High-Level Synthesis for Efficient Accelerator Design

by

Sung Jin Kim

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 05/06/2015

The dissertation is approved by the following members of the Final Oral Committee:
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
Mikko Lipasti, Professor, Electrical and Computer Engineering
Kewal Saluja, Professor, Electrical and Computer Engineering
Azadeh Davoodi, Associate Professor, Electrical and Computer Engineering
Cristian Estan, Software Engineer, Google

ACKNOWLEDGMENTS

I would like to thank everyone who helped to complete my dissertation. Without the guidance of my committee members, help from friends, and support from my wife and family, I would never have finished my dissertation.

First, I would like to thank my advisor, Dr. Karu Sankaralingam, who had continuously encouraged me to study and research. Without your excellent guidance, caring, and patience, I would have never completed my research.

I would like to thank all committee members: Dr. Mikko Lipasti, Dr. Kewal Saluja, Dr. Azadeh Davoodi, and Dr. Cristian Estan. Dr. Lipasti pushed me to rethink the evaluation methodology when I was in trouble to analyze the result. Dr. Saluja and Dr. Davoodi were great listener and gave me many inspiring comments for my research. I especially thank to Dr. Estan for giving me numerous advices at the beginning of my research.

I also thank to the Vertical group. In particular, Dr. Chen-Han Ho, who was my collaborator for the research, gave me significant insights and thoughts to achieve my research. I have learned many skills to proceed my research. Thanks to Lorenzo De Carli for the great help and discussions when I have had much trouble for my research. Amit, Vijay, Tony, Newsha, Vinay, and Clint: thank you for your feedback and support.

Thanks to the faculty of Chung-Ang University. Dr. Joonki Paik, who was my advisor for master degree, have encouraged me to study abroad. Without your advice, I could never decided and be prepared to be attend to a PhD program. Also, I appreciate my friends in my country. Shichang gave me valuable comments and taught me the skill to utilize tools that I need. Tae-keun continuously provided new technonlgies and information, and it was useful for pursuit of my research. Aram, Jihong, Kyungwon, and Hyungsoo always stood by me though the good time and bad time.

Finally, I would like to thank my parents and sisters. They were always supporting me and encouraging me with their best wishes. Especially, I would like to appreciate my wife, Hyun A. She was always there cheering me up and my strongest support.

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

For several decades, the research of general-purpose microprocessors has been central to the organization of efficient computing systems. However, computer architects face the limitations of innovation and research alternative techniques to keep improving computing systems. Today, a custom hardware implementation and specialized accelerators are becoming a widely-used approach to overcome these limitations. While hardware accelerators are useful for efficient computing, its entry barrier due to design complexity is higher than software implementation for a general-purpose microprocessor. High-Level Synthesis (HLS) is a well-known custom hardware design base on the software approach to relaxed design complexity.

In this work, we propose new HLS approaches to generate hardware accelerators, depending on the characteristics of the implemented software: SoftWare Synthesis for network Loop (SWSL) and cache-coherent High-Level Synthesis (ccHLS). Depending on the properties of the target software, the implementation method should be different and it requires different HLS techniques to meet the requirement of these goals. Data structure lookups are among the most expensive operations on a router and its function is to achieve high throughput with low latency and power to support massive input packet processing. Hence, accelerators from HLS for network lookup should consider ways to minimize latency and maximize throughput. SWSL accepts a specialized programming model for a specialized architecture called PLUG and retains its simple programming model. SWSL generates entire lookup chains performing aggressive pipelining to achieve high throughput.

On the other hand, conventional programs normally focus on loops as accelerating regions which requires more computation performance than non-accelerating regions. However, this approach requires the host processor to execute non-accelerating code region and the accelerator's memory as cache because the host processor and accelerators are running separately. Thus, the execution model of most accelerators is 1) read data from the accelerator's memory to inject into the accelerator hardware and 2) write data to the accelerator's memory to update the computation result. Its mechanism leads to some cache coherency between the cache in a host processor and the accelerator's memory. In addition, it is not cost-effective because it requires additional memory for accelerators. ccHLS generates fixed-function accelerators sharing data cache with the host processor starting with C/C++ source code. It is basically organized as two accelerators following a decoupled access/execute model, which are the fixed-function compute accelerator and the fixed-function memory access accelerator. The former has a task to speed up computation of a target source code and the latter has the role of data movement including cache accesses in the host processor directly. The fixed-function memory access accelerator is organized by an address generation unit for address computation and FSM to control memory access. The host processor, except the data cache, is turned off during acceleration.

In this dissertation, we observed following: First, SWSL gives 2 ~4× lower latency and 3 ~4× reduced chip area with reasonable power consumption compared with a previously proposed solution. Second, ccHLS provides reasonable factors by using fixed-function accelerators from software implementation of applications. We expect cost-effective and performance/energy improvement design for acceleration. Based on our simulation result, compared to 2-wide and 4-wide out-of-order processors, we observe 2.55× and 1.57× speedup and 3.96× and 7.20× energy reduction.

1 INTRODUCTION

By increasing the number of transistors on a chip, as predicted by Moore's Law [59], computing systems have flourished in the 20th century. In terms of performance with Moore's Law, computