning to maximize communication parallelism [138, 139]. The third technique involves moving memories or functional blocks between buses [140]. The work in [141] develops new communication architecture using these bus transformations [141]. These techniques aim to achieve high performance with less physical cost. Nevertheless, all of the above-mention methods are limited to hierarchical shared bus architecture, which is shown in Fig. 2.7(a).

Multi-layer Bus-based Communication Architecture Exploration

To obtain a higher performance platform through communication architecture, several design space exploration methods focus on various kinds of multi-layer bus architectures. A flat multi-layer bus architecture as shown in Fig. 2.7(b), referred to as multi-layer bus or crossbar bus, includes one bus matrix fabric, aka crossbar. Each master and slave are connected to each port of the crossbar. A cascaded multi-layer bus as shown in Fig. 2.7(c), referred to as cascaded multi-layer bus or cascaded crossbar bus, includes more than one crossbar connected with each other via bus bridges. Most studies optimize the topology of multi-layer bus by clustering master and slave hardware components in the system.

Murali *et al.* propose an application-specific on-chip crossbar generation method considering floorplanning [142]. It clusters the master and slave cores based on a pair-wise overlap traffic trace among different cores of the full crossbar. Although the method can generate many crossbar topologies with multiple master clusters, multiple slave clusters, and subsystem clusters, it does not explore the parameters of the crossbar, which have a large impact on the design quality.

Pasricha *et al*. propose a bus matrix generation method based on AMBA advanced high-performance bus (AHB) [143] and AXI [144] that explores optimal bus matrix architecture [71, 145]. It optimizes bus topology using a static branch and bound-based slave clustering and explores its parameters e.g. bus frequency. The method uses static bandwidth analysis to prevent exploration in an invalid design space. In addition, it minimizes wire congestion by removing unnecessary buses on the bus matrix fabric. Even though the method successfully finds the topology and parameters of the bus matrix, the optimization is limited to only a fixed set of functional blocks.

Lee *et al*. develop a bus matrix synthesis method that determines bus matrix topology via an interface selection [146]. The method optimizes multi-layer bus by systematically selecting master or slave interface for each hardware component. Then, it clusters the master and slave components into multiple master and AHB subsystem clusters by analyzing communication conflicts of the hardware components that are merged into the same cluster. However, it does not consider grouping multiple slaves in the same cluster to further reduce physical cost of bus matrix fabric.

The studies on cascaded multi-layer bus aim to simplify each crossbar. Joo *et al*. explore cascaded crossbar in terms of topology, frequency, arbitration, and off-chip memory allocation [147]. Jun *et al*. exploit locality principle to merge crossbars in order to reduce bus matrix [148]. The target architectures of both studies are different from this thesis. They focus only on cascaded crossbar switches. However, this thesis focuses on optimizing a multi-layer bus architecture shown in Fig. 2.7(b) by clustering components attached to the bus matrix fabric.

Cilardo *et al*. propose a communication architecture synthesis method including hierarchical shared bus, multi-layer bus, and cascaded multi-layer bus [33]. The method determines crossbar topology by clustering master-slave components and optimizes scheduling to satisfy temporal bounds. This work comes closest to the research in this dissertation. Nevertheless, there are two differences: (1) it does not consider multiple master clusters; (2) it does not mention about how to choose bus protocols and parameters.

This thesis focuses on exploring the topology and parameters of a flat multi-layer bus. It partitions master and slave components into multiple master, multiple slave, and AHB subsystem clusters in order to determines the topology. It selects parameters, such as protocols, frequency, and bus width, through a mapping using parameter set search tree.

2.3.3 Architecture Exploration for CNN-based Platform

DeepBurning automation tool provides a platform to implement NN in FPGA and application-specific integrated circuit (ASIC) [149]. It analyzes NN model, maps the NN's computation onto the hardware building blocks in its NN component library to leverage several parallelisms, and generates RTL, control flow

and memory image for input data and weights. The output architectures are optimized for each NN. However, the method focuses on only the NN itself, and does not consider other processing within an application, such as pre-processing and post-processing.

Hong *et al*. propose a dataflow modeling method for applications that contain loop structure like NN [150]. It specifies loop structures in an SDF graph and maps them on to the hardware in order to maximize usage of the given architecture. This allows the exploitation of multiple parallelisms within an NN. Nevertheless, this method targets only the multi-core platform.

Tsimpourlas *et al*. develop a design space exploration framework for CNN targeting edge devices [151]. Given a CNN-based application and task mapping, this method explores the parameters of the architecture, e.g. the number of PEs, to optimize the architecture in terms of execution time and energy consumption. However, it does not mention application tasks' mapping. It optimizes the architecture of existing platform, such as Intel's Movidius Myriad2. In addition, it does not consider accelerating the CNN computation with the concurrency of data transfer. In other words, it does not take multi-layer bus into account.

Chapter 3

An Efficient Performance Estimation Method for Configurable Multi-layer Bus-based SoCs

Chapter 3 describes the proposed efficient performance estimation method for configurable multi-layer bus-based SoC. It is suitable for evaluating system performance in an early stage of the design process because it is both fast and accurate. First, this chapter explains bus architecture and standard bus protocol. Second, it defines the MoC and the architectural model. Then, the problem formulation and procedures of the proposed performance estimation method are explained in details. Next, the experiments are conducted by modeling advanced microcontroller bus architecture (AMBA) advanced high-performance bus (AHB) and advanced peripheral bus (APB), and applying the proposed method to JPEG encoder application. Finally, this chapter is summarized.

3.1 Motivation and Objective

When the number of processing cores grows from tens to hundreds, a multi-layer bus architecture is introduced in addition to hierarchical shared bus architecture that becomes systems' bottleneck due to a massive amount of data communications. Standard specifications for the multi-layer bus are developed such as AMBA's multi-layer AHB [143] and AXI [144]. However, a full bus matrix contains a massive amount of wires, which leads to routing and power consumption problems. Therefore, apart from the regulations and communication methods, the specifications also define configurations of the multi-layer bus as well as the model of the bus matrix, which represents the interconnect of the multi-layer bus. The configurations reduce the number of wires on the bus matrix because they merge some master and slave layers and the buses on the bus matrix that do not conduct any data transfer are removed. This wire reduction eases the routing and power problems. Nevertheless, the performance of the communication might degrade

depending on the selected configurations. Therefore, it is important to evaluate the multi-layer bus configurations and find an optimal architecture that can satisfy SoC design constraints because the topology, configuration, and protocol of the bus architecture affect the design quality of SoC.

In architecture exploration, performance evaluation is one of the critical parts of the exploration process because system performance, aka execution time, is one of the most important metrics of design quality and performance evaluation consumes a lot of time. Furthermore, an enormous design space of IP selection and bus architectures, including hierarchical shared bus and configurable multi-layer bus, must be evaluated. Therefore, a fast performance estimation approach that can evaluate various architectures accurately is desired.

This chapter proposes an efficient performance estimation method for configurable multi-layer bus-based SoC. The proposed method acquires data flow information from a system-level profiling and constructs a system-level execution dependency graph (SL-EDG). Then, it constructs and analyzes an architecture-level execution dependency graph (AL-EDG) of each architecture to estimate execution time. During the performance analysis, the behavior details of shared buses and multi-layer bus are determined based on the analyzed dynamic bus contention and bus protocols' features.

The key features of the proposed method are as follows:

1. predicting the behavior of shared buses and multi-layer bus during the performance estimation according to the analyzed dynamic bus contention and bus protocols' features;

2. estimating the performance of various architectures according to the speculated buses' behavior by analyzing an architecture-dependent execution graph;

3. by defining the protocol's specific parameters and behavior, the proposed method is applicable to estimate the performance of various bus protocols.

The proposed method estimates the performance of SoC accurately within a short time compared to the RTL simulation.

3.2 Bus Architecture

This section explains the target bus architectures, which includes hierarchical shared bus and multi-layer bus. Their architectures and an example of a standard bus protocol, specifically, AMBA AHB and APB [143, 152, 153], are described in terms of bus model and communication method.

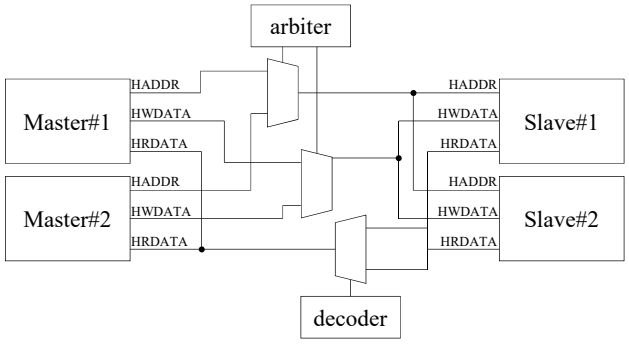


Figure 3.1: AHB bus model.

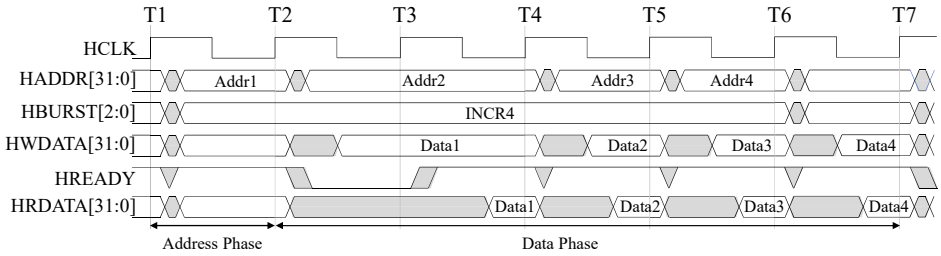


Figure 3.2: Waveform of AHB's four-beat incrementing burst operation.

3.2.1 Hierarchical Shared Bus Architecture

A hierarchical shared bus architecture consists of multiple shared buses. On a shared bus, multiple masters share the same bus to communicate with the target slave and one communication is active at a time. These shared buses are connected to each other with a bus bridge, forming a bus hierarchy in the architecture as shown in Fig. 2.7(a). Two examples of a standard hierarchical shared bus are AMBA AHB and APB [152, 153].

Advanced High-speed Bus (AHB)

The AHB is a multi-master high-performance interconnect. It is comprised of one address bus and two data buses as shown in Fig. 3.1 and operates in half-duplex mode at the high clock frequency. The data transfer takes place between a pair of master and slave when the master's bus request is granted by the arbiter. An AHB bus can handle up to 16 masters.

To accelerate the consecutive data transfer, the AHB enables the pipeline and the burst transfer mode. Figure 3.2 shows a four-beat incrementing burst transfer. The address phase consumes one clock cycle and the data phase consumes at least one clock cycle per one data depending on the slave's status. The pipeline mode allows the last data cycle to overlap with the address cycle of the next transfer in consecutive data transfer.

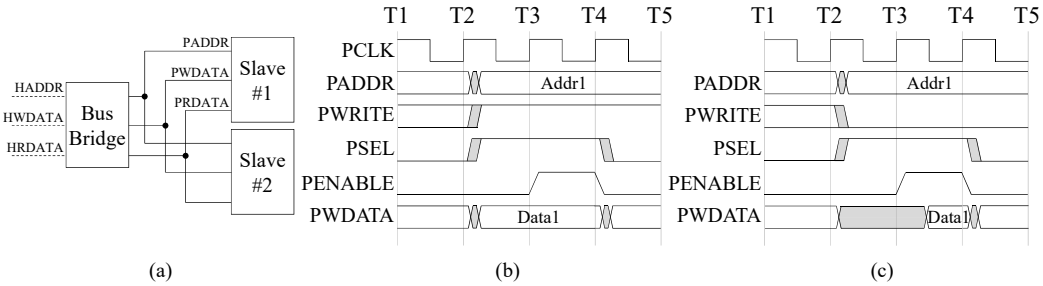


Figure 3.3: Specification of APB: (a) APB bus model; (b) waveform of APB's write transfer; (c) waveform of APB's read transfer.

The AHB provides a split-retry mechanism in the case that slave is unable to complete the request immediately. The mechanism allows the bus to be released for other transfers while the slave is preparing for the request.

This thesis models AHB's behavior with three assumptions. First, the data phase takes one cycle for transferring one data because it is assumed that the data transfer takes place when both master and slave are ready. Second, the arbitration scheme is assumed to be a fixed priority. Third, each transaction is executed with a defined-length incrementing burst protocol.

Advanced Peripheral Bus (APB)

The APB connects low bandwidth peripherals to separate them from the backbone AHB. Figure 3.3(a) illustrates the APB bus model. Any incoming transfer to the APB must go through a bus bridge, which operates as the only bus master. A transfer on the APB bus takes two clock cycles per one data as shown in Fig. 3.3(b) for the write operation and Fig. 3.3(c) for the read operation.

Data Transfer via Multiple Buses

A transfer via two buses goes through a bus bridge. For a transfer via an AHB and an APB, the bus bridge converts between AHB and APB protocol. The waveform of a write operation in Fig. 3.4(a) shows that the operation consumes two cycles on the AHB for the first address and data, and two cycles per data on the APB. Similarly, the waveform of a read operation in Fig. 3.4(b) shows that the operation consumes one cycle on the AHB for the first address and two cycles per data on the APB. Although a transfer through two AHB buses is not defined in the specification, the AMBA design kit [154] provides an AHB-to-AHB bus bridge with zero-cycle overhead.

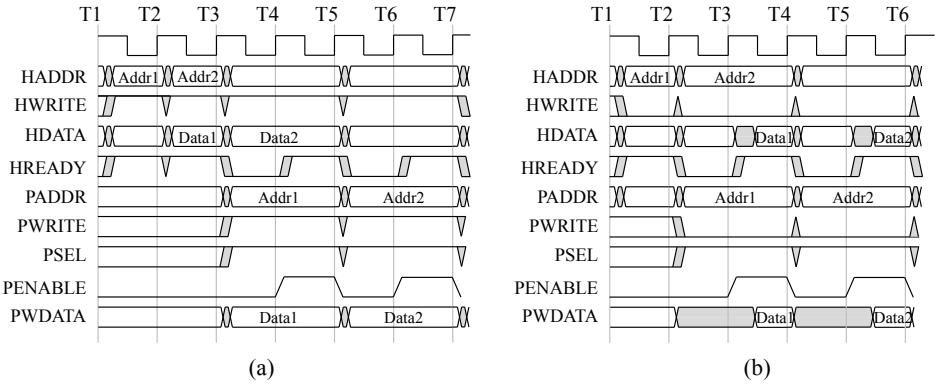


Figure 3.4: Waveform of transfer via AHB and APB: (a) waveform of write transfer; (b) waveform of read transfer.

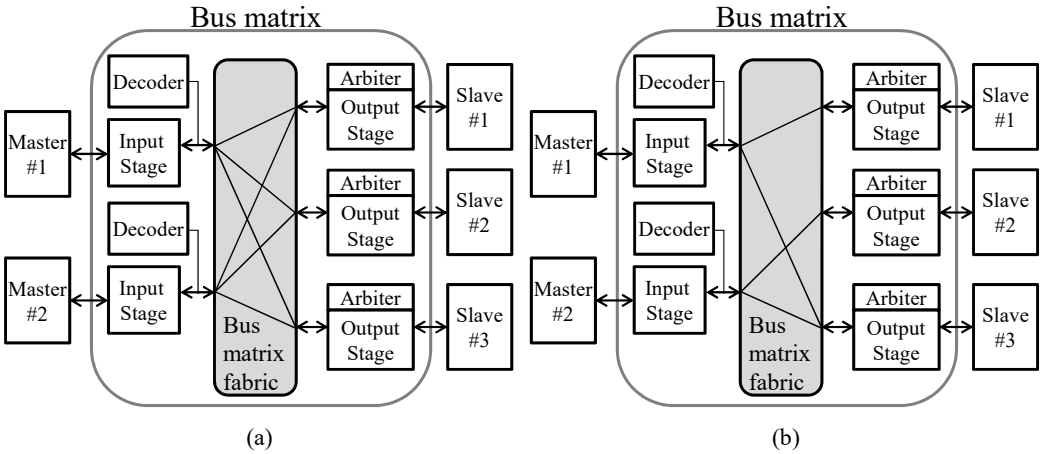


Figure 3.5: Bus matrix topology of multi-layer bus: (a) a full bus matrix topology; (b) a maximally connected bus matrix topology.

3.2.2 Multi-layer Bus Architecture

A multi-layer bus increases multiple-master system's bandwidth by parallelizing multiple communications on the bus matrix architecture. Masters and slaves are connected to master and slave layers of the bus matrix, respectively. Consequently, multiple communications can be active at a time. However, if a shared slave serves more than one master, the bus matrix ought to arbitrate those masters which can access the slave layer of the bus matrix that the slave resides.

The AMBA specification offers a multi-layer AHB with the bus matrix topology to handle a system that requires the high-bandwidth [143]. It consists of five components. First, an input stage buffers address and control signals of an incoming transfer when the slave is busy. Second, a decoder generates slave select signals, and selects response and read data signals. Third, an output stage selects

address, control and write data signals and send the signals to each slave layer. Fourth, an arbiter determines which master can get an access to each slave layer. The arbitration is distributed to the arbiter at each slave layer. Fifth, a bus matrix fabric contains buses that are paths for data transfer.

Figure 3.5(a) shows the full bus matrix topology, where every master layer is connected with every slave layer by buses on the bus matrix fabric. In the case that the buses are not used, they can be removed. Such topology is called a **maximally connected bus matrix** shown in Fig. 3.5(b).

In this thesis, the arbitration scheme of a bus matrix is assumed to be a fixed burst priority. Since the hierarchical shared bus employs a preemptive fixed priority arbitration while the multi-layer bus employs a non-preemptive fixed burst priority arbitration, each transaction through the multi-layer bus must be a lock transaction on the hierarchical shared bus. To sum up, each transaction using a bus matrix is assumed to be a locked defined-length incrementing burst transfer.

Configurable Multi-layer Bus Architecture

To further optimize the multi-layer bus that accommodates many masters and slaves, several bus matrix configurations are introduced [62, 143]. The bus architecture with these configurations is referred to as a **configurable multi-layer bus architecture** and a group of masters and slaves connected to a master layer or a slave layer is referred to as a **cluster**.

In addition to a single-master (Layer 1 of Fig. 3.6) or single-slave (Layer 5 of Fig. 3.6) configuration, each layer of bus matrix can connect to a cluster with one of the following configurations:

- **Multiple masters on one master layer** : A multiple-master cluster includes more than one master as shown in Layer 2 of Fig. 3.6, in which assumes that two masters share the same master Layer 2 of the bus matrix. An arbiter on the master layer determines which master gains a bus access at a time.

- **Multiple slaves on one slave layer** : A multiple-slave cluster combines more than one slave as shown in Layer 6 of Fig. 3.6, where two slaves are merged into one cluster and the slaves appear as one slave to the bus matrix.

- **Local slave** : When a slave is accessed by only one master or multiple masters in the same cluster, the slave can be made local to those masters by attaching it to the master layer as shown in Layer 3 of Fig. 3.6. This is advantageous in terms of reducing bus matrix complexity.

- **Subsystems** : A subsystem integrates masters and slaves into the same cluster and constructs a subsystem as shown in Layer 4 of Fig. 3.6.

In this thesis, the bus matrix configuration is customized through port, direct memory access controller (DMAC) and memory clustering. The configuration

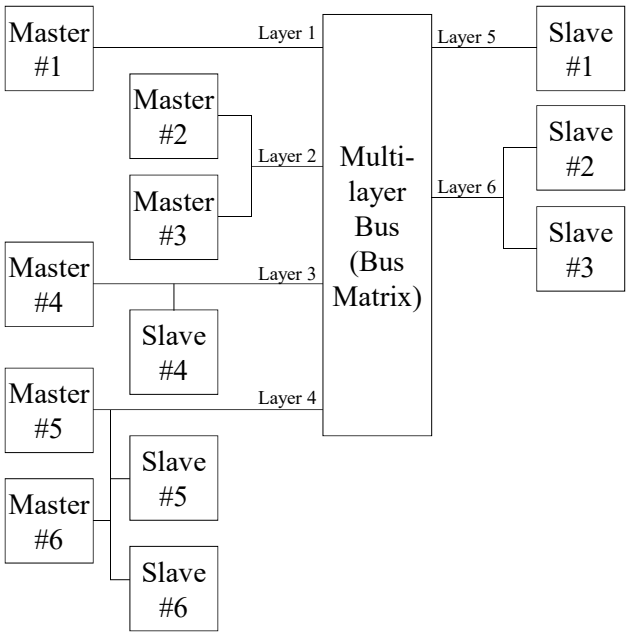


Figure 3.6: Multi-layer AHB bus configuration. Layer1 connects to a single-master cluster. Layer2 connects to a multiple-master cluster. Layer3 connects to a local-slave cluster. Layer4 connects to a subsystem cluster. Layer5 connects to a single-slave cluster. Layer6 connects to a multiple-slave cluster.

depends on the type of components residing in each cluster. Furthermore, the local slave configuration and the subsystem configuration are considered as subsystems for simplicity in modeling.

3.3 Definitions

First, this section explains MoC, architectural model of the configurable multi-layer bus-based SoC, and defines the proposed performance estimation method.

3.3.1 Model of Computation (MoC)

A Kahn-Process Network-based acyclic directed graph called system-level model (SLM) is used as our MoC to specify the behavior of a target system in terms of sequential data computation processes and unbounded first in first out (FIFO) communication channels. An SLM is described as a loosely-timed model according to the transaction-level modeling (TLM) 2.0 specification [37]. The processes that represent data processing are untimed, while the entry points and exit points of channels that represent data transfers are explicitly noted by event triggers.

An SLM is represented by $M_{sl} = (P, C)$, which means that SLM M_{sl} is com-

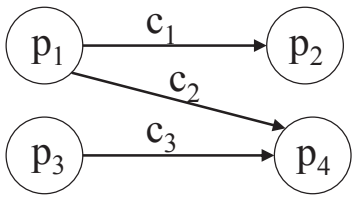


Figure 3.7: An example of SLM.

posed of a process set $P = \{p_i | i = 0, 1, 2, ...\}$ and a channel set $C = \{c_j | j = 0, 1, 2, ...\}$. $c_j = (p_m, p_n)$ represents the channel from process $p_m \in P$ to $p_n \in P$. The data to be used in a process is received through the input channels, executed inside the process and the result is transmitted via the output channels. A write operation to an output channel is a non-blocking operation, while a read operation from an input channel is a blocking one. Additionally, s_j and p_{c_j} represent the data size and the execution priority of channel c_j, respectively.

Figure 3.7 shows an example of an SLM, M_{sl}, consisting of four processes and three channels. An arrow represents the direction of data flow in each channel. For instance, channel c_1, c_2 and c_3 are data communications from process p_1 to p_2, p_1 to p_4 and p_3 to p_4, respectively.

3.3.2 Architectural Model

A configurable multi-layer bus-based architecture consists of IP modules, DMACs, memories, shared buses and/or a multi-layer bus. A multi-layer bus is composed of a bus matrix and the buses on it, which allow parallel communications in a system with multiple masters and slaves. An architecture may contain heterogeneous configurations of a multi-layer bus [143], such as multiple masters, multiple slaves, local slave, and subsystems. Some of the configurations degrade performance, but removing unnecessary buses on the bus matrix and optimizing the bus matrix with these configurations give a benefit in area and ease of routing.

The configurable multi-layer bus-based architectural platform is formalized as an architectural model called architecture-level model (ALM). An ALM describes the components and organization of an architecture, including the information about the process-to-functional-block and channel-to-port mapping decisions. One channel is accounted for the point that the data flows into it, called source of channel, and the point that the data flows out of it, called destination of channel, to be mapped on to the ports that are responsible for the transfers. An ALM is defined with a 7-tuple, (F, PT, D, M, B, BM, BB), each of which has the following characteristics:

- F is a set of IP modules' instances, called functional blocks, that undertake the execution of the system-level processes. $fb_i = (j, P_{fb_i}, f_{fb_i}, e_{(p_k, fb_i)}) \in F$

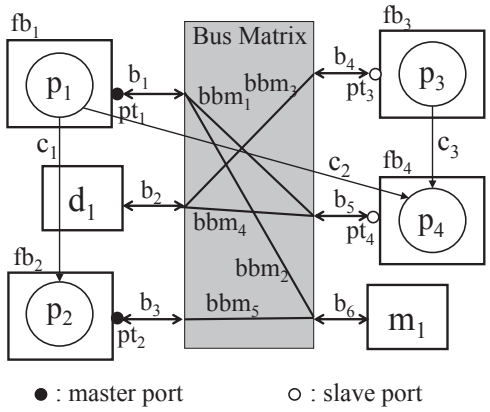


Figure 3.8: An example of ALM.

indicates that functional block i is an implementation of IP j and undertakes the processes in set P_{fb_i}. f_{fb_i} and $e_{(p_k,fb_i)}$ represent the execution frequency of fb_i and execution cycle of process $p_k \in P_{fb_i}$ on functional block fb_i, respectively.

- PT is a set of ports $pt_i = (fb_j, b_k, n_q, n_r)$. A port connects a functional block fb_j to a shared bus b_k and functions as a master or a slave on the connecting bus. A port contains n_q receive buffers and n_r transmit buffers for multiple buffering.

- D is a set of DMACs, $d_i = (C_{d_i})$. C_{d_i} refers to a set of source and destination of channels that require d_i to initiate the transfer. A DMAC controller functions as a bus master to transfer data from a requesting slave, store data in its buffer temporarily and send data to another slave upon request.

- M is a set of memories, $m_i = (C_{m_i}, n_{c_i})$. C_{m_i} refers to the set of source and destination of channels that are conducted via a memory, which serves as a slave in the system. The memory space is divided into storage blocks, and the number of storage blocks to store data of c_i is represented by n_{c_i}.

- B is a set of shared buses, $b_i = (w_{b_i}, f_{b_i}, pr_{b_i}, bmp_{b_i})$, where w_{b_i}, f_{b_i} and pr_{b_i} are b_i's data bus width, frequency and protocol. bmp_{b_i} indicates the port of bus matrix that b_i is connected.

- BM represents multi-layer bus' bus matrix, which is defined with a 4-tuple, $(w_{bm}, f_{bm}, pr_{bm}, BBM)$, where w_{bm}, f_{bm} and pr_{bm} are data bus width, frequency and protocol, respectively. BBM is a set of buses, bbm_i, that route bus matrix's master layers to slave layers on the fabric of multi-layer bus.

- BB is a set of bus bridges. $bb_i = (b_j, b_k)$, represents the bus bridge that

connects its master interface to b_j and its slave interface to b_k, implying that bb_i functions as a bus master on b_j and as a slave on b_k.

Figure 3.8 shows an example of an ALM, M_{al}, which is composed of four functional blocks, four ports, a DMAC, a memory, six shared buses and five buses on bus matrix of multi-layer bus. Processes and channels of M_{sl} in Fig. 3.7 are mapped onto components in M_{al}. The process-to-functional block mapping information specifies that p_1, p_2, p_3 and p_4 are mapped onto fb_1, fb_2, fb_3 and fb_4, respectively. Similarly, the channel-to-port mapping indicates that the sources of c_1 and c_2, symbolized with c_{1s} and c_{2s}, are mapped onto master port pt_1, the source of c_3, c_{3s}, is mapped onto slave port pt_3, the destination of c_1, c_{1d}, is mapped onto master port pt_2, while the destinations of c_2 and c_3, c_{2d} and c_{3d}, are mapped onto slave port pt_4.

From the channel-to-port mapping, C_{d_i} and C_{m_i}, the communication path of each channel is determined considering the master-slave communication regulation of bus protocols. In Fig. 3.8, channel c_2's communication path is "$pt_1 \rightarrow pt_4$". In the case of c_1, $C_{m_1} = \{c_{1s}\}$ because memory is necessary as an intermediate slave of the communication between two master ports. Therefore, the communication path of c_1 becomes "$pt_1 \rightarrow m_1 \rightarrow pt_2$", meaning that the data transfer is conducted from pt_1 to m_1 and from m_1 to pt_2. Likewise, $C_{d_1} = \{c_{3s}\}$ because a DMAC is needed as a master in the communication of c_3 and the communication path becomes "$pt_3 \rightarrow d_1 \rightarrow pt_4$". A communication path may traverse more than one DMAC or memory due to the placement of ports on the buses connected to the bus matrix. Besides, a transfer in a sub-path, e.g. "$pt_1 \rightarrow m_1$", may involve multiple buses and buses on bus matrix.

3.3.3 Definition of the Proposed Efficient Performance Estimation Method

- Input
  1. M_{sl} : An SLM describing behavior of a system.
  2. M_{al} : An ALM specifying components and mappings of an architecture.
- Output
  Total execution time of a system described by M_{sl} when executed on architecture M_{al}, considering concurrent data processing and transfers.

3.4 Performance Estimation Method for Configurable Multi-layer Bus-based SoC

There are four procedures in the proposed efficient performance estimation method.

1. System-level profiling - The SLM is simulated in order to gather profiling information, which includes data processing timings, transfer timings and the amount of transferred data.

2. SL-EDG construction - A graph representing execution dependencies in system level between data processing and transfers is constructed from the profiling information.

3. AL-EDG construction - A graph representing architecture-dependent execution orders between data processing and transfers is constructed from the SL-EDG and the ALM.

4. AL-EDG analysis - Performance of each ALM is estimated by analyzing corresponding the AL-EDG to obtain the architecture-dependent data processing and transfer timings.

Since both time-consuming profiling procedure and SL-EDG construction procedure are architecture-independent, they are done only once for all ALMs of the same SLM and input data. Therefore, it is possible to quickly estimate the performance of various architectures by iteratively constructing and analyzing AL-EDG without simulating individual architectures.

3.4.1 System-level Profiling using SystemC

To gather system information, the system is profiled using SystemC simulation as the method of Ueda *et al*. [52]. Monitor_process class and monitor_channel class are extended from SystemC's sc_module and sc_prim_channel, respectively, because SystemC can model hardware's parallel execution [37]. Each process of SLM is implemented with the monitor_process class to capture timings of data processing. Likewise, each channel is implemented with the monitor_channel class to monitor the amount of transferred data and data transfer timings, which are recorded when there are both read access and write access to the channel.

System-level profiling is proceeded by compiling the SLM's code and executing its binary, meaning that it is done in a loosely-timed manner. Consequently, all profiling information is quickly gathered using SystemC simulator.

3.4.2 SL-EDG Construction

An SL-EDG is a graph that represents data processing, data transfers and their execution dependencies in system-level. It is constructed based on the profiling information in the same way as the method of Ueda *et al*. [52]. Its construction is independent of hardware architecture, so does the number of its vertices. The SL-EDG is represented by $G_{sl} = (V_{sl}, E_{sl})$. It is comprised of the set of system-level vertices $V_{sl} = \{v_{p_{(i,k)}} \vee v_{c_{(j,l)}} \mid i, j, k, l \in \mathbb{N}\}$, and the set of system-level edges $E_{sl} = \{(v_{p_{(i,k)}}, v_{p_{(i,k+1)}}) \vee (v_{c_{(j,l)}}, v_{c_{(j,l+1)}}) \vee (v_{p_{(i,k)}}, v_{c_{(j,l)}}) \vee (v_{c_{(j,l)}}, v_{p_{(i,k)}}) \mid i, j, k, l \in \mathbb{N}\}$.

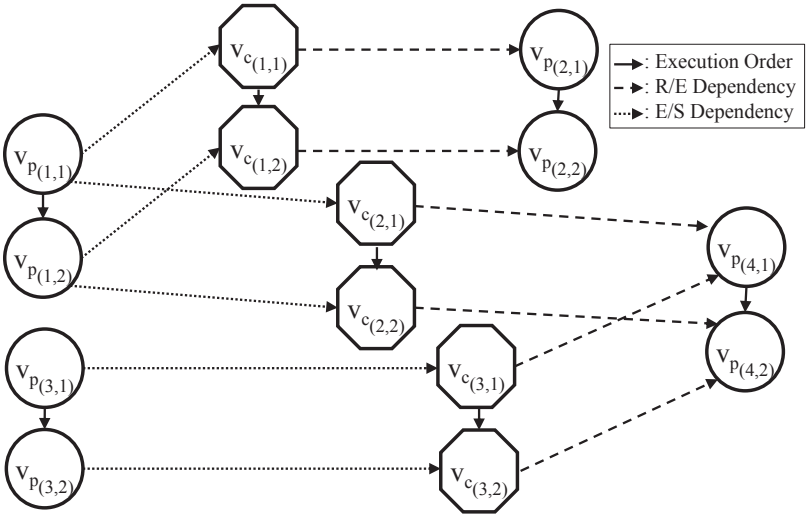


Figure 3.9: An example of SL-EDG.

Figure 3.9 illustrates the example of SL-EDG corresponding to SLM in Fig. 3.7. Assuming that each process executes its data processing twice and each channel transfers data twice. The circular nodes denoted by $v_{p_{(i,k)}}$ represent the vertex of p_i's k-th data processing and the octagonal nodes denoted by $v_{c_{(j,l)}}$ represent the vertex of c_j's l-th data transfer. The solid arrows represent execution orders. The dashed arrows represent R/E dependency-edges, indicating that the execution of data processing starts after the data has been received, and the dotted arrows represent E/S dependency-edges, indicating that the data transmission starts after the execution of data processing is over.

3.4.3 AL-EDG Construction

An AL-EDG is a graph that represents data processing, data transfers, and their execution dependencies according to components of the architecture specified by an ALM. In addition to the vertices and edges of SL-EDG, AL-EDG also consists of vertices and edges involving DMACs and memories that fulfill bus protocol's regulation about master-slave communication of each data transfer. The number of its vertices depends on the components and organization of the ALM. AL-EDG is represented by $G_{al} = (V_{al}, E_{al})$, where V_{al} and E_{al} are AL-EDG's vertex set and edge set, respectively.

The AL-EDG is constructed by the following steps;

1. Copy SL-EDG as AL-EDG. Let V_{al} be V_{sl} and E_{al} be E_{sl}.

2. Alter V_{al} and E_{al} so that the AL-EDG also includes the dependencies of data transfers raised by communication paths. For every channel $c_i = (p_u, p_x) \in C$, do as follows;

(a) If $c_{is} \in C_{d_k}$, meaning that DMAC d_k initiated c_i's transfer to a port mapped to the source of channel c_i, do as follows;

i. Make vertices $v_{d_{(k,l)}}$ representing processing on d_k, and vertices $v_{c'''_{(i,j)}}$ representing additional data transfers of c_i. Then, add them to V_{al}. Make edges $(v_{c'''_{(i,j)}}, v_{c'''_{(i,j+1)}})$ representing execution orders between data transfers of c_i, and add them to E_{al}.

ii. Delete edges $(v_{c_{(i,j)}}, v_{p_{(x,y)}})$ from E_{al} and add edges $(v_{c_{(i,j)}}, v_{d(k,l)})$, $(v_{d_{(k,l)}}, v_{c'''_{(i,j)}})$ and $(v_{c'''_{(i,j)}}, v_{p_{(x,y)}})$, which represent execution dependencies in c_i's communication path traversing d_k, to E_{al}.

(b) If $c_{is} \notin C_{d_k}$ and $c_{is} \in C_{m_q}$, meaning that a port mapped to the source of channel c_i establishes c_i's transfer to memory m_q, do as follows;

i. Make vertices $v_{m_{(q,r)}}$ representing processing on m_q, and vertices $v_{c'''_{(i,j)}}$ representing additional data transfers of c_i. Then, add them to V_{al}. Make edges $(v_{c'''_{(i,j)}}, v_{c'''_{(i,j+1)}})$ representing execution orders between data transfers of c_i, and add them to E_{al}.

ii. Delete edges $(v_{c_{(i,j)}}, v_{p_{(x,y)}})$ from E_{al} and add edges $(v_{c_{(i,j)}}, v_{m(q,r)})$, $(v_{m_{(q,r)}}, v_{c'''_{(i,j)}})$ and $(v_{c'''_{(i,j)}}, v_{p_{(x,y)}})$, which represent execution dependencies in c_i's communication path traversing m_q, to E_{al}.

(c) If $c_{is} \in C_{d_k}$ and $c_{is} \in C_{m_q}$, meaning that the communication of c_i traverses memory m_q after DMAC d_k, do as follows;

i. Make vertices $v_{m_{(q,r)}}$ representing processing on m_q, and vertices $v_{c'_{(i,j)}}$ representing additional data transfers of c_i. Then, add them to V_{al}. Make edges $(v_{c'_{(i,j)}}, v_{c'_{(i,j+1)}})$ representing execution orders between data transfers of c_i, and add them to E_{al}.

ii. Delete edges $(v_{d_{(k,l)}}, v'''_{c_{(i,j)}})$ from E_{al} and add edges $(v_{d_{(k,l)}}, v'_{c(i,j)})$, $(v'_{c_{(i,j)}}, v_{m_{(q,r)}})$ and $(v_{m_{(q,r)}}, v_{c'''_{(i,j)}})$, which represent execution dependencies in c_i's communication path traversing m_q after d_k, to E_{al}.

(d) If $c_{id} \in C_{d_s}$, meaning that the transfer of c_i traverses DMAC d_s after memory m_q, do as follows;

i. Make vertices $v_{d_{(s,t)}}$ representing processing on d_s, and vertices $v_{c''_{(i,j)}}$ representing additional data transfers of c_i. Then, add them to V_{al}. Make edges $(v_{c''_{(i,j)}}, v_{c''_{(i,j+1)}})$ representing execution orders between data transfers of c_i, and add them to E_{al}.

ii. Delete edges $(v_{m_{(q,r)}}, v'''_{c_{(i,j)}})$ from E_{al} and add edges $(v_{m_{(q,r)}}, v''_{c(i,j)})$, $(v''_{c_{(i,j)}}, v_{d_{(s,t)}})$ and $(v_{d_{(s,t)}}, v_{c'''_{(i,j)}})$, which represent execution dependencies in c_i's communication path traversing d_s after m_q, to E_{al}.

3. Divide the vertices into groups of functional blocks V_{fb_i}, buses V_{b_i}, buses on bus matrix V_{bbm_i}, DMACs V_{d_i}, and memories V_{m_i}, that undertake their executions. The channel vertices must be included in the groups of all buses undertaking their executions.

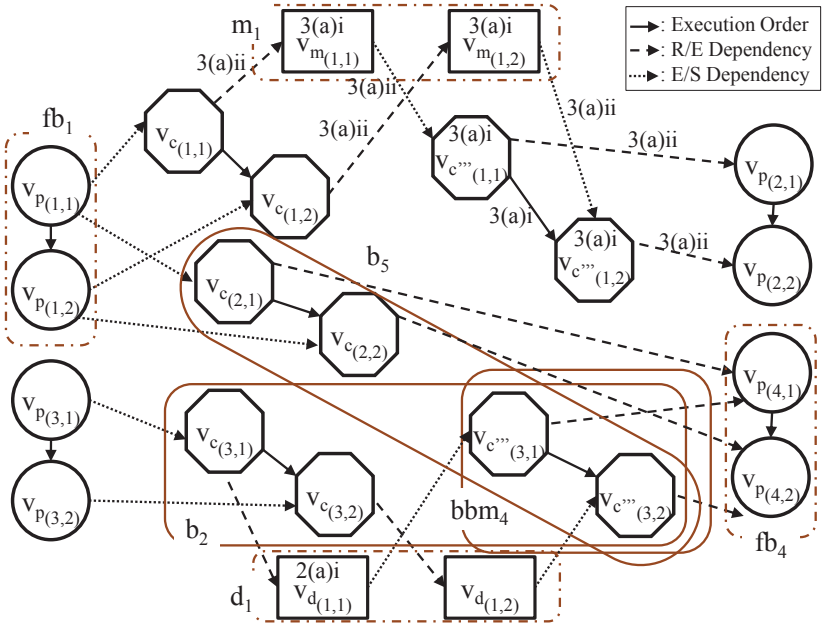


Figure 3.10: An example of AL-EDG.

In the following, the AL-EDG shown in Fig. 3.10 is constructed for the ALM shown in Fig. 3.8. First, the SL-EDG in Fig. 3.9 is copied as an initial AL-EDG. Since $C_{d_1} = \{c_{3s}\}$, $v_{d_{(1,1)}}$, $v_{d_{(1,2)}}$, $v_{c'''_{(3,1)}}$ and $v_{c'''_{(3,2)}}$ are generated into the graph in step 2(a)i. Then, in step 2(a)ii, edges $(v_{c_{(3,1)}}, v_{p_{(2,1)}})$ and $(v_{c_{(3,2)}}, v_{p_{(2,2)}})$ are removed, and edges $(v_{c_{(3,1)}}, v_{d(1,1)})$, $(v_{c_{(3,2)}}, v_{d(1,2)})$, $(v_{d_{(1,1)}}, v_{c'''_{(3,1)}})$, $(v_{d_{(1,2)}}, v_{c'''_{(3,2)}})$, $(v_{c'''_{(3,1)}}, v_{p_{(2,1)}})$ and $(v_{c'''_{(3,2)}}, v_{p_{(2,2)}})$ are added to the AL-EDG. Similarly, because $C_{m_1} = \{c_{1s}\}$, the graph is modified according to steps 2(b)i and 2(b)ii as marked. Finally, the vertices are grouped. $v_{p_{(1,1)}}$ and $v_{p_{(1,2)}}$ are put into V_{fb_1}, the group of process vertices undertaken by fb_1. Similarly, $v_{d_{(1,1)}}$ and $v_{d_{(1,2)}}$ are grouped into V_{d_1}, the group of DMAC vertices undertaken by d_1. V_{b_2}, the group of channel vertices undertaken by b_2, includes vertices $v_{c_{(3,1)}}$, $v_{c_{(3,2)}}$, $v_{c'''_{(3,1)}}$ and $v_{c'''_{(3,2)}}$. Likewise, V_{b_5} includes vertices $v_{c_{(2,1)}}$, $v_{c_{(2,2)}}$, $v_{c'''_{(3,1)}}$ and $v_{c'''_{(3,2)}}$. In the example, $v_{c'''_{(3,1)}}$ and $v_{c'''_{(3,2)}}$, also in V_{bbm_4}, the group of bbm_4, are included in three groups because the transfer from DMAC to fb_4 uses b_2, b_5 and bbm_4.

3.4.4 AL-EDG Analysis

AL-EDG analysis estimates the performance by predicting the behavior of the target system on a specified architecture. This research aims to speculate multi-layer and shared bus behavior based on the analyzed dynamic bus contention arising from arbitration, traffic congestion, advanced bus features and communication protocols. Therefore, the general concepts of bus protocols and advanced bus features of both multi-layer bus and shared bus are modeled so that the AL-EDG

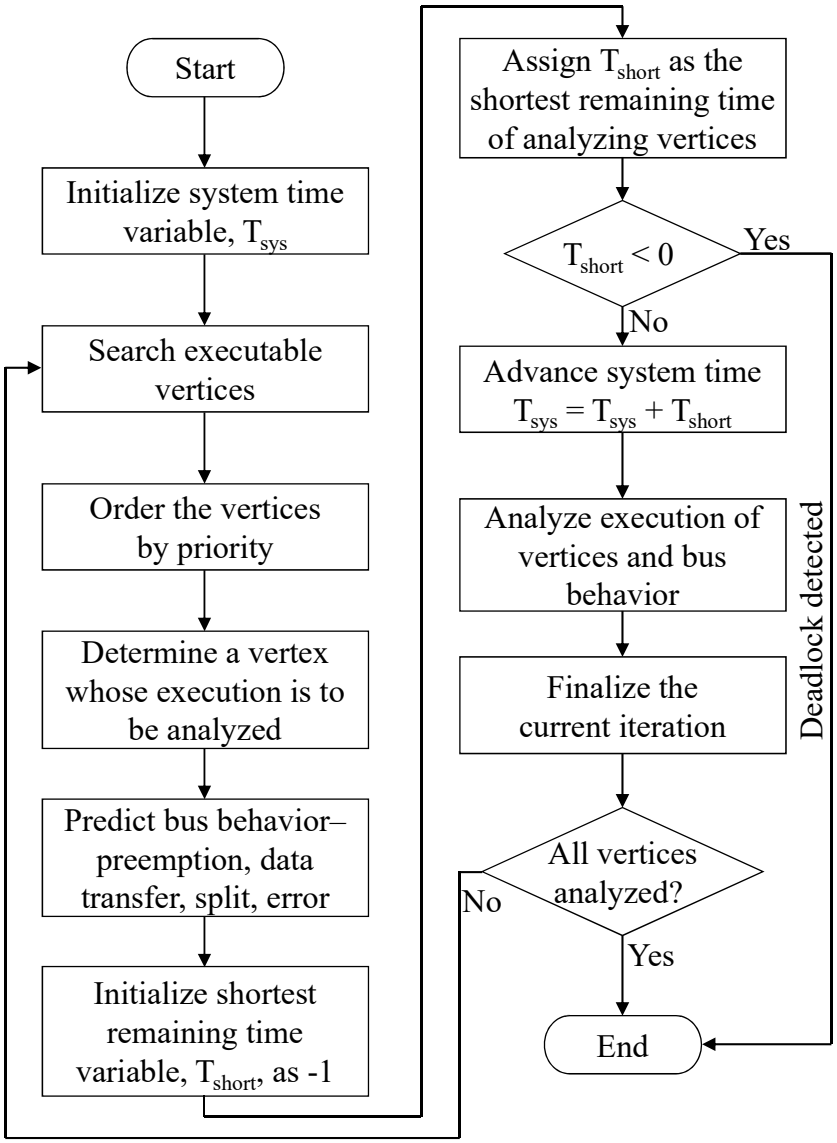


Figure 3.11: The flow of AL-EDG analysis.

analysis recognizes dynamic bus contention. As a result, the proposed method earns an advantage in terms of accuracy and performance estimation speed.

The AL-EDG analysis flow, shown in Fig. 3.11, begins with system time variable initialization. Then, the analysis steps are iterated to execute data processing and transfers until every vertex in V_{al} exhausts or the deadlock is detected.

First, the analysis finds the executable vertices of the current iteration. A vertex in V_{fb_i}, V_{d_j} and V_{m_k} is classified as executable when it have no edge from other vertices, and added to executable vertex set $V_{exe_{fb_i}}$, $V_{exe_{d_j}}$ and $V_{exe_{m_k}}$, respectively. In order to model the regulation of bus protocol that bus master initiates a communication to a slave, a channel vertex is considered as executable based on the master's request. That is, a channel vertex whose transfer is initiated by a master port becomes executable when the status of one of the master port's receive buffers is empty for a read transaction or the vertex has no edge from other vertices for a write transaction. A channel vertex whose transfer is initiated by a DMAC is executable when it has no edge from other vertices and for a write transfer, the status of one of the slave port's receive buffers is empty. Then, the vertices are added to executable vertex set $V_{exe_{b_i}}$ and $V_{exe_{bbm_j}}$. The existence of multiple vertices within an executable vertex set of bus or bus on bus matrix implies simultaneous bus requests and activities. For that reason, together with the current bus activity, bus contention is detected.

In the next step, the vertices in each executable vertex set are reordered by priority, which is decided according to scheduling and arbitration policy at analysis time. Priorities of process vertices are determined from user-defined process priorities on each functional block. Priorities of channel vertices in $V_{exe_{b_i}}$ of the shared buses on the master layer side of bus matrix and in $V_{exe_{bbm_j}}$ of the buses on the bus matrix are determined based on the priorities of bus master and bus bridge's master interface designated by the system designer. On the other hand, priorities of those in $V_{exe_{b_i}}$ of the bus connected to the slave layer of bus matrix depend on priorities of the bus matrix's master layer.

Then, a vertex whose execution will be analyzed is selected from each executable vertex set. The analysis program selects the process vertex that has the highest priority and the status of one of the target port's transmit buffers is empty from $V_{exe_{fb_i}}$, and the channel vertex that has the highest priority and the master is not banned by bus' arbiter from $V_{exe_{b_i}}$ and $V_{exe_{bbm_j}}$. However, some channel vertices in $V_{exe_{b_i}}$ and $V_{exe_{bbm_j}}$ depend on the selected vertices of other $V_{exe_{b_i}}$ and $V_{exe_{bbm_j}}$. For instance, the channel vertex whose transfer involves bus bridge $bb_q = (b_r, b_s)$ can be selected from $V_{exe_{b_r}}$ only when it holds the highest priority among the vertices in $V_{exe_{b_s}}$.

The analysis predicts the dynamic behavior of bus architecture from the selected channel vertices and the speculated bus contention. Bus activity is determined when a channel vertex is selected from every executable vertex set of buses and bus on bus matrix that the vertex belongs to. Split or retry response is the mechanism that allows the shared bus and bus on bus matrix to be released when

the slave cannot conduct normal data transfer immediately. The split or retry response's operation is diagnosed when the status of slave's receive buffer is not empty for the write operation or there exist incoming edges to the channel vertex for the read operation. In the analysis of such cases, the slave is assumed to response with retry when the transfer traverses bus on bus matrix, and with split otherwise. Bus preemption is detected if the selected vertex of a bus' executable set holds a higher priority than the one analyzed as executing on the bus and the transfer on the bus is not analyzed as a lock transfer. Otherwise, the occurrence of normal data transfer is determined.

Next, after initializing the shortest remaining time variable, T_{short}, as -1, the remaining operation time of each analyzing vertex is computed by deducting elapsed time from estimated total operation time of the vertex and bus activity, and the shortest time is assigned to T_{short}. The system time is advanced by the time assigned to T_{short}. However, since there is a chance that the analyzing system falls in a deadlock state, the analysis program detects the deadlock if T_{short} is less than 0, terminates immediately and reports the deadlock condition to the user. In this case, the user may modify the channel priority description and run the analysis again in order to resolve the deadlock.

Total time of each vertex is determined according to its type. Total data processing time of each process vertex is calculated in advance by the following equation;

$$t_p = \frac{e_{(p_i, fb_j)}}{f_{fb_j}} \tag{3.1}$$

The total processing time of DMAC and memory vertices are assumed to be 0 since both components only temporarily store data in their internal storage. On the other hand, total bus usage time is determined based on the predicted bus activity in every iteration so that dynamic bus contention effects are recognized. The time for split operation is determined as in Eq. (3.2).

$$t_s = \frac{S + C_c + C_a}{min(f_{b_i})} \tag{3.2}$$

S and C_c are the overhead of split operation and protocol conversion, respectively, while C_a is the number of address cycles and $min(f_{b_i})$ represents the lowest bus frequency among the frequency of buses that the split operation takes place. The time for retry operation is determined similarly as in Eq. (3.3), where R denotes the overhead of retry operation.

$$t_r = \frac{R + C_c + C_a}{min(f_{b_i})} \tag{3.3}$$

Finally, the calculation of data transfer time is shown in Eq. (3.4),

$$t_d = \frac{D \times C_d \times B + C_c + C_a}{min(f_{b_i})} \tag{3.4}$$

$$D = \frac{w_{c_i}}{min(w_{b_i})} \tag{3.5}$$

where C_d and B are the number of clock cycles in one data cycle and the number of burst beats, respectively. D is the number of data cycles required to transfer one data, determined by Eq. (3.5), where w_{c_i} is the number of bits of one data transferred by the analyzing channel vertex, and w_{b_i} represents the width of the narrowest data bus among the buses and bus matrix that the transfer takes place. The number of address cycles implies the pipeline nature of the bus. It is counted as 0 if the new data transfer is consecutive to the previous one or as the number of protocol's address cycles, otherwise. The number of burst beats is determined according to the number of remaining data to be analyzed and bus preemption.

By analyzing the selected vertices and the speculated bus behavior, the analysis program advances elapsed time of the analyzing vertices and bus activities by T_{short} and keeps track of system's resource status. For the process and channel vertices analyzed for the first time, the IDs of the vertices are registered to models of the target storage to track the status of buffers in ports, DMACs and storage blocks in memories. An ID is unregistered from the storage model when the dependent vertices are analyzed as completed, implying that the data is processed or transferred. The storage model with no ID registered implies that the status is empty. Since data transfer of a channel vertex might be separated into several burst transfer's analysis, the number of remaining transfer data is deducted by the number of burst beats, B, when elapsed time becomes equal to total operation time. The vertices are recognized as completed when their elapsed time becomes equal to the total operation time except for the channel vertices that the number of remaining data must also become 0. In each channel vertex's analysis, the bus bridge in use is marked as active. Shared bus and bus on bus matrix resources are marked as active, lock, split and retry to represents the data transfer, lock transfer, split and retry operation on the bus, respectively. Additionally, the bus master whose transfer is split is marked as banned from arbitration. Lastly, the completed vertices and related edges are removed from the G_{al}.

Finally, the status of each resource is finalized according to the bus protocol at the end of each iteration, e.g. unban bus master, etc. If there are no vertices left in the G_{al}, the analysis returns T_{sys} as system time. Otherwise, the analysis loops from searching the executable vertices step.

3.4.5 Computational Complexity

The computational complexity of the AL-EDG analysis is derived from the flow explained previously and the program implemented to estimate the performance of multi-layer AHB bus-based SoC, which is described thoroughly in section 3.5.

The asymptotic notation $O(n^3)$ expresses the proposed AL-EDG analysis's computational complexity as a function of the number of AL-EDG vertices, n. The analysis repeats from searching executable vertices to finalizing iteration for at

most kn iterations, where k is a constant indicating the number of loops spent for analyzing a vertex. In each iteration, the most complex step in the computational time aspect is ordering the executable vertices, where the worst case consumes n^2 time complexity. Therefore, the complexity of the proposed analysis in the worst-case is cubic w.r.t. the number of AL-EDG vertices.

However, the computational complexity becomes $O(n^2)$ in most cases that the processes execute iteratively and each of the components in an architecture undertakes only a few numbers of processes and channels. Consequently, a few executable vertices exist in the executable vertex sets at a time and ordering the executable vertices consumes only n order of time complexity. The $O(n^2)$ complexity of most cases will be presented in section 3.5.5.

The complexity applies to the situation that the application size, e.g. the size of an image in the image processing, causes both the number of vertices in SL-EDG and AL-EDG to grow. In other words, AL-EDG analysis runtime increases by $O(n^3)$ when estimating the performance of various-sized applications executed on the same architecture.

3.5 Case Study

To show that the proposed method is efficient in architecture exploration of electronic system level (ESL), the proposed analysis flow is applied for performance analysis of multi-layer AHB bus-based SoC. The proposed method is also applicable to shared bus-based architecture and not limited to AHB and APB bus protocol.

The efficiency of the proposed performance estimation method is investigated in two aspects. The first one is the accuracy of the performance estimated by the proposed method when compared with the performance obtained from the RTL simulation. The second one is the speedup of the proposed method over the RTL simulation, which is measured from the runtime of both tools.

3.5.1 Modeling of Multi-layer AHB and APB Protocol

In order to apply the proposed flow to analyze the performance of a multi-layer AHB bus-based system, protocol's parameters and protocol related variable values are defined based on AHB and APB protocol of the AMBA specification [152] as shown in Table 3.1 and Table 3.2. The split and retry responses of AHB requires two cycles as the overhead, and therefore, both overhead of split operation S and overhead of retry operation R are set to 2. The values for protocol related variables are determined according to system status during the analysis but restricted to a certain set of values. The number of clock cycles in one AHB and APB data cycle C_d is 1 and 2 under the assumption that there is no wait cycle. The overhead of protocol conversion differs by protocol pairs and direction of data flow. For AHB-APB protocol conversion, C_C is 1 for a write transfer and 0 for a read

Table 3.1: List of protocol's parameters

Parameter	Value
S	2
R	2

Table 3.2: List of protocol related variable values

Variable	Value
C_d	1,2
C_c	0,1
C_a	0,1
B	1,2,4,8,16

transfer, split response and retry response. The number of address cycles C_a can be either 0 when pipelined, otherwise, the 1 with no wait cycle. The number of burst beats B is typically 1, 2, 4, 8 and 16. However, if bus preemption occurs, B can be any integer no more than 16.

There are three additional multi-layer AHB protocol conditions. Firstly, every communication via bus matrix must be locked because the arbiters of multi-layer bus do not allow preemption. Consequently, the shared buses used for the transfer must be marked as lock. Secondly, 1-cycle-idle phase exists between two bus requests due to the state machine of AHB master interface. Therefore, a channel vertex is removed from executable vertex sets for one clock after the analysis of the previous operation has finished. Finally, the arbitration policy of buses and buses on bus matrix is restricted to fixed-priority policy.

The following describes the analysis of the G_{al} in Fig. 3.10. Let the priorities of fb_1, fb_2 and d_1 be 3, 2, 1, respectively, so do the priorities of c_1, c_2 and c_3. Assume that the mapped functional blocks of p_1, p_2, p_3 and p_4 spend 100, 80, 120 and 140 ns for data processing calculated by Eq. (3.1) and the amount of data transferred in c_1, c_2 and c_3 are 16, 16 and 32, respectively. The system operates at 50 MHz, and there is one receive and one transmit buffer in each port. The execution Gantt chart is shown in Fig. 3.12.

After T_{sys} is initialized, the executable vertices are searched throughout G_{al}. In the first iteration, $v_{p_{(1,1)}}$ and $v_{p_{(3,1)}}$ have no source edge, so they are added to $V_{exe_{fb_1}}$ and $V_{exe_{fb_3}}$, respectively. Moreover, since pt_2 has an empty receive buffer, $v'''_{c_{(1,1)}}$ is analyzed to have a bus request raised, and is added to $V_{exe_{b_3}}$, $V_{exe_{b_6}}$ and $V_{exe_{bbm_5}}$. Then, $v_{p_{(1,1)}}$, $v_{p_{(3,1)}}$ and $v'''_{c_{(1,1)}}$ are selected to be analyzed. Next, the analysis predicts the bus activity of b_3, b_6 and bbm_5 to be retry response because there exists an edge to $v'''_{c_{(1,1)}}$.

The shortest remaining time of analyzing vertices is determined. The remaining

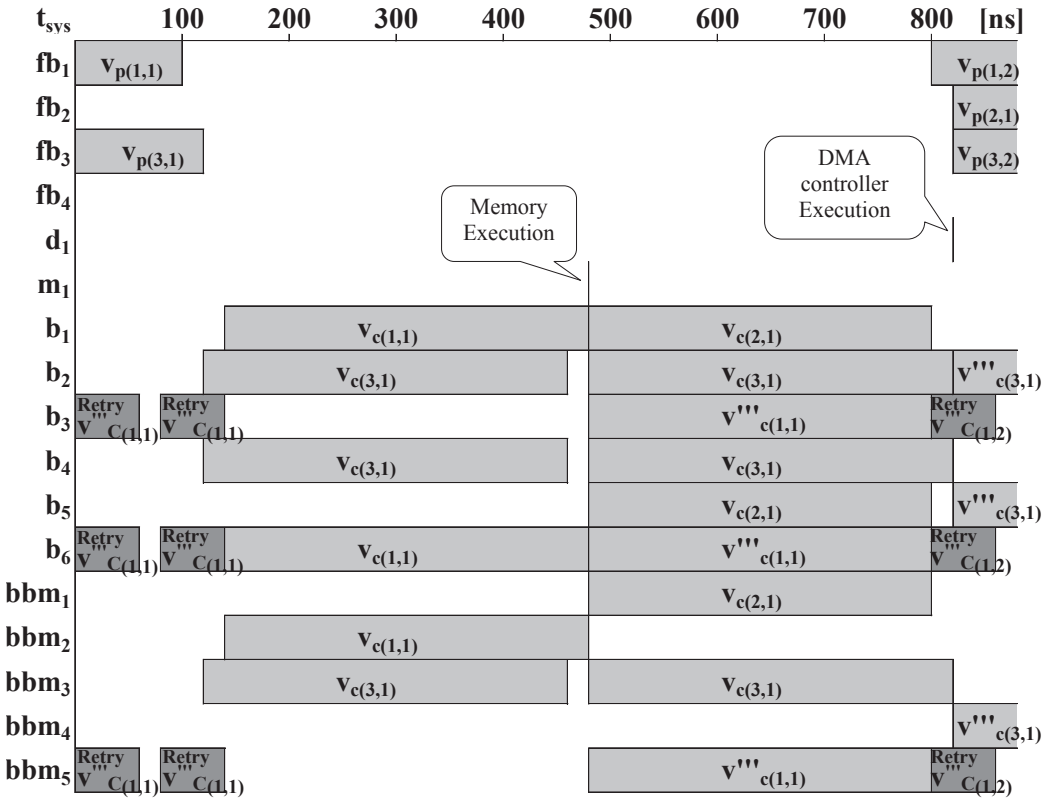


Figure 3.12: An example of AL-EDG analysis.

time of $v_{p_{(1,1)}}$ and $v_{p_{(3,1)}}$ are 100 and 120 ns, respectively, while the retry response of $v'''_{c_{(1,1)}}$ takes 60 ns according to Eq. (3.3) computed with one address cycle. For that reason, T_{short} becomes 60 and T_{sys} is advanced.

The vertices and bus behavior are analyzed. The ID of $v_{p_{(1,1)}}$ and $v_{p_{(3,1)}}$ are registered in the model of pt_1's and pt_3's transmit buffer. At the same time, the retry response finishes.

To finalize this iteration, $v'''_{c_{(1,1)}}$ is excluded from the analysis for 1 cycle due to AHB interface's idle phase. Consequently, T_{sys} is advanced by 20 ns in the second iteration.

In the third iteration, $v'''_{c_{(1,1)}}$ is reconsidered and analyzed to be responded with retry. T_{short} becomes 20 ns because the analysis of $v_{p_{1,1}}$ has ended, and therefore, vertex $v_{p_{(1,1)}}$, edges ($v_{p_{(1,1)}}$,$v_{c_{(1,1)}}$) and ($v_{p_{(1,1)}}$,$v_{c_{(2,1)}}$) are removed from G_{al}.

In the fourth iteration, $v_{c_{(1,1)}}$ becomes executable and is added into $V_{exe_{b_1}}$, $V_{exe_{b_6}}$ and $V_{exe_{bbm_2}}$, so does $v_{c_{(2,1)}}$ which is added into $V_{exe_{b_1}}$, $V_{exe_{b_5}}$ and $V_{exe_{bbm_1}}$. Since both channel vertices are initiated from the same port, but c_1 holds a higher priority, $v_{c_{(1,1)}}$ is selected on b_1, b_6 and bbm_2, while $v_{c_{(2,1)}}$ is selected on only b_5 and bbm_1. Consequently, $v_{c_{(2,1)}}$ is ignored in this iteration. Unfortunately, because b_6 is occupied with retry response operation, $v_{c_{(1,1)}}$ is not analyzed. T_{sys} is advanced to 120

Figure 3.13: An SLM of JPEG encoder.

Table 3.3: Information of data in channels

	width [bit]	#data
c_0	24	64
c_1	8	64
c_2	12	64
c_3	12	64
c_4	12	64
c_5	8	256

ns, and vertex $v_{p(3,1)}$ and edges ($v_{p(3,1)}$,$v_{c(3,1)}$) are removed from G_{al}.

Then, $v_{c(3,1)}$ is analyzed to transfer data for 16 burst beats, which takes 340 ns for one address cycle and 16 data cycles in the fifth iteration. However, the second retry operation of $v_{c(1,1)}$ remains only $20ns$, so T_{sys} becomes 140 ns.

In the sixth iteration, The analysis of $v_{c(1,1)}$'s transfer starts and lasts for 340 ns. At $T_{sys} = 460$, transfer of the first 16 data of $v_{c(3,1)}$ finishes, but since there are 16 data left to be transferred, the vertex has to be considered again after 20 ns of 1-clock-cycle idle phase. At the same time, the analysis of $v_{c(1,1)}$ remains 20 ns too, so T_{short} becomes 20 ns.

At $T_{sys} = 480$, the seventh iteration takes place to analyze the execution of memory vertex $v_{m(1,1)}$. T_{short} equals 0 ns and the eight iteration begins at the same point of time. The executable vertex $v_{c(2,1)}$ is selected to be analyzed and its execution time is 320 ns because the address cycle overlaps with the last data cycle of $v_{c(1,1)}$. Meanwhile, $v_{c(3,1)}$ is analyzed again and the transfer time becomes 340 ns. The analysis proceeds in the same fashion until G_{al} exhausts.

3.5.2 Experimental Environment Setup

The effectiveness of the proposed performance estimation method is studied through a JPEG encoder application. Figure 3.13 shows an SLM of JPEG encoder, consisting of seven processes and six channels. The processes are Block Splitting (BS), Color Transformation (CT), Discrete Cosine Transformation (DCT), Quantization (Q), ZigZag ordering (ZZ), Variable Length Coding (VLC) and file WRiTing (WRT). The data width and the amount of data in one system-level transaction of each channel are shown in Table 3.3. The images used in the experiments are processed in an 8×8-pixel block unit and without downsampling.

Table 3.4: Information of functional blocks and its ports

FB name*	Exe.cycle [cycle]	Port
BS	67	1 Slave
CT	68	1 Master
DCT	368	1 Slave
ZZ	67	1 Slave
Q	68	1 Master
VLC	200-265	1 Master
WRT	258	1 Slave

The experiments were conducted on a 3.60 GHz Intel Xeon, 32 GB memory and 64-bit CentOS5 machine. The estimation method is implemented in C language and the SLM was implemented with SystemC 2.3.0 [37]. The source code of the proposed method and the SLM was compiled with gnu gcc 4.1.2. RTL simulation tool is ModelSim SE-64 10.3.

3.5.3 Accuracy Measurement

The results of the AL-EDG analysis for performance estimation method are compared with the performance results of the RTL simulation to measure the accuracy of the analysis. The performance of eight architectures, each of which executes the same aforementioned seven processes of JPEG encoder, are evaluated. The execution cycle shown in Table 3.4 is the worst-case execution cycle. Moreover, each architecture is comprised of the same set of functional blocks and ports, whose information is shown in Table 3.4, a DMAC, and a memory. The DMAC functions as a master to initiate the communication between DCT and ZZ functional block. The memory stores data transferring between Q and VLC functional block. The width of the buses, shared bus and bus matrix is 32 bits and each architecture is operated at 50MHz. The architectures contain the same functional blocks, DMAC, and memories, but their ports are attached to the bus architecture differently and various configurable multi-layer bus architectures are represented. Therefore, the difference in performance of the architectures is solely affected by the communication architecture.

The performance estimation by the proposed method proceeds as follows; First, the encoding of an image was profiled to collect the data processing timing, data transfer timing and the amount of transferred data, and the SL-EDG was constructed accordingly. For each ALM of the architectures under evaluation, an AL-EDG was constructed and analyzed to obtain the estimated performance. Table 3.5 shows the number of vertices in the SL-EDGs and AL-EDGs. Normally, the number of vertices in an AL-EDG depends on the components and communication path of every channel, but the number of vertices of AL-EDGs in the

Table 3.5: The number of vertices in SL-EDG and AL-EDG

Image size	256 × 512	512 × 512	512 × 1,024	1,024 × 1,024
SL-EDG	59,542	118,972	238,234	476,628
AL-EDG	84,118	168,124	336,538	673,236

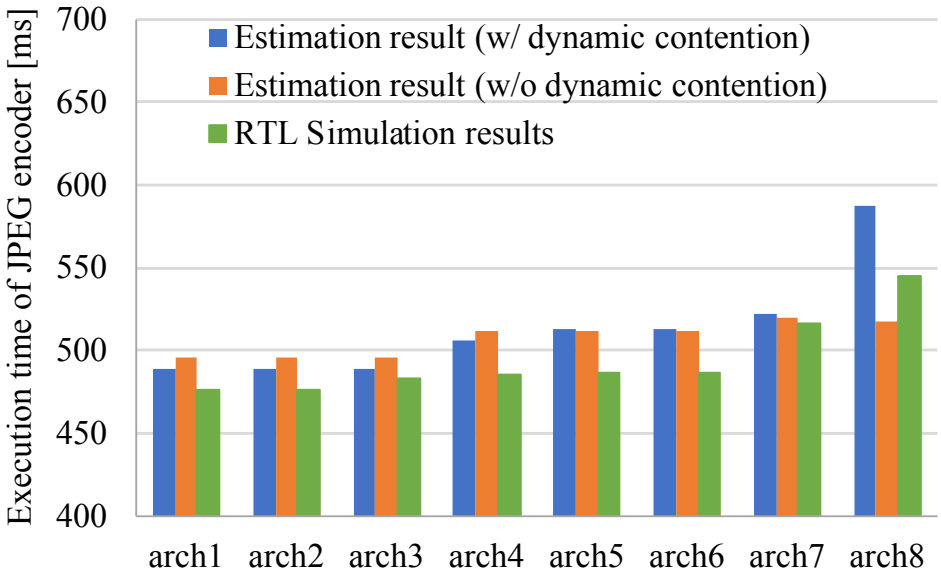


Figure 3.14: Performance results estimated by the proposed method, the method w/o considering dynamic bus contention and RTL simulation (1, 024×1, 024-pixel image).

experiments are equal because the only difference is the architecture organization. However, the vertices are divided into the groups of components that undertake them differently. The experiments were conducted with four images of different sizes.

Figure 3.14 shows the performance results yielded from RTL simulation, the proposed method considering dynamic bus contention and the method that does not consider dynamic bus contention. The results of the proposed method are represented in the graph as estimation results (w/ dynamic contention) and those of the method that does not consider dynamic bus contention is represented as estimation results (w/o dynamic contention).

The performance results of the proposed method considering dynamic bus contention are compared with the results of RTL simulation in order to evaluate the estimation error. Architectures noted as arch3 and arch7, respectively illustrated in Fig. 3.15**(c)** and Fig. 3.15**(g)**, contain multi-layer bus with heterogeneous configurations and incur the smallest error of 1.5%. On the other hands, arch8, illustrated in Fig. 3.15**(h)**, is the shared bus-based architecture and incurs the biggest error of 7.6%. When bus contention is detected to occur on a shared bus, the analy-

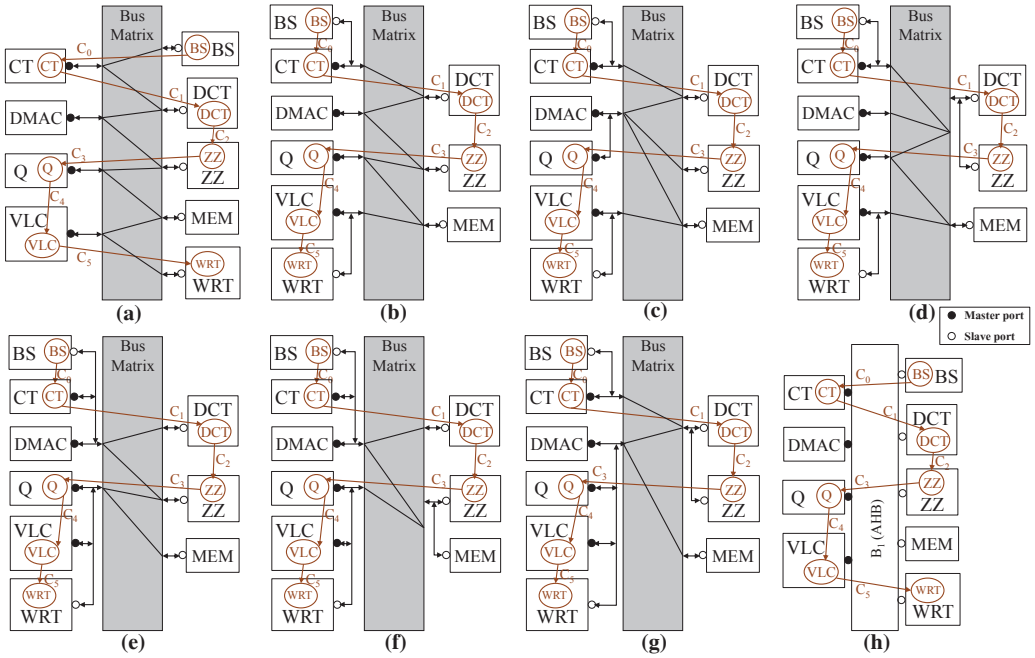


Figure 3.15: ALM of architectures in the experiments: (a) arch1; (b) arch2; (c) arch3; (d) arch4; (e) arch5; (f) arch6; (g) arch7; (h) arch8.

sis program speculates a bus activity as well as the operation time, which raises communication-related timing errors of each bus. These errors are accumulated especially when bus contention arises repeatedly. For that reason, the accuracy of the proposed method becomes worse in the case of a single shared bus-based architecture, arch8. In the case of multi-layer bus-based architecture, the errors are distributed to several shared buses and affect the estimated system time parallelly. According to the timing results, a larger amount of bus contention is found when analyzing the performance of arch6, arch5, and arch4, respectively, so the errors of these architectures are bigger than those of the other architectures that contain multi-layer bus.

Figure 3.16 illustrates the mean values and the ranges of errors in eight architectures. The proposed method evaluates system performance with only 1-8% difference from the RTL simulation and the mean error appear as 3.8%. Its overestimation is due to the fact that the longest execution time of VLC functional block, aka the Worst-Case Execution Time (WCET), is constantly used in the estimation. On the other hand, the execution cycle of VLC varies because its execution behavior depends on the processed data.

During the performance analysis, taking dynamic bus contention, i.e. bus requests and current bus activity, and dynamic bus behavior, i.e. dynamic address phase calculation, split, retry and preemption operation, into account benefits the estimation results in many aspects. Figure 3.14 also illustrated the estimation re-

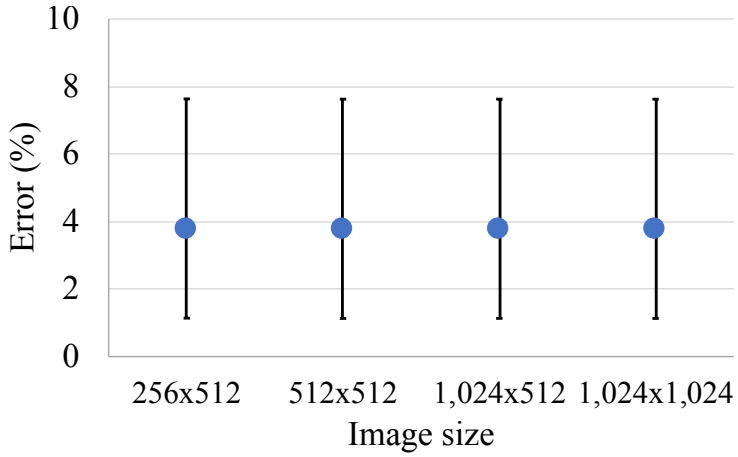


Figure 3.16: Error bar shows the error of the estimation.

sults when neither dynamic bus contention nor probable dynamic bus behavior is recognized. They are labeled as estimation results (w/o dynamic contention). The first benefit is that analyzing system performance without considering dynamic bus contention and behavior may cause underestimation in arch8 because it assumes too optimistic bus contention. On the contrary, the address phase is not recognized dynamically when ignoring bus requests and current activity, so more errors incur in the estimation results of arch1, arch2, arch3, and arch4. This leads to a wider error range of -5.2% to 5.2%, which reduces the reliability of the estimation, and insufficient design, which is unacceptable because it might not satisfy the design constraints. The second benefit is that the proposed method's estimation results are 1-2% more accurate than the results when bus contention and behavior are ignored. For instance, the estimation errors of arch1 from the method with and without the consideration regarding dynamic contention are 2.5% and 4.0%, respectively. However, since data processing of JPEG encoder application dominates data communications between IPs, the impact of considering dynamic bus contention and behavior is not so large.

3.5.4 Tool Runtime and Speedup

Figure 3.17 illustrates the time spent for system-level profiling and SL-EDG construction w.r.t. the number of pixels in the sample images. The procedures are done as fast as within two minutes for the image as large as total 2,359,296 pixels (1,536×1,536 pixels) because the profiling is conducted in a loosely-timed TLM manner. However, the time spent for the two procedures tends to grow linearly for the bigger size of image.

The AL-EDG construction and analysis achieved much faster in evaluating the performance of an individual architecture, which is proven by the runtime speedup value as high as 152.6 times over the simulation. Figure 3.18 shows the relation-

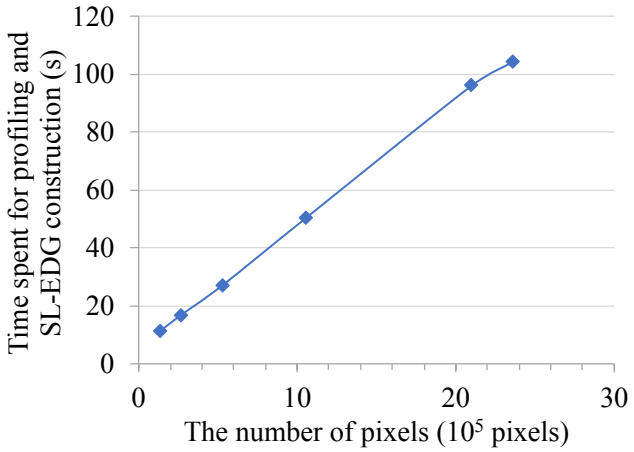


Figure 3.17: Runtime for profiling and construction of SL-EDG.

ship between the number of pixels and the tool runtime in estimating the performance of each architecture candidate. The circles and squares represent the tool runtime of the proposed estimation method and the RTL simulation, respectively. The stars show the average speedup of the architecture w.r.t. the number of pixels and the relevant error bar indicates the range of speedup values. The experiments demonstrate that the speedup varies from 17.4-152.6 times by the combination and organization of functional blocks, DMACs, memories and buses as well as the size of the sample image. The bigger the image is, the less the average speedup becomes. This is because the runtime of the analysis program grows by a quadratic function as the image becomes bigger, while the RTL simulation runtime grows linearly in our case study.

Figure 3.19 draws the relationship between tools' runtime and the number of pixels. It shows that the overall procedure of the efficient performance estimation method has achieved the maximum speedup of 25.6 times over the overall RTL simulation in evaluating the performance of eight architectures. A bigger speedup value can be gained when evaluating a larger number of architectures due to the fact that the proposed method conduct the profiling and SL-EDG construction procedure only once for one image. On the other hands, the speedup slightly drops when approximating the performance of architectures encoding the bigger image. However, the proposed method is able to evaluate the performance of a large number of architectures within a much shorter time that the RTL simulation which takes an unbearably long time.

3.5.5 Discussion

The abstraction level of the proposed method is between untimed- and timed-model. The reason is that a loosely-timed simulation takes place in the system-level profiling procedure, and then, the static analysis is executed repeatedly to

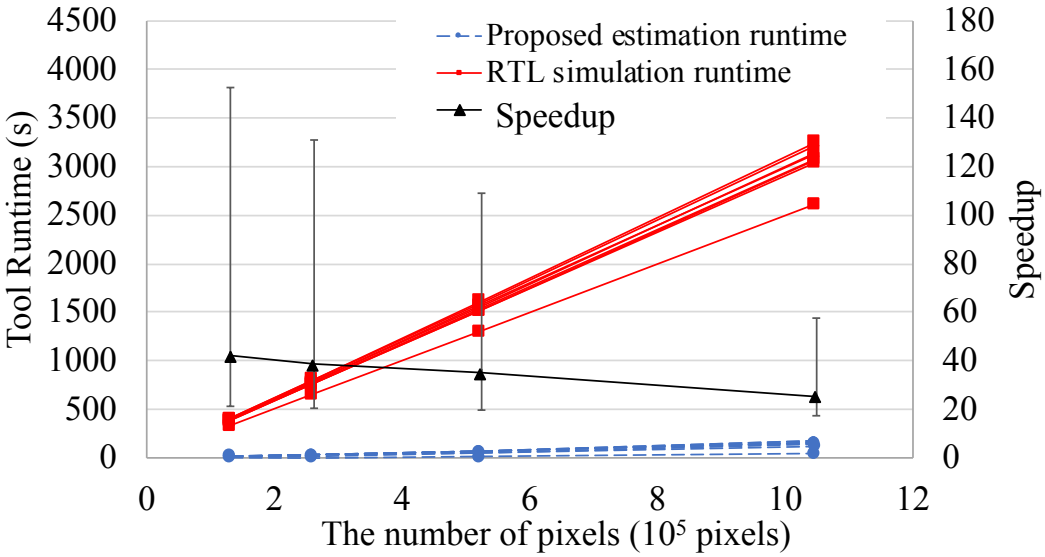


Figure 3.18: Average speedup in estimating performance of eight architectures.

estimate the performance of architectures. Therefore, one of the most obvious advantages of the proposed method over the dynamic simulation methods, e.g. RTL, cycle accurate (CA) and bus cycle accurate (BCA) simulation, is that it requires less modeling effort. Unlike dynamic simulations, in which models for each architecture must be implemented, the SLM in the proposed method is created and profiled only once and the information can be utilized for performance estimation of every ALM. Consequently, days of modeling effort can be saved because inferior architectures are discriminated in an early design stage.

Figure 3.20 illustrates the natural logarithm relationship between the number of AL-EDG vertices, n, and the runtime of the proposed method spent on AL-EDG analysis for eight individual architectures encoding the 512×512 pixel-image. The slopes of lines in the graph shows that the runtime of the experiments increases by only $O(n^{1.65})$. The reason is that in the conducted estimations, the number of executable vertices in each executable vertex set of the functional blocks, DMACs, memories, shared buses and buses on the bus matrix of the multi-layer bus in an architecture is scheduled to as few as no more than five process or channel vertices during each iteration. Consequently, the complexity of the most complex step, ordering the executable vertices, is reduced to n and then, the overall complexity becomes n^2.

The scalability of the proposed method in terms of tool runtime depends on the number of AL-EDG vertices as shown in section 3.4.5, and the computational complexity is $O(n^3)$. For AI applications, which have a larger number of AL-EDG vertices than the JPEG encoder, the tool runtime is expected to show a similar trend as in Fig. 3.20. Nevertheless, it is worth noting that the tool runtime also depends on the architecture itself.

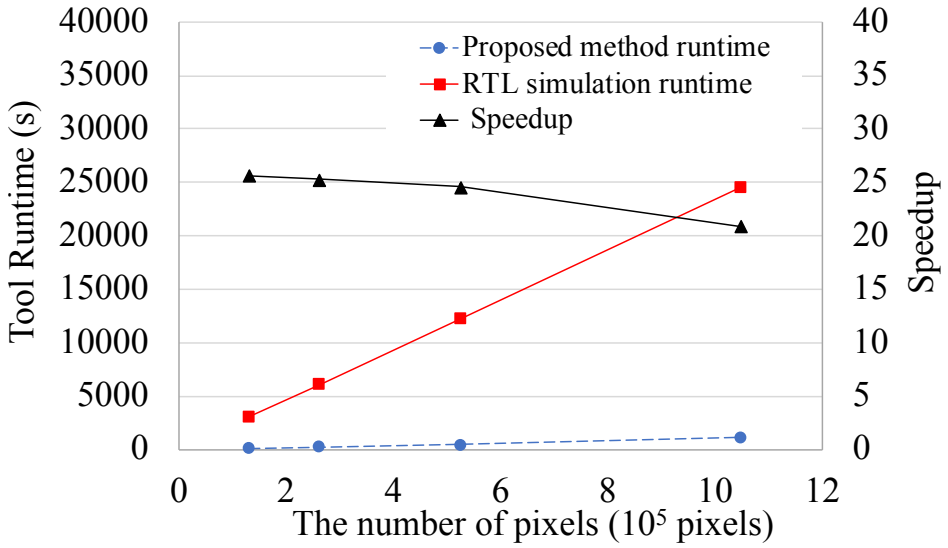


Figure 3.19: The proposed method's overall speedup.

As you can see in Fig. 3.18 and Fig. 3.19, the proposed method is faster than the conventional RTL simulation. The proposed method uses a small amount of memory because it does not use the real image data. It uses only the execution order and the amount of the transferred data obtained from system-level profiling. In the experiments, the proposed method uses less memory than the RTL simulation. Therefore, the proposed method is able to evaluate the performance of the architecture encoding a larger image in a short time even when the RTL simulation takes too much time.

In section 3.5.3 and section 3.5.4, experiments were conducted only to the architectures with the same set of functional blocks, DMACs, and memories. The proposed estimation method achieves 25.6 times faster estimation compared to RTL simulation with small error. Furthermore, the proposed method also works well with different sets of components by repeating only the AL-EDG analysis procedures.

The runtime of the proposed method is roughly compared with the CA simulation time in order to estimate the speedup. The experiments were conducted again on the Pentium4 workstation, running at 3.4 GHz. Then, the results are evaluated against the CA simulation time results presented by Martin *et al.* [155]. It is found that the proposed method has gained 30-35 times of runtime speedup over the CA simulation.

3.6 Conclusion

This chapter proposed an efficient performance estimation method for a configurable multi-layer bus-based architecture by utilizing system-level data flow infor-

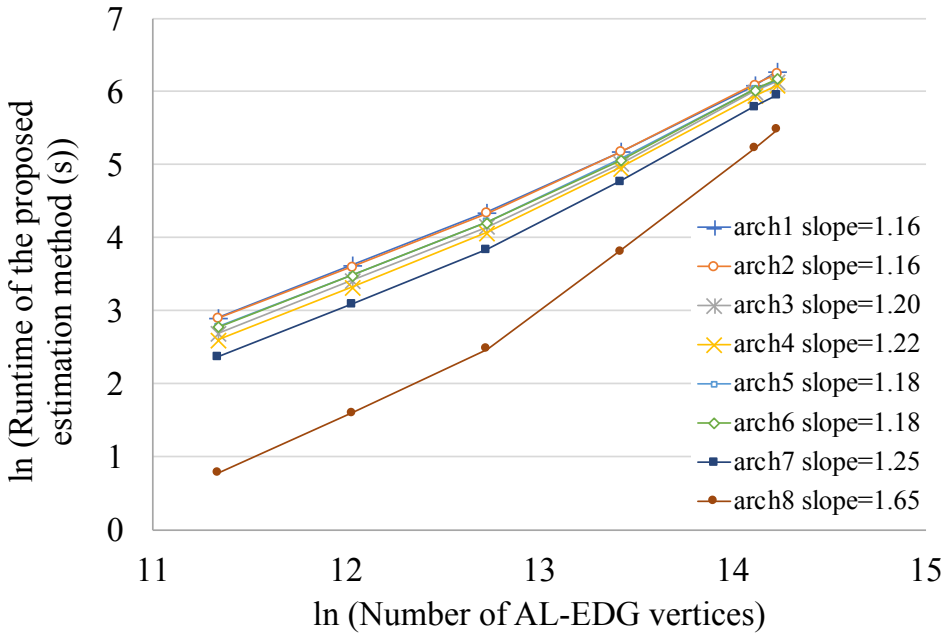


Figure 3.20: AL-EDG analysis' runtime of individual architecture.

mation. The flow of performance analysis takes the outstanding behavior details of bus protocol into account so that it can recognize the dynamic bus contention. The proposed method is fast and accurate. It estimates the performance within 8% of error compared to the conventional RTL simulation. Furthermore, the AL-EDG construction and analysis have achieved the speedup of 152.6 times over RTL simulation in estimating the performance of one architecture. The experimental results also show that the proposed performance estimation method including system-level profiling, SL-EDG construction, and AL-EDG construction and analysis of eight architectures has achieved the overall speedup of 25.6 times over eight RTL simulations. When evaluating more architectures, the proposed method repeats only the AL-EDG construction and analysis procedures. Therefore, the larger the number of architectures becomes, the bigger overall speedup value is gained. In the future, the statistical analysis shall be applied to the proposed method to assure the estimated performance statistically.

Chapter 4

Parallelism-flexible Convolution Core for Sparse Convolutional Neural Network

Chapter 4 proposes a parallelism-flexible convolution core for sparse CNN that leverages multiple types of parallelism flexibly and weight sparsity efficiently to achieve high performance. First, this chapter describes CNN and the prior arts of CNN accelerators. Next, it introduces flexible parallelism concept and explains architecture organization and operations of the proposed parallelism-flexible convolution core in alternating dataflow to exploit multiple types of parallelism, and eliminating redundant operations due to weight sparsity. Then, the proposed convolution core was evaluated on 13 convolutional layers in a sparse VGG-16 benchmark. Finally, this chapter is summarized.

4.1 Motivation and Objective

In modern AI platforms, data processing at the edge and embedded systems requires high-performance computing devices. CNN, which is one of the most vigorous AI algorithms, evolves day-by-day for a vast number of applications especially in image and video analytic domains, such as surveillance systems and autonomous driving, because of their remarkable classification performance shown in several image recognition studies [16–18] on ImageNet benchmark [113]. The processing of these applications usually takes place at the edge (near sensor, such as camera) or on embedded systems in order to achieve a real-time response. Unfortunately, CNN comes with the cost of an excessive computation that becomes critical for real-time and low-power inference processing on both edge and embedded systems. Most processing time of CNN is consumed by the convolutional layers. In order to accelerate its computation, CNN requires high-performance and low-power accelerator to deliver its superior ability.

High-performance CNN accelerators bring about real-time ability with the ex-

ploitation of four major techniques. First, data-reuse maximization focuses on reusing input feature maps (IFMs), kernels and output feature maps (OFMs). It is employed by several low-power architectures [34, 101, 108, 110] because it reduces high-latency and energy-consuming external memory access. Second, data precision minimization aims to reduce data bit width while the recognition accuracy is maintained [21, 22]. Third, calculation-skip maximization reduces the calculation by omitting zero-operand MACC, which is the result from weight pruning process [20, 23]. This allows several architectures to achieve performance improvement by the degree of sparsity [110, 112]. Fourth, parallel calculation maximization leverages various types of parallelism in CNN. Recent publications exploit specific types of parallelism and schedule the computation accordingly to maximally utilize the multipliers in PEs [34, 36, 101]. Typically, the accelerators exploit multiple techniques.

There exist two main problems that prevent CNN accelerators from achieving superior performance. First, most CNN accelerators fail to maximize parallel calculation of all the convolutional layers due to the fact that the type of fruitful parallelism varies by the size and number of IFMs and OFMs, while the dataflow and computation scheduling (mapping parallel operations to the multipliers) remain fixed throughout all the layers. Specifically, the accelerator has difficulty in adjusting its dataflow and scheduling according to layer specification, which results in low multiplier utilization and low performance in some layers. For example, even though the architecture proposed in [34] exploits various types of parallelism, it cannot achieve high multiplier utilization in the first layer because of the fixed dataflow and scheduling. To resolve this problem, flexible scheduling, aka flexible parallelism, is required to improve the low multiplier utilization with various types of parallelism according to the layer specification. The second problem addresses the difficulty of effectively employing the calculation-skip maximization and parallel calculation maximization techniques, specifically flexible parallelism, at the same time. For example, parallelizing multiplications comprising one output may occupy more multipliers, but it cannot fully leverage zero-skipping without complicating data control because the scheduling is regulated by the pre-defined dataflow. As a consequence, the dataflow to exploit flexible parallelism may reduce calculation-skip ability.

This chapter includes the following contributions:

1. it introduces a flexible parallelism concept to maximize multiplier utilization;

2. it proposes a parallelism-flexible convolution core for sparse CNN that efficiently exploits weight sparsity by skipping zero-operand computation;

3. to show the effectiveness, the parallelism-flexible convolution core for sparse CNN was implemented and evaluated using RTL simulations, and synthesized for Intel's Arria10 FPGA and Stratix10 FPGA.

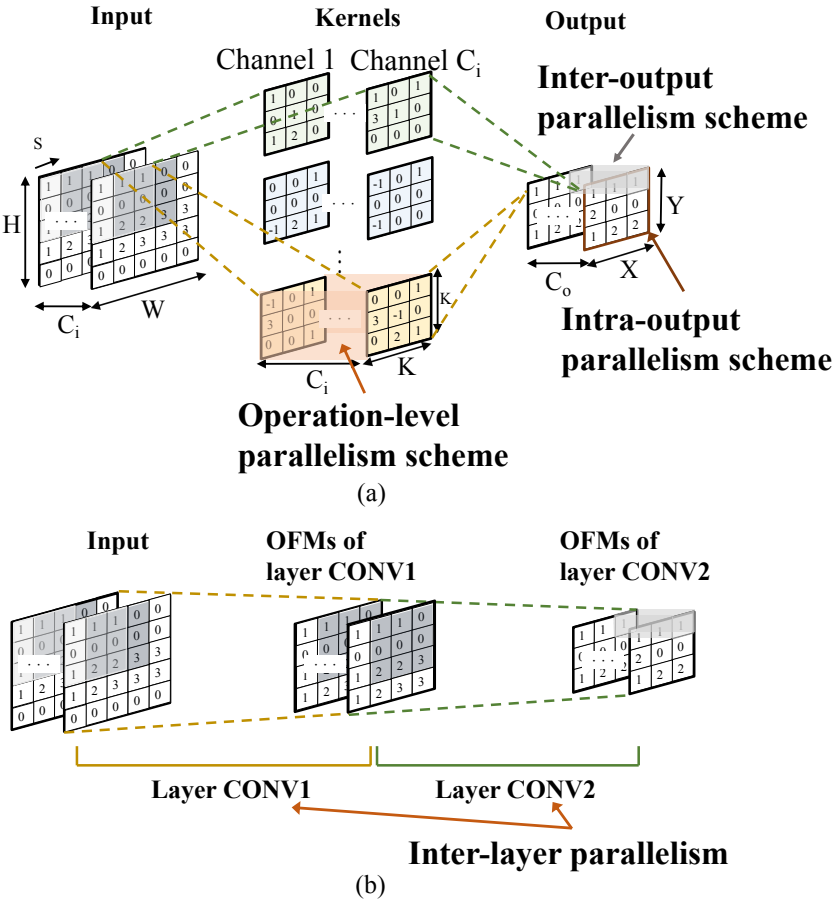


Figure 4.1: The computation of convolutional layers and their parallelism: (a) inter-layer parallelism; (b) inter-output, intra-output, and operation-level parallelism.

4.2 Convolutional Neural Network (CNN)

4.2.1 Terminology of CNN

Typically, a CNN consists of four kinds of layers: (1) convolutional layer, which functions as a feature extractor; (2) pooling layer, which subsamples the extracted features; (3) normalization layer, which normalizes feature correlations; (4) fully-connected layer, which produces non-linear activations for regression or classification problems. This paper focuses on accelerating the multi-channel two-dimensional convolution of the convolutional layers, which is computation-intensive and time-consuming.

Figure 4.1(a) illustrates the terminology of a convolutional layer, where H and W are height and width of an IFM, C_i is the number of input channels, S is the stride (the number of pixels to shift the kernel in convolution), K is the kernel size

(a kernel includes $C_i \times K \times K$ weights), X and Y are height and width of an OFM, and C_o is the number of output channels, which is equal to the number of kernels. Each activation of the OFMs is computed by a deep nested loop according to the following MACC equations:

$$A_o^v(x, y) = f(F_o^v(x, y)) \tag{4.1}$$

$$F_o^v(x, y) = b^v + \sum_{t=1}^{C_i} \sum_{m=1}^{K} \sum_{n=1}^{K} k_v^t(m, n) \times F_i^t(x \times S + m, y \times S + n) \tag{4.2}$$

where $A_o^v(x, y)$ is the activation at position (x, y) of the OFM v, f is an activation function, $F_o^v(x, y)$ is the result of convolution at position (x, y) between IFMs and kernel v, b^v is a bias of kernel v, $k_v^t(m, n)$ is the weight at position (m, n) in channel t of kernel v, and $F_i^t(x \times S + m, y \times S + n)$ is the activation at position $(x \times S + m, y \times S + n)$ of IFM t.

4.2.2 Parallelism in CNN

There are four types of parallelism incorporated with convolutional layers: inter-layer, inter-output, intra-output, and operation-level parallelism. Inter-layer parallelism is the parallelism that executes the convolution of multiple layers in a pipeline manner as shown in Fig. 4.1(b). The latter three types comprise an intra-layer parallelism, in which the MACCs within the same layer are computed in parallel. The inter- and intra-output parallelisms are the parallelism between multiple OFMs (the C_o axis in Fig. 4.1(a)) and output activations within an OFM (the X-Y plane in Fig. 4.1(a)), respectively. The operation-level parallelism is the most fine-grained type that parallelizes the multiplications of the same output activation. It occupies most multipliers when accelerating a typical dense CNN, in which all weights are non-zero.

Enjoyable type of parallelism in each layer varies throughout the CNN by layer specification, such as size and number of OFMs. When the size of OFMs is large, the intra-output parallelism is efficient in terms of multiplier utilization. However, multiplier utilization decreases as the OFMs become smaller. In this case, inter-output parallelism can complement the small amount of intra-output parallelism, and hence, increase multiplier utilization.

4.3 Compressed CNN Model

To reduce the required bandwidth in reading CNN model to CNN accelerator and simply exploit weight sparsity, kernels of a sparse CNN model from quantization and weight pruning is compressed into a channel-major modified compressed sparse column format [112] layer by layer. Each channel of kernels in each convolutional layer is compressed as a non-zero weight vector, $\boldsymbol{w}$, which includes

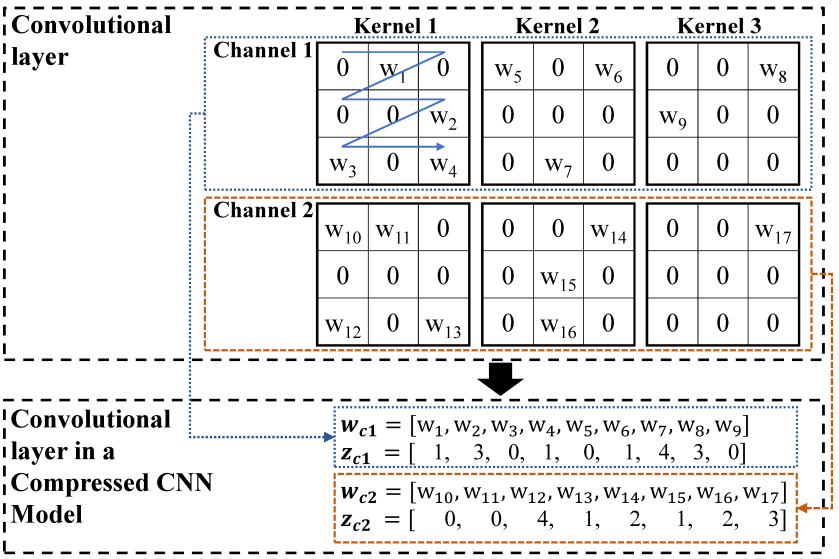


Figure 4.2: An example of compressing a convolutional layer to a compressed CNN model.

non-zero weight elements, and a leading-zero vector, z, which includes the number of zero-valued weights preceding the non-zero weight at the same vector index as w.

For example, a convolutional layer that contains three kernels, each of which includes two channels of 3×3-weights, is compressed as shown in Fig. 4.2. The notation w_i represents the i^{th} non-zero weight. The kernels are compressed channel by channel. The non-zero weight vector of channel 1, w_{c_1}, includes non-zero weights in order as shown by the bold arrow (written in channel 1 of kernel 1) from kernel 1 to kernel 3. The corresponding leading-zero vector, z_{c_1}, includes the number of leading zeros of w_1, w_2, w_3, and so on, respectively. The number of leading zeros is counted continuously regardless of the change of kernels. For that reason, the number of leading zeros of w_8 is 3 since there is one 0 after w_7 in kernel 2 and two 0 before w_8 in kernel 3. The weights of channel 2 are compressed similarly.

4.4 Overview of The Proposed Parallelism-flexible Convolution Core

Figure 4.3(a) illustrates an overall architecture of the proposed parallelism-flexible convolution core, which includes five key components. The memory controller reads and writes data from/to external memory, such as DRAM, through a DDR memory interface. It forwards incoming data, including compressed CNN model, layer specification, parallelism in effect and degree of parallelism (denoted as Par-

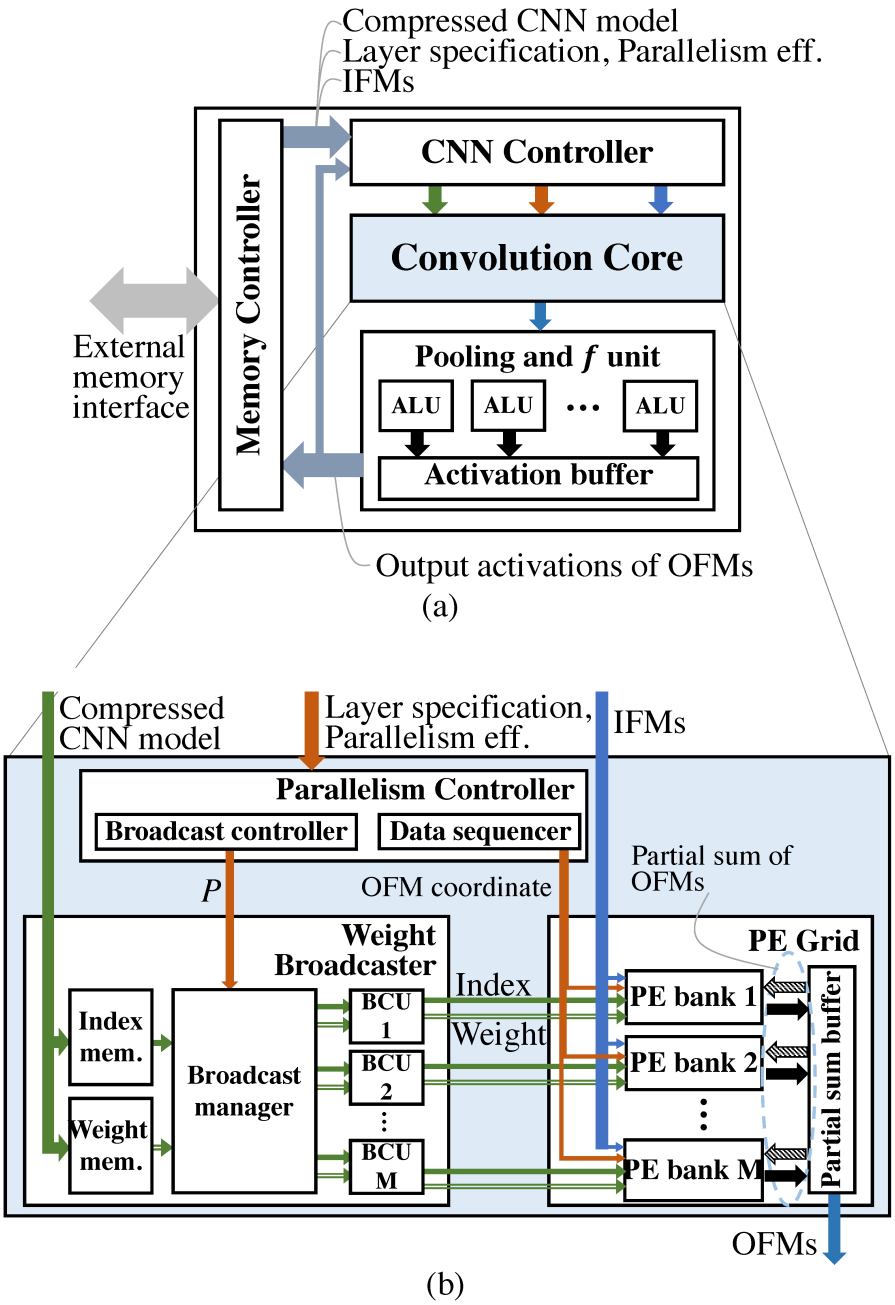


Figure 4.3: Architecture of the proposed parallelism-flexible convolution core: (a) an overall architecture; (b) architecture of the proposed parallelism-flexible convolution core for sparse CNN.

allelism eff. in the figure; see section 4.5.3 for detail), and IFMs, to the CNN controller. Parallelism in effect and degree of parallelism refer to types of parallelism that the proposed convolution core exploits in computing a certain layer and degree of inter-output parallelism (the number of OFMs to be computed simultaneously), respectively. To perform convolution in a layer-wise manner, the CNN controller controls the execution of the accelerator from the layer specification and parallelism in effect, forwards the compressed CNN model to the convolution core, and manages the incoming IFMs using line buffer. The convolution core performs convolution and stores the intermediate results in the partial sum buffer in Fig. 4.3(b). The pooling and f unit include multiple arithmetic logic units (ALUs), which subsample and apply activation function to the OFMs, and activation buffer, which stores the output activations. Finally, the activations are either moved to external memory or reused as IFMs of the next layer.

Since convolutional layers consume most CNN computation time, this work focuses on the convolution core that accelerates the convolutional layers. It efficiently leverages both multiple types of parallelism and weight sparsity of the compressed CNN model according to the size of OFMs. The proposed convolution core is applicable to various sizes of IFMs, sizes of OFMs, numbers of input channels, numbers of output channels, kernel sizes and strides. Other CNN processing, i.e. activation function, pooling layers, and fully-connected layers, are lightweight processing, and hence they can be computed on either general purpose processors or specialized hardware such as EIE [112].

4.5 Parallelism-flexible Convolution Core for Sparse CNN

To overcome the problems in exploiting multiple types of parallelism and its integration with calculation-skip technique, the proposed convolution core flexibly adjusts its dataflow and scheduling to multiple types of parallelism, i.e. intra- and inter-output parallelism, with various degrees of parallelism layer by layer, and eliminates the operations related to zero-valued weights through output-stationary data- flow pattern. Compared to the conventional accelerators, the proposed convolution core uses the weight broadcaster and the parallelism controller to enable such abilities. The weight broadcaster and parallelism controller compensate the decreased multiplier occupancy due to reduced intra-output parallelism by broadcasting different kernels and assigning repeated OFM coordinates to the PE grid, respectively, to increase inter-output parallelism according to the degree of parallelism, P. At the same time, the weight broadcaster distributes only non-zero weights and their indices, which are calculated from the leading-zero vector, to the PE grid. The irregular access to IFMs due to weight sparsity is made simple with local indexing to the addresses in local input buffer near PEs. Hence, the output-stationary dataflow pattern that is regulated by the weight broadcaster and the par-

allelism controller efficiently integrates flexible parallelism and calculation-skip techniques.

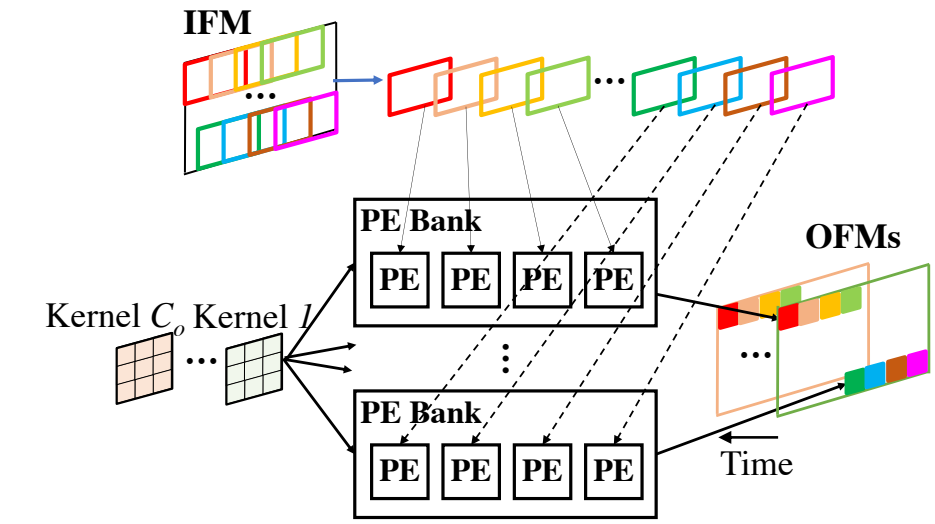
4.5.1 Flexible Parallelism Concept

To maximize parallel calculation (multiplier utilization) in every convolutional layer throughout the CNN, flexible parallelism changes dataflow and scheduling of the convolution layer by layer. Figure 4.4 illustrates an example of MACC scheduling. A PE bank means a group of PEs that convolute the IFM pixels with the same kernel. The scheduling in which the architecture exploits only intra-output parallelism as parallelism in effect with $P = 1$ is shown in Fig. 4.4(a). All the PEs convolute IFM's sliding windows with the same kernel to compute distinct OFM pixels simultaneously and convolute with all the kernels consecutively to compute all the OFMs. The scheduling in which the architecture exploits both intra- and inter-output parallelism simultaneously, aka multi-parallelism, as parallelism in effect with $P > 1$ is shown in Fig. 4.4(b). PEs within a PE bank compute distinct OFM pixels with the same kernel at the same time to realize intra-output parallelism, while different PE banks compute distinct OFMs with P different kernels at the same time to realize inter-output parallelism and each PE bank computes OFMs with $\frac{C_o}{P}$ kernels sequentially. Depending on P, several PE banks convolute distinct OFM pixels with the same kernel to increase intra-output parallelism when P is small, and convolute IFM with more distinct kernels to increase inter-output parallelism when P is large.

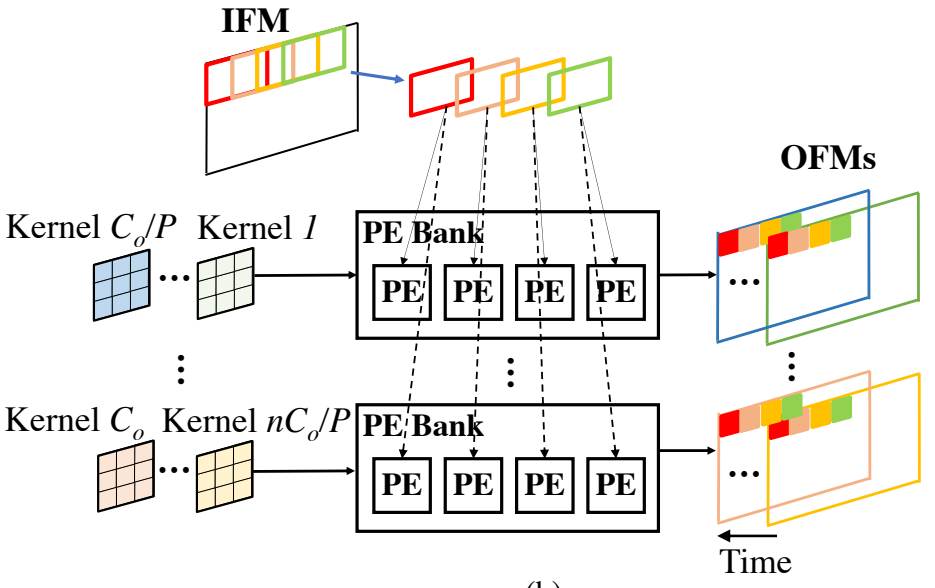
The parallelism in effect and degree of parallelism are determined in advance in order to maximize multiplier utilization (see section 4.5.4). In addition, if the size of OFMs or P is large, IFMs and OFMs are partitioned into tiles (data tiling) so that multipliers and buffer can accommodate parallel MACCs and data, respectively. For example, let us assume that there are 50 PEs. If an OFM consists of 100 output activations, the OFM is partitioned into two tiles to be able to map on 50 PEs in case of $P = 1$. As P grows larger, the OFM is further partitioned into four tiles in case of $P = 2$ and so on. The PEs process one tile at a time.

4.5.2 Operations of the Convolution Core

The OFMs of each layer are computed as shown in Algorithm 1. First, IFMs and OFMs are divided into T equal data tiles. Then, the algorithm loops through all C_i IFMs of each tile in the second loop in order to maximally reuse each IFM. To implement multi-parallelism, the proposed convolution core flexibly unrolls the third and fourth loop layer by layer according to P. Unrolling the third loop parallelizes the convolution of P different kernels to realize inter-output parallelism. Hence, P different OFMs are computed on PEs in P different PE banks simultaneously. Each PE is assigned to compute $\lceil \frac{C_o}{P} \rceil$ kernels. The fourth loop iterates all the outputs at different OFM coordinates, $F_o(x, y)$ of tile T_s. Unrolling this



(a)



(b)

Figure 4.4: The flexible parallelism concept: (a) exploitation of intra-output parallelism; (b) exploitation of intra- and inter-output parallelism.

Algorithm 1: Processing of a convolutional layer on the proposed parallelism-flexible convolution core

Input: IFMs: F_i, Non-zero weights of kernels: k, Bias vector: b
Output: OFMs: F_o

```
1  Initialize F_o with b;
2  for s ←1 to T do
       // Loop all Tiles
3      for t ←1 to C_i do
           // Loop all IFMs
4          for u ←1 to P do
               // Loop all degree of parallelism
5              for F_o(x, y) ∈ T_s do
                   // Loop all outputs in tile
6                  for K_v^t ∈ K_u do
                       // Loop ⌈C_o/P⌉ kernels
7                      for k_v^t(m, n) ∈ K_v^t and k_v^t(m, n) > 0 do
                           // Loop all non-zero weights in kernel
8                          F_o^v(x, y) += k_v^t(m, n) × F_i^t(x × S + m, y × S + n);
9                      end
10                 end
11             end
12         end
13     end
14 end
```

loop and mapping each output on different PEs realize intra-output parallelism. Next, in line 5, the algorithm iterates over each kernel, K_v^t, in the set of kernels assigned to sequentially compute within one PE in the third loop, which is denoted as K_u. Finally, the most inner loop sequentially accumulates the result at coordinate (x, y) of OFM v, $F_o^v(x, y)$, with the multiplication results of the non-zero weight elements, $k_v^t(m, n)$, and the corresponding IFM, $F_i^t(x \times S + m, y \times S + n)$.

4.5.3 Architecture Organization

The architecture of the proposed convolution core is illustrated in Fig. 4.3(b). It receives compressed CNN model, Parallelism Eff., and IFMs as input. The layer specification includes the size and number of kernels, IFMs, and OFMs. Parallelism Eff. includes parallelism in effect and degree of parallelism.

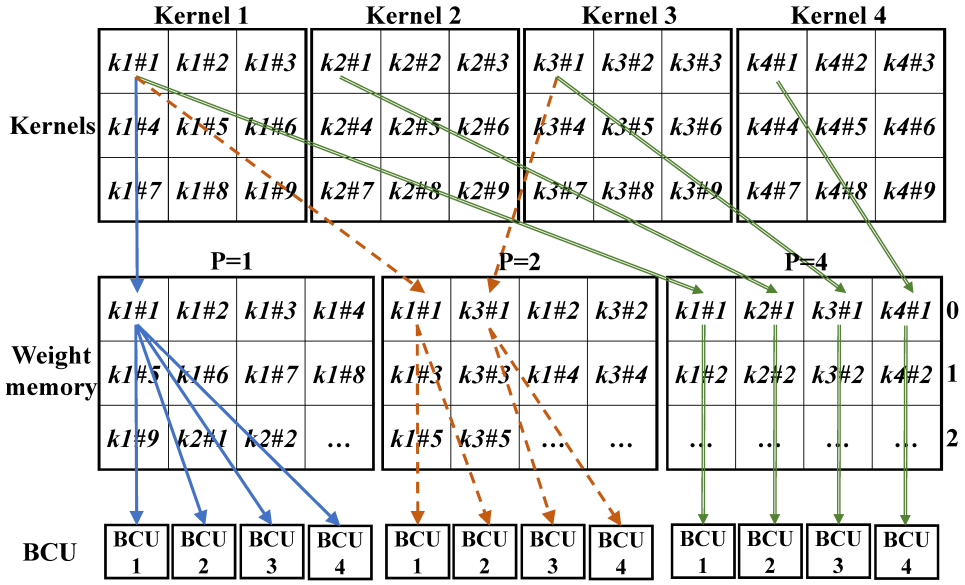


Figure 4.5: Example of weight arrangement of four kernels in weight memory, so that BCUs can broadcast weights from different kernels at the same time.

Parallelism Controller

The parallelism controller is responsible for alternating the dataflow on the convolution core. It is composed of a broadcast controller and a data sequencer. Both work according to the parallelism in effect and the degree of parallelism, P.

The broadcast controller forwards P to the weight broadcaster to control the dataflow of kernels. It forwards 1 as P if the parallelism in effect is intra-output parallelism and P, where $P > 1$, if the parallelism in effect is multi-parallelism with the degree of parallelism P.

The data sequencer alternates the dataflow of IFMs through the assignment of OFM coordinates to be computed by each PE. For intra-output parallelism ($P = 1$), the data sequencer assigns different coordinates to all the PEs. For multi-parallelism, the data sequencer assigns different OFM coordinates to PEs in $\lfloor \frac{M}{P} \rfloor$ PE banks, where M is the number of PE banks, and duplicates the same coordinates P times. For example, assuming that $P = 2$, the OFM coordinates assigned to PE bank #1 to PE bank #$(\frac{M}{2})$ are different, but are the same as the ones assigned to PE bank #$(\frac{M}{2} + 1)$ to PE bank #M. If data tiling is necessary, the data sequencer repeats the coordinate assignment process for all the tiles after the convolution of the previous tile has completed.

Weight Broadcaster

The weight broadcaster is composed of a weight memory, an index memory, a broadcast manager, and multiple broadcast units (BCUs). First, the compressed

CNN model of each layer is loaded into the weight memory and index memory (Weight mem. and Index mem. in Fig. 4.3(b), respectively) channel by channel. Next, upon the completion of storing IFM into the local input buffer, the broadcast manager reads $\boldsymbol{w}$ and $\boldsymbol{z}$ of a channel from the memories and distributes them to BCUs according to P. Finally, each BCU decompresses the compressed CNN model from $\boldsymbol{w}$ and $\boldsymbol{z}$, and broadcasts them consecutively to a PE bank.

For ease of distributing the compressed CNN model to BCUs in order to exploit multi-parallelism, $\boldsymbol{w}$ and $\boldsymbol{z}$ are re-ordered in advance according to P in such a way that weights and indices from P different kernels can be read at the same time. When $P = 1$, all the BCUs broadcast the same weight value, so the weights and indices in one channel of all the kernels are ordered contiguously. On the other hand, when $P > 1$, the weights and indices from different kernels that must be broadcasted at the same time are ordered in the same memory word. Figure 4.5 illustrates an example of weight arrangement in the weight memory and weight distribution to BCUs when assuming that there are four BCUs, one memory word stores four weights, and P equals to 1, 2, and 4. The weights are re-ordered and distributed as follows:

- When $P = 1$: the first memory word contains $k1\#1$, $k1\#2$, $k1\#3$, and $k1\#4$. First, the weight $k1\#1$ is distributed to all the BCUs, then, followed by $k1\#2$, and so on.
- When $P = 2$: the first memory word contains $k1\#1$, $k3\#1$, $k1\#2$, and $k3\#2$. First, the weight $k1\#1$ is distributed to BCU#1 and BCU#2 and the weight $k3\#1$ is distributed to BCU#3 and BCU#4 at the same time, then followed by $k1\#2$ and $k3\#2$, and so on.
- When $P = 4$: the first memory word contains $k1\#1$, $k2\#1$, $k3\#1$, and $k4\#1$. The weight $k1\#1$, $k2\#2$, $k3\#3$, and $k4\#4$ are distributed to BCU#1 through BCU#4, respectively.

Consequently, multiple kernels can be convoluted simultaneously when $P > 1$. Hence, the weight broadcaster can alter the dataflow of kernels to enable multi-parallelism.

To leverage sparsity, each BCU decompresses the compressed CNN model by extracting only non-zero weights from $\boldsymbol{w}$ and accumulates their indices from $\boldsymbol{z}$ in order. Then, non-zero weights and indices are broadcasted to a PE bank consecutively so that the PEs continuously perform MACCs related to non-zero weights while the ones related to zero-valued weights are skipped.

Processing Element Grid

A PE grid consists of multiple PE banks that perform MACCs and a partial sum buffer that stores accumulation results of the previous input channels. One PE bank is connected to one BCU, so the number of PE banks and the number of

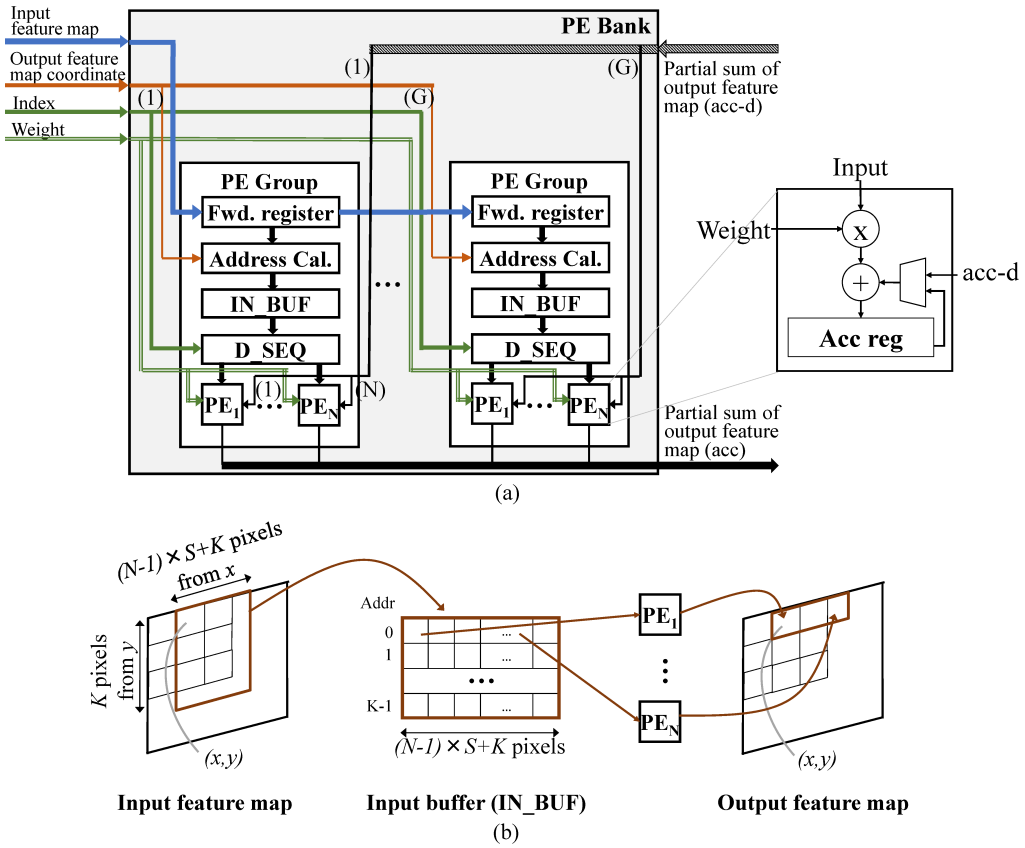


Figure 4.6: Architecture of a PE bank: (a) an overview architecture of a PE bank; (b) data layout of the local input buffer (IN_BUF).

BCUs are equal. Each PE bank receives IFMs from CNN controller, OFM coordinates from data sequencer, and pairs of weight and index from the corresponding BCU.

As shown in Fig. 4.6(a), a PE bank includes G groups of PE, aka PE groups, that compute different OFM pixels. Every PE group within a PE bank consumes the same pair of weight and index but unique OFM coordinates.

A PE group consists of a forward register, an address calculator unit, a local input buffer, a local data sequencer and PEs, which are denoted as Fwd. register, Address Cal., IN_BUF, D_SEQ and $\mathrm{PE_i}$ in Fig. 4.6(a), respectively. The forward register receives IFMs and forwards them to the neighbor PE group in order to reduce physical wire delay. The address calculator determines the address of the required IFM pixels from the OFM coordinate assigned to the PE group. The target IFM pixels of one channel are stored in the local input buffer in order to reuse them for the computation of all kernels. Assuming that there are N PEs in one PE group, $(N-1) \times S + K$ consecutive OFM pixels of the same row starting from the assigned OFM coordinate are computed within a PE group. The local input buffer is registers that store K rows of N overlapping IFM windows, where K is the kernel size and S is the stride. Specifically, it stores input pixels x to $x + (N-1) \times S + K$ of row y to $y + K - 1$ in total of $K \times (N-1) \times S + K$ IFM pixels as shown in Fig. 4.6(b) when the assigned OFM coordinate is (x, y). The local data sequencer selects data from the local input buffer and passes them to PEs. Each PE is composed of a multiplier, an adder, and an accumulation register. It multiplies the selected data with the broadcasted non-zero weight and accumulates the result with the partial sum result from either the partial sum buffer if the weight is the first one of a kernel or the local accumulation register otherwise. The accumulation result is stored in the accumulation register, denoted as Acc reg in Fig. 4.6(a), and it is written to the partial sum buffer after the PE finishes the accumulation of all the weights within one channel of each kernel. These operations are pipelined in order to compute MACC in every clock cycle and achieve high frequency.

Figure 4.7 illustrates data layout in the partial sum buffer. When $P = 1$, all output activations of one OFM in one tile are stored in the same address of the partial sum buffer, and then C_o addresses are required. On the other hand, when $P > 1$, all the output activations of P OFMs in one tile are stored in the same address of the partial sum buffer and $\frac{C_o}{P}$ addresses are used. Note that a tile is smaller when P grows larger. Partial sum buffer is divided into M banks to store the results from each PE bank. That is because PE banks may perform convolution on different kernels, so they may access M different addresses at the same time, while PEs within a PE bank do the convolution with the same kernel and they store the results to the same address.

To handle irregularity in accessing IFM pixels caused by weight sparsity, the irregular data access is made local in the PE groups. The local data sequencer selects IFM pixels in the local input buffer using the index of each non-zero weight

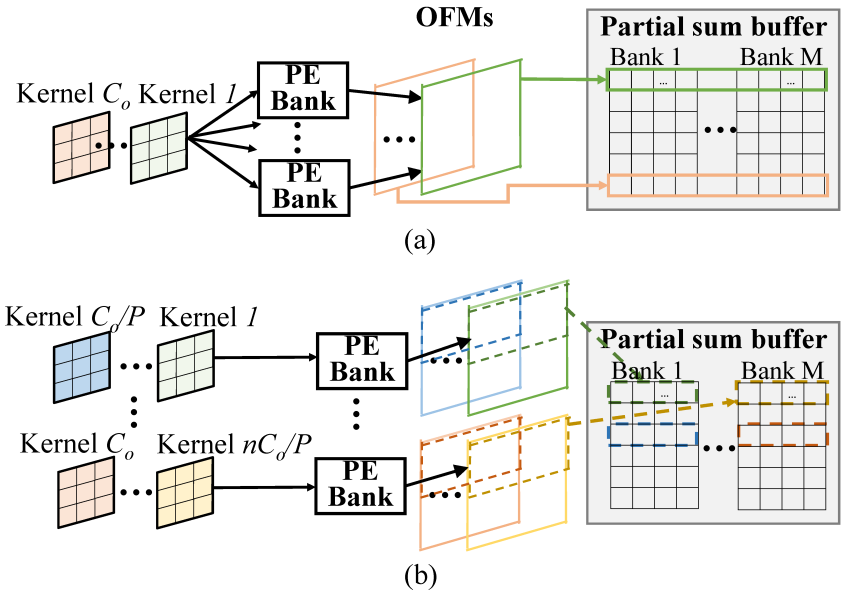


Figure 4.7: The data layout in partial sum buffer assuming the number of output activations in an OFM equals to the total number of PEs: (a) when $P = 1$, all output activations of one OFM are stored in the same address and C_o addresses are required; (b) when $P > 1$, all output activations of P OFMs in one tile are stored in the same address and $\frac{C_o}{P}$ addresses are required.

for addressing. Since the local input buffer is register array, the irregular access is simple and fast. The PE computes only the MACCs related with the non-zero weights. The accumulation result of the previous channels is read from the partial sum buffer when the first non-zero weight of a kernel is received. The accumulation result up to the current channel is written to the partial sum buffer when the last non-zero weight of a kernel is received.

To hide the latency of data transfer from external memory, weight memory (Weight mem.), index memory (Index mem.), local input buffer (IN_BUF), and partial sum buffer are implemented with double buffer. Here, double buffer is introduced because it enables continuous convolution by providing the second buffer to prefetch the next input data while the first input data is being computed. Figure 4.8 illustrates data load, compute, and store timing of the proposed convolution core. First, the compressed CNN model and IFMs of input channel 1 of the first tile are pre-fetched from external memory to weight mem.#0 and IN_BUF#0, respectively. Then, the PE grid performs MACCs on the pre-fetched data and stores the results in partial sum buffer#0. At the same time, the compressed CNN model and IFM pixels of input channel 2 of the first tile are loaded into weight mem.#1 and IN_BUF#1, respectively. While the PE grid is computing the last channel of the tile, the first input channel of IFM pixels of the next tile is loaded into the next available IN_BUF and the first input channel of compressed CNN

model is re-loaded to the next available Weight mem. The results of the second tile will be stored in partial sum buffer#1 so that the result of the first tile in partial sum buffer#0 is transferred to the external memory or fed back to the CNN controller as IFMs of the next layer.

4.5.4 Determination of Parallelism in Effect and Degree of Parallelism

The parallelism in effect and the degree of parallelism, P, of a layer, which are the value that maximizes PE utilization, U, are determined in advance based on the layer specification, the number of BCUs, and the number of PEs in layer-wise. In this context, PE utilization means the percentage between the total number of MACCs of a sparse layer and the total available PE cycles, and it is defined as follows:

$$U = \frac{X \times Y \times C_o \times K \times K \times C_i \times R \times 100}{N \times G \times M \times E}, \tag{4.3}$$

where X, Y, C_o, K, C_i are the same as defined in Fig. 4.1(a), R is the ratio of number of non-zero weights and the number of all weights, N, G, M refers to architecture's parameter in section 4.5.3, and E is the estimated number of cycles in computing the convolutional layer as follows:

$$E = \lceil \frac{C_o \times K \times K \times C_i \times R \times T}{P} \rceil + H \times T \times C_i, \tag{4.4}$$

$$T = \lceil \lceil \frac{X}{N} \rceil \times \frac{Y}{G \times \lfloor \frac{M}{P} \rfloor} \rceil, \tag{4.5}$$

The first term of E is the theoretical time for computing the layer with P, and the second term is the total overhead for decompressing and broadcasting the compressed CNN model.

The overhead, H, incurs once for one loop of all the IFMs (see Algorithm 1) as illustrated in Fig. 4.8, and it is constant regardless of layer specification. The existence of the overhead term means that larger P incurs more decompressing overhead even though inter-output parallelism improves multiplier utilization theoretically.

4.6 Experimental Methodology

To demonstrate the merits of the proposed parallelism-flexible convolution core for sparse CNN, performance, resource usage on FPGA, and power consumption are evaluated. This section explains the workload, architecture configuration and evaluation method in the experiments.

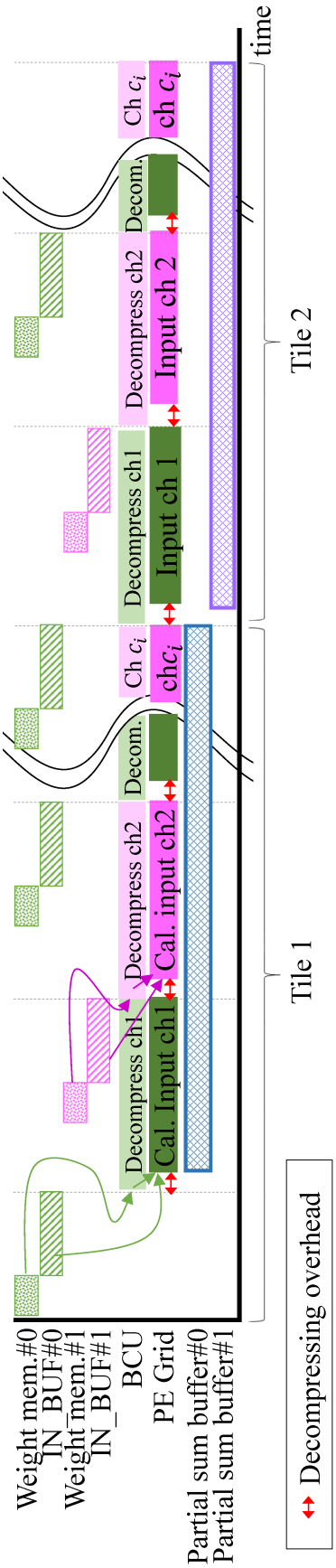


Figure 4.8: Timing of data loading, computing, and storing data of the convolution core using double buffering.

Table 4.1: Parameters of the implemented convolution core

Parameter	Value
M	16
G	4
N	16
Total PE (multiplier)	1,024

4.6.1 Workload

In the experiment, the execution time in computing the convolutional layers of VGG-16 [17] is measured. It was chosen because of three reasons. First, VGG-16 includes convolutional layers with various sizes of IFMs, sizes of OFMs, numbers of input channels, and numbers of output channels, which means that it possesses different dominant parallelism within the same network. For example, the dominant parallelism of the shallow layers, such as conv1_1 or conv1_2, is intra-output parallelism because their size of OFMs is as large as 224×224 pixels. On the contrary, the inter-output parallelism is dominant in the deep layers like conv5_1, conv5_2, and conv5_3 because the number of kernels is larger than the size of OFMs. Second, VGG-16 serves as the backbone of many CNNs, such as SSD [156]. Third, VGG-16 is sparsified by several techniques and its state-of-the-art sparsity is published in [20].

The sparse VGG model was generated by removing the small-valued weights according to the sparsity reported in [20]. The model was compressed into the compressed CNN model using 16-bit and 4-bit for each weight and index, respectively. The arithmetic precision is 16-bit and 32-bit fixed-point for multiplication and accumulation, respectively.

4.6.2 Architecture Configuration

The proposed convolution core is implemented with parameters as shown in Table 4.1. A forward register is inserted every one other PE groups in order to save registers. A PE is implemented with one 16x16-bit multiplier, one 32-bit adders, and one 32-bit register for accumulating the results.

In the evaluation, the proposed convolution core executes the convolution layer by layer. The compressed CNN model, IFMs, and OFMs of each layer are transferred between the proposed convolution core and external memory.

The proposed convolution core executes the convolution in accordance to the determination of parallelism in effect and degree of parallelism, P, as described in section 4.5.4. This experiment considers P as 1, 2, 4, 8 or 16. Figure 4.9 shows the relationship between P and the estimated PE utilization of VGG-16's conv1_1, conv2_1, conv3_1, conv4_1, and conv5_1 layers. The rest of the layers exhibits similar relationship as the layer computing the same size of OFMs. The

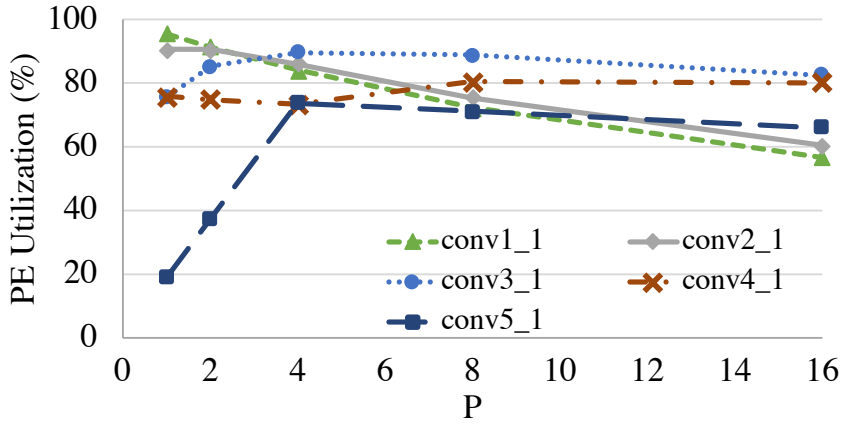


Figure 4.9: The estimated PE utilization when $P = 1, 2, 4, 8$ for conv1_1, conv2_-1, conv3_1, conv4_1, and conv5_1 of VGG-16.

Table 4.2: The parallelism in effect and degree of parallelism for convolutional layers of VGG-16

Layer	**Parallelism in effect**	**Degree of parallelism (P)**
conv1_1 ~ conv1_2	Intra-output parallelism	1
conv2_1 ~ conv2_2	Multi-parallelism	2
conv3_1 ~ conv3_3	Multi-parallelism	4
conv4_1 ~ conv4_3	Multi-parallelism	8
conv5_1 ~ conv5_3	Multi-parallelism	4

overhead of decompressing the compressed CNN model, H, is 16 cycles in the implementation that is designed to achieve high frequency. As a result, increasing P incurs more overhead cycles, and hence the PE utilization may degrade. For conv1_1, intra-output parallelism occupies all the PEs since the size of OFMs is large. The utilization is less than 100% due to the decompressing overhead. For conv2_1, conv3_1, and conv4_1, even though the size of OFMs is large, there exist a small amount of idle PEs when employing only intra-output parallelism. The size of conv5_1's OFMs is very small compared to the number of PEs. Therefore, intra-output parallelism is not utilized to its full capacity. Therefore, employing multi-parallelism by increasing P for conv2_1 through conv5_3 improves multiplier utilization. According to the estimated PE utilization based on Eq. (4.3), Table 4.2 summarizes the parallelism in effect and degree of parallelism, P.

4.6.3 Evaluation Method

Performance The execution cycles and giga MACCs per second (GMACS) were measured using RTL simulation.

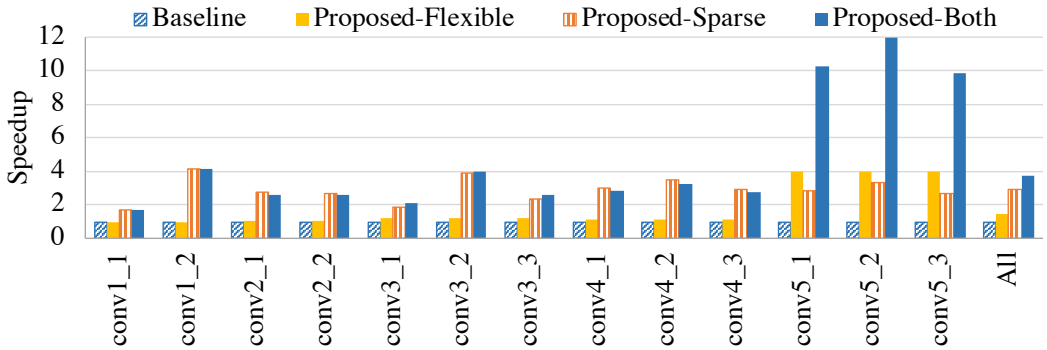


Figure 4.10: Speedup of the proposed parallelism-flexible convolution core by layer of VGG-16 compared to the baseline architecture.

Resource usage on FPGA The resource usage was reported from the HDL synthesis results using Quartus Prime software.

Power consumption on FPGA The power consumption was reported from the power analysis tool of Quartus Prime software.

4.7 Evaluation Results on VGG-16

4.7.1 Performance

To illustrate the effectiveness of flexible parallelism and weight sparsity, the performance of the proposed convolution core that exploits only flexible parallelism, only weight sparsity, and both techniques were compared with the baseline architecture. The baseline architecture is the architecture that exploits only intra-output parallelism and does not skip zero-operand MACC. The parallelism in effect and P is shown in Table 4.2.

Speedup

The speedup of the proposed convolution core over the baseline architecture in computing VGG-16's convolutional layers is shown in Fig. 4.10. The results of the baseline architecture are denoted by Baseline, and the results of the proposed convolution core that employs only flexible parallelism, only weight sparsity, and both techniques are denoted by Proposed-Flexible, Proposed-Sparse, and Proposed-Both, respectively.

By exploiting flexible parallelism, the Proposed-Flexible achieves 1.42x speedup over the Baseline in the total of all the layers. For layer group conv1_x, the Proposed-Flexible does not gain speedup because the intra-output parallelism already occupies all the PEs. On the other hand, the intra-output parallelism in layer conv2_1 through conv5_3 leaves some PEs idle. By occupying them with

inter-output parallelism, the Proposed-Flexible gains speedup over the Baseline. In the layer groups of conv2_x, conv3_x and conv4_x, 5%, 23% and 23% of the PEs, respectively, are idle in the dense CNN computation with the Baseline. The Proposed-Flexible gains 1.13x speedup in average by occupying those idle PEs. Layer group conv5_x takes advantage of the flexible parallelism the most because there are as much as 81% of idle PEs when only intra-output parallelism is exploited. It gains 3.96x speedup compared to the Baseline. Such speedup is achieved because the proposed convolution core can flexibly alternate the dataflow to the degree of parallelism of multi-parallelism that is the most beneficial for each convolutional layer.

The performance of Proposed-Sparse achieves 2.96x speedup in the total of all layers over the Baseline. By leveraging weight sparsity, it can reduce the execution cycles by the degree of sparsity and gain speedup in every layer. Thus, skipping zero-operand MACCs is highly effective in acceleration.

The Proposed-Both achieves 3.73x speedup in the total of all the layers since it leverages flexible parallelism and weight sparsity with simple dataflow control. The speedup of layer group conv1_x comes from the weight sparsity only, while the speedup of other layers comes from both techniques. In layer group conv5_-x, the speedup mainly comes from flexible parallelism. The maximum speedup of 11.95x is achieved in layer conv5_2. On the other hand, it is noticeable that the Proposed-Both gains less speedup than the Proposed-Sparse in some layers, such as layer group conv4_x. Furthermore, considering 1.42x and 2.96x speedup in the total of all the layers from both techniques, higher total speedup at 4.17x was expected. Such speedup was not achieved because of two reasons: (1) the imbalance workload of sparse CNN leaves some PEs idle in order to wait for the others to finish their workload of the same channel when inter-output parallelism is leveraged; (2) the decompressing overhead exists and becomes larger when the exploitation of higher P requires data tiling. These insufficiencies will be discussed in section 4.9.2.

Active Multiplier Utilization

To confirm that the proposed convolution core can improve PE utilization in computing sparse CNN, active multiplier utilization, which is defined as the percentage of the number of MACCs incorporated with non-zero weights and the total available multiplier cycles, was examined. Since a PE includes one multiplier and one adder in the proposed convolution core, multiplier utilization and PE utilization are the same in this context.

Figure 4.11 shows active multiplier utilization, which is calculated as in Eq. (4.3) with E as the number of execution cycles from the simulation. The active multiplier utilization results of the Baseline and the Proposed-Flexible are quite low in all the layers because they compute a sparse CNN in the same way as a dense CNN. In other words, they compute zero-operand MACCs.

Compared to the Baseline, active multiplier utilization of the Proposed-Flexible

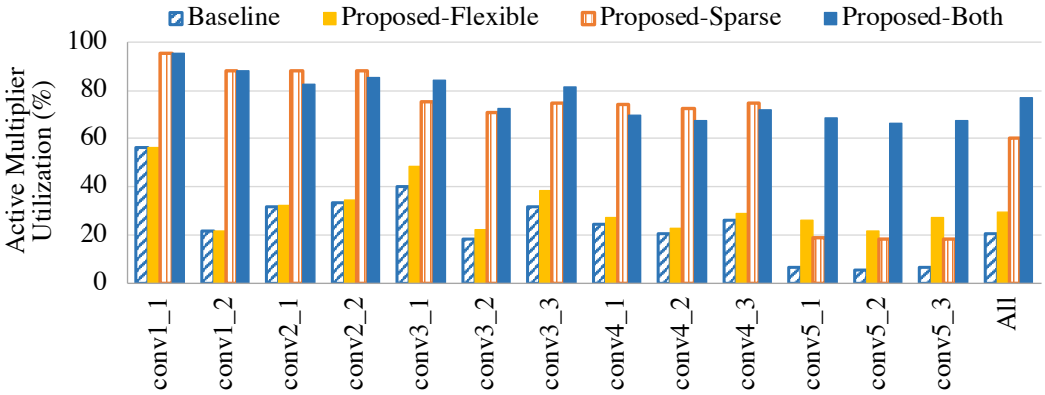


Figure 4.11: Active multiplier utilization of the proposed parallelism-flexible convolution core for each layer of VGG-16 compared to the baseline architecture.

improves in layer conv2_1 through conv5_3, where multi-parallelism is applied. In layer group conv5_x, the utilization increases by approximately 4x as expected when exploiting multi-parallelism with P equals to 4 and the utilization of the Baseline is under $\frac{100}{P}\%$. The improvement is less than the degree of P in layer conv2_1 to conv4_3 because 77% to 95% of multipliers are occupied considering that they execute as dense CNN, which leaves only a small room for improvement.

The active multiplier utilization rises dramatically when exploiting weight sparsity because all the MACCs that take place when using the Proposed-Sparse and the Proposed-Both are meaningful. The active multiplier utilization of the Proposed-Both reaches almost 70% in every layer and as high as 77% in total. It is higher than the Proposed-Sparse except for the layers that achieve lower speedup because of multi-parallelism. Here, the multipliers of Proposed-Both are not fully utilized from two causes.

1. Architectural fragmentation refers to the fact that PEs are idle because the parameters in Table 4.1 limit scheduling. There exist two types of fragmentation. First, fine-grained fragmentation refers to the case that some PEs within a PE group are idle when the number of the dimension X of OFM is indivisible by N because the local input buffer limits that all the PEs in the same PE group must process the OFM pixels from the same row. Second, medium-grained fragmentation refers to the situation that some PE groups are idle when the number of OFM pixels in one tile is indivisible by $N \times G$ because the BCUs and PE banks are connected one-to-one, so inter-output parallelism cannot be scheduled within the same PE bank to occupy the idle PE groups. This issue will be explained further in section 4.9.2;

2. Imbalance workload as mentioned above.

Required External Memory Bandwidth

The maximum bandwidth of the proposed convolution core for VGG-16 is 100 Gbps (512 bits data bus operating at 200 MHz), which is achievable in most FPGA boards [157, 158]. The required bandwidth of each VGG-16's convolutional layer is calculated as follows:

$$bandwidth = \frac{total\ bits\ from\ external\ memory}{ideal\ computation\ time}, \tag{4.6}$$

where *total bits from external memory* includes total bits of compressed CNN model, Bit_{model}, and total bits of IFMs, Bit_{IFM}. The Bit_{model} is calculated as follows:

$$Bit_{model} = \#weight_{nz} \times T \times (Bit_w + Bit_z), \tag{4.7}$$

where $\#weight_{nz}$ is the number of non-zero weights in a layer, T is number of tiles as in Eq. (4.5) (the compressed CNN model is reloaded for every tiles), Bit_w is the number of bits per one weight, which is 16 bits, and Bit_z is the number of bits per one leading-zero value, which is 4 bits. The Bit_{IFM} is calculated as follows:

$$Bit_{IFM} = W \times H \times C_i \times Bit_{data}, \tag{4.8}$$

where W, H, C_i are as in Fig. 4.1(a), and Bit_{data} is the number of data bits, which is 16 bits. The *ideal computation time* is the time for computing MACCs of the sparse CNN with 1,024 PEs at 200 MHz.

The result of the calculation shows that the required bandwidth is 29.2 Gbps, which is low compared to the maximum bandwidth. Therefore, data transfer time can be hidden by double buffering, and hence, does not affect the performance of the proposed convolution core.

4.7.2 Resource Usage and Power Consumption on FPGA

Table 4.3 shows the Arria10 GX1150 FPGA's resource usage of the implementation of the proposed convolution core with 1,024 PEs that is optimized for VGG-like convolutional layers (kernel size is 3×3 and stride is 1). The resource usage for the parallelism controller and the weight broadcaster (Parallelism Cntl and Broadcaster in Table 4.3) is 5, 2, 3, and 3% of LUTs, registers, DSPs, and M20K block RAM (BRAM), respectively. It shows that the proposed convolution core can leverage both flexible parallelism and weight sparsity of sparse CNN simply by adjusting the dataflow with a very small resource usage.

The result shows that BRAM, which is used up to 65%, is the bottleneck. It is used for storing the compressed CNN model of each layer and the partial sum of OFMs. A large BRAM usage comes from two reasons. First, the design requires a wide bit width memory. The weight and index memory for storing the compressed CNN model consumes 104 blocks of BRAM (52 blocks each for each memory). It requires wide bit width to support the maximum degree of inter-output parallelism

Table 4.3: Resource usage of the implementation of the proposed convolution core with 1,024 PEs optimized for VGG-like convolutional layers on Intel's Arria10 GX1150

Module	LUTs	Registers	DSPs	M20K
Parallelism Cntl	10,719 (2%)	16,892 (1%)	49 (3%)	0 (0%)
Broadcaster	27,725 (3%)	17,156 (1%)	1 (0%)	104 (3%)
PE Grid	202,309 (24%)	344,416 (20%)	576 (38%)	1,664 (61%)
Core (Total)	240,753 (29%)	378,543 (22%)	626 (41%)	1,768 (64%)

according to the number of BCUs. The number of BRAMs can be reduced when the number of BCUs decreases. Likewise, the partial sum buffer, which consumes 1,664 blocks of BRAM (26 blocks for each PE groups), requires wide bit width because the bit width of the partial sum is as high as 32 bits. Second, the BRAMs for partial sum buffer for 1,024 PEs is prepared for the worst-case scenario that P equals to 1 in the 512-kernel layers. In that case, it requires $32,024$ bits with 512×2 addresses in total to accommodate the data with double buffer. However, flexible parallelism technique employed by the proposed convolution core may reduce the required BRAMs when computing VGG-16. This issue is discussed in section 4.9.2.

The power consumption of the proposed convolution core is 25 Watts. It is considered efficient for edge processing platform or embedded systems considering its deliverable performance.

4.8 Comparison with Prior CNN Accelerators

The design quality of the proposed convolution core was compared with prior FPGA-based CNN accelerators. First, a comparison is made in terms of performance, i.e. active multiplier utilization and effective GMACS, to demonstrate that the proposed convolution core can increase multiplier utilization, and consequently, improve the GMACS performance. Then, a comparison with the prior arts at their best performance is made to show the efficiency of the proposed convolution core. Since the definition of PE is different among FPGA-based CNN accelerators, the resource for convolution means multiplier in this section. Noted that prior accelerators report giga operations per second (GOPS), where one GMACS is equivalent to two GOPS.

To make a fair comparison, the prior accelerators are selected based on types of parallelism that they exploit and whether they provide their results on VGG-16. Their description is as follows:

- **Caffeine** [34] implements inter-output and operation-level parallelism of multiple IFMs with the factor of 32×32 for unrolling the parallelism of

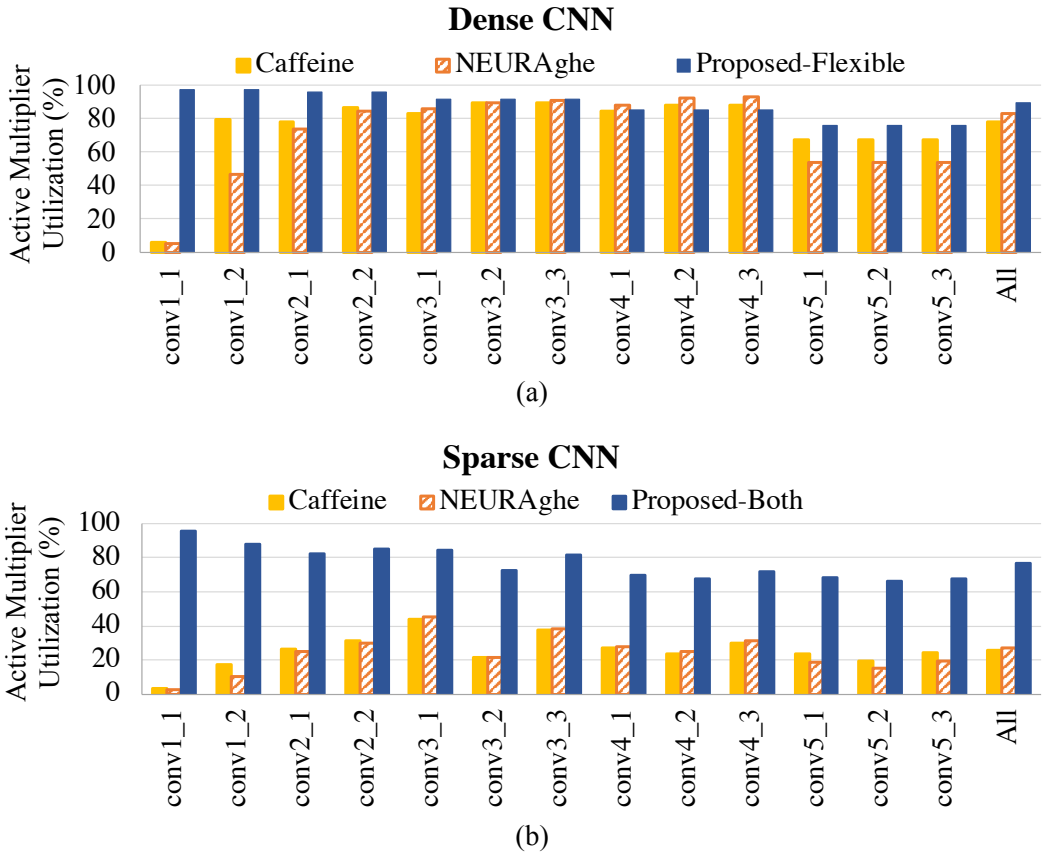


Figure 4.12: Active multiplier utilization of Caffeine, NEURAghe, and the proposed parallelism-flexible convolution core by layer of VGG-16: (a) in computing dense CNN; (b) in computing sparse CNN.

OFMs and IFMs in total of 1,024 multipliers. Its operating frequency is 200 MHz on Xilinx's Ultrascale KU060.

- **NEURAghe** [35] implements intra-output and operation-level parallelism. It includes 16 SoP modules, each of which contains 54 multipliers, in total of 864 multipliers. It was not scaled to 1,024 multipliers due to architecture constraints. Their reported performance at 140 MHz operating frequency is scaled to the performance at 200 MHz as follows:

$$GMACS_{200MHz} = \frac{GMACS_{140MHz} * 200}{140}, \tag{4.9}$$

where $GMACS_{200MHz}$ and $GMACS_{140MHz}$ are GMACS at 200 MHz and 140 MHz, respectively.

Active Multiplier Utilization To show that the proposed convolution core can efficiently utilize multipliers, the active multiplier utilization is illustrated in two

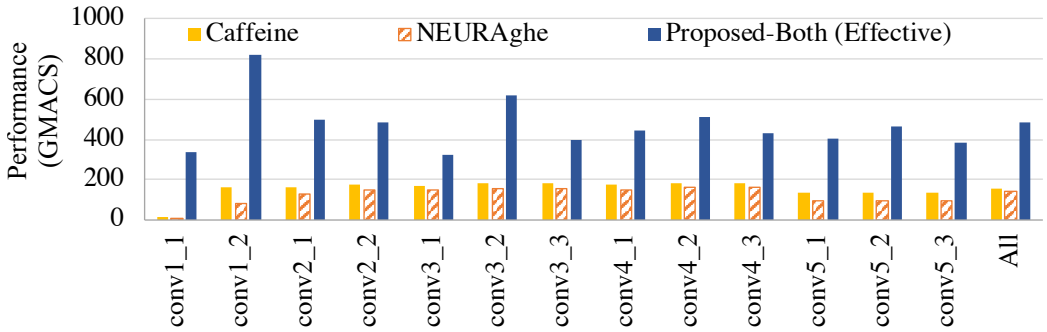


Figure 4.13: Performance in GMACS of Caffeine, NEURAghe, and the proposed parallelism-flexible convolution core by layer of VGG-16.

aspects: (1) in computing a dense CNN; (2) in computing a sparse CNN. In this context, the active multiplier utilization is not equivalent to PE utilization since the definition of PE varies between the chosen accelerators. For Caffeine and NEURAghe, the active multiplier utilization, $U_{Est.}$, is calculated from the performance in GMACS as follow:

$$U_{Est.} = \frac{GMACS_{200MHz} \times 100}{\#MUL \times f}, \tag{4.10}$$

where $\#MUL$ and f are the number of multipliers and operating frequency, respectively.

First, Fig. 4.12(a) shows active multiplier utilization of the Caffeine, NEURAghe, and Proposed-Flexible in computing a dense CNN to demonstrate the utilization improvement from flexible parallelism. Note that the utilization of Proposed-Flexible here is different from the previous section because the one in the previous section is the utilization in computing a sparse CNN, so the number of meaningful MACCs is less than that of a dense CNN. The figure shows that the Proposed-Flexible utilizes the multipliers better than both Caffeine and NEURAghe in almost all the layers and in the total across all the layers. That is because the Proposed-Flexible alternates the parallelism in effect and P to use the parallelism that theoretically results in the highest active multiplier utilization. However, the utilization of layer group conv4_x is slightly lower than Caffeine and NEURAghe due to the effect of decompressing overhead and fine-grained PE fragmentation.

Second, Fig. 4.12(b) shows active multiplier utilization in computing a sparse CNN. The superior active multiplier utilization of the proposed convolution core shows that most multiplier cycles are spent on meaningful MACCs, unlike the architectures that exploit operation-level parallelism and waste time on zero-operand MACCs. Furthermore, the results also imply that flexible parallelism works well with weight sparsity since the utilization of all layers is relatively high.

GMACS In Fig. 4.13, the performance in GMACS is illustrated. The Proposed-Both (Effective) refers to the equivalent effective GMACS that is achievable from

Table 4.4: Comparison with prior FPGA work

	[34]	[35]	[36]	[109]	Proposed
Device	Zynq KU060	Zynq XC7Z045	Zynq XC7Z045	Arria10 SX660	Arria10 GX1150
Exploited parallelism	Fixed operation-level, inter-output	Fixed operation-level, intra-output	Fixed operation-level, intra-output, inter-output	Fixed operation-level, intra-output, inter-output	Flexible intra-output, inter-output
Frequency	200 MHz	140 MHz	150 MHz	120 MHz	200 MHz
#Multiplier (#DSPs)	1,024 (1,058)	864 (864)	1,152 (780)	- -	1,024 (626)
Power (Watt)	26	10	9.63	-	25
Effective GOPS	310	170	188	53	960
Resource Efficiency	0.31	0.20	0.16	-	0.94
Power Efficiency	12.4*	17.0*	19.50*	-	38.4**

Resource Efficiency is GOPS/Multiplier and Power Efficiency is GOPS/Watt

*The power consumption is measured for the entire system of the CNN accelerator

**The power consumption is measured when there is only the convolution or core on FPGA

leveraging weight sparsity. Since the proposed convolution core skips all zero-operand MACCs, it can achieve a superior GMACS compared to other accelerators. The effective GMACS of the proposed convolution core in computing all 13 convolutional layers of VGG-16 is 480.7 GMACS.

To understand the usability of the proposed convolution core, a comparison with prior FPGA-based accelerators is made according to the reported implementation. In addition to the above-mentioned accelerators, the proposed convolution core is also compared to the accelerators in [36] and [109] (512-opt-pr variant). While Caffeine [34], NEURAghe [35] and the work in [36] compute a dense CNN, the work in [109] and the proposed convolution core can skip MACCs related to zero-valued weights.

Table 4.4 presents the comparison of the proposed convolution core with prior FPGA-based CNN accelerators, which are implemented for 16-bit fixed-point arithmetic precision. #Multiplier refers to the number of multipliers for MACCs on the design, which is calculated based on the parameters described in each paper. #DSP refers to the number of DSPs utilized on each accelerator as reported. Note that a Xilinx's DSP and an Intel's DSP can accommodate one and two 16-bit

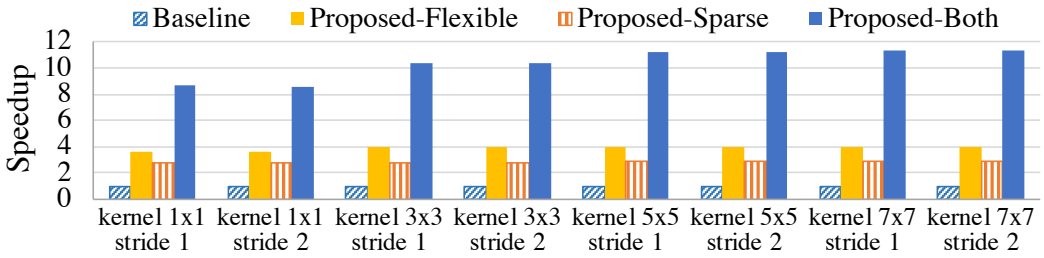


Figure 4.14: Speedup of the proposed parallelism-flexible convolution core by kernel size and stride compared to the baseline architecture.

fixed-point MACCs, respectively. The GOPS is evaluated from 13 convolutional layers of VGG-16.

Compared to other CNN accelerators, the proposed convolution core outperforms them in terms of effective GOPS performance, effective resource efficiency and effective power efficiency. It achieves 3x, 5x, 5x and 18x better performance than the Caffeine, NEURAghe, the work in [36], and the work in [109], respectively. In the case of [109], the low performance despite the fact that it can leverage sparsity is partially due to a relatively low frequency, which might be the results from high-level synthesis. It seems that the high effective GOPS of the proposed convolution core is the result of high frequency. However, when the NEURAghe is scaled as they claim to a larger FPGA, which may bring the frequency up to 200 MHz and double its performance, the proposed convolution core still outperforms in terms of effective GOPS. Similarly, the proposed convolution core achieved the highest effective resource efficiency. The high effective power efficiency of the proposed convolution core implies that the architecture is capable of processing one image with a lower power budget. This means that the proposed convolution core is efficient for being a platform at the edge or on embedded systems.

4.9 Applicability to Modern State-of-the-art CNNs

4.9.1 Evaluation

Based on our survey, the convolutional layer specification of modern state-of-the-art CNNs varies by kernel size and stride in addition to the size of IFMs, the size of OFMs, the number of input channels and the number of output channels. Except for AlexNet [16] which contains kernel size of 11 and stride 4 in the first layer, most modern state-of-the-art CNNs, such as YOLOv2 [13], FCN [159] and ResNet [18], contain convolutional layers with kernel size between 1×1 to 7×7 and stride of 1 to 2.

The concept of the proposed convolution core is effective for not only various sizes of IFMs, sizes of OFMs, numbers of input channels, and numbers of

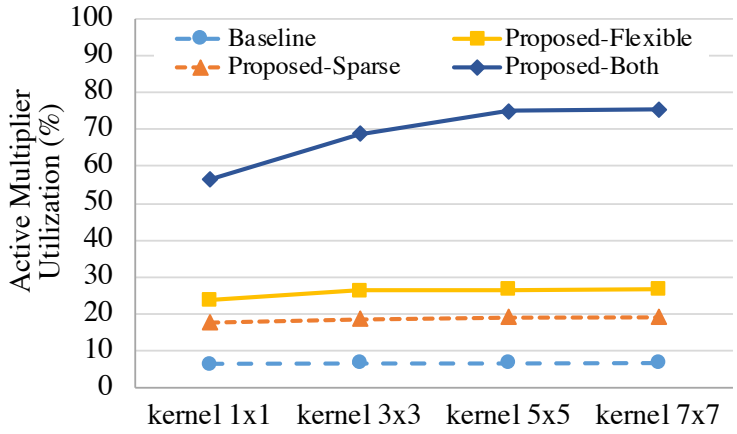


Figure 4.15: Active multiplier utilization of the proposed parallelism-flexible convolution core by kernel size compared to the baseline architecture when stride is 1. The active multiplier utilization is the same when stride is 2.

output channels as shown in the experiment on VGG-16, but the proposed convolution core also gains speedup and multiplier utilization despite various kernel sizes and strides. To show that the proposed convolution core can handle a wide range of modern CNNs, the implementation of the proposed convolution core was extended to various kernel sizes and strides by (1) adding logic for model decompression for various kernel sizes in the weight broadcaster; (2) enlarging local input buffer of each PE group to accommodate data for kernel size up to 7×7 and stride up to 2; (3) adding logic for selecting IFM pixels from local input buffer according to the index of sparse weights. The kernel size up to 7×7 and stride up to 2 were chosen because larger kernel size and stride are rare (no such CNN hyperparameters in YOLOv2, FCN or ResNet) although they can be handled by extending the implementation in a similar manner. In the evaluation of the extended implementation, a sparse model of convolutional layers was generated by zeroing out small values from a randomly generated kernels. The layer specification, i.e. X, Y, C_i, C_o and R, are fixed according to the conv5_1 of VGG-16 because its speedup is achieved from both sparsity and flexible parallelism. Likewise, P is chosen as 4 since it is independent of kernel size and stride.

The speedup and active multiplier utilization are shown in Fig. 4.14 and Fig. 4.15, respectively. Despite different strides, the speedup and active multiplier utilization are the same because the total number of MACCs is equal for the same size of OFMs and kernel size. For different kernel sizes, the Proposed-Flexible, Proposed-Sparse, and Proposed-Both achieve similar speedup and active multiplier utilization since performance improvement comes from sparsity and P, which are not affected by kernel size. Nevertheless, as the kernel size grows, more speedup and active multiplier utilization are achieved because they suffer less from imbalance workload. The performance improvement in both VGG-16

Table 4.5: Resource usage of the extended implementation of the proposed convolution core with 1,024 PEs on Intel's Stratix10 GX2800

Module	LUTs	Registers	DSPs	M20K
Parallelism Cntl	12,750 (1%)	20,143 (1%)	49 (1%)	0 (0%)
Broadcaster	30,804 (2%)	18,934 (1%)	1 (0%)	104 (1%)
PE Grid	328,430 (18%)	736,013 (20%)	576 (10%)	1,664 (15%)
Core (Total)	374,208 (21%)	774,782 (21%)	626 (11%)	1,768 (16%)

benchmark and this experiment is achieved by the concept of the parallelism-flexible convolution core, and hence they are not affected by the extension of the implementation.

The synthesis results in Table 4.5 shows the required resources. The increased resources in PE Grid originate from a larger local input buffer and IFM pixel selection from the local input buffer. The increased resources in weight broadcaster and parallelism controller comes from accumulating the index of sparse weight during model decompression and assigning OFM coordinates for a larger size of OFMs, respectively. The extended implementation of the proposed convolution core is synthesized for Intel's Stratix10 GX2800 FPGA. We have tried to evaluate the extended implementation on Arria10, but the required resources exceed the capacity of Arria10 GX1150 FPGA, whereas the numbers in Table 4.5 seems likely to accommodate in the capacity of Arria10. This probably comes from the architectural difference between Arria10 and Stratix10; for instance, Stratix10 has special registers on routing network called HyperFlex. For kernel size of 7×7 and stride of 2, the size of input buffer increases by 4.8 times compared to the case of the kernel size of 3×3 and stride of 1. Consequently, the LUTs for selecting IFM pixels from input buffer also increase despite the simple and fast access.

As the state-of-the-art CNNs, such as YOLOv2 and ResNet, have as much as 1k or 2k output channels in a convolutional layer, a large number of output channels can be handled by either increasing the size of output buffer or using $P > 1$. In the extended implementation, the output buffer size is kept as 512 addresses because such a large number of output channels usually occurs in deep layers, where the size of OFMs is small and degree of parallelism P is more than 1.

The results have shown that the proposed convolution core is useful for various layer specifications. It is applicable to accelerating the convolutional layers for various state-of-the-art CNNs, such as YOLOv2, FCN, ResNet.

4.9.2 Discussion

This section analyzes the insufficiencies and bottleneck of the proposed convolution core. Then, it discusses possible solutions and improvement.

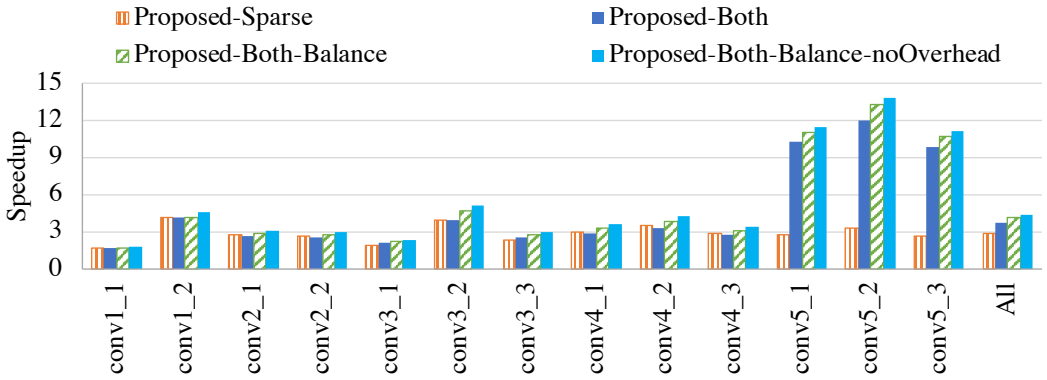


Figure 4.16: Speedup of the proposed parallelism-flexible convolution core by layer of VGG-16 in ideal execution scenario.

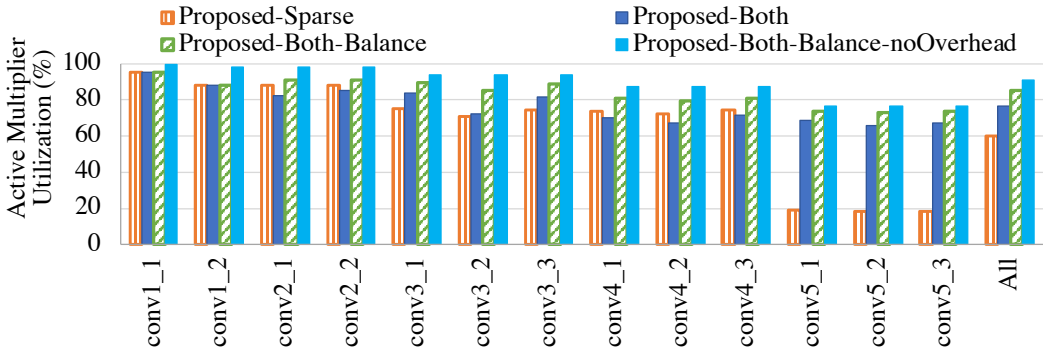


Figure 4.17: Active multiplier utilization of the proposed parallelism-flexible convolution core by layer of VGG-16 in ideal execution scenario.

Performance

There exist three insufficiencies that prevent the proposed convolution core from bringing about its peak performance. First, idle PE cycles arise from the imbalance workload. Second, decompressing the compressed CNN model incurs the decompressing overhead. Third, the architectural fragmentation constraints the scheduling of parallelism.

The first insufficiency is that the imbalance workload of sparse kernels increases idle PE cycles when the proposed convolution core exploits inter-output parallelism. That is because the proposed convolution core unrolls the degree-of-parallelism loop in line 3 in Algorithm 1 to implement inter-output parallelism. If the total number of non-zero weights in all the kernels (loop in line 5 and 6) that belong to each iteration of line 3 is not equal, PEs are idle in order to wait for PEs in other iterations to finish their workload. To investigate the effect of the imbalance workload, an artificial sparse VGG-like CNN which makes the workload in every kernel equal is generated. The performance is measured and shown

as Proposed-Both-Balance in Fig. 4.16. The result shows that the overall performance is improved by 9%. In addition, the Proposed-Both-Balance outperforms the Proposed-Sparse in every layer, which implies that the flexible parallelism can improve the performance of every layer. Figure 4.17 illustrates active multiplier utilization, which shows that the Proposed-Both-Balance utilizes PEs better because no PE waits for the others. This problem can be solved in either hardware or software. In hardware, the kernels should be divided into P partitions with an arbitrary number of kernels per partition in such a way that the workload is balanced. However, this may cause complication in storing the results to the partial sum buffer because the partitioning may vary in every input channel. In software, the CNN sparsification process should constraint the number of non-zero weights of each kernel so that it results in a balanced workload.

Second, the existence of decompressing overhead degrades both performance and active multiplier utilization because the PEs are idle during those cycles. As shown in Fig. 4.8, the overhead occurs once every input channel as a pipeline latency. This means that more data tiles due to a large P incur more overhead, which degrades the advantage of flexible parallelism. Figure 4.16 and Fig. 4.17 show that the speedup and utilization of the ideal execution (Proposed-Both-Balance-noOverhead) improve and the effect of overhead is illustrated with the difference of Proposed-Both-Balance and Proposed-Both-Balance-noOverhead. A 16-cycle decompressing overhead comes from the pipeline for decompressing the compressed CNN model that aims to achieve high frequency. As a consequence, decreasing this overhead may degrade the operating frequency, which results in longer execution time despite the reduced execution cycles.

Third, the proposed convolution core suffers from architectural fragmentation that prevents PE occupancy during convolution cycles. As mentioned above, there are two types of fragmentation: fine-grained and medium-grained. The example of fine-grained fragmentation is layer group conv4_x, where 28 pixels in one row of OFM leave four idle PEs out of 32 PEs in two PE groups, each of which contains 16 PEs. It adds up to at least 12.5% of all PEs. They cannot be occupied due to the local input buffer limitation. Medium-grained fragmentation occurs in layer group conv5_x, where 14×14 output pixels of one OFM occupy only 196 PEs out of 256 PEs in four PE banks. The effect is as large as 24% of all PEs, which is the main reason for no more than 76% of active multiplier utilization. The architecture is unable to schedule neither inter- nor intra-output parallelism due to the limitation in one-to-one connection to the BCU and dimension of OFM. The effect of this problem can be mitigated by choosing the parameter that is suitable for certain CNN.

Resource Usage

In the implementation, the partial sum buffer is designed to support the worst case. However, the number of words can be reduced by the factor of P when executing the proposed convolution core with $P > 1$ in the layers that contain a large number

of kernels. In other words, the required number of words can be reduced to the maximum of $\frac{C_o}{P}$ across the CNN.

4.10 Conclusion and Future Work

To achieve high performance, the proposed parallelism-flexible convolution core for sparse CNN accelerator exploits multiple types of parallelism flexibly layer by layer to maximize multiplier utilization and skips redundant MACCs due to weight sparsity. The integration of both techniques with parallelism controller and weight broadcaster that are not complicated in terms of dataflow control and resource usage improves performance significantly by 4x speedup over the baseline architecture and 3x in effective GMACS over prior arts of CNN accelerator. To maximally take advantage of the proposed convolution core, the constrained sparsification process remains as the future work.

Chapter 5

An Architecture Exploration of SoCs for CNN-based AI Platform

Chapter 5 proposes an architecture design space exploration method of SoCs for CNN-based AI platform concerning a configurable multi-layer bus with hierarchical shared bus subsystems. First, the CNN modeling is described in terms of modeling granularity and data tiling. Next, the model is defined and the multi-objective architecture exploration problem is formulated. Then, the design quality evaluation method and architecture exploration method are proposed. Finally, the experiments show the validity and design space coverage.

5.1 Motivation and Objective

The advancement of semi-conductor process technology has made it feasible to fabricate a large scale integrated (LSI) circuit on a chip. The chip becomes powerful as it has acquired multi-functional processing capability, while the multi-function dilates the complexity of the designing a system. In addition, strict constraints of design quality, which are high-performance, small area and low energy consumption, are raised at the same time to further complicate the SoC design.

Exploring the enormous design space is cumbersome, but crucial in finding optimal architecture. Since IP and bus architecture combination generates a vast amount of architecture, finding optimal design in an early design stage (before proceeding to low-level design, such as RTL) can shorten design time. The complexity lies in an efficient way to select and evaluate IP, bus architecture and their parameters.

The problems in exploring an SoCs design for CNN-based AI Platform are two folds: efficiently discover architecture that parallelizes computation-intensive convolutional layer and determine bus architecture that is capable of transferring a massive amount of data incorporated with CNN processing. The first problem relates to granularity of system-level modeling. A CNN is usually modeled layer by layer (one layer as one process; coarse-grained modeling granularity), which

regulates the parallelization of data computation to process-level. For that reason, when the workload of processes in the system is imbalanced, for instance, the amount of computation of a convolutional layer is much more than the amount of computation of a pooling layer, intra-layer parallelism is not considered for performance improvement with the IP selection. On the other hand, modeling the data computation in finer granularity, such as OFM-wise (the computation of one OFM as a process) or operation-wise (one MACC as a process) granularity, allows the parallelization of intra-layer MACCs with the price of designer's system-level modeling effort and unnecessarily large design space. The second problem discloses the difficulty in discovering high-performance communication architecture, aka bus architecture, between components within the SoC. Standard specifications for the multi-layer bus are developed such as AMBAs multilayer AHB [143] and AXI [144]. Since a full bus matrix contains a massive amount of wires, which leads to routing problems, the specifications also define configurations of the multi-layer bus and the model of bus matrix in addition to the regulations and communication methods in order to reduce the number of wires on the bus matrix. Therefore, it is important to find a multi-layer bus configuration that can satisfy design constraints because the topology, configuration and protocol of the communication architecture affect the design quality.

This chapter proposes an architecture exploration method to find Pareto-optimal SoC architectures for CNN-based AI Platform. The proposed method solves multi-objective design space exploration with traversal through parameter trees. This chapter includes the following contributions:

1. in addition to parameterizing SoC architecture itself, the IPs are also parameterized and explored, so that a computation-intensive process, such as convolutional layers, can be scheduled on multiple instances and variable number of PEs. This allows the exploration to consider the parallelization of intra-layer MACCs within a process with a coarse-grained modeling granularity, which reduces the cumbersome of hand-crafted medium- and fine-grained modeling and IP selection;

2. both topology and specification, e.g. bus width and execution frequency, of hierarchical shared bus and multi-layer bus architecture are parameterized in terms of three-step channel mapping, mappings related to bus matrix and bus parameter mappings. With this method, multiple configurations of multi-layer bus can be easily explored.

5.2 Modeling CNN

As mentioned in section 4.2.1, a typical CNN is composed of convolutional, pooling, normalization and fully-connected layers. In application design and system-level modeling, a CNN is usually modeled layer by layer because layer-wise modeling is provided by most deep learning libraries, such as caffe [160]. However,

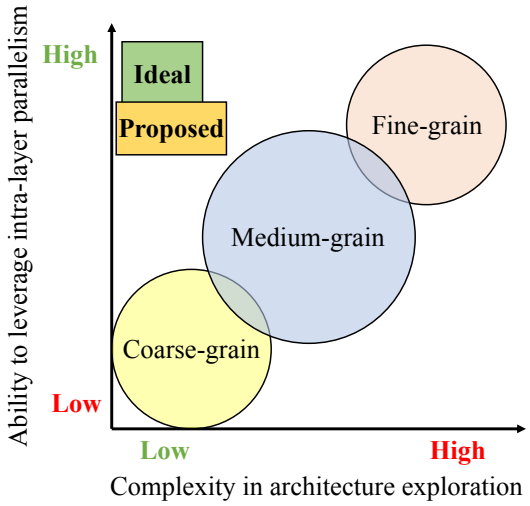


Figure 5.1: The relationship between the ability to leverage intra-layer parallelism and complexity in architecture exploration of modeling granularity.

modeling granularity of CNN has two effects in architecture exploration: the ability to leverage intra-layer parallelism and complexity in architecture exploration. The ability to leverage intra-layer parallelism means the ability of the architecture exploration to realize the parallelization of intra-layer MACCs within one layer. The complexity in architecture exploration means the effort to model and explore the design space. In concrete, it implies the number of processes and channels within an SLM. The larger the number is, the more modeling effort it requires, and the larger set of processes and channels in IP and bus architecture selection it has, respectively.

5.2.1 Modeling Granularity

Figure 5.1 shows the relationship between the ability to leverage intra-layer parallelism and complexity in architecture exploration when employing coarse-grained, medium-grained, and fine-grained granularity. The higher ability to leverage intra-layer parallelism and the less complexity are desired.

In coarse-grained modeling granularity, each process models the behavior of each CNN's layer and is mapped onto an IP in system-level design. Modeling CNN with this granularity is convenient for designers since it conforms with deep learning frameworks in application design. It also introduces low complexity in architecture exploration because the number of processes to map to IPs is not more than the number of CNN's layers. However, due to the fact that some layers, such as convolutional layers, include more operations than the other, mapping process to IP introduces unbalance workload to the system, which may degrade the capability of the architecture. In other words, modeling with coarse-grained

granularity restricts the ability to leverage intra-layer parallelism during the architecture exploration.

In medium-grained modeling granularity, each of the computation-intensive layers like convolutional layers is modeled into several processes by OFMs or IFMs or even output pixels. The finer modeling granularity enables intra-layer parallelism to be mapped onto multiple IPs, and hence, increases the ability to leverage intra-layer parallelism. On the other hand, the increased number of processes complicates the architecture exploration, especially if the processes of the same layer are mapped to IPs non-systematically.

The fine-grained modeling granularity models CNN's layers in an operation-wise manner. It allows the architecture exploration to leverage intra-layer parallelism in several levels, i.e. inter-output, intra-output and operation-level parallelism. However, there can be a vast number of processes, which increases complexity in architecture exploration.

Ideally, the modeling granularity should enable the architecture exploration to leverage high intra-layer parallelism and low exploration complexity. The green rectangle labeled with Ideal in Fig. 5.1 represents the ideal relationship between the two properties of the exploration.

The proposed architecture exploration method achieves two desired properties through modeling the CNN with the coarse-grained SLM and parameterized IPs. Each process of the coarse-grained SLM is mapped onto the IPs that is parameterized in terms of the number of instances and PEs, especially CNN accelerators. The coarse-grained SLM enables low complexity in the exploration, while IP parameterization increases the ability to leverage intra-layer parallelism by taking advantage of multiple instances or PEs. Nonetheless, to obtain both properties, it is necessary to model data tiling behavior in order to analyze parallel computation on multiple instances of functional blocks.

5.2.2 Nature of Data Tiling in CNN

In the processing of CNN, specifically convolutional layers, many CNN accelerators partition IFMs and OFMs into multiple blocks, aka tiles, to take advantage of data locality [34, 105, 109]. They reuse IFMs, OFMs and kernels within the same layer, and reuse OFMs as IFMs of the next layer [110]. The benefits include small on-chip memory and reducing external memory access.

In the processing of convolutional layer, data tiling divides the IFMs and weight into multiple tiles by unrolling the loops of convolution algorithm. A design space exploration searches the tiling factor space using a roofline model [161], which is an analytical model, in order to select the tiling factor that maximizes computational throughput and external memory access [105, 162, 163]. It offers multi-dimensional data tiling.

This research models the behavior of data tiling in the H and W dimension (in Fig. 4.1(a)) of IFMs to take advantage of intra-layer parallelism. Figure 5.2

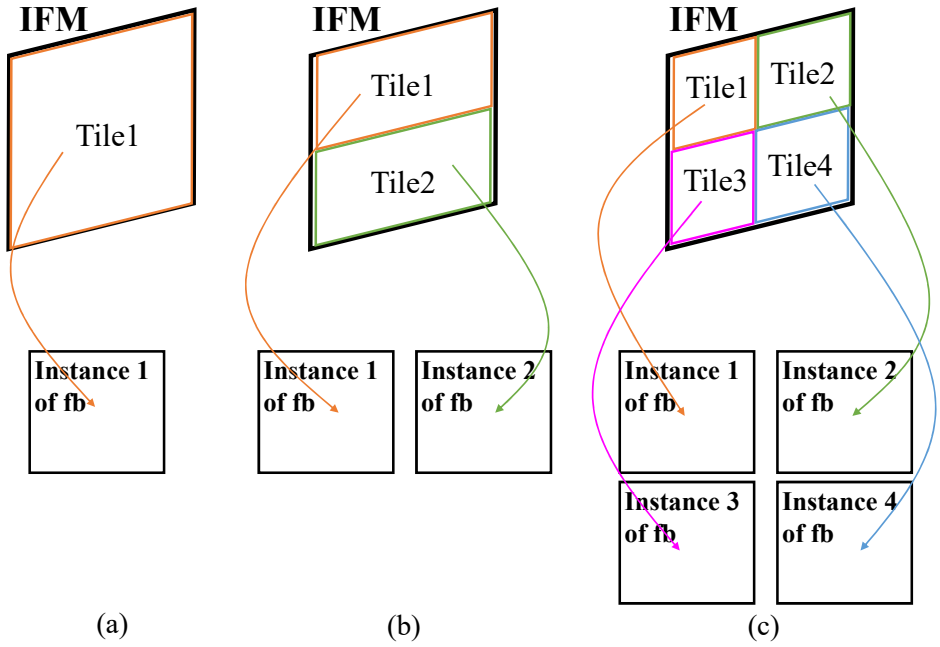


Figure 5.2: An example of mapping data tiles of a convolutional layer onto multiple instances of CNN accelerator functional block.

illustrates the example of mapping a convolutional layer onto multiple instances of CNN accelerator functional block. Assuming that the layer is mapped onto one, two and four instances. In Fig. 5.2(a), since there is only one instance, the IFM is divided into one tile and is mapped to instance 1. In Fig. 5.2(b) and Fig. 5.2(c), the IFM is divided into two and four tiles, respectively. Each tile is mapped on to one instance. Every instance can operate simultaneously, hence computing intra-layer MACCs in parallel. The intra-layer parallelism is leveraged more as the number of instances grows. Tiling dimensions that incorporate the dimension of weights are not considered because the number of MACCs varies depending on sparsity in the model.

5.3 Model Definitions

This section defines the MoC and architectural model employed by the proposed architecture exploration method, which is extended from section 3.3.1 and section 3.3.2, respectively. The extension of MoC includes the layer specification of deep learning model, e.g. type of layer, the number of hidden nodes of NNs or the number and size of CNN kernels. The extension of architectural model includes the parameterization of IP.

5.3.1 Model of Computation (MoC)

Based on the model of SLM, M_{sl}, discussed in section 3.3.1, a process $p_i \in P$ is described further in terms of the layer-wise description of deep learning model. Assuming that a process related to the deep learning model in an application describes data computation of one layer, it is represented with $p_i = (t_p, L_i)$, where t_p refers to the type of layer of p_i, e.g. convolutional layer, pooling layer or etc., and L_i refers to the layer specification. For example, for convolutional layer, L_i includes H, W, C_i, S, K, X, Y and C_o as described chapter 4.

5.3.2 Architectural Model

Based on the architectural model, M_{al}, that describes the parameterized SoC architecture, the IP modules in the architectural model are further parameterized into the number of module instances and number of PEs. Each functional block is represented with $fb_i = (j, N_{fb_i}, N_{pe}, P_{fb_i}, f_{fb_i}, e_{(p_k,fb_i)}) \in F$, which indicates that the functional block i, fb_i, is N_{fb_i} instances of IP j and each instance consists of N_{pe_i} PEs. In this way, an intra-layer computations within a process can be parallelized using multiple instances without modeling those vast amount of intra-layer computations as individual processes. Since processes of convolutional layers consume most processing time, this thesis applies N_{fb_i} and N_{pe_i} to only the functional blocks of CNN accelerators. For other functional blocks, both N_{fb_i} and N_{pe_i} are accounted as 1. Furthermore, memory IP is additionally defined as the IP for storing data. It can be either on-chip memory or interface of off-chip memory. Its model has a unique characteristic that the memory IP does not contain buffer and can store all the data within the same frame. In addition, bus width is considered as address bus width, w_{a,b_i} and data bus width, w_{b_i} for shared bus, and address bus width, $w_{a,bm}$ and data bus width, w_{bm} for multi-layer bus.

5.4 Problem Formulation of a Multi-objective Architecture Exploration

Figure 5.3 shows the overview of the proposed architecture exploration method. This section describe the inputs, output and objective functions in details.

5.4.1 Input

There are five inputs to the architecture exploration as follows.

1. **IP database** keeps the information about IPs registered by the designer. It includes following IP information.
    - **IP gate count** : the number of logic gate used for the IP implementation

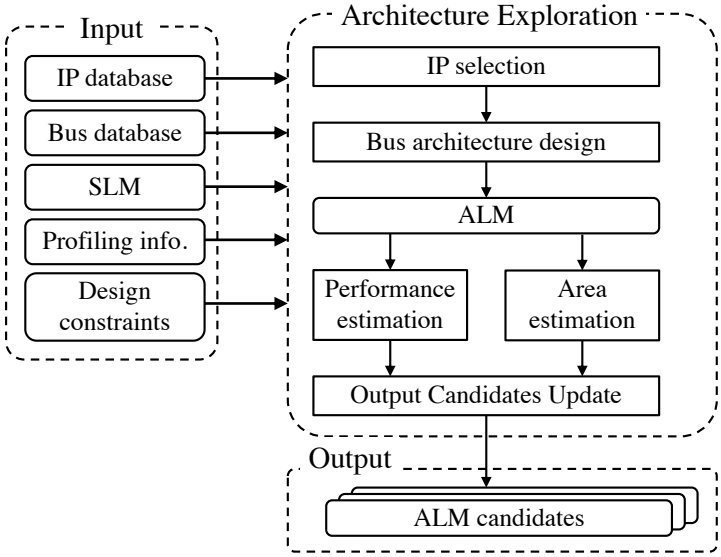


Figure 5.3: Overview of the proposed architecture exploration method.

- **Execution frequency candidate** : the candidates of execution frequency at which the IP can operate
- **Executable process** : the processes that can be executed on each IP
- **Execution cycle** : the execution cycle spent for each executable process on the IP.
- **The number of master ports and slave ports** : the number of ports of each standard protocol, consisting of the number of AHB master ports, the number of AHB slave ports and the number of APB slave ports.
- **Other IP parameters** : the list and candidates of IP's parameter, such as the number of instances and the number of PEs.

2. **Bus database** specifies the information of bus protocols, which consists of the bus protocol name, the data bus width candidates, the address bus width candidates, the execution frequency candidates and the maximum number of master and slave interfaces.

3. **SLM** (M_{sl}) defines system level behavior of the target system. It is described in SystemC [37], where the process represents the data processing and the channel represents the data communication.

4. **Profiling information** (Profiling info. in Fig. 5.3) includes the execution order and the amount of transfer data. It is extracted from the system level profiling of the SLM using the loosely-timed transaction level simulation explained in section 3.4.1.

5. **Design constraints** are raised to the architecture exploration by system designer. The necessary constraints are the maximum number of buses in each cluster, the maximum number of buffers, the maximum number of storage blocks, and the maximum number of bus bridges. Additionally, a designer can also raise the execution time constraint and the area constraint.

5.4.2 Objective functions of Architecture Exploration

This thesis considers two design quality metrics as objective functions for architecture exploration.

1. **Performance function** is described in terms of execution time. The execution time is estimated by the AL-EDG analysis method explained in chapter 3 and section 5.5.1.

2. **Hardware area function** is the function of hardware area. It is estimated in the architecture level and its estimation method is described in section 5.5.2.

5.4.3 Output

Output of the proposed architecture exploration method is **ALM candidates** that represent all Pareto solutions. Each ALM candidate, M_{al}, includes an ALM as described in section 3.3.2 and section 5.3.2, estimated execution time, and hardware area. Each Pareto solution holds the design trade-off between the execution time and the area of the multi-objective architecture exploration.

5.5 Design Quality Evaluation

The proposed architecture exploration method evaluates two design qualities: performance and hardware area. Performance is estimated in terms of execution time using an efficient performance estimation method based on chapter 3 and considering the behavior related to data tiling. Area is estimated from the hardware components comprising an architecture described by ALM.

5.5.1 Performance Estimation

This work estimates performance of each architecture using the method proposed in section 3.4, which is consisted of four procedures. The system-level profiling and SL-EDG construction are proceeded as the method of Ueda *et al.* [52]. The AL-EDG is constructed and analyzed considering data tiling.

Figure 5.4 shows an example of an SLM containing a process of a convolutional layer and its associated processes. The $Data1$ (p_0) and $Data2$ (p_5) processes store

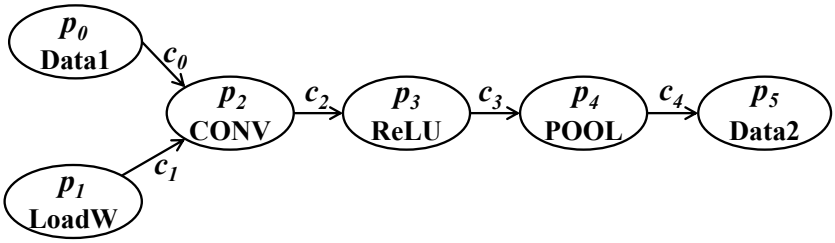


Figure 5.4: An example of an SLM containing a process of a convolutional layer and its associated processes.

OFMs and fetch them as IFMs of the next layer. The *LoadW* (p_1) process loads weights (kernels) of the following convolutional layer, *CONV*. The *CONV* (p_2), *ReLU* (p_3) and *POOL* (p_4) processes are the processes of convolutional, ReLU and pooling layer, respectively.

SL-EDG is represented by $G_{sl} = (V_{sl}, E_{sl})$ as defined in 3.4.2. An example of SL-EDG of the above SLM is illustrated in Fig. 5.5. Two schemes for modeling convolutional layers are (1) IFM-major scheme, which models one data processing using one IFM channel and (2) OFM-major scheme, which models one data processing for computing one OFM channel. Assuming that a convolutional layer uses 2-channel IFMs to produce 3-channel OFMs. The circular node denoted by $v_{p_{(i,k)}}$ represents the vertex of p_i's k^{th} data processing and the octagonal node denoted by $v_{c_{(j,l)}}$ represents the vertex of c_j's l^{th} data transfer. p_1 and p_2 consist of two vertices (2-channel IFMs) as in Fig. 5.5(a) in the modeling with the IFM-major scheme, while they consist of three vertices (3-channel OFMs) as in Fig. 5.5(b) in the modeling with the OFM-major scheme. p_3 and p_4 consist of three vertices (3-channel OFMs from p_2 as input), and p_0 and p_5 consist of one vertex since they are process for storing data. The processes that are not loading weights, convolutional, activation, and pooling layers are referred to as typical processes.

Applying data tiling behavior to processes of convolutional layers also affects the processes and channels associated with convolutional layers. The processes associated with a convolutional layer include (1) process of loading weights of the convolutional layers that precedes the process of convolutional layer, e.g. *LoadW* process in Fig. 5.4; (2) process of activation layer, such as ReLU layer, that follows the process of convolutional layer, e.g. *ReLU* process; (3) process of pooling layer that follows the process of convolutional layer or the activation layer in (2), e.g. *POOL* process. The associated channels include (1) channel that transfers IFMs data to convolutional layer, e.g. channel c_0; (2) channel that transfers weights to convolutional layer, e.g. channel c_1; (3) channels between the mentioned associated processes, e.g. channel c_2 and c_3; (4) channel that transfers OFMs data to memory or the next convolutional layer, e.g. channel c_4. In order to analyze data tiling behavior, the AL-EDG construction partitions the mentioned processes and channels according to tiles of data and the ALM described in section 5.3.2.

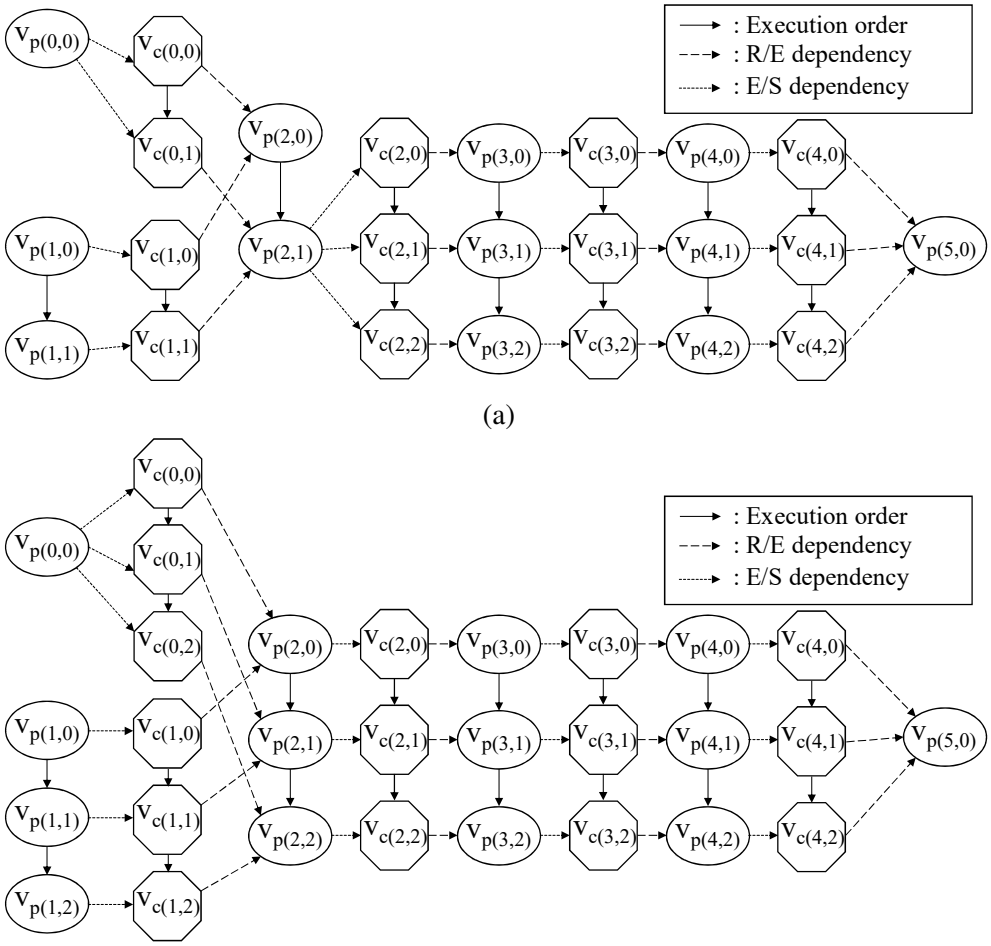


Figure 5.5: An example of an SL-EDG of SLM in Fig. 5.3: (a) convolutional layer modeled with IFM-major scheme; (b) convolutional layer modeled with OFM-major scheme.

AL-EDG Construction

In addition to the vertices and edges of SL-EDG and vertices and edges associated with DMAC and memories as explained in section 3.4.3, AL-EDG also consists of the additional vertices and edges due to data tiling and the number of instances of functional blocks. AL-EDG is represented by $G_{al} = (V_{al}, E_{al})$, where V_{al} and E_{al} are AL-EDG's vertex set and edge set, respectively. In the AL-EDG, $v_{p_{(i\_q,k\_r)}}$ represents the vertex of the r^{th} partition of p_i's k^{th} data processing, which is scheduled on the q^{th} instance of the functional block and $v_{c_{(j,l\_s)}}$ represents the vertex of c_j's s^{th} partition of the l^{th} data transfer. The AL-EDG is constructed as follows;

1. Copy vertices and edges of SL-EDG as those of AL-EDG as follows:

(a) Copy $v_{p(i,k)} \in V_{sl}$ as $v_{p(i\_0,k\_0)}$ and add to V_{al}.

(b) Copy $v_{c(j,l\_0)} \in V_{sl}$ as $v_{c(j,l\_0)}$ and add to V_{al}.

(c) Copy $(v_{p(i,k)}, v_{p(i,k+1)}) \in E_{sl}$ as $(v_{p(i\_0,k\_0)}, v_{p(i\_0,k+1\_0)})$ and add to E_{al}.

(d) Copy $(v_{c(j,l\_0)}, v_{c(j,l+1\_0)}) \in E_{sl}$ as $(v_{c(j,l\_0)}, v_{c(j,l+1\_0)})$ and add to E_{al}.

(e) Copy $(v_{p(i,k)}, v_{c(j,l\_0)}) \in E_{sl}$ as $(v_{p(i\_0,k\_0)}, v_{c(j,l\_0)})$ and add to E_{al}.

(f) Copy $(v_{c(j,l\_0)}, v_{p(i,k)}) \in E_{sl}$ as $(v_{c(j,l\_0)}, v_{p(i\_0,k\_0)})$ and add to E_{al}.

2. Alter V_{al} and E_{sl} so that the AL-EDG includes vertices and edges involving data tiling. It is assumed that a convolutional layer receives C_i-channel IFMs as input and generates C_o-channel OFMs as output. For every process p_i, do as follows;

   (a) If p_i is a process of convolutional layer that is mapped onto Q instances of CNN accelerator IP's functional block and is divided into R tiles per one instance of functional blocks, which means that data of the layer is divided into $Q \times R$ tiles, do as follows;

      i. For each vertex $v_{p(i\_0,k\_0)}$, make vertices $v_{p(i\_q,k\_r)}$, where $q = 0, ..., Q-1$ and $r = 0, ..., R-1$, and add them to V_{al}. Each of them represents the processing of each data tile on each instance of the mapped functional block. The amount of computation of each new vertex $v_{p(i\_q,k\_r)}$ is $\frac{1}{Q \times R}$ times of the amount of computation of the original vertex $v_{p(i\_0,k\_0)}$.

      ii. For each edge $(v_{c\,j,l\_0}, v_{p(i\_0,k\_0)})$, add edges $(v_{c\,j,l\_0}, v_{p(i\_q,k\_r)})$, which represent R/E dependencies of each data tile, to E_{al}.

      iii. For each edge $(v_{p(i\_0,k\_0)}, v_{c(j,l\_0)})$, add edges $(v_{p(i\_q,k\_r)}, v_{c\,j,l\_0})$, which represent E/S dependencies of each data tile, to E_{al}.

      iv. Make edges that represent execution orders between data processing, which depends on modeling scheme. For IFM-major scheme, make edges $(v_{p(i\_q,k\_r)}, v_{p(i\_q,k+1\_r)})$ and $(v_{p(i\_q,C_i-1\_r)}, v_{p(i\_q,0\_r+1)})$. For OFM-major scheme, make edges $(v_{p(i\_q,k\_r)}, v_{p(i\_q,k\_r+1)})$ and $(v_{p(i\_q,k\_R-1)}, v_{p(i\_q,k+1\_0)})$.

   Figure 5.6 (a) shows these steps for vertices of p_2 when $Q = 2$ and $R = 2$. Each vertex of p_2 is partitioned into four vertices, for example, the $v_{p(2\_0,0\_0)}$ is partitioned into $v_{p(2\_0,0\_0)}$, $v_{p(2\_0,0\_1)}$, $v_{p(2\_0,0\_2)}$, and $v_{p(2\_0,0\_3)}$, each of which is responsible for each data tile. The vertices of p_2 become eight vertices of four data tiles undertaken by two instances of the functional block.

   (b) If p_i is a process of loading weights of the following convolutional layers, which is mapped onto Q instances of functional block of CNN accelerator IP, and the modeling scheme is IFM-major scheme, do as follows;

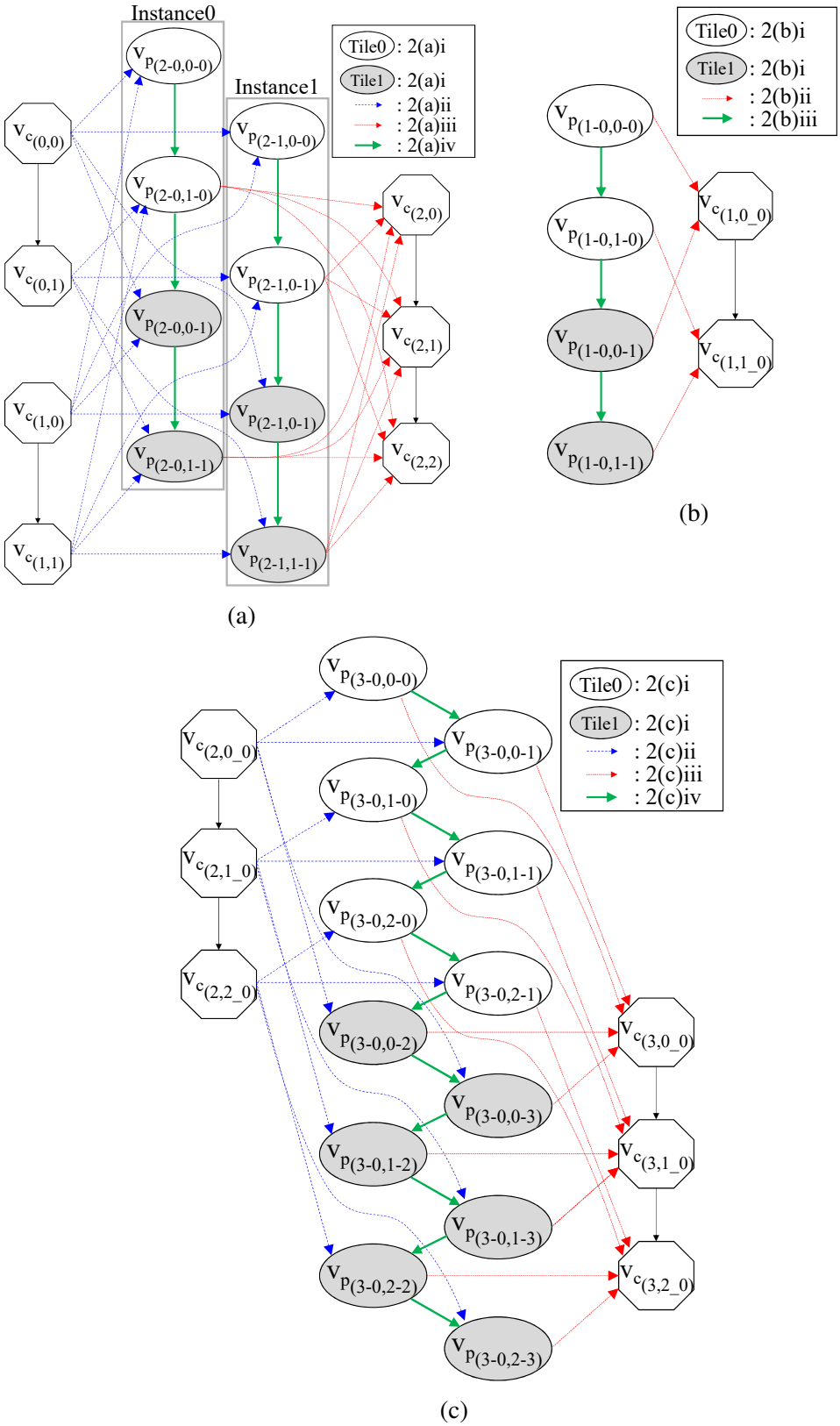


Figure 5.6: An example of AL-EDG construction in step 2 of the SLM in Fig. 5.5(a), which is implemented with IFM-major scheme.

i. For each vertex $v_{p(i\_0,k\_0)}$, duplicate it as vertices $v_{p(i\_0,k\_r)}$ where $r = 0, ..., R-1$, and add them to V_{al}. Each of them represents the processing of loading weights for each tile.

ii. For each edge $(v_{p(i\_0,k\_0)}, v_{c(j,l\_0)})$, add edges $(v_{p(i\_0,k\_r)}, v_{c_{j,l\_0}})$ to E_{al}.

iii. Make edges $(v_{p(i\_0,k\_r)}, v_{p(i\_0,k+1\_r)})$ and $(v_{p(i\_0,C_i-1\_r)}, v_{p(i\_0,0\_r+1)})$ to represent execution orders between data processing.

Figure 5.6 (b) shows these steps for vertices of p_1. Each vertex of p_1 is duplicated into two vertices, for example, the $v_{p(1\_0,0\_0)}$ is duplicated as $v_{p(1\_0,0\_0)}$ and $v_{p(1\_0,0\_1)}$. Each vertex of p_1 is responsible for loading weights for each data tile.

(c) If p_i is a process of activation layer, such as ReLU layer, or pooling layer, or normalization layer, the preceding process of convolutional layer is mapped onto Q instances of functional block of CNN accelerator IP, do as follows;

i. For each vertex $v_{p(i\_0,k\_0)}$, make vertices $v_{p(i\_0,k\_r)}$ where $r = 0, ..., (Q-1) \times (R-1)$, and add them to V_{al}. Each of them represents the processing of each data tile. The amount of computation of each new vertex $v_{p(i\_0,k\_r)}$ is $\frac{1}{Q \times R}$ times of the amount of computation of the original vertex $v_{p(i\_0,k\_0)}$.

ii. For each edge $(v_{c_{j,l\_0}}, v_{p(i\_0,k\_0)})$, add edges $(v_{c_{j,l\_0}}, v_{p(i\_0,k\_r)})$ to E_{al}.

iii. For each edge $(v_{p(i\_0,k\_0)}, v_{c(j,l\_0)})$, add edges $(v_{p(i\_0,k\_r)}, v_{c_{j,l\_0}})$ to E_{al}.

iv. Make edges that represent execution orders between data processing, which differs by modeling scheme.
For IFM-major scheme, make edges $(v_{p(i\_0,k\_r)}, v_{p(i\_0,k\_r+1)})$ to represent the order of process vertices of the same tile and input channel from different instances of functional block of CNN accelerator IP, edges $(v_{p(i\_0,k\_r)}, v_{p(i\_0,k+1\_r-Q+1)})$ to represent the order of process vertices of the same tile but different input channels, and edges $(v_{p(i\_q,C_o-1\_r)}, v_{p(i\_q,0\_r+1)})$ to represent the order of process vertices from different tiles.
For OFM-major scheme, make edges $(v_{p(i\_0,k\_r)}, v_{p(i\_0,k\_r+1)})$ and $(v_{p(i\_0,k\_((Q-1)\times(R-1)))}, v_{p(i\_0,k+1\_0)})$.

Figure 5.6 (c) shows these steps for vertices of p_3. Each vertex of p_3 is partitioned into four vertices, each of which is responsible for each data tile. The vertices of p_3 become twelve vertices from two instances of the functional block.

3. Alter V_{al} and E_{sl} so that the AL-EDG includes channel vertices and edges involving data tiling. For every channel $c_j = (p_u, p_x) \in C$, do as follows;

(a) If p_u is a process of loading weights and p_x is a process of convolutional layer mapped onto Q instances of functional block of CNN

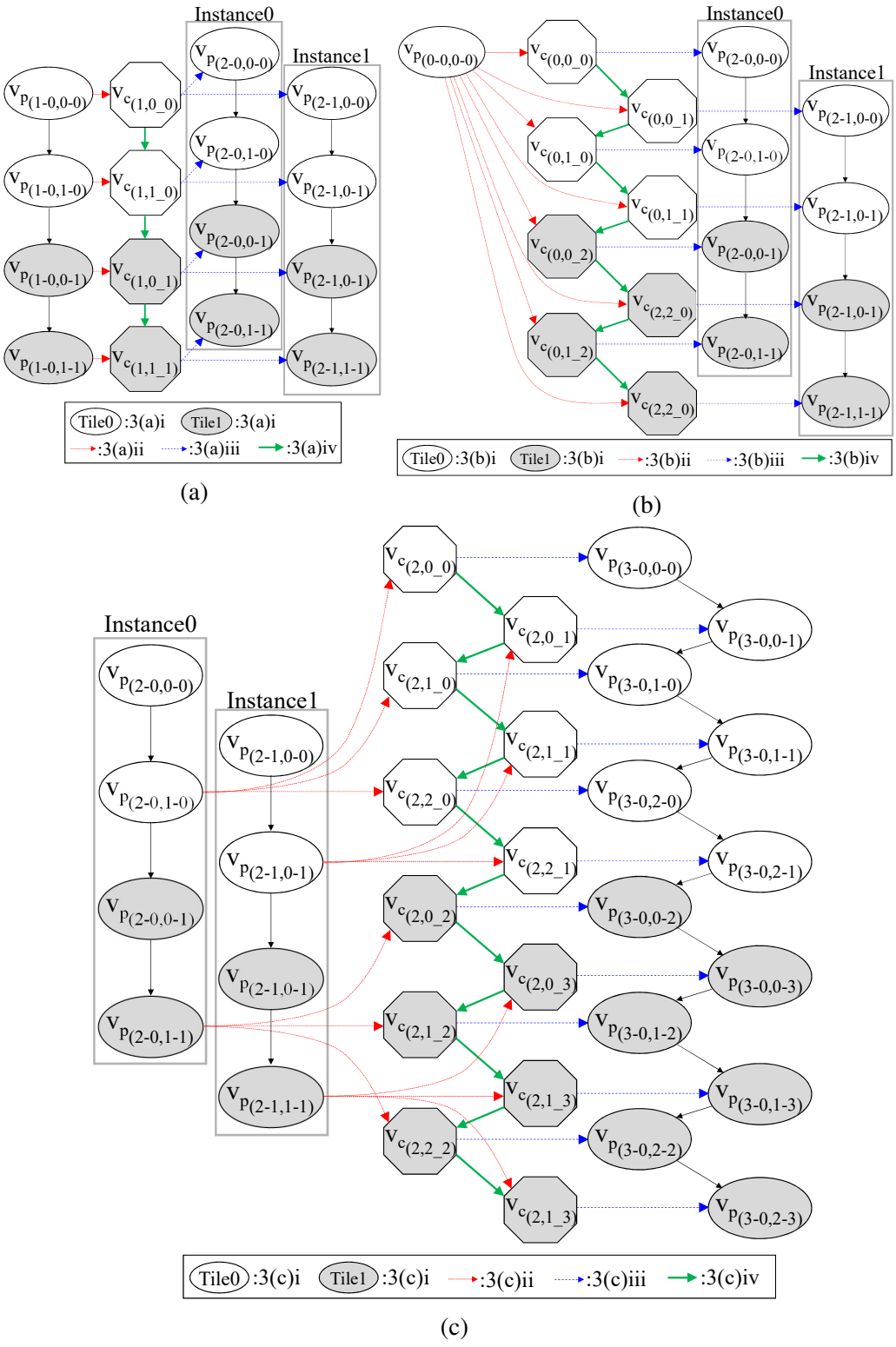


Figure 5.7: An example of AL-EDG construction in step 3 of The SLM in Fig. 5.5(a), which is implemented with IFM-major scheme.

accelerator IP, and the modeling scheme is IFM-major scheme, do as follows;

i. For each vertex $v_{c_{j,l\_0}}$, duplicate it as vertices $v_{c_{j,l\_s}}$, where $s = 0, ..., R-1$, and add them to V_{al}. The number of data to transfer of each $v_{c_{j,l\_s}}$ is equal to the number of data of $v_{c_{j,l\_0}}$.
ii. For each $(v_{p_{(u\_0,k\_r)}}, v_{c_{j,l\_0}})$ edge, alter it to be $(v_{p_{(u\_0,k\_r)}}, v_{c_{j,l\_s}})$, where $r = s$, to represent the E/S dependencies.
iii. For each $(v_{c_{j,l\_0}}, v_{p_{(x\_q,k\_r)}})$, alter it to be $(v_{c_{j,l\_s}}, v_{p_{(x\_q,k\_r)}})$, where $r = s$, to represent the R/E dependencies.
iv. Make edges $(v_{c_{j,l\_s}}, v_{c_{j,l+1\_s}})$ and $(v_{c_{j,C_i-1\_s}}, v_{c_{j,0\_s+1}})$ to represent execution orders between data transfers.

Figure 5.7 (a) shows these steps for vertices of c_1. Each vertex of c_1 is duplicated into two vertices, for example, the $v_{c_{(1,0\_0)}}$ is duplicated as $v_{c_{(1,0\_0)}}$ and $v_{c_{(1,0\_1)}}$. Each vertex of c_1 is responsible for transferring weights for each data tile.

(b) If p_u is a typical process and p_x is a process of convolutional layer mapped onto Q instances of functional block of CNN accelerator IP, do as follows;

i. For each vertex $v_{c_{j,l\_0}}$, make vertices $v_{c_{j,l\_s}}$, where $s = 0, ..., (Q-1)\times(R-1)$, and add them to V_{al}. The number of data to transfer of each $v_{c_{j,l\_s}}$ is $\frac{1}{Q\times R}$ times of the number of data of $v_{c_{j,l\_0}}$. It is assumed that each tile is considerably large compared to the overlapping data of sliding window, so the overlapping data is ignored in the estimation.
ii. For each $(v_{p_{(u\_0,k\_0)}}, v_{c_{j,l\_0}})$ edge, alter it to be $(v_{p_{(u\_0,k\_0)}}, v_{c_{j,l\_s}})$.
iii. For each $(v_{c_{j,l\_0}}, v_{p_{(x\_q,k\_r)}})$ edge, alter it to be $(v_{c_{j,l\_s}}, v_{p_{(x\_q,k\_r)}})$, where $s = (r \times Q) + q$.
iv. Make edges that represent execution orders between data transfers, which differs by modeling scheme.
For IFM-major scheme, make edges $(v_{c_{(i,k\_r)}}, v_{c_{(i,k\_r+1)}})$ to represent the order of channel vertices of the same tile and input channel to different instances of functional block of CNN accelerator IP, make edges $(v_{c_{(i,k\_r)}}, v_{c_{(i,k+1\_r-Q+1)}})$ to represent the order of channel vertices of the same tile but different input channels, and make edges $(v_{c_{(i,C_o-1\_r)}}, v_{c_{(i,0\_r+1)}})$ to represent the order of channel vertices from different tiles.
For OFM-major scheme, make edges $(v_{c_{j,l\_s}}, v_{c_{j,l\_s+1}})$ and $(v_{c_{j,l\_((Q-1)\times(R-1))}}, v_{c_{j,l+1\_0}})$.

Figure 5.7 (b) shows these steps for vertices of c_0. Each vertex of c_0 is partitioned into four vertices, for example, the $v_{c_{(0,0\_0)}}$ is partitioned into $v_{c_{(0,0\_0)}}$, $v_{c_{(0,0\_1)}}$, $v_{c_{(0,0\_2)}}$ and $v_{c_{(0,0\_3)}}$. Each vertex of c_0 is responsible for transferring weights for each data tile.

(c) If p_u is a process of convolutional layer mapped onto the Q instances of functional block of CNN accelerator IP or activation layer or pooling layer, and p_x is a typical process or activation layer or pooling layer, do as follows;

i. For each vertex $v_{c_{j,l\_0}}$, make vertices $v_{c_{j,l\_s}}$, where $s = 0, ..., (Q-1) \times (R-1)$, and add them to V_{al}. The number of data to transfer of each $v_{c_{j,l\_s}}$ is $\frac{1}{Q \times R}$ times of the number of data of $v_{c_{j,l\_0}}$.

ii. If p_u is a process of convolutional layer, for each $(v_{p_{(u\_q,k\_r)}}, v_{c_{j,l\_0}})$ edge, alter it to be $(v_{p_{(u\_q,k\_r)}}, v_{c_{j,l\_s}})$, where $s = (r \times Q) + q$. If p_u is a process of activation or pooling layer, for each $(v_{p_{(u\_0,k\_r)}}, v_{c_{j,l\_0}})$ edge, alter it to be $(v_{p_{(u\_0,k\_r)}}, v_{c_{j,l\_s}})$, where $r = s$.

iii. If p_x is a typical process, for each $(v_{c_{j,l\_0}}, v_{p_{(x\_0,k\_0)}})$, add edges $(v_{c_{j,l\_s}}, v_{p_{(x\_0,k\_0)}})$. If p_x is a process of activation layer or pooling layer, for each $(v_{c_{j,l\_0}}, v_{p_{(x\_0,k\_r)}})$, alter it to be $(v_{c_{j,l\_s}}, v_{p_{(x\_0,k\_r)}})$, where $s = r$.

iv. Make edges that represent execution orders between data transfers, which depends on modeling scheme.
For IFM-major scheme, make edges $(v_{c_{(i,k\_r)}}, v_{c_{(i,k\_r+1)}})$ to represent the order of channel vertices of the same tile and input channel to different instances of functional block of CNN accelerator IP, make edges $(v_{c_{(i,k\_r)}}, v_{c_{(i,k+1\_r-Q+1)}})$ to represent the order of channel vertices of the same tile but different input channels, and make edges $(v_{c_{(i,C_o-1\_r)}}, v_{c_{(i,0\_r+1)}})$ to represent the order of channel vertices from different tiles.
For OFM-major scheme, make edges $(v_{c_{j,l\_s}}, v_{c_{j,l\_s+1}})$ and $(v_{c_{j,l\_((Q-1)\times(R-1))}}, v_{c_{j,l+1\_0}})$.

Figure 5.7 (c) shows these steps for vertices of c_2. Each vertex of c_2 is partitioned into four vertices, for example, the $v_{c_{(2,0\_0)}}$ is partitioned into $v_{c_{(2,0\_0)}}$, $v_{c_{(2,0\_1)}}$, $v_{c_{(2,0\_2)}}$ and $v_{c_{(2,0\_3)}}$.

4. Alter V_{al} and E_{al} so that the AL-EDG also includes the dependencies of data transfers raised by communication paths as explained in step 2 of section 3.4.3. On the other hand, in step 2(a)ii and 2(b)ii, if there are multiple $(v_{c_{(j,l_s)}}, v_{p_{(i_q,k_r)}})$ edges, one $(v_{c_{(j,l_s)}}, v_{c'''_{(j,l_s)}})$ and multiple $(v_{c'''_{(j,l_s)}}, v_{p_{(i_q,k_r)}})$ edges will be added.

5. Divide the vertices into groups. The vertices of functional blocks are divided by functional blocks and instances. The group $V_{fb_{i\_q}}$ represents the vertex group that are undertaken by instance q of functional block i. The vertices of other components are divided as explained in step 3 of section 3.4.3.

AL-EDG Analysis

The AL-EDG is analyzed to estimate execution time of the system given as the ALM. The analysis proceeds with similar procedures as explained in section 3.4.4. Two additional points are applied in order to analyze the behavior of intra-layer parallelism due to data tiling.

First, to analyze the intra-layer parallelism undertaken by different instances of the same functional block, the analysis finds the executable vertices and determines a vertex whose execution will be analyzed on each instance of each functional block. For each instance, a vertex in $V_{fb_{i\_q}}$ is classified as executable when it has no edge from other vertices. It is added to executable vertex set, $V_{exe_{fb_{i\_q}}}$, and then, the analysis chooses a vertex in $V_{exe_{fb_{i\_q}}}$ to analyze its execution in each iteration. Consequently, vertices of the same process that are undertaken by multiple instances can be analyzed at the same time, which corresponds to the analysis of intra-layer parallelism.

Second, total data processing time of each process vertex is calculated in accordance to data tiling. It is calculated by the following equation:

$$t_p = \frac{\alpha \times e_{(p_j, fb_i)}}{f_{fb_i}}, \tag{5.1}$$

where α is the processing time factor and depends on the IP, $e_{(p_j,fb_i)}$ is the execution cycle of process p_j on functional block fb_i, and f_{fb_i} is the execution frequency of fb_i. For typical processes, α is 1. For processes involved in convolutional layers, α is set in accordance with IP.

5.5.2 Hardware Area Estimation

Area of an architecture, $A(M_{al})$, is the summation of areas of all the hardware components within an ALM, M_{al}. It is estimated as shown in Eq. (5.2).

$$\begin{aligned} A(M_{al}) = &\sum_{fb_i \in F} A(fb_i) + \sum_{pt_i \in PT} A(pt_i) + \sum_{d_i \in D} A(d_i) + \sum_{m_i \in M} A(m_i) + \\ &A(BM) + \sum_{b_i \in B} A(b_i) + \sum_{bb_i \in BB} A(bb_i), \end{aligned} \tag{5.2}$$

where $A(fb_i)$, $A(pt_i)$, $A(d_i)$, $A(m_i)$, $A(BM)$, $A(b_i)$, and $A(bb_i)$ represent the area function of functional blocks, ports, DMACs, memories, a bus matrix, shared buses, and bus bridges, respectively.

Area of Functional Block

Assuming that functional block i that is an implementation of IP j, the area of functional block i, $A(fb_i)$, is estimated as follows:

$$A(fb_i) = g_{ip_j} \times g_{nand}, \tag{5.3}$$

where g_{ip_j} is the gate count of IP j that is registered in the IP database and g_{nand} is the area of one NAND gate of the target process technology. For CNN accelerator IP, the area is estimated as follows:

$$A(fb_i) = (g_{ip_j} + g_{pe} \times N_{pe_i}) \times g_{nand}, \tag{5.4}$$

where g_{ip_j} is the gate count of IP j that is registered in the IP database, which excludes the gate count of PE, and g_{pe} is the gate count of PE of IP j.

Area of Port

The area of each port, $A(pt_i)$, is the summation of the area of protocol interface of port, $A_{inf}(pt_i)$, and the buffer area within port, $A_{buf}(pt_i)$, as follows:

$$A(pt_i) = A_{inf}(pt_i) + A_{buf}(pt_i), \tag{5.5}$$

The area of protocol interface is the product of the IP's protocol interface gate count of the connecting functional block and the area of one NAND gate. The protocol interface gate count is registered in the IP database.

The buffer area is the summation of the area of receive buffers and transmit buffers in each port. The area of receive buffer is the product of SRAM area that is large enough to store the biggest amount of data for a process execution and the number of receive buffer. Likewise, the area of transmit buffer is the product of SRAM area that is large enough to store the biggest amount of generated data and the number of transmit buffer. The SRAM area is obtained from the SRAM library, which is prepared in advance. However, it is accounted that the port that is connected to functional block of memory IP does not contain buffer.

Area of DMAC

Figure 5.8 (a) shows the model of a DMAC, which is composed of a master interface, an internal control logic circuit and a data buffer. It is assumed that the control program is included as part of the control circuit and the protocol of DMAC's master interface is the same protocol as of the connecting bus. The area of DMAC, d_i, is estimated with the following equation:

$$A(d_i) = A_{inf}(d_i) + A_{logic}(d_i) + A_{buf}(d_i), \tag{5.6}$$

where $A_{inf}(d_i)$, $A_{logic}(d_i)$ and $A_{buf}(d_i)$ are the area of the master interface, the internal control logic circuit and the data buffer, respectively. Here, the buffer area is assumed to be significantly larger than the others. Therefore, this research considers the area of a DMAC the area of buffer, which is derived from the above mentioned SRAM library.

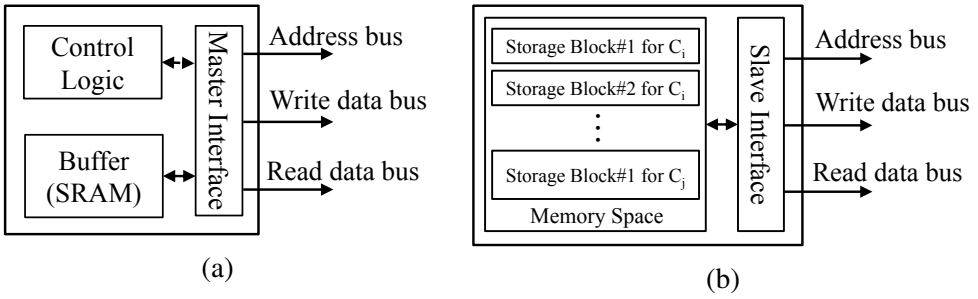


Figure 5.8: Architecture model in this research: (a) model of a DMAC; (b) model of a memory.

Area of Memory

The model of memory associated with data transfer is shown in Fig. 5.8 (b). It consists of a slave interface and a memory space, which is divided into storage blocks. It is assumed that a storage block is specific to a channel and the slave interface of a memory is the same protocol as of the connecting bus. The area of memory, m_i, is estimated with the following equation:

$$A(m_i) = A_{inf}(m_i) + \sum_{c_j \in C_{m_i}} A_{sb}(c_j) \times n_{c_j}, \tag{5.7}$$

where $A_{inf}(m_i)$ is the slave interface area, C_{m_i} refers to the set of channels that are conducted via a memory, $A_{sb}(c_j)$ is the area of storage block for c_j and n_{c_j} is the number of storage blocks for c_j. Similar to DMAC, the size of storage blocks is considered significantly larger than the interface area. Therefore, this research estimates the area of memory as the area of storage blocks, which is derived from the SRAM library mentioned earlier.

Area of Bus Matrix

Figure 3.5 illustrates the multi-layer matrix model. It consists of a decoder for each master layer, an arbiter and a bus matrix fabric, which includes input stages and output stages that select the signals for each master and slave layer.

The bus matrix area is estimated according to the model by summing the area of all the components as follows:

$$A(BM) = \sum A_{dec} + \sum A_{arb} + A_{fabric}, \tag{5.8}$$

where $\sum A_{dec}$ is the summation of the area of all the decoders, $\sum A_{arb}$ is the summation of the area of all the arbiters, and A_{fabric} is the area of bus matrix fabric. The areas of input stages and output stages are included in the area of bus matrix fabric.

The areas of decoder, arbiter, and bus matrix fabric are derived from the linear equation of the areas extracted from the logic synthesis result of each component. The area of decoder at each master layer varies by the number of bus matrix's slave layers and the number of slaves in each master layer. The area of arbiter at each slave layer varies by the number of master layers that need to access each slave layer. Finally, the area of bus matrix fabric varies by the number of buses on the bus matrix and the bus width.

Area of Shared Bus

The area of shared bus, $A(b_i)$, is basically the summation of the wire area, $A_{wire}(b_i)$, and the bus logic area, $A_{logic}(b_i)$, as shown in Eq. (5.9).

$$A(b_i) = A_{wire}(b_i) + A_{logic}(b_i), \tag{5.9}$$

It differs by bus protocol according to protocol models in Fig. 3.1 and Fig. 3.3(a). The area of shared bus containing bus masters, such as AHB, comes from wire area and bus logic area. The area of peripheral bus containing only bus slaves, such as APB, includes only the wire area.

The wire area of each bus is calculated from wire length, l_{b_i}, wire pitch, W_{pitch}, and bus width, w_{b_i}. Wire length is derived based on the method of [131] and [164] as shown in Eq. (5.10).

$$l_{b_i} = (0.9 + 0.55\sqrt{n_{pin_i}})\sqrt{A_{b_i}}, \tag{5.10}$$

where n_{pin_i} is the number of pins to which the bus connects and A_{b_i} represents the summation of area of all the modules connecting to the target bus as follows.

$$\begin{aligned} A_{b_i} = &\sum_{fb_j \in F_{b_i}} A(fb_j) + \sum_{pt_j \in PT_{b_i}} A(pt_j) + \\ &\sum_{d_j \in D_{b_i}} A(d_j) + \sum_{m_j \in M_{b_i}} A(m_j) + \sum_{bb_j \in BB_{b_i}} A(bb_j), \end{aligned} \tag{5.11}$$

where F_{b_i}, PT_{b_i}, D_{b_i}, M_{b_i} and BB_{b_i} are the function blocks, ports, DMACs, memories and bus bridges that are connected to bus b_i. Then, wire area is estimated as follows:

$$A_{wire}(b_i) = l_{b_i} \times W_{pitch} \times (w_{a,b_i} + 2 \times w_{b_i}) \times (1 - R_{over}), \tag{5.12}$$

where W_{pitch} and R_{over} represent wiring pitch and rate of over-the-cell wires, both of which are specified by the designer.

The bus logic area of an AHB is the summation of the area of three multiplexers, one arbiter and one decoder. The area of a multiplexer comes from the multiplexer library prepared in advance from the logic synthesis result. The areas of arbiter and decoder are obtained in the same way as the areas of arbiter and decoder of bus matrix.

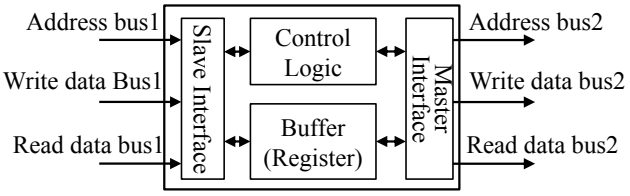


Figure 5.9: Architecture model of a bus bridge.

Area of Bus Bridge

The model of bus bridge is shown in Fig. 5.9. Its area, $A(bb_i)$, is estimated as the summation of area of its components as follows:

$$A(bb_i) = A_{infm}(bb_i) + A_{infs}(bb_i) + A_{logic}(bb_i) + A_{buf}(bb_i), \tag{5.13}$$

where $A_{infm}(bb_i)$, $A_{infs}(bb_i)$, $A_{logic}(bb_i)$ and $A_{buf}(bb_i)$ are the area of master interface, slave interface, control logic circuit and buffer. The interface area and the control logic circuit area are obtained from the bus bridge library, which is created by synthesizing the interface area and the control logic circuit area. The area of buffer is derived from the register area which equals to the widest bus width connected to the bus bridge. The register area is calculated from the linear equation extracted from the synthesized register area.

5.6 Architecture Exploration of SoCs for CNN-based AI Platform

Exploring SoCs architecture for CNN-based AI platform concerns four important points below.

1. The exploration that finds the architecture that leverage the intra-layer parallelism even though the system is modeled in coarse-grained granularity
2. Ports, buses, buses on bus matrix, bus bridges, DMACs and memories composing a communication path from the source port to the destination port of each channel
3. The architecture's port clustering, and the placement of DMACs and memories for the cluster organization of the bus matrix topology
4. The selection of bus protocols and their parameters for both shared buses and the multi-layer bus

Point 1 considers leveraging intra-layer parallelism capability through the parameterization of functional blocks. This saves human and time resource in implementing a medium- or fine-grained SLM and exploring the design space from a

cumbersome SLM. Point 2 and 3 together determine the communication path of each channel, which is crucial for the ALM organization of the target architecture. Point 4 focuses on exploring bus protocols and the parameters of bus architecture based on the selected bus protocols because bus protocols determine not only the valid parameter values of a bus, e.g. bus width, but also its characteristics.

This research explores the architecture design space in two folds. First, it determines IPs and bus architecture on the SoCs by mapping processes and channels of the SLM to IPs and bus architecture. It determines the functional blocks and communication path, such as ports, DMACs and memories, and then, organizes ports, DMACs and memories in each cluster of bus matrix via channel mapping and the automatic placement of DMACs and memories. Second, it maps parameters of SoC architecture to the available candidates. The parameters for parallelizing the MACCs of intra-layer parallelism are determined, and the parameters for shared buses and multi-layer bus are selected based on bus protocols. Both are done by the traversal through a parameter set search tree as described in section 5.6.2.

5.6.1 SoC Architecture Parameterization

In addition to selecting the IPs and bus architecture on the SoCs by mapping SLM to the architecture, the proposed method also explores parameters that describe both architectural and operational properties of the components. The components on the SoC for CNN-based AI platform are parameterized for mapping as follows:

1. **Parameters of functional blocks**
   - The number of instances of IP j that is implemented as functional block i (N_{fb_i})
   - The number of PEs within an instance of functional block i (N_{pe_i})
   - Execution frequency of functional block (f_{fb_i})
2. **Parameters of shared buses**
   - Bus protocol (pr_{b_i})
   - Data bus width (w_{b_i})
   - Address bus width (w_{a,b_i})
   - Execution frequency of shared bus (f_{b_i})
3. **Parameters of multi-layer bus**
   - Multi-layer bus protocol (pr_{bm})
   - Data bus width of bus matrix (w_{bm})
   - Address bus width of bus matrix ($w_{a,bm}$)
   - Execution frequency of bus matrix (f_{bm})

4. **Parameters of buffers in ports**
   - The number of receive buffers (n_q)
   - The number of transmit buffers (n_r)
5. **Parameters of memory that is associated with data transfer**
   - The number of storage blocks (n_{c_i})

5.6.2 Parameter Set Search Tree

The proposed architecture exploration method explores the design space by traversing through a parameter set search tree with depth-first search. The parameter set search tree is composed of multiple parameter mapping trees by concatenating the leaf of the preceding tree to the root of the following tree.

IPs and Bus Architecture Selection Trees

1. **Process mapping**
   The process mapping determines a functional block instance of an IP to undertake the execution of each process. The functional blocks are selected from the IPs in the IP database that can execute each process in the same fashioned as in the method of Ueda *et al.* as shown in Fig. 2.4. The depth for this search tree is $|P|$.
2. **Channel mapping**
   The channel mapping determines the communication path for interprocess communication. Sources and destinations of channels are mapped on to ports, clusters and buses. The depth of the three channel mapping trees is $2 \times |C|$.
   (a) **Channel-to-port mapping**
   First, the source of a channel is mapped on to a port for data transmission from a functional block that generates the data. Then, the destination of the channel is mapped on to a port for data reception from the functional block. The mapped port is selected from a master port or a slave port of the functional block according to the number of ports of the IP registered in the IP database. A channel-to-port mapping tree is shown in Fig. 5.10. It is assumed that p_1 is mapped on to $fb_1(I_2, 1)$, p_2 is mapped on to $fb_2(I_1, 1)$, and both IP I_1 and I_2 have a master port and a slave port. In Fig. 5.10, the source of channel c_1, remarked by c_{1s}, is mapped on to either a master port or a slave port of fb_1. Then, the destination of channel c_1, remarked by c_{1d}, is mapped on to either a master port or a slave port of fb_2. pt_{iM} and pt_{iS} represent the i^{th} master port and the i^{th} slave port, respectively. The

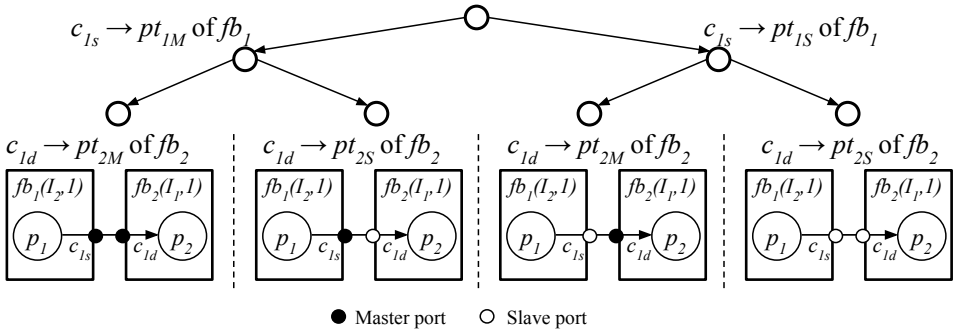


Figure 5.10: An example of channel-to-port mapping tree.

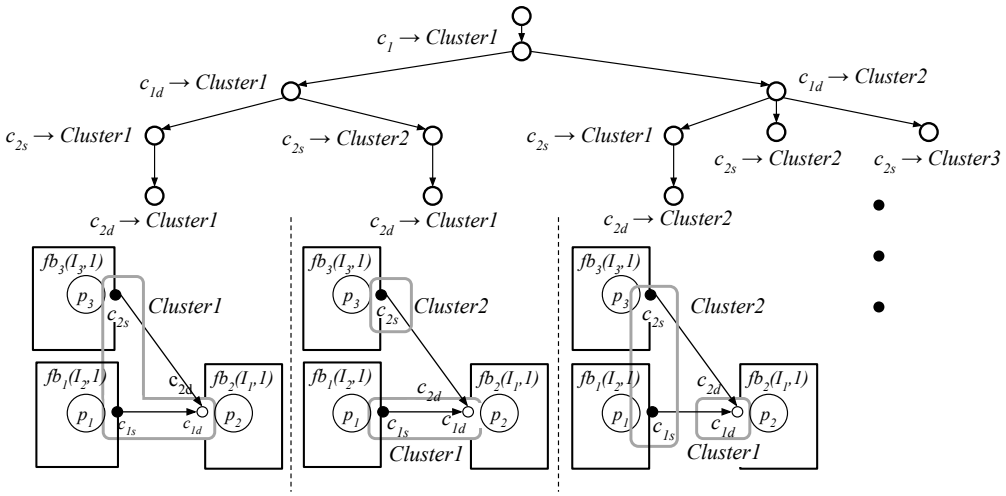


Figure 5.11: An example of channel-to-cluster mapping tree.

black dot represents a master port and the white dot represents a slave port. The maximum number of ports is $2 \times |C|$ and is reached when the source and destination of each channel are mapped to its own port.

(b) **Channel-to-cluster mapping**

The channel-to-cluster maps the source and the destination of each channel into clusters. It represents the selection of a port's cluster. Each cluster is a group of ports, DMACs and memories connected to either a master layer or a slave layer of the bus matrix. As a result of channel-to-cluster mapping, a cluster with one or more master ports is considered a master cluster and connected with the bus matrix as a master layer, while a cluster that has no master port is recognized as a slave cluster and connected with the bus matrix as a slave layer. Figure 5.11 shows an example of the channel-to-cluster mapping of a system with two channels, which are mapped on to three ports. First, the source of channel c_1, c_{1s}, is mapped on to $Cluster1$. Then, the desti-

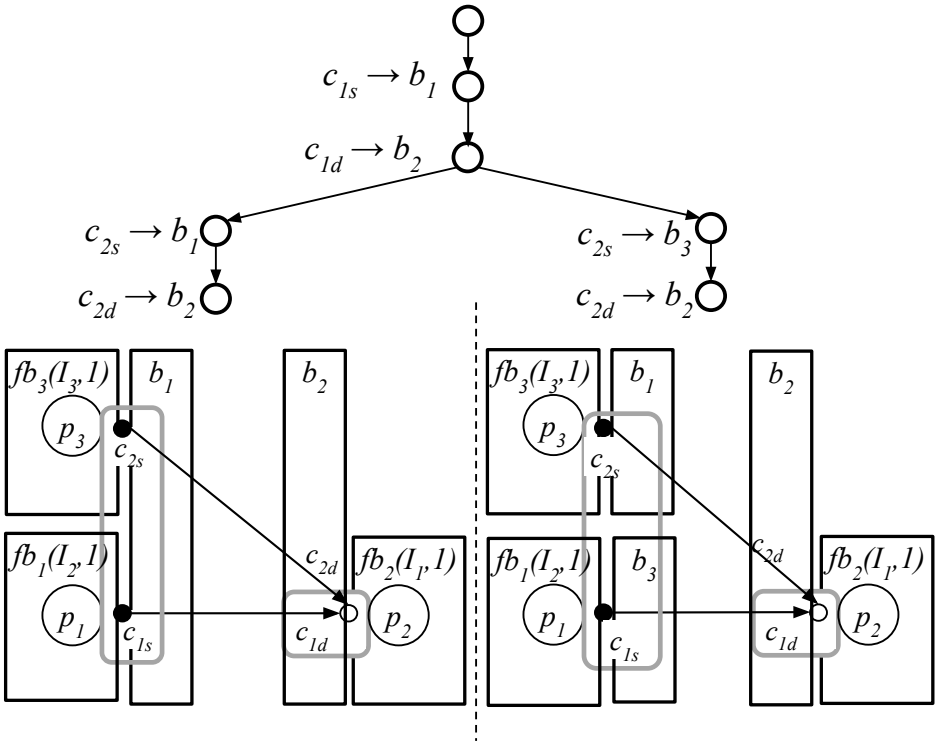


Figure 5.12: An example of channel-to-bus mapping tree, where a channel is mapped on to a bus in the same cluster or a new bus.

nation of channel c_1, c_{1d}, is mapped on to the existing cluster *Cluster*1 or a new cluster *Cluster*2. Likewise, the next source of channel c_2, c_{2s}, can be in the existing clusters or a new cluster. Finally, since the destination of channel c_2, c_{2d}, is mapped on to the same port as c_{1d}, it is mapped on to the same cluster as c_{1d}.

(c) **Channel-to-bus mapping**
The source and destination of each channel are mapped on to buses, representing those that are connected to each mapped port. A channel can be mapped only on to a bus that is in the same cluster or a new bus. The number of buses in a cluster that contains only slave ports, i.e. a slave cluster, is limited to one because only one master accesses a slave layer at a time. Consequently, more buses would not relieve any bus contention. An example of the channel-to-bus mapping proposed in this research is illustrated in Fig. 5.12. In the figure, the source of channel c_1, c_{1s}, is first mapped on to a new bus b_1. Then, since the destination of channel c_1, c_{1d}, is in a different cluster from c_{1s}, it cannot be mapped on to b_1. Therefore, it is mapped on to a new bus b_2. The next source of channel c_2, c_{2s}, is in the same cluster as c_{1s} is, so it can be mapped on to either bus b_1 or a new bus b_3. Finally, because

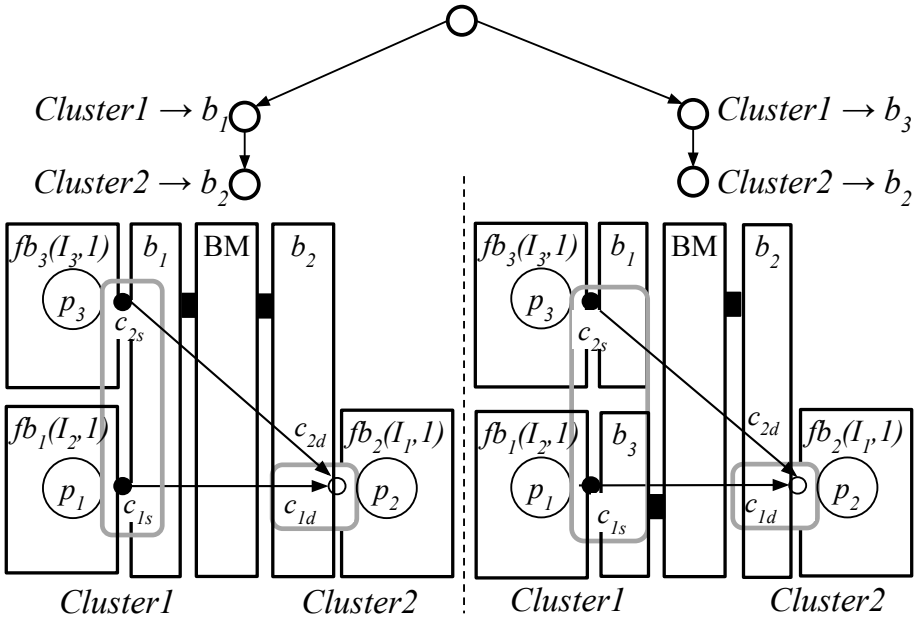


Figure 5.13: An example of cluster-to-bus matrix mapping tree.

the destination of channel c_2, c_{2d}, is mapped on to the same port as c_{1d} is, it is also mapped on to bus b_2.

3. **Cluster-to-bus matrix mapping**
   In the multi-layer architecture, one bus in each cluster is connected to the bus matrix. The cluster-to-bus matrix mapping selects a shared bus in each cluster to connect to a port of bus matrix. Figure 5.13 illustrates the cluster-to-bus matrix mapping tree. *Cluster*1, a master cluster, consists of two buses while *Cluster*2, a slave cluster, contains one bus. First, the mapping proceeds by selecting either bus b_1 or bus b_3 of *Cluster*1 to connect with the bus matrix, *BM*. Then, bus b_2 is chosen to connect with the bus matrix because it is the only bus in *Cluster*2. The depth of this search tree equals to the number of clusters.

4. **DMAC and memory placement**
   Since most bus specifications provide a master-slave communication scheme, the communication of channels that both source and destination are mapped on to two master ports or two slave ports cannot take place. A memory or a DMA controller must be inserted as part of the communication path. In a hierarchical shared bus architecture, either a DMAC or a memory would suffice the master-slave regulation. However, a multi-layer bus architecture also regulates that the communication must occur within the same master layer or between a master and a slave layer. For example, when a master in a master cluster communicates with a slave in another master cluster, an

Table 5.1: DMAC and memory placement to suffice master-slave communication scheme of a multi-layer bus

Port and cluster type		Same cluster?	DMAC or memory insertion
pt_s	pt_d		
M (MC)	M (MC)	Yes	A memory on a bus in pt_s's MC
		No	One memory in one of the SCs
M (MC)	S (MC)	Yes	No insertion
M (MC)	S (SC)	No	No insertion
S (MC)	M (MC)	Yes	No insertion
S (SC)	M (MC)	No	No insertion
S (MC)	S (MC)	Yes	A DMAC on a bus in pt_s's MC
S (MC)	S (SC)	No	A DMAC on a bus in pt_s's MC
S (SC)	S (MC)	No	A DMAC on a bus in pt_d's MC
S (SC)	S (SC)	Yes	A DMAC on a bus in one of the MCs
		No	A DMAC on a bus in one of the MCs

memory is needed in one of the slave clusters and a DMAC is needed for data transfer between the memory and the target slave port.

The DMAC and memory placement explores the placement of DMACs and memories that are inserted as a communication path of the channels according to the result of channel mapping. Since insertion adds latency to the communication path, the DMAC and memory placement inserts no more than one component for each channel as shown in Table 5.1. It disregards the result of channel mapping that requires two or more additional components for one of the channels. In the table, pt_s and pt_d refer to the port of the source and the destination of a channel, respectively. The symbols M, S, MC, and SC stand for a master port, a slave port, a master cluster, and a slave cluster, respectively. For example, the M (MC) in the column pt_s means that pt_s is a master port in a master cluster.

The clusters resulting from the DMAC and memory placement are interpreted as follows.

- A cluster with a single master port or a single DMAC is a **single master cluster**.
- A cluster with more than one master ports or DMAC but no slave port or memory is a **multiple master cluster**.
- A cluster with at least one master port or DMAC and at least one slave port or memory is a **shared bus or hierarchical shared bus subsystem**.

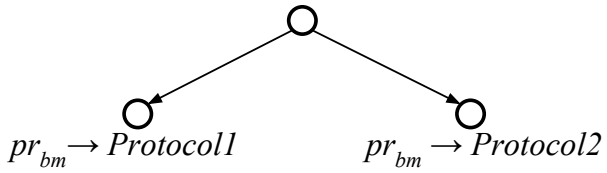


Figure 5.14: An example of bus matrix protocol mapping tree.

- A cluster with a single slave port or a single memory is a **single slave cluster**.
- A cluster with more than one slave ports or memories but no master port or DMAC is a **multiple slave cluster**.

Upon the completion of the placement, the necessary bus bridges and buses on the bus matrix fabric are automatically placed into the architecture. A bus bridge is required when a master (master port or DMAC) and a slave (slave port or memory) reside on different buses or the bus of a master or a slave is not connected to the bus matrix. The necessary buses on the bus matrix fabric are determined to form a maximally connected bus matrix of a partial bus matrix.

The depth of this tree varies according to the result of channel mapping. The maximum depth is $3 \times |C|$, which means that every channel is mapped to slave ports in different master clusters that need two DMAC and one memory insertion.

5. **Bus matrix protocol mapping**
   The bus matrix protocol mapping determines a multi-layer bus protocol for a bus matrix, pr_{bm}. Figure 5.14 illustrates an example when there are two protocols for bus matrix. This tree is organized with nodes that map bus matrix to each protocol. In this research, the protocol for a multi-layer bus and its limitation is as follows.

   - **Multi-layer AHB**
     The bus matrix of multi-layer AHB can connect with at most 16 master clusters and 16 slave clusters.

   If the architecture includes only hierarchical shared buses, this tree is skipped. For multi-layer bus-based architecture, the depth of this search tree is $|BM|$.

6. **Shared bus protocol mapping**
   The shared bus protocol mapping determines a protocol for each shared bus in the constructing architecture. The bus connection regulations are different depending on bus protocol. In order to map a bus on to a bus protocol, all of the following conditions must be satisfied.

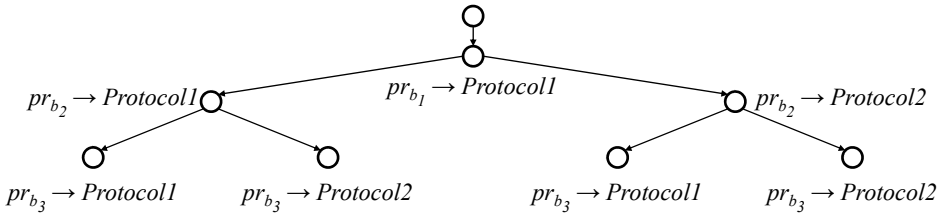


Figure 5.15: An example of bus protocol mapping tree.

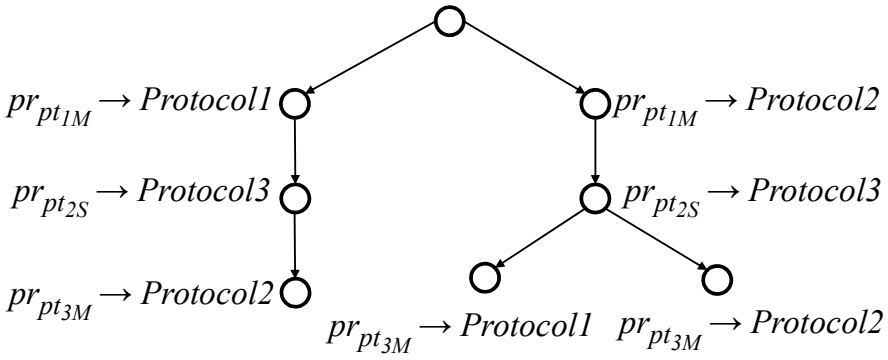


Figure 5.16: An example of port protocol mapping tree.

- **AHB**
  The bus must connect to no more than 16 masters.
- **APB**
  The bus has a bus bridge as its only master. Additionally, it must not connect to the bus matrix unless it is a bus in the slave cluster.

Figure 5.15 illustrates an example when there are two bus protocols: *Protocol*1 as *AHB* and *Protocol*2 as *APB*, respectively. Let us assume that b_1 resides in master clusters and is connected to master ports, while b_2 and b_3 are in a slave cluster. b_1 can only be mapped to *Protocol*1 because it does not satisfy the condition of *Protocol*2, while b_2 and b_3 can be mapped to both protocols. The depth of this tree is $|B|$.

7. **Port protocol mapping**
   The port protocol mapping selects a protocol for a port of functional block from one of the port protocols that are registered for each IP in the IP database. Figure 5.16 illustrates an example of port protocol mapping. Assuming that an IP of functional block fb_1 that is connected to pt_1M has one master port of *Protocol*1, two master port of *Protocol*2 and one slave port of *Protocol*3. The protocol of pt_1M, pr_{pt_1M}, is selected to be either *Protocol*1 or *Protocol*2, which are protocols for master port. Then, an IP of functional block fb_2 that is connected to pt_2S has one master port of

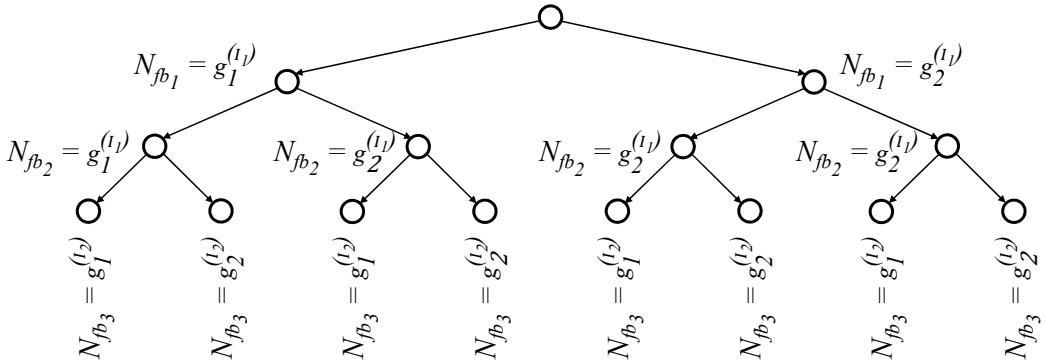


Figure 5.17: An example of the number of functional block instance mapping tree.

*Protocol*2 and one slave port of *Protocol*3. The protocol of pt_2S, pr_{pt_2S}, is mapped to *Protocol*3, since it is the only slave port's protocol of fb_2. Next, assume that pt_3M is connected to fb_1. Its protocol, pr_{pt_3M} can be mapped to only *Protocol*2 if pr_{pt_1M} is *Protocol*1 because there is no master port of *Protocol*1 left, and pr_{pt_3M} can be mapped to either *Protocol*1 or *Protocol*2 if pr_{pt_1M} is *Protocol*2. In this research, the port protocol is either an AHB master port, an AHB slave port or an APB slave port. The depth of this search tree is $|PT|$

Parameter Mapping Trees

1. **Number of functional block instance mapping**
   This mapping selects the number of functional block instances, N_{fb_i}, for each functional block, fb_i. The number of instance candidates of each functional block is raised in the IP database. Figure 5.17 shows the example of the mapping, where $g_j^{(I_i)}$ represents the j^{th} candidate of the number of instances of IP I_i. It is assumed that (1) there are two candidates for each IP; (2) fb_1 and fb_2 is implemented from IP I_1; (3) fb_3 is implemented from IP I_2. Generally, the depth of this tree is $|F|$. However, since this thesis applies N_{fb_i} to only the functional blocks of CNN accelerators, the depth equals to the number of CNN accelerators.

2. **Number of PE mapping**
   This mapping selects the number of PEs, N_{pe_i}, for each functional block, fb_i. The number of PE candidates of each functional block is raised in the IP database according to the IP that the functional block implemented. Figure 5.18 shows an example of the mapping, where $h_j^{(I_i)}$ represents the j^{th} candidate of the number of PEs of IP I_k. Assume that there are three functional blocks. The depth of this tree is the same as the depth of the number of functional block instance mapping tree.

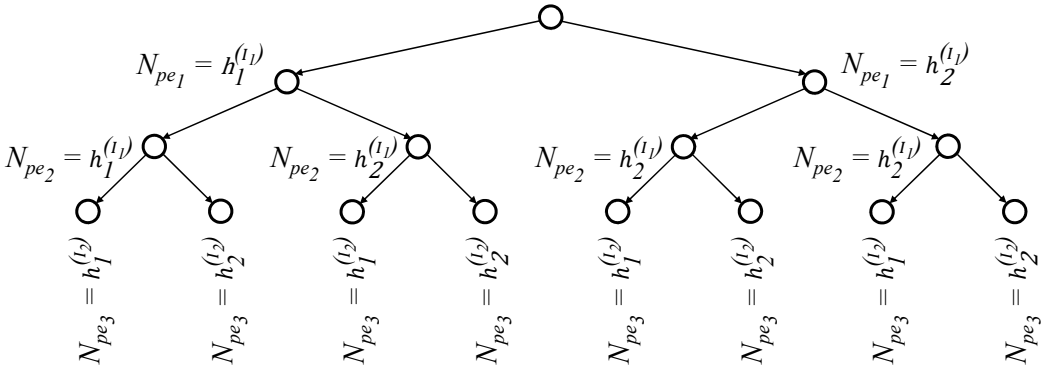


Figure 5.18: An example of the number of PE mapping tree.

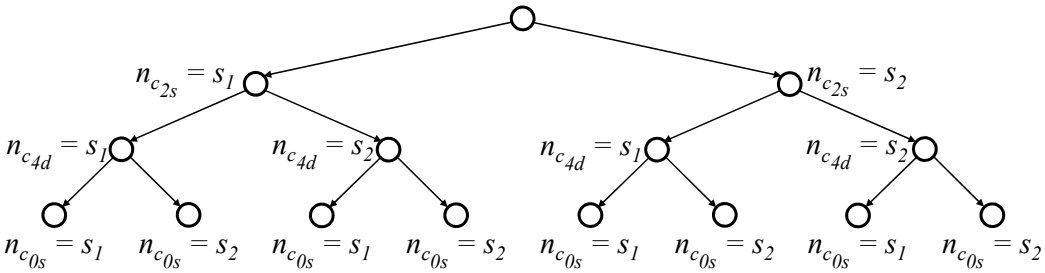


Figure 5.19: An example of the number of memory's storage block mapping tree.

3. **Number of memory's storage block mapping**
   For each memory inserted during the DMAC and memory placement, the number of its storage blocks for each channel is determined by the number of memory's storage block mapping. The candidates for the number of memory's storage block is raised by the designer. The depth of this tree is the number of channels that require memory for data transfer as inserted in step DMAC and memory placement. The maximum depth of this tree is $|C|$ when all channels require a memory. Figure 5.19 illustrates the number of memory's storage block mapping tree. Assume that there are three candidates of the number of memory's storage block is s_i and there are two memories in the system, where memory m_1 undertakes the communication of channel c_{2s} and c_{4d} ($C_{m_1} = \{c_{2s}, c_{4d}\}$) and memory m_2 undertakes channel c_{0s} ($C_{m_2} = \{c_{0s}\}$). In the figure, $n_{c_{2s}}$, $n_{c_{4d}}$, and $n_{c_{0s}}$ are the number of storage block of c_{2s} of m_1, c_{4d} of m_1, and c_{0s} of m_2, respectively.

4. **Bus matrix's bus width mapping**
   The bus matrix's bus width mapping selects a bus width for the buses on bus matrix. The data bus width and the address bus width are mapped separately and the bus width is selected from the bus width candidates of the bus protocol in the bus database. For example, the protocol of bus matrix

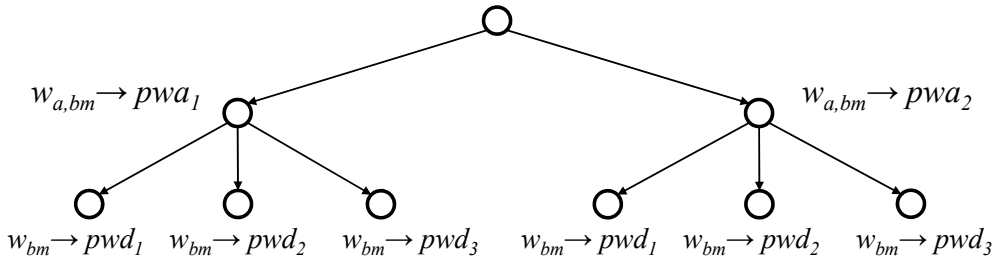


Figure 5.20: An example of bus matrix's bus width mapping tree.

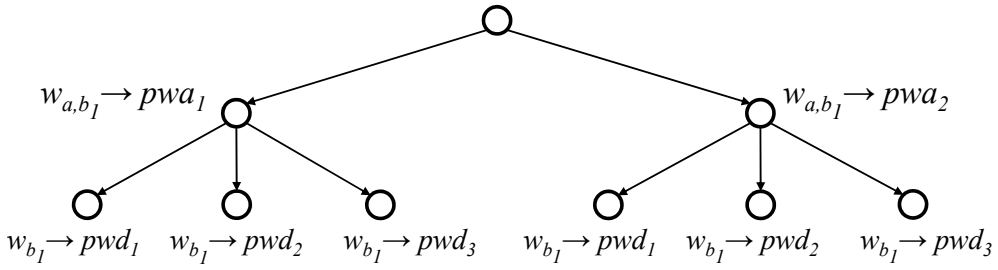


Figure 5.21: An example of shared bus width mapping tree.

specifies two address and three data bus width candidates. Figure 5.20 illustrates the bus matrix's bus width mapping tree, where $w_{a,bm}$ denotes the address bus width and w_{bm} denotes the data bus width. In the figure, pwa_i and pwd_j denote the address and data bus width candidates of the mapped protocol, respectively. For hierarchical shared bus-based architecture, this tree is skipped. For multi-layer bus-based architecture, the depth of this search tree is $2 \times |BM|$.

5. **Shared bus width mapping**
   The shared bus width mapping chooses a bus width for each shared bus, which is selected from the candidates of the bus protocol. This tree resembles the bus width mapping tree in Fig. 5.21, but the data bus width and the address bus width are mapped separately like bus matrix's bus width mapping. Therefore, the depth of this tree equals to $2 \times |B|$

6. **Functional block's execution frequency mapping**
   Each functional block operates at the execution frequency determined by functional block's execution frequency mapping as shown in Fig. 2.5. The frequency is selected from the execution frequency candidates registered in the IP database. The depth of this tree is $|F|$.

7. **Bus matrix's execution frequency mapping**
   The bus matrix, including all the buses on bus matrix, operates at the execution frequency determined in bus matrix's execution frequency mapping.

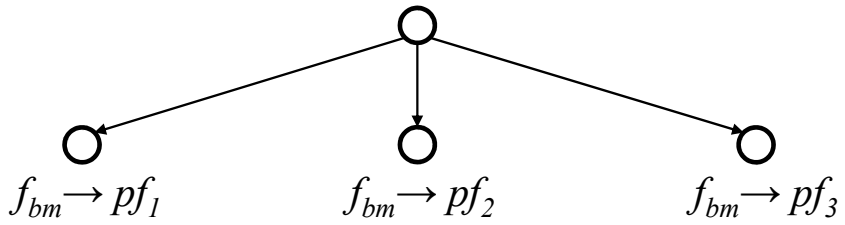


Figure 5.22: An example of bus matrix's execution frequency mapping tree.

The execution frequency is selected from the candidates of the mapped bus protocol in the bus database. Figure 5.22 illustrates the execution frequency mapping of bus matrix, where pf_i denotes the execution frequency candidates of bus matrix's mapped protocol and it is assumed that there are three candidates. For hierarchical shared bus-based architecture, this tree is skipped. For multi-layer bus-based architecture, the depth of this search tree is $|BM|$.

8. **Shared bus' execution frequency mapping**
   Each shared bus in the architecture operates at the execution frequency determined in shared bus' execution frequency mapping. The execution frequency of a bus is selected from the candidates of the bus protocol in the bus database and is mapped in similar manner as in Fig. 2.6. The depth of this tree is $|B|$.

9. **Number of buffer mapping**
   The number of buffer mapping determines the number of buffers in each port of a functional block, except for functional blocks of memory IP. The number of buffer candidates is raised by the designer. The number of receive buffers and the number of transmit buffers are mapped separately. Figure 5.23 illustrates the number of buffer mapping tree, where n_{q_i} and n_{r_i} refer to the number of receive buffers and transmit buffers in port pt_i, respectively, assuming that there are two candidates represented by n_i. The depth of this search tree is $2 \times |PT|$.

5.6.3 Pruning Parameter Set Search Tree

Pruning non-optimal architecture in the early stage can accelerate the exploration, which results in a short exploration time. The parameter set search tree pruning eliminates tree branches that do not produce Pareto solutions using the branch and bound algorithm. As a result, all of the descendants of the search tree from the pruned node are eliminated. The parameter set search tree is pruned when one of the following conditions is met.

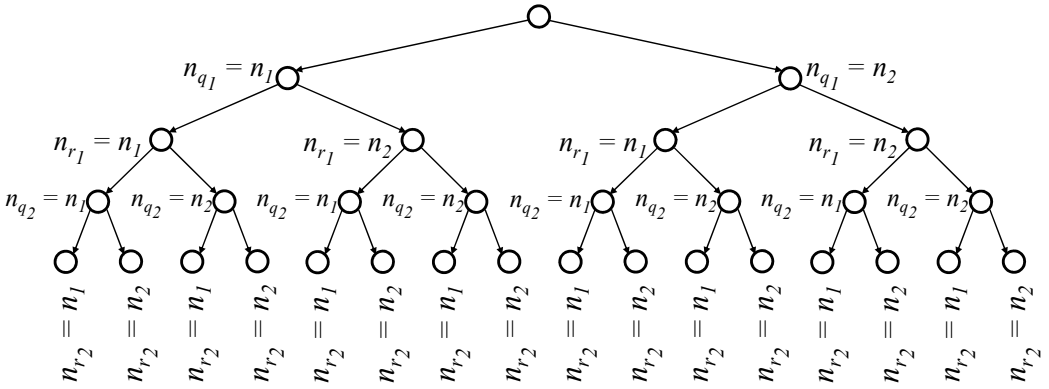


Figure 5.23: An example of the number of buffer mapping tree.

- One or both lower bounds of the execution time and the hardware area of the current search node exceed the design constraints.
- If the lower bound and the upper bound of the execution time are equal, the smallest hardware area architecture is selected for evaluation as one of the architecture candidates.
- Both lower bounds of the execution time and the hardware area of the current search node exceed the explored optimal architecture.
- A deadlock incurs in the lower and upper bound estimation.

The first three conditions eliminate architecture search nodes by the design quality. The forth one eliminates architecture candidates by a deadlock, which occurs at the execution time estimation of the architecture. Although the descendants of the node that incur deadlock may or may not incur deadlock, all of them are also eliminated in the pruning process since there are high possibility that the mapping causes the deadlock in all the descendants.

5.6.4 Order of Parameter Mapping Trees

IPs and bus architecture selection trees determine the organization of the hardware components within an architecture. Parameter mapping trees determine the parameters, such as functional blocks' frequency, bus frequency, bus width, and etc. Therefore, the parameters cannot be explored before the organization of the hardware components is decided.

The hardware components include functional blocks, ports for data transfer, buses, DMACs, memories, and bus bridges. Since channel mapping, specifically channel-to-port mapping, depends on the IP specification of the functional blocks, channel mapping cannot proceed before process mapping. Likewise, cluster-to-bus matrix mapping depends on channel mapping, specifically channel-to-cluster

mapping, it cannot proceed before channel mapping. The objective of DMAC and memory placement is to fulfill the master-slave communication scheme, so it proceeds after the communication path of each channel is decided. Therefore, its exploration takes place after channel mapping and cluster-to-bus matrix mapping. Next, bus matrix protocol mapping, shared bus protocol mapping, and port protocol mapping depend on the bus matrix, buses, and ports, respectively, so they must be done after channel mapping and cluster-to-bus matrix mapping. There is not preference regarding the order of these three mapping.

To take advantage of the pruning based on the lower bound of the hardware area, the parameter trees that affect more on the area should be explored before the execution frequency because pruning early in the exploration leads to a smaller number of search nodes. From the observation, the CNN accelerator consumes more area that other components, so the number of functional block instance mapping and the number of PE mapping should be explored first. The number of memory's storage blocks, the number of buffers, bus matrix's bus width, and shared bus width affect the area in order. However, exploring the number of buffers early may lead to a larger number of nodes because its tree tends to be the highest, and hence, it has the largest number of leaves. Therefore, it is explored last. There is no preference in the order of functional block's, bus matrix's, bus' execution frequency mapping.

Based on these findings, the proposed method explores parameters in the following order. First, IPs and bus architecture selection trees are ordered as follows: (1) process mapping; (2) channel mapping; (3) cluster-to-bus matrix mapping; (4) DMAC and memory placement; (5) bus matrix protocol mapping; (6) shared bus protocol mapping; (7) port protocol mapping. Second, parameter trees are ordered as follows: (1) number of functional block instance mapping; (2) number of PE mapping; (3) number of memory's storage block mapping; (4) bus matrix's bus width mapping; (5) shared bus width mapping; (6) functional block's execution frequency mapping; (7) bus matrix's execution frequency mapping; (8) shared bus' execution frequency mapping; (9) number of buffer mapping. The proposed method explores based on the larger values first to take advantage of pruning by constraints on design qualities and the explored optimal architecture.

5.7 Case Study

This section evaluates the benefits of the proposed architecture exploration method. To apply the proposed method to exploring SoC architecture for CNN-based AI platform, first, the parallelism-flexible convolution core is modeled as a parameterized IP for supporting multiple types of parallelism of convolutional layers. Then, the experiments show that the proposed method can discover Pareto-optimal architectures with parameterized IP and several configurations of multi-layer bus. The environment of the experimental platform is set as shown in Table 5.2. In this experiment, the estimation parameters are based on a CMOS 0.18 μm process

Table 5.2: Environment of the experimental platform

Hardware	
Machine	
CPU	Intel Xeon CPU E7-8880 2.30GHz
Memory	1 TB
OS	64 bits CentOS 6.10
Software	
SystemC	version 2.3.1a
C compiler	GNU GCC compiler 4.4.7

technology. Here, it should be noted that the design space of the proposed method is significantly extended for CNN-based AI platform with multi-layer bus and data tiling, and it is incompatible with other methods such as [131]. Therefore, quantitative comparisons are not performed.

5.7.1 Modeling Parallelism-flexible Convolution Core

In order to apply the proposed method for CNN-based AI platform, this section models the proposed parallelism-flexible convolution core in terms of modeling scheme, behavior and architecture. The modeling scheme relates to modeling the CNN-based application in SLM. The behavioral modeling relates to estimating the design quality, specifically execution time, of the system. The architectural modeling relates to parameterizing the architecture for architecture exploration.

First, a process of a convolutional layer is modeled with the IFM-major scheme in the SLM. According to Algorithm 1, the proposed convolution core computes all OFMs of a tile from each IFM first as the IFM loop locates outer than the OFM loop. This conforms with the IFM-major modeling scheme, which models the processing of an IFM as one process execution vertex in the SL-EDG.

Second, in terms of behavior modeling, the processing time factor, α, equals to $\frac{1}{P}$, where P is the degree of parallelism explained in Chapter 4. From Algorithm 1, the processing time for one IFM depends on the number of non-zero weights of that channel and P. Since the proposed convolution core increases inter-output parallelism by the degree P, the processing time of each convolutional layers' process vertex in the AL-EDG is defined as follows:

$$t_p = \frac{1}{P} \times \frac{e_{(p_j, fb_i)}}{f_{fb_i}}, \tag{5.14}$$

where $e_{(p_j, fb_i)}$ is equivalence to the number of non-zero weights of that channel. In the proposed method, P is determined as the value that maximizes the PE utilization, U in Eq. (4.3). However, the utilization also considers the number of

instance of IP implemented as functional block i, N_{fb_i}. For that reason, the utilization is redefined as follows:

$$U = \frac{X \times Y \times C_o \times K \times K \times C_i \times R \times 100}{N_{fb_i} \times N \times G \times M \times E}, \tag{5.15}$$

where X, Y, C_o, K, C_i, R, N, G, and M are the same as defined in Chapter 4. The estimated number of cycles in computing the convolutional layer, E, is calculated in a similar way as Eq. (4.4), but T is replaced with the number of tiles calculated on one instance of fb_i, T', which is defined as follows:

$$T' = \lceil \lceil \lceil \frac{X}{N} \rceil \times \frac{Y}{G \times \lfloor \frac{M}{P} \rfloor} \rceil \times \frac{1}{N_{fb_i}} \rceil. \tag{5.16}$$

In the experiments, P is considered as one of the following candidates: $1, 2, 4, 8, 16$.

Third, in terms of architecture, the proposed convolution core is parameterized according to the N, G, and M described in Chapter 4. The product of N, G, and M is the total number of PEs. For that reason, the number of PE mapping tree is replaced with three mapping trees: the number of PE bank mapping tree (N_{M_i} mapping tree), the number of PE groups in one PE bank mapping tree (N_{G_i} mapping tree), and the number of PEs in one PE group mapping tree (N_{N_i} mapping tree). The area is calculated as follows:

$$A(fb_i) = (g_{ip_j} + (g_M + (g_G + g_N \times N_{N_i}) \times N_{G_i}) \times N_{M_i}) \times g_{nand}, \tag{5.17}$$

where g_M, g_G, and g_N are the area of a PE bank (excluding PE groups), a PE group (excluding PEs), and a PE, respectively. In addition, in order to take advantage of a large-bit width bus, multiple data are merged for transferring in the same cycle to maximally use the bus capacity.

5.7.2 Experiment 1 : Validity of the Proposed Architecture Exploration Method

This experiment shows results to confirm the validity of the proposed architecture exploration method. Two types of architecture exploration were conducted: exhaustive exploration and exploration with pruning. The results were compared in terms of time for architecture exploration and exploration coverage.

Objective

- To show that the search with pruning effectively explores the architecture design space and discovers Pareto-optimal candidates.
- To show that the proposed method can explore varieties of architectures by customizing the number of instances and architectural parameter of functional blocks

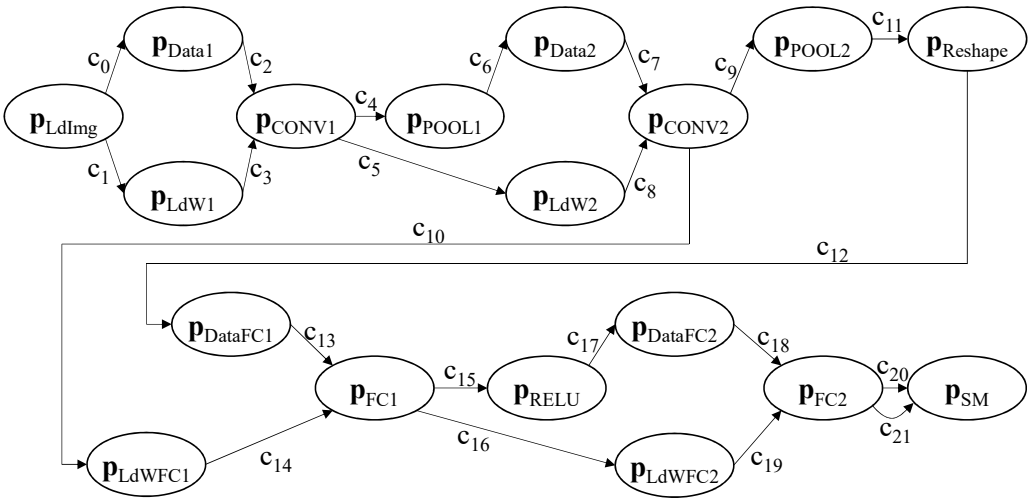


Figure 5.24: An SLM of Lenet-5.

Target System

The target system is a LeNet-5 CNN [165], which is illustrated in Fig. 5.24. The SLM is composed of 18 processes and 22 channels. The input is an one-channel of 28×28 pixels image and the output is ten softmax values indicating the probability of the image being ten classes.

The input image data is read from external memory (p_{LdImg}) and written to a storage (p_{Data1}). Likewise, the weights of the first convolutional layer are read from external memory (p_{LdW1}). Then, the image data and the weights are convoluted (p_{CONV1}). Next, pooling operation is performed on the convoluted results (p_{POOL1}) and written to a storage (p_{Data2}). The p_{LdW2}, p_{CONV2}, and p_{POOL2} operate similarly. Before writing the pooling results to a storage ($p_{DataFC1}$), they are reshaped into a vector ($p_{reshape}$). For the fully-connected layers, the weights of each layer are loaded (p_{LdWFC1}, p_{LdWFC2}) and the matrix multiplication of fully-connected layers takes place (p_{FC1}, p_{FC2}). The results of p_{FC1} are activated with RELU function (p_{RELU}) and stored to a storage ($p_{DataFC2}$). The results of p_{FC2} are activated with softmax function and written to external memory (p_{SM}).

The channels perform inter-process data transfers. Noted that c_1, c_5, c_{10}, c_{16}, and c_{21} are one-bit signals, each of which indicates that the preceding convolutional layer or fully-connected layer is done, and hence, the weights of the next layer can start loading. These channels are mapped to a dedicated one-bit port and signal. They are irrelevant to multi-layer bus.

Experimental Settings

Here, three experimental settings are described: estimation parameters, the candidates for architecture exploration, and the design constraints. First, the estimation

Table 5.3: Estimation parameter value for CMOS 0.18 μm. process technology

Parameter Name	Parameter Value	Notes
Area per gate	9.8 μm^2	NAND gate area
Wire pitch	0.56 μm.	METAL3 layer wire pitch
Over-the-cell ratio	0.95	-

Table 5.4: IP database in experiment 1

IP	Area (gate)	Freq. Cand. (MHz)	Ports	Mappable processes [name(cycle)]	Other parameters [name(cand.)]
IP_1	3,294	200	1 AHB Master	LdImg(20)	
IP_2	45,553	200	1 AHB Slave	Data1(1) Data2(1) DataFC1(1) DataFC2(1)	
IP_3	300	200	1 AHB Slave	LdW1(20) LdW2(20) LdWFC1(20) LdWFC2(20)	
IP_4	10,000 (g_{ip}) 3,000 (g_M) 500 (g_G) 30 (g_N)	200	3 AHB Master	CONV1(330) CONV2(150)	N_{fb_i}(1,2) N_{M_i}(8,16) N_{G_i}(4) N_{N_i}(8,16)
IP_5	2,000	200	1 AHB Master	POOL1(3) POOL2(1)	
IP_6	500	200	1 AHB Slave	Reshape(1)	
IP_7	3,294	200	1 AHB Master	FC1(16) FC2(10)	
IP_8	1,500	200	1 AHB Master	RELU(16)	
IP_9	3,000	200	1 AHB Slave	SM(10)	

Table 5.5: Bus database in experiment 1

Protocol Name	Bus Type	Bus Width Candidate [bit]	Frequency Candidate [MHz]	Max. Master Master Number	Max. Slave Number
AHB	Shared	256	200	16	-
AHB	Multi-layer	256	200	8	8

Table 5.6: Functional block constraint in experiment 1

Functional Block	**Mapped Processes**
$fb_1 = (IP_1, 1)$	LdImg
$fb_2 = (IP_2, 1)$	Data1, DataFC1
$fb_3 = (IP_2, 2)$	Data2, DataFC2
$fb_4 = (IP_3, 1)$	LdW1, LdW2, LdWFC1, LdWFC2
$fb_5 = (IP_4, 1)$	CONV1, CONV2
$fb_6 = (IP_5, 1)$	POOL1, POOL2
$fb_7 = (IP_6, 1)$	Reshape
$fb_8 = (IP_7, 1)$	FC1, FC2
$fb_9 = (IP_8, 1)$	RELU
$fb_{10} = (IP_9, 1)$	SM

Table 5.7: Port constraint in experiment 1

Port	Protocol	Connected fb_i	Cluster	n_q	n_r	Mapped Channels
pt_1	AHB (M)	fb_1	1	0	1	c_{0s}
pt_2	AHB (S)	fb_2	2	1	1	c_{0d}, c_{2s}, c_{12d}, c_{13s}
pt_3	AHB (S)	fb_3	3	1	1	c_{17d}, c_{18s}, c_{6d}, c_{7s}
pt_4	AHB (S)	fb_4	4	1	1	c_{19s}, c_{3s}, c_{14s}, c_{8s}
pt_5	AHB (M)	fb_5	5	2	0	c_{2d}, c_{7d}
pt_6	AHB (M)	fb_5	6	2	0	c_{3d}, c_{8d}
pt_7	AHB (M)	fb_5	7	0	2	c_{4s}, c_{9s}
pt_8	AHB (M)	fb_6	8	1	1	c_{4d}, c_{6s}, c_{9d}, c_{11s}
pt_9	AHB (S)	fb_7	9	1	1	c_{11d}, c_{12s}
pt_{10}	AHB (M)	fb_8	-	1	1	c_{20s}, c_{19d}, c_{15s}, c_{18d}, c_{14d}, c_{13d}
pt_{11}	AHB (M)	fb_9	-	1	1	c_{15d}, c_{17s}
pt_{12}	AHB (S)	fb_{10}	-	1	0	c_{20d}

Table 5.8: Time for architecture exploration

Method	# of leaves	# of estimation	Total time for expl.
Exhaustive expl.	2,140,577	2,139,768	14.5 hours
Expl. with pruning	917,893	1,657,035	11.5 hours

parameters are process technology-specific parameters used in the area estimation. Second, the candidates for architecture exploration are user-defined, such as IP candidates and bus candidates. Third, the design constraints include design quality constraints and architectural constraints.

The parameters are shown in Table 5.3.

The candidates for architecture exploration include IPs in IP database, bus protocols in bus database, candidates of the number of buffers, and candidates of the number of memory's storage blocks. The IP database and bus database used in experiment 1 are shown in Table 5.4 and Table 5.5, respectively. The IP_4 is the parallelism-flexible convolution core proposed in chapter 4.

This experiment raises three types of design constraints: (1) functional block constraint, which indicates the IP instances and their mapped processes; (2) port constraint, which indicates the ports used for each channel; (3) bus constraint, which indicates the maximum number of buses. First, Table 5.6 shows the functional block constraint, where $fb_i = (IP_j, k)$ means fb_i is the k^{th} instance of the IP_j. The IPs undertake processes of the same processing but locates in different CNN layers because the experiment processes only one image so they can operate without the effect in performance. The functional block set is fixed, but the functional blocks' parameter, i.e. N_{fb_i}, N_{M_i}, N_{G_i}, andN_{N_i}, are not constrained. Second, Table 5.7 lists the port constraint in terms of connected functional block, protocol, cluster, the number of buffers (n_q and n_r), and the mapped channels. The M and S after the port's protocol indicate that the port is either a master or a slave. Note that the dedicated ports, which are mapped to the channels of 1-bit signals, are omitted since they are not clustered to be mapped onto the bus matrix. Third, the bus constraint specifies that the maximum number of buses in one cluster is 1. The candidate for the number of buffers and the number of memory's storage blocks is set to 1. However, this experiment does not raise any constraints about the design qualities.

Time for Architecture Exploration

Table 5.8 shows the experimental results for exploring the architecture of the LeNet-5 with the proposed method. Exhaustive expl. denotes the exhaustive exploration that traverses every node in the parameter set search tree. Expl. with pruning denotes the exploration that traverses the parameter set search tree with candidate pruning. In the table, "# of leaves", "# of estimation", and "Total time

for expl." refer to the number of visited leaves, the number of design quality estimation, and the total time for architecture exploration, respectively.

The experimental results show that the exhaustive exploration takes a longer time to discover the same Pareto-optimal architectures as the exploration with pruning. That is because the exhaustive exploration searches and estimates the design qualities for every leaves, while the exploration with pruning disregards the leaves in the branches of the tree that does not yield Pareto-optimal solutions. However, the number of estimation in the exploration with pruning is more than the number of leaves because the exploration also performs the estimation at the nodes of the tree for pruning. The experiment shows only 21% of time reduction thanks to pruning because this test case gives only a few candidates for the exploration and a lot of design constraints, which limit the design space. However, the time reduction will be significant when exploring a large number of candidates and less design constraints.

Since a strict port constraint in Table 5.7 is raised for this experiment, the design space is small and takes only less than 15 hours. However, when the ports are not constrained to any cluster (removing the numbers in column "Cluster" in Table 5.7), the design space contains 33.8 million architectures, which is 15.7 times larger than the size of the design space shown with the column "# of leaves" of exhaustive exploration in Table 5.8. Consequently, considering the design space of the same set of ports and mapped channels without constraining ports to clusters, it may take up to 9.5 days and 7.6 days with the exhaustive exploration and the exploration with pruning, respectively. The design space is even larger and its exploration consumes longer exploration time when the port constraint is removed. Furthermore, when the AI system grows larger, the exploration time takes even longer with the larger number of functional blocks and ports.

Even though the proposed method can explore and evaluate each architecture very quickly compared to the low-level design, this experiment shows that the proposed method may consume weeks or months to explore a huge design space of the AI applications. However, such an exploration time in the early stage of the design is too long and barely acceptable considering the tight time-to-market. Therefore, the proposed method still needs improvements in terms of the time to explore a large design space. The improvements can be achieved by introducing incremental computation for successive architectures with a small difference, more aggressive pruning, and parallel traversal of the parameter set search tree.

Varieties of the Explored Architectures

Figure 5.25 shows the 17 explored Pareto-optimal architectures plotted in a trade-off relationship between application's execution time and area. The proposed architecture exploration method explored various parameters of functional blocks. The architectures which contain a larger number of PEs consume more area, but execute the LeNet-5 faster. Table 5.9 shows the parameters of fb_5, which includes N_{fb_5}, N_{M_5}, N_{G_5}, andN_{N_5}. It shows that the proposed method can customize the pa-

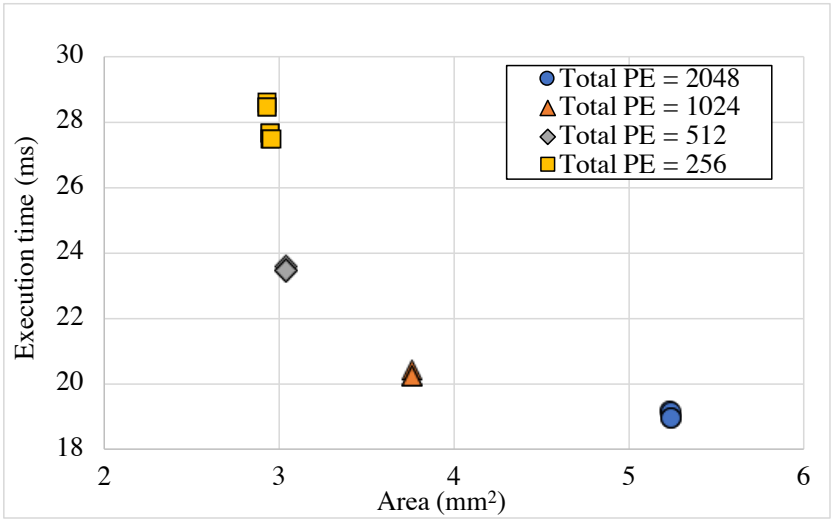


Figure 5.25: Pareto-optimal architectures resulted from experiment 1.

Table 5.9: Parameters of fb_5 of the Pareto-optimal architecture

Arch.	N_{fb_5}	N_{M_5}	N_{G_5}	N_{N_5}
Total PE = 2,048				
arch1, arch2, arch3, arch4	2	16	4	16
Total PE = 1,024				
arch5, arch6, arch7	1	16	4	16
Total PE = 512				
arch8, arch9, arch10	1	8	4	16
Total PE = 256				
arch11, arch12, arch13, arch14 arch15, arch16, arch17	1	8	4	8

rameters of functional blocks, and hence, it provides varieties of Pareto-optimal architectures with different parameter combinations. The architectures containing the same combination of parameters have different area and execution time because of the results from clustering port pt_{10}, pt_{11}, and pt_{12} into different clusters, which leads to a different multi-layer bus configuration.

5.7.3 Experiment 2 : Architecture Exploration for Large CNN Application

This experiment shows the Pareto-optimal architectures found by the proposed architecture exploration method in terms of execution time and hardware area. This experiment explores the design space with pruning.

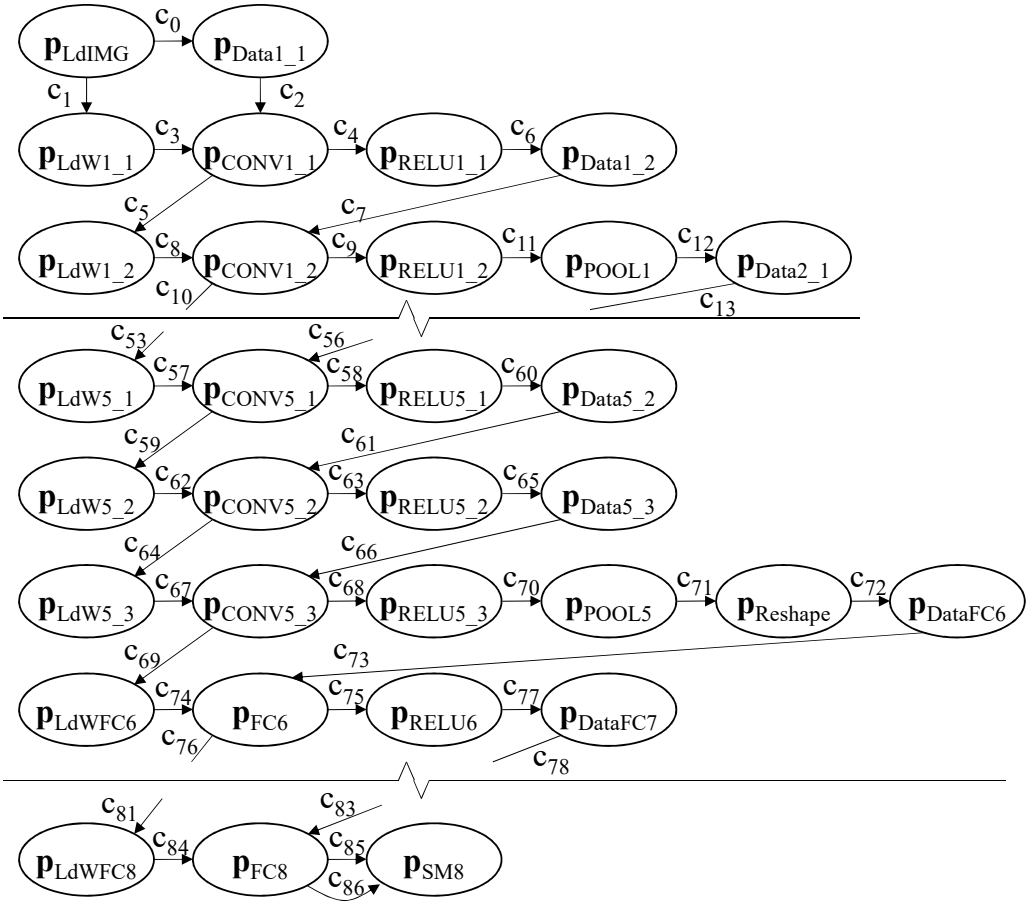


Figure 5.26: An SLM of VGG-16.

Table 5.10: IP database in experiment 2

IP	Area (gate)	Freq. Cand. (MHz)	Ports	Other parameters [name(cand.)]
IP_1	6,588	400	1 AHB Master	
IP_2	60,000	200	1 AHB Slave	
IP_3	300	200	1 AHB Slave	
IP_4	10,000 (g_{ip}) 3,000 (g_M) 500 (g_G) 30 (g_N)	200	3 AHB Master	N_{fb_i}(1,2,4) N_{M_i}(16) N_{G_i}(4) N_{N_i}(16)
IP_5	2,500	400	1 AHB Slave	
IP_6	3,000	200	1 AHB Master	
IP_7	500	200	1 AHB Slave	
IP_8	20,000	1,000	1 AHB Master	
IP_9	7,500	400	1 AHB Slave	

Objective

- To show that the proposed method can explore architectures including various number of instances of functional blocks
- To show that the proposed method can explore varieties of multi-layer bus configurations.

Target System

The target system is a VGG-16 CNN [17], which is illustrated in Fig. 5.26. The SLM is composed of 71 processes and 87 channels. The input is a three-channel of 224×224 pixels image and the output is a 1,000 softmax values indicating the probability of the image being 1,000 classes of ImageNet [113]. The operations are similar to the LeNet-5, but the VGG-16 contains more layers in different order. Figure 5.26 omits some layers of the VGG-16 since they order in similar way as the ones shown in the figure. The channels indicating that the preceding convolutional layer or fully-connected layer is done, such as c_1, c_5, c_{53}, c_{59}, c_{64}, c_{69}, and c_{81} in Fig. 5.26, are mapped to dedicated one-bit ports and signals.

Experimental Settings

Similar to the experiment 1, there are three experimental settings. The estimation parameters are the same as described in Table 5.3.

Table 5.11: IP functionality (mappable processes) in experiment 2

IP	**Mappable Process (cycle)**
IP_1	LdIMG(500)
IP_2	Data1_1(1), Data1_2(1), Data2_1(1), Data2_2(1), Data3_1(1), Data3_2(1), Data3_3(1), Data4_1(1), Data4_2(1), Data4_3(1), Data5_1(1), Data5_2(1), Data5_3(1), DataFC6(1), DataFC7(1), DataFC8(1)
IP_3	LdW1_1(34),LdW1_2(25), LdW2_1(36), LdW2_2(37), LdW3_1(68), LdW3_2(42), LdW3_3(58), LdW4_1(78), LdW4_2(69), LdW4_3(82), LdW5_1(83), LdW5_2(73), LdW5_3(85), LdWFC6(20), LdWFC7(20), LdWFC8(20)
IP_4	CONV1_1(351), CONV1_2(143), CONV2_1(408), CONV2_2(431), CONV3_1(1238),CONV3_2(569), CONV3_3(984), CONV4_1(1491), CONV4_2(1261), CONV4_3(1583), CONV5_1(1629), CONV5_2(1353), CONV5_3(1675)
IP_5	RELU1_x(196), RELU2_x(49), RELU3_x(13), RELU4_x(4), RELU5_x(1), RELU6(16), RELU7(16)
IP_6	POOL1(98), POOL2(25), POOL3(7), POOL4(2), POOL5(1)
IP_7	Reshape(1)
IP_8	FC6(25088), FC7(4087), FC8(1000)
IP_9	SM8(400)

Table 5.12: Bus database in experiment 2

Protocol Name	Bus Type	Bus Width Candidate [bit]	Frequency Candidate [MHz]	Max. Master Master Number	Max. Slave Number
AHB	Shared	1,024	400	16	-
AHB	Multi-layer	1,024	400	8	8

Table 5.13: Functional block constraint in experiment 2

Functional Block	Mapped Processes
$fb_1 = (IP_1, 1)$	LdImg
$fb_2 = (IP_2, 1)$	Data1_1, Data2_1, Data3_1, Data3_3, Data4_2, Data5_1, Data5_3, DataFC7
$fb_3 = (IP_2, 2)$	Data1_2, Data2_2, Data3_2, Data4_1, Data4_3, Data5_2, DataFC6, DataFC8
$fb_4 = (IP_3, 1)$	LdW1_x, LdW3_x, LdW3_x, LdW4_x, LdW5_x, LdWFC6, LdWFC7, LdWFC8
$fb_5 = (IP_4, 1)$	CONV1_x, CONV2_x, CONV3_x, CONV4_x, CONV5_x
$fb_6 = (IP_5, 1)$	RELU1_x, RELU2_x, RELU3_x, RELU4_x, RELU5_x, RELU6,RELU7
$fb_7 = (IP_6, 1)$	POOL1, POOL2, POOL3, POOL4, POOL5
$fb_8 = (IP_7, 1)$	Reshape
$fb_9 = (IP_8, 1)$	FC6, FC7, FC8
$fb_{10} = (IP_{10}, 1)$	SM8

The IP database in this experiment is shown in Table 5.10 and the list of mappable processes of each IP is shown in Table 5.11. The IP_4 is the parallelism-flexible convolution core proposed in chapter 4. The proposed method considers data tiling, so the number of cycles is the number per one tile. The bus database is shown in Table 5.12. The candidate for the number of buffers and the number of memory's storage blocks are fixed to 1.

This experiment raises similar design constraints as the experiment 1. Table 5.13 and Table 5.14 show the functional block constraint and port constraint, respectively. The port constraint is given to the three ports connected to fb_5 so that it conforms to the parallelism-flexible convolution core except for that the ports are not constrained to any cluster. The bus constraints of the proposed method are the same as experiment 1.

Table 5.14: Port constraint in experiment 2

Port	Protocol	Connected fb_i	Cluster	n_q	n_r	Mapped Channels
pt_1	AHB (M)	fb_1	-	0	1	c_{0s}
pt_2	AHB (S)	fb_2	-	1	1	c_{0d}, c_{2s}, c_{12d}, c_{13s}, c_{23d}, c_{24s}, c_{33d}, c_{34s}, c_{44d}, c_{45s}, c_{55d}, c_{56s}, c_{65d}, c_{66s}, c_{77d}, c_{78s}
pt_3	AHB (S)	fb_3	-	1	1	c_{6d}, c_{7s}, c_{17d}, c_{18s}, c_{28d}, c_{29s}, c_{39d}, c_{40s}, c_{49d}, c_{50s}, c_{60d}, c_{61s}, c_{72d}, c_{73s}, c_{82d}, c_{83s}
pt_4	AHB (S)	fb_4	-	1	1	c_{3s}, c_{8s}, c_{14s}, c_{19s}, c_{25s}, c_{30s}, c_{35s}, c_{41s}, c_{46s}, c_{51s}, c_{57s}, c_{62s}, c_{67s}, c_{74s}, c_{79s}, c_{84s}
pt_5	AHB (M)	fb_5	0	2	0	c_{3d}, c_{8d}, c_{14d}, c_{19d}, c_{25d}, c_{30d}, c_{35d}, c_{41d}, c_{46d}, c_{51d}, c_{57d}, c_{62d}, c_{67d}
pt_6	AHB (M)	fb_5	1	2	0	c_{2d}, c_{7d}, c_{13d}, c_{18d}, c_{24d}, c_{29d}, c_{34d}, c_{40d}, c_{45d}, c_{50d}, c_{56d}, c_{61d}, c_{66d}
pt_7	AHB (M)	fb_5	2	0	2	c_{4s}, c_{9s}, c_{15s}, c_{20s}, c_{26s}, c_{31s}, c_{36s}, c_{42s}, c_{47s}, c_{52s}, c_{58s}, c_{63s}, c_{68s}
pt_8	AHB (S)	fb_6	-	1	1	c_{4d}, c_{6s}, c_{9d},c_{11s}, c_{15d}, c_{17s}, c_{20d},c_{22s}, c_{26d}, c_{28s}, c_{31d}, c_{33s}, c_{36d}, c_{38s}, c_{42d}, c_{44s}, c_{47d}, c_{49s}, c_{52d}, c_{54s}, c_{58d}, c_{60s}, c_{63d}, c_{65s}, c_{68d}, c_{70s}, c_{75d}, c_{77s}, c_{80d}, c_{82s}
pt_9	AHB (M)	fb_7	-	1	1	c_{11d}, c_{12s}, c_{22d}, c_{23s}, c_{38d}, c_{39s}, c_{54d}, c_{55s}, c_{70d}, c_{71s}
pt_{10}	AHB (S)	fb_8	-	1	1	c_{71d},c_{72s}
pt_{11}	AHB (M)	fb_9	-	1	1	c_{73d}, c_{74d}, c_{75s}, c_{78d}, c_{79d}, c_{80s}, c_{83d}, c_{84d}, c_{85s}
pt_{12}	AHB (S)	fb_{10}	-	1	0	c_{85d}

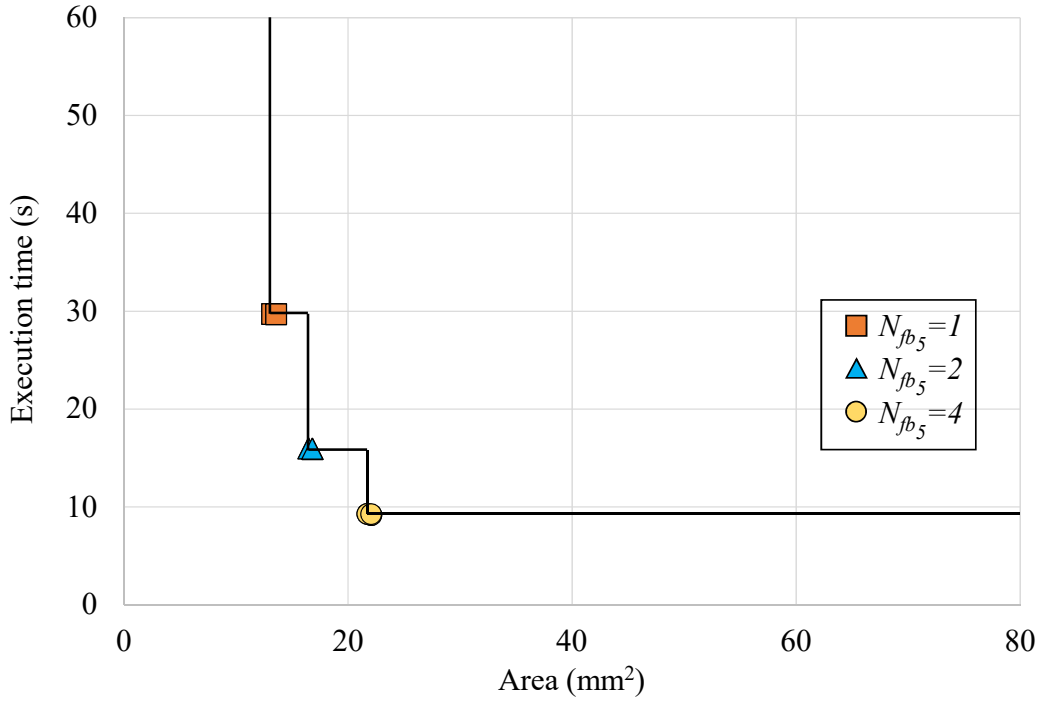


Figure 5.27: The trade-off relationship between area and execution time of the Pareto-optimal architectures discovered by the proposed method.

Pareto-optimal Architectures

Figure 5.27 shows the trade-off relationship between area and execution time of the Pareto-optimal architectures discovered by the proposed method. All of these architectures are multi-layer bus-based architectures. They are divided into three groups by the number of instances of fb_5, N_{fb_5}. The circle, triangle, and rectangle marks represent the architectures that N_{fb_5} is 4, 2, and 1, respectively. Multiple instances of fb_5 parallelize intra-layer MACCs of the CONV process.

The Pareto-optimal architectures exploit the following three characteristics for achieving high performance. First, multiple instances of fb_5 parallelize intra-layer MACCs of the CONV processes, which results in a short time for computing a convolutional layer. Second, the multi-layer bus parallelizes the data transfer. Third, the proposed method considers data tiling so that the data transfer and data processing of a process are partitioned and can be paralleled. Additionally, the data tiling concept of the proposed method also reduces the size of buffer for each functional block.

Varieties of Explored Architectures

Figure 5.28 shows an example of the architectures from the proposed method, indicating that the proposed method can explore varieties of architectures in terms

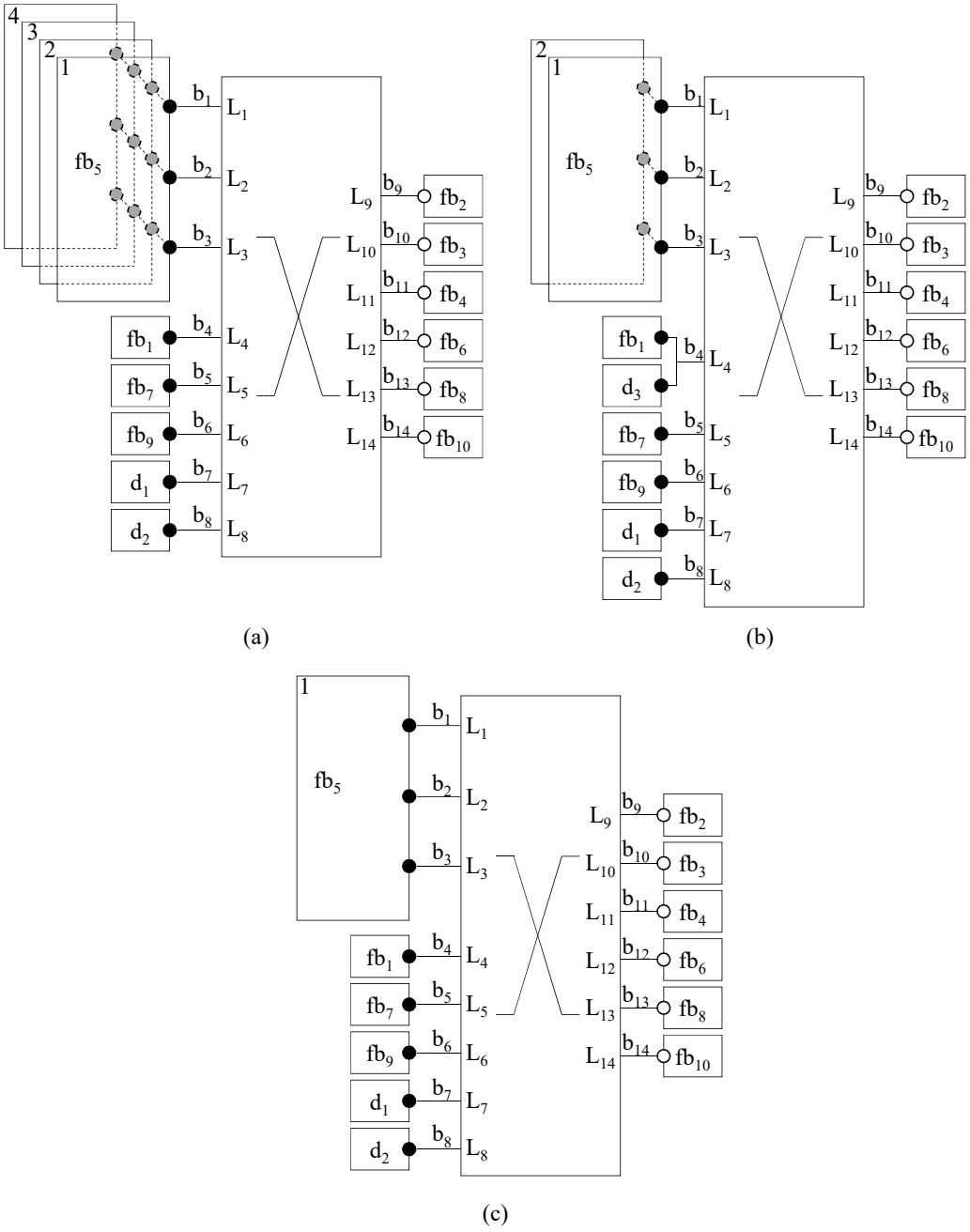


Figure 5.28: Example of architectures discovered by the proposed method.

of the number of instances of functional blocks and the multi-layer bus configurations. The three architectures are multi-layer bus-based architectures.

In terms of the number of instances of functional blocks, the architecture in Fig. 5.28(a) contains four instances of fb_5 for processing convolutional layers, while Fig. 5.28(b) and Fig. 5.28(c) contain two and one instances, respectively. The variety in the number of instance affects the area and execution time significantly as shown by the circle, triangle, and rectangle marks in Fig. 5.27.

In terms of multi-layer bus configurations, Fig. 5.28(a) shows an architecture that connects each port to a layer of multi-layer bus. In Fig. 5.28(b), both the master port of fb_1 and a DMAC d_3 are connected to bus b_4 and layer L_4 of the bus matrix. The cluster of the master port of fb_1 and d_3 constructs a multiple master configuration on the bus matrix. Other configurations including multiple slave and AHB-subsystem configurations are also explored by the proposed method.

5.8 Conclusion

This chapter proposed an architecture design space exploration method with two major features that are suitable for discovering high-performance SoC architecture for CNN-based AI platform. First, it parameterizes and explores optimal parameter values so that it finds architectures that well leverage intra-process parallelism, specifically multiple types of parallelism in convolutional layers. Second, it partitions hardware components in the architecture into clusters to explore optimal bus architecture, i.e. configurable multi-layer bus, so that it finds bus architecture that is capable of conducting a vast amount of parallel data transfers. The results show that the proposed method discovers varieties of architecture including varieties of functional blocks and multi-layer bus configurations. The future works include energy consumption estimation because it is one of the most important design qualities, additional techniques that accelerate the design space exploration, and the exploration of other types of communication architecture, such as Network-on-chip (NoC).

Chapter 6

Conclusion and Future Work

This chapter concludes the discussion in this thesis and describes future work.

6.1 Conclusion

The need for high-performance AI-based edge computing devices brings about a grand challenge in designing an optimal SoC under the strict design quality constraints. To effectively design such edge devices, there are three requirements: (1) quickly evaluate the design quality of each architecture in the design space; (2) accelerate the computation of deep learning algorithms; (3) efficiently explore the design space to find optimal architectures. This thesis contributes to the method of designing an SoC architecture for CNN-based AI applications from a system-level description so that the architecture achieves high-performance in performing the inference processing at the edge. The target architecture consists of IP modules, including CNN accelerators, DMACs, memories, shared buses and/or a configurable multi-layer bus. The difficulties lie upon the fact that the design space of the target architecture is extremely large, the architectures are complicated, and consequently, the design quality estimation is time-consuming. This thesis tackled with these problems in order to fulfill the requirements by proposing (1) an efficient performance estimation method for a configurable multi-layer bus-based architecture with system-level profiling; (2) a parallelism-flexible convolution core to accelerate the convolutional layers of CNN; (3) an architecture exploration method that effectively finds the architectures that leverage parallelisms in both computation and data communication.

Chapter 3 proposes an efficient performance estimation method for a configurable multi-layer bus-based architecture. The proposed method estimates the execution time of an application on several architectures by profiling the execution behavior of the application, constructing an SL-EDG, constructing an AL-EDG for each architecture, and estimates the execution time by analyzing the AL-EDG. This method is fast for estimating a large number of architectures because the first two procedures, which are time-consuming, take place only once for an applica-

tion, and the latter two procedures, which are fast, repeat for various architectures. The performance analysis considers bus protocol behavior and dynamic bus contention so that it can accurately estimate the execution time. The experiments have shown that the proposed method can estimate the execution time within 8% of error and achieve 25.6 times tool-runtime speedup compared to the conventional RTL simulation. To sum up, the proposed performance estimation method is both fast and accurate, so it is efficient as a design evaluation method for architecture exploration.

Chapter 4 proposes a parallelism-flexible convolution core for sparse CNN to accelerate the convolutional layers of CNN with two techniques. First, it maximizes parallel calculation by leveraging multiple types of parallelism with the parallelism controller and the weight broadcaster that alternate dataflow layer by layer. Second, it skips MACCs related to zero-valued weights to reduce the amount of computation with the weight broadcaster that broadcasts only non-zero weights to the PEs. The results show that it achieves 4x speedup over the baseline architecture and 3x in effective GMACS over prior arts of CNN accelerator on a VGG-16 benchmark. The proposed convolution core can accelerate the CNN effectively.

An architecture design space exploration method proposed in chapter 5 considers parallelism in intra-process computation, inter-process computation, and data transfers to search Pareto-optimal architecture candidates. It takes intra-process and inter-process computation into account by exploring parameterized IPs that allow one process to be executed on multiple instances of functional blocks. It parallelizes data transfers through clustering and mapping the transfers to the configurable multi-layer bus. The results show that the proposed method discovers varieties of architectures, including varieties of functional block parameters and multi-layer bus configurations. Hence, the proposed architecture exploration method is suitable for discovering SoC architectures for CNN-based AI platform.

To design a high-performance SoC for CNN-based AI platform, this thesis proposes a fast architecture design space exploration method that searches hardware components, organization, and parameters of the SoC. First, the efficient performance estimation method enables the execution time evaluation of a large number of architecture candidates. Second, the parallelism-flexible convolution core provides a high-performance CNN accelerator. Third, a fast architecture exploration method systematically explores and prunes the design space of varieties of architectures including parameterized IPs, specifically CNN accelerator, and configurable multi-layer bus. Hence, the fast design quality evaluation method, the high-performance CNN accelerator, and the systematic exploration lead to the discovery of superior architectures for CNN-based AI platform within a short time.

6.2 Future Work

To further enhance the proposed architecture design space exploration method of an SoC for a CNN-based AI platform, the following topics remain as future work.

6.2.1 Extension of Communication Architecture

In this thesis, the bus architecture is limited to only hierarchical shared bus and multi-layer bus, but there are more high-performance bus architectures used in the embedded systems, such as NoC. These bus architectures provide high parallelism capability with unique specifications such as routing and switching functions of the NoC. Such bus architectures can be supported by (1) modeling their specific functions in order to analyze parallel data transfer behavior; (2) extending the parameter set search tree to explore the bus architecture-specific parameters.

6.2.2 Statistical Performance Estimation Approach

The performance of many applications, such as encoding and decoding, varies by the given data that the worst-case estimation proposed in this thesis may provide a too pessimistic design quality evaluation. For that reason, applying a statistical analysis to the proposed performance estimation method is a promising direction. It can be employed by analyzing performance distribution and it can assure the estimated performance with confidence.

6.2.3 Constrained Neural Network Sparsification

In order to maximally take advantage of the proposed parallelism-flexible convolution core, a constrained neural network sparsification process should be developed. The sparsification technique should consider a balanced distribution of the zero-valued weights across all channels or kernels so that all PEs are responsible for an equal amount of workload. As a result of a balanced workload, there exists a minimum number of idle PE cycles because the PEs do not have to wait for the others to finish their workload.

6.2.4 Process and Communication Scheduling

This thesis schedules the execution of process and communication by priority. However, other scheduling schemes are also widely used in the SoC design, such as round robin, which is a dynamic priority mechanism. The scheduling has a great effect on the system performance. Therefore, considering the scheduling can lead to the improvement of the architectures found in the design space.

6.2.5 Energy Consumption Estimation Model

Another significant design quality that should be evaluated is the energy consumption of multi-layer bus architecture. Nowadays, in the SoC industry, the power and energy problem have become significant because high-performance systems produce heat and consume a lot of energy. Finding a low energy design is highly demanded. Therefore, an energy consumption estimation model should be developed.

6.2.6 Acceleration of Design Space Exploration

To further accelerate the exploration of a huge design space, additional techniques can be introduced to the proposed architecture exploration method, such as incremental computation for successive architectures with a small difference, more aggressive pruning, and parallel traversal of the parameter set search tree.

Bibliography

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[2] W. Chan, N. Jaitly, Q. Le, and O. Vinyals, "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition," in *Proceedings of 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2016, pp. 4960–4964.

[3] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *ArXiv:1412.5567 [cs.CL]*, Dec. 2014.

[4] H. Zen and H. Sak, "Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis," in *Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2015, pp. 4470–4474. DOI: 10.1109/ICASSP.2015.7178816.

[5] H. Zen, Y. Agiomyrgiannakis, N. Egberts, F. Henderson, and P. Szczepaniak, "Fast, compact, and high quality LSTM-RNN based statistical parametric speech synthesizers for mobile devices," in *Proceedings of INTERSPEECH*, 2016.

[6] T. Wen, M. Gasic, N. Mrksic, P. Su, D. Vandyke, and S. J. Young, "Semantically conditioned LSTM-based natural language generation for spoken dialogue systems," in *Proceedings of 2015 Conference on Empirical Methods in Natural Language Processing*, Sept. 2015, pp. 1711–1721.

[7] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, "Spoken language understanding using long short-term memory neural networks," in *Proceedings of 2014 IEEE Spoken Language Technology Workshop (SLT)*, Dec. 2014, pp. 189–194. DOI: 10.1109/SLT.2014.7078572.

[8] B. E. Bejnordi, M. Veta, P. J. van Diest, and et al, "Diagnostic assessment of deep learning algorithms for detection of lymph node metastases in women with breast cancer," *JAMA*, vol. 318, no. 22, pp. 2199–2210, 2017. DOI: 10.1001/jama.2017.14585.

[9] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, pp. 115–, Jan. 2017.

[10] J. Powles and H. Hodson, "Google deepmind and healthcare in an age of algorithms," *Health and Technology*, vol. 7, no. 4, pp. 351–367, Dec. 2017. DOI: 10.1007/s12553-017-0179-1.

[11] *IBM Watson Health*, https://www.ibm.com/watson/health/.

[12] Y. Cheng, G. Li, H. Chen, S. X. Tan, and H. Yu, "DEEPEYE: A compact and accurate video comprehension at terminal devices compressed with quantization and tensorization," *ArXiv:1805.07935 [cs.CV]*, 2018.

[13] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *ArXiv: 1612.08242 [cs.CV]*, 2016.

[14] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28*, Curran Associates, Inc., 2015, pp. 91–99.

[15] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *ArXiv:1612.03144 [cs.CV]*, 2016.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, Lake Tahoe, Nevada, 2012, pp. 1097–1105.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ArXiv:1409.1556 [cs.CV]*, 2014.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *ArXiv:1512.03385 [cs.CV]*, 2015.

[19] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

[21] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 4820–4828. DOI: 10.1109/CVPR.2016.521.

[22] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, New York, NY, USA: JMLR.org, 2016, pp. 2849–2858.

[23] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 806–814.

[24] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *ArXiv:1802.05668 [cs.NE]*, 2018.

[25] Y. Choi, M. El-Khamy, and J. Lee, "Universal deep neural network compression," *ArXiv:1802.02271 [cs.CV]*, 2018.

[26] Y. He and S. Han, "ADC: Automated deep compression and acceleration with reinforcement learning," *ArXiv:1802.03494 [cs.CV]*, 2018.

[27] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep3: Leveraging three levels of parallelism for efficient deep learning," in *Proceedings of 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6. DOI: 10.1145/3061639.3062225.

[28] G. Madl, S. Pasricha, L. A. D. Bathen, N. Dutt, and Q. Zhu, "Formal performance evaluation of AMBA-based system-on-chip designs," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, Seoul, Korea, 2006, pp. 311–320.

[29] C. Shih, Y. Lai, and J. R. Jiang, "SPOCK: static performance analysis and deadlock verification for efficient asynchronous circuit synthesis," in *Proceedings of 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2015, pp. 442–449. DOI: 10.1109/ICCAD.2015.7372603.

[30] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using petri nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440–449, Sep. 1980. DOI: 10.1109/TSE.1980.230492.

[31] M. Li, T. Achteren, E. Brockmeyer, and F. Catthoor, "Statistical performance analysis and estimation for parallel multimedia processing," English, *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 105–116, 2010.

[32] Y.-S. Cho, E.-J. Choi, and K.-R. Cho, "Modeling and analysis of the system bus latency on the SoC platform," in *Proceedings of the 2006 International Workshop on System-level Interconnect Prediction*, Munich, Germany, 2006, pp. 67–74.

[33] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo, "Exploiting concurrency for the automated synthesis of MPSoC interconnects," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, 57:1–57:24, Apr. 2015. DOI: 10.1145/2700075.

[34] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proceedings of 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[35] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "NEURAghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs," *ArXiv:1712.00994 [cs.NE]*, 2017.

[36] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, California, USA: ACM, 2016, pp. 26–35. DOI: 10.1145/2847263.2847265.

[37] Accellera Systems Initiative, *IEEE Standard for Standard SystemC Language Reference Manual*,
[Online] Available : http://standards.ieee.org, Jan. 2012.

[38] D. Verkest, K. Rompaey, I. Bolsens, and H. Man, "CoWare–A design environment for heterogeneous hardware/software systems," English, *Design Automation for Embedded Systems*, vol. 1, no. 4, pp. 357–386, 1996.

[39] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada, "RTOS-centric hardware/software cosimulator for embedded system design," in *Proceeding of International Conference on Hardware/Software Codesign and System Synthesis 2004 (CODES+ISSS '04)*, 2004, pp. 158–163.

[40] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya, "Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference 2001 (ASP-DAC)*, Feb. 2001, pp. 63–68. DOI: 10.1109/ASPDAC.2001.913282.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. DOI: 10.1145/2024716.2024718.

[42] B. Kleinert, S. WeiSS, F. Schäfer, J. Bakakeu, and D. Fey, "Adaptive synchronization interface for hardware-software co-simulation based on SystemC and QEMU," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, Prague, Czech Republic, 2016, pp. 28–36.

[43] F. Cucchetto, A. Lonardi, and G. Pravadelli, "A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms," in *Proceedings of 2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2014, pp. 1–6. DOI: 10.1109/VLSI-SoC.2014.7004154.

[44] S. Kyle, I. Böhm, B. Franke, H. Leather, and N. Topham, "Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation," *SIGPLAN Not.*, vol. 47, no. 5, pp. 21–30, Jun. 2012. DOI: 10.1145/2345141.2248422.

[45] O. Matoussi and F. Pétrot, "Loop aware IR-level annotation framework for performance estimation in native simulation," in *Proceedings of 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 220–225. DOI: 10.1109/ASPDAC.2017.7858323.

[46] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang, "Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES 2002)*, Estes Park, Colorado, 2002, pp. 13–18.

[47] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, vol. 4, Apr. 1994.

[48] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[49] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2002, pp. 620–627. DOI: 10.1109/DATE.2002.998365.

[50] A. Bouchhima, S. Yoo, and A. Jerraya, "Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model," in *Proceedings of Asia and South Pacific Design Automation Conference 2004 (ASP-DAC)*, Jan. 2004, pp. 469–474.

[51] S. Yoo and A. Jerraya, "Hardware/software cosimulation from interface perspective," *Computers and Digital Techniques, IEE Proceedings* -, vol. 152, no. 3, pp. 369–379, May 2005.

[52] K. Ueda, K. Sakanushi, Y. Takeuchi, and M. Imai, "Architecture-level performance estimation method based on system-level profiling," *Computers and Digital Techniques, IEE Proceedings* -, vol. 152, no. 1, pp. 12–19, Jan. 2005.

[53] R. Domer, "The SpecC language," in *Tutorial Guide of IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, May 2001, pp. 5.1.1–5.1.12. DOI: 10.1109/TUTCAS.2001.946956.

[54] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proceedings of 2004 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Feb. 2004, 752–757 Vol.2. DOI: 10.1109/DATE.2004.1268966.

[55] D. Kim, S. Ha, and R. Gupta, "CATS: cycle accurate transaction-driven simulation with multiple processor simulators," in *Proceedings of 2007 Design, Automation Test in Europe Conference Exhibition (DATE)*, Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364685.

[56] G. Schirner, A. Gerstlauer, and R. Dömer, "Fast and accurate processor models for efficient mpsoc design," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 15, no. 2, 10:1–10:26, Mar. 2010. DOI: 10.1145/1698759.1698760.

[57] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *Proceedings of 2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2015, pp. 1698–1707. DOI: 10.7873/DATE.2015.1105.

[58] F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, and A. A. Jerraya, "Flexible MPSoC platform with fast interconnect exploration for optimal system performance for a specific application," in *Proceedings of 2006 Design, Automation and Test in Europe (DATE): Designers' Forum*, Munich, Germany, 2006, pp. 166–171.

[59] S. M. Aziz, "A cycle-accurate transaction level SystemC model for a serial communication bus," *Comput. Electr. Eng.*, vol. 35, no. 5, pp. 790–802, Sep. 2009. DOI: 10.1016/j.compeleceng.2008.11.029.

[60] C. K. Lo and R. S. Tsay, "Automatic generation of Cycle Accurate and Cycle Count Accurate transaction level bus models from a formal model," in *Proceedings of 2009 Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2009, pp. 558–563. DOI: 10.1109/ASPDAC.2009.4796539.

[61] A. Baganne, I. Bennour, M. Elmarzougui, R. Gaiech, and E. Martin, "A multi-level design flow for incorporating IP cores: Case study of 1D wavelet IP integration," in *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2003, 250–255 suppl. DOI: 10.1109/DATE.2003.1253837.

[62] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[63] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *Proceedings of 2008 Design, Automation and Test in Europe (DATE)*, Mar. 2008, pp. 9–14. DOI: 10.1109/DATE.2008.4484652.

[64] R. B. Atitallah, S. Niar, S. Meftali, and J.-L. Dekeyser, "An MPSoC performance estimation framework using transaction level modeling," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, 2007, pp. 525–533. DOI: 10.1109/RTCSA.2007.21.

[65] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proceedings of 2008 Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2008, pp. 3–8. DOI: 10.1145/1403375.1403380.

[66] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti, "Transaction-level models for AMBA bus architecture using SystemC 2.0 [SOC applications]," in *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2003, 26–31 suppl. DOI: 10.1109/DATE.2003.1186667.

[67] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai, "A practical approach for bus architecture optimization at transaction level," in *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2003, 176–181 suppl. DOI: 10.1109/DATE.2003.1253825.

[68] G. Schirner and R. Domer, "Quantitative analysis of transaction level models for the AMBA bus," in *Proceedings of 2006 Design Automation Test in Europe Conference (DATE)*, Mar. 2006, pp. 1–6. DOI: 10.1109/DATE.2006.244108.

[69] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based communication architectures at the CCATB abstraction," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, 22:1–22:32, Jan. 2008.

[70] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Using TLM for exploring bus-based SoC communication architectures," in *Proceedings of 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, Jul. 2005, pp. 79–85. DOI: 10.1109/ASAP.2005.65.

[71] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "BMSYN: Bus matrix communication architecture synthesis for MPSoC," *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, vol. 26, no. 8, pp. 1454–1464, Aug. 2007.

[72] M. Li, C. Lo, L. Chen, J. Yeh, and R. Tsay, "A cycle count accurate TLM bus modeling approach," in *Proceedings of 2013 International Symposium on VLSI Design, Automation, and Test (VLSI-DAT)*, Apr. 2013, pp. 1–4. DOI: 10.1109/VLDI-DAT.2013.6533807.

[73] C.-K. Lo, M.-L. Li, L.-C. Chen, Y.-S. Lu, R.-S. Tsay, H.-Y. Huang, and J.-C. Yeh, "Automatic generation of high-speed accurate TLM models for out-of-order pipelined bus," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 1s, 37:1–37:25, Dec. 2013. DOI: 10.1145/2536747.2536759.

[74] H. Javaid, Y. Yachide, S. M. M. Shwe, H. Bokhari, and S. Parameswaran, "FALCON: A framework for hierarchical computation of metrics for component-based parameterized SoCs," in *Proceedings of 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2014, pp. 1–6. DOI: 10.1145/2593069.2593138.

[75] L.-C. Chen, H.-I. Wu, and R. Tsay, "Automatic timing-coherent transactor generation for mixed-level simulations," in *Proceedings of The 20th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2015, pp. 588–593. DOI: 10.1109/ASPDAC.2015.7059072.

[76] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-decoupled parallel SystemC simulation," in *Proceedings of 2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2014, pp. 1–4. DOI: 10.7873/DATE.2014.204.

[77] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-link: Parallel SystemC simulation using time-decoupled segments," in *Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 493–498.

[78] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," in *Proceedings of 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, Jun. 2009, pp. 80–87. DOI: 10.1109/PADS.2009.25.

[79] I. M. Pessoa, A. Mello, A. Greiner, and F. Pêcheux, "Parallel TLM simulation of MPSoC on SMP workstations: Influence of communication locality," in *Proceedings of 2010 International Conference on Microelectronics*, Dec. 2010, pp. 359–362. DOI: 10.1109/ICM.2010.5696160.

[80] N. Ventroux and T. Sassolas, "A new parallel SystemC kernel leveraging manycore architectures," in *Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 487–492.

[81] T. Schmidt, G. Liu, and R. Dömer, "Exploiting thread and data level parallelism for ultimate parallel SystemC simulation," in *Proceedings of 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6. DOI: 10.1145/3061639.3062243.

[82] Y. Lai, C. Chuang, and J. R. Jiang, "A general framework for efficient performance analysis of acyclic asynchronous pipelines," in *Proceedings of 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2015, pp. 736–743. DOI: 10.1109/ICCAD.2015.7372643.

[83] C. Shih, C. Shih, and J. R. Jiang, "Closing the accuracy gap of static performance analysis of asynchronous circuits," in *Proceedings of 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6. DOI: 10.1145/3061639.3062211.

[84] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Proceedings of 2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2012, pp. 641–646. DOI: 10.1109/DATE.2012.6176550.

[85] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Proceedings of 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6. DOI: 10.1145/2897937.2897977.

[86] Y. Cho, G. Lee, S. Yoo, K. Choi, and N.-E. Zergainoh, "Scheduling and timing analysis of HW/SW on-chip communication in MPSoC design," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE): Designers' Forum - Volume 2*, 2003, pp. 20 132–.

[87] C. Lee, S. Kim, and S. Ha, "A systematic design space exploration of MPSoC based on synchronous data flow specification," *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 193–213, 2010.

[88] F. Shafiq, T. Isshiki, D. Li, and H. Kunieda, "A fast trace aware statistical based prediction model with burst traffic modeling for contention stall in a priority based MPSoC bus," *IPSJ Transactions on System LSI Design Methodology*, vol. 9, pp. 37–48, 2016. DOI: 10.2197/ipsjtsldm.9.37.

[89] S. Kim and S. Ha, "System-level performance analysis of multiprocessor system-on-chips by combining analytical model and execution time variation," *Microprocessors and Microsystems*, vol. 38, no. 3, pp. 233–245, 2014. DOI: https://doi.org/10.1016/j.micpro.2014.02.003.

[90] 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, pp. 157–179, Sep. 2005. DOI: 10.1007/s10617-006-9589-4.

[91] F. Shafiq, T. Isshiki, and D. Li, "An accurate and fast trace-aware performance estimation model for prioritized MPSoC bus with multiple interfering bus-masters," *IPSJ Transactions on System LSI Design Methodology*, vol. 10, pp. 13–27, 2017. DOI: 10.2197/ipsjtsldm.10.13.

[92] 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. DOI: 10.2197/ipsjtsldm.5.44.

[93] T. Schmidt, G. Liu, and R. Dömer, "Hybrid analysis of SystemC models for fast and accurate parallel simulation," in *Proceedings of 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 226–231. DOI: 10.1109/ASPDAC.2017.7858324.

[94] M. Takahashi, H. Miyajima, and M. Fukui, "An efficient power and performance evaluation method with reconfigurable bus architecture model," in *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2003)*, Apr. 2003, pp. 345–350.

[95] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 20, no. 6, pp. 768–783, Nov. 2006.

[96] K. Kim and D. D. Gajski, "Trace-driven performance estimation of multicore platforms," in *Proceedings of 2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2014, pp. 627–630. DOI: 10.1109/MWSCAS.2014.6908493.

[97] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Proceedings of 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Jul. 2009, pp. 53–60.

[98] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, Jun. 2010.

[99] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, Jun. 2014, pp. 696–701. DOI: 10.1109/CVPRW.2014.106.

[100] L. Cavigelli and L. Benini, "A 803 GOp/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, 2016.

[101] K. Ando, K. Orimo, K. Ueyoshi, M. Ikebe, T. Asai, and M. Motomura, "Reconfigurable processor array architecture for deep convolutional neural networks," in *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2016)*, 2016.

[102] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proceedings of 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2015, pp. 92–104.

[103] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*, Lille, France, 2015, pp. 1737–1746.

[104] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proceedings of 2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct. 2013, pp. 13–19.

[105] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*, Monterey, California, USA, 2015, pp. 161–170.

[106] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGPLAN Not.*, vol. 49, no. 4, pp. 269–284, Feb. 2014.

[107] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proceedings of 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 609–622.

[108] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[109] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson, "FPGA-based CNN inference accelerator synthesized from multi-threaded C software," in *Proceedings of 2017 30th IEEE International System-on-Chip Conference (SOCC)*, Sep. 2017, pp. 268–273.

[110] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, Toronto, ON, Canada, 2017, pp. 27–40. DOI: 10.1145/3079856.3080254.

[111] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *ArXiv:1604.03168 [cs.CV]*, 2016.

[112] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of 2016 ACM/IEEE 43nd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[113] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proceedings of 2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[114] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proceedings of 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[115] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 553–564. DOI: 10.1109/HPCA.2017.29.

[116] K. Neubauer, C. Haubelt, P. Wanko, and T. Schaub, "Utilizing quad-trees for efficient design space exploration with partial assignment evaluation," in *Proceedings of 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2018, pp. 434–439. DOI: 10.1109/ASPDAC.2018.8297362.

[117] J. Wu, P. Wang, S.-K. Lam, and T. Srikanthan, "Efficient heuristic and tabu search for hardware/software partitioning," *J. Supercomput.*, vol. 66, no. 1, pp. 118–134, Oct. 2013. DOI: 10.1007/s11227-013-0888-9.

[118] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," *SIGPLAN Not.*, vol. 37, no. 7, pp. 18–27, Jun. 2002. DOI: 10.1145/566225.513835.

[119] Z. J. Jia, A. Núñez, T. Bautista, and A. D. Pimentel, "A two-phase design space exploration strategy for system-level real-time application mapping onto mpsoc," *Microprocess. Microsyst.*, vol. 38, no. 1, pp. 9–21, Feb. 2014. DOI: 10.1016/j.micpro.2013.10.005.

[120] Á. Hegedüs, Á. Horváth, I. Ráth, and D. Varró, "A model-driven framework for guided design space exploration," in *Proceedings of 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov. 2011, pp. 173–182. DOI: 10.1109/ASE.2011.6100051.

[121] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, "Adaptive threshold non-Pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs," in *Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 918–923.

[122] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 2013 50th Annual Design Automation Conference (DAC)*, Austin, Texas, 2013, 50:1–50:7. DOI: 10.1145/2463209.2488795.

[123] D. Li, S. Yao, Y. Liu, S. Wang, and X. Sun, "Efficient design space exploration via statistical sampling and AdaBoost learning," in *Proceedings of 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6. DOI: 10.1145/2897937.2898012.

[124] M. Palesi and T. Givargis, "Multi-objective design space exploration using genetic algorithms," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, 2002, pp. 67–72. DOI: 10.1145/774789.774804.

[125] M. Thompson and A. D. Pimentel, "Exploiting domain knowledge in system-level MPSoC design space exploration," *J. Syst. Archit.*, vol. 59, no. 7, pp. 351–360, Aug. 2013. DOI: 10.1016/j.sysarc.2013.05.023.

[126] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/-software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5–32, Jan. 1997. DOI: 10.1023/A:1008857008151.

[127] L. Zhang, C.-Z. Xu, Z. Tian, and T. Li, "Hardware/software partitioning based on greedy algorithm and simulated annealing algorithm," *Journal of Computer Applications*, vol. 33, no. 07, 1898, p. 1898, 2013. DOI: 10.11772/j.issn.1001-9081.2013.07.1898.

[128] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 358–374, Jun. 2006. DOI: 10.1109/TEVC.2005.860766.

[129] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip," *IEEE Trans. VLSI Syst.*, vol. 10, no. 4, pp. 416–422, Aug. 2002.

[130] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 744–749.

[131] K. Ueda, "An embedded system design methodology based on system-level profiling," PhD thesis, Graduate School of Information Science and Technology, Osaka University, 2006.

[132] J. Gong, D. D. Gajski, and S. Bakshi, "Model refinement for hardware-software codesign," in *Proceedings of ED TC European Design and Test Conference*, Mar. 1996, pp. 270–274. DOI: 10.1109/EDTC.1996.494312.

[133] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp. 952–961, Jun. 2004. DOI: 10.1109/TCAD.2004.828127.

[134] S. Pandey, M. Glesner, and M. Muhlhauser, "Performance aware on-chip communication synthesis and optimization for shared multi-bus based architecture," in *Proceedings of 2005 18th Symposium on Integrated Circuits and Systems Design*, Sep. 2005, pp. 230–235. DOI: 10.1109/SBCCI.2005.4286862.

[135] S. Kim and S. Ha, "Efficient exploration of bus-based system-on-chip architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 7, pp. 681–692, Jul. 2006. DOI: 10.1109/TVLSI.2006.878260.

[136] T. van Meeuwen, A. Vandecappelle, A. van Zelst, F. Catthoor, and D. Verkest, "System-level interconnect architecture exploration for custom memory organizations," in *Proceedings of International Symposium on System Synthesis*, Sep. 2001, pp. 13–18. DOI: 10.1145/500001.500005.

[137] Y. Niu, J. Bian, H. Wang, K. Tong, and L. Zhu, "SLCAO: An effective system level communication architectures optimization methodology for system-on- chips," in *Proceedings of 2005 6th International Conference on ASIC*, vol. 1, Oct. 2005, pp. 33–36. DOI: 10.1109/ICASIC.2005.1611263.

[138] T. Seceleanu, V. Leppanen, J. Suomi, and O. Nevalainen, "Resource allocation methodology for the segmented bus platform," in *Proceedings of 2005 IEEE International SOC Conference*, Sep. 2005, pp. 129–132. DOI: 10.1109/SOCC.2005.1554479.

[139] S. Srinivasan, F. Angiolini, M. Ruggiero, L. Benini, and N. Vijaykrishnan, "Simultaneous memory and bus partitioning for SoC architectures," in *Proceedings of 2005 IEEE International SOC Conference*, Sep. 2005, pp. 125–128. DOI: 10.1109/SOCC.2005.1554478.

[140] B. H. Meyer and D. E. Thomas, "Simultaneous synthesis of buses, data mapping and memory allocation for MPSoC," in *Proceedings of 2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2007, pp. 3–8. DOI: 10.1145/1289816.1289822.

[141] L. Chiou, Y. Chen, and C. Lee, "System-level bus-based communication architecture exploration using a pseudoparallel algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 8, pp. 1213–1223, Aug. 2009. DOI: 10.1109/TCAD.2009.2021733.

[142] L. B. S. Murali and G. D. Micheli, "An application-specific design methodology for on-chip crossbar generation," *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, vol. 26, no. 7, pp. 1283–1296, Jul. 2007.

[143] ARM, *Multi-layer AHB overview*,
[Online] Available : http://infocenter.arm.com, 2004.

[144] ARM, *AMBA AXI and ACE protocol specification*,
[Online] Available : http://www.arm.com/products/system-ip/amba/amba-open-specifications.php, 2011.

[145] S. Pasricha, Y. Park, F. J. Kurdahi, and N. Dutt, "CAPPS: A framework for powerperformance tradeoffs in bus-matrix-based on-chip communication architecture synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 2, pp. 209–221, Feb. 2010. DOI: 10.1109/TVLSI.2008.2009304.

[146] G. Lee, Y. Ahn, S. Lee, J. Son, K. Yoon, and K. Choi, "Communication architecture design for reconfigurable multimedia SoC platform," *Design Automation for Embedded System*, vol. 14, no. 1, pp. 1–20, Mar. 2010.

[147] Y.-P. Joo, S. Kim, and S. Ha, "Efficient hierarchical bus-matrix architecture exploration of processor pool-based MPSoC," *Design Automation for Embedded Systems*, vol. 16, no. 4, pp. 293–317, Nov. 2012. DOI: 10.1007/s10617-013-9110-9.

[148] M. Jun, D. Woo, and E.-Y. Chung, "Partial connection-aware topology synthesis for on-chip cascaded crossbar network," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 73–86, Jan. 2012. DOI: 10.1109/TC.2010.211.

[149] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proceedings of 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6. DOI: 10.1145/2897937.2898002.

[150] H. Hong, H. Oh, and S. Ha, "Hierarchical dataflow modeling of iterative applications," in *Proceedings of 54th Annual Design Automation Conference 2017 (DAC)*, Austin, TX, USA: ACM, 2017, 39:1–39:6. DOI: 10.1145/3061639.3062260.

[151] F. Tsimpourlas, L. Papadopoulos, A. Bartsokas, and D. Soudris, "A design space exploration framework for convolutional neural networks implemented on edge devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2212–2221, Nov. 2018. DOI: 10.1109/TCAD.2018.2857280.

[152] ARM, *AMBA specification (rev 2.0)*, [Online] Available : http://infocenter.arm.com, 1999.

[153] ARM, *ARM AMBA5 AHB protocol*, [Online] Available : http://infocenter.arm.com, 2015.

[154] ARM, *AMBA design kit : Technical reference manual*, [Online] Available : http://infocenter.arm.com, 2013.

[155] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of 2006 43rd ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 274–279.

[156] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proceedings of the 14th European Conference on Computer Vision (ECCV 2016)*, 2016, pp. 21–37.

[157] *Intel programmable acceleration card with intel arria 10 gx fpga*, www.altera.com/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html.

[158] *Nallatech 510t compute acceleration card*, www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card.

[159] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, Apr. 2017.

[160] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *ArXiv:1408.5093 [cs.CV]*, 2014.

[161] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. DOI: 10.1145/1498765.1498785.

[162] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *Proceedings of 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2016, pp. 575–580. DOI: 10.1109/ASPDAC.2016.7428073.

[163] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, "Curbing the roofline: A scalable and flexible architecture for CNNs on FPGA," in *Proceedings of the ACM International Conference on Computing Frontiers (CF'16)*, Como, Italy, 2016, pp. 376–383. DOI: 10.1145/2903150.2911715.

[164] P. Zarkesh-Ha, J. A. Davis, and J. D. Meindl, "Prediction of net-length distribution for global interconnects in a heterogeneous system-on-a-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 6, pp. 649–659, Dec. 2000. DOI: 10.1109/92.902259.

[165] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. DOI: 10.1109/5.726791.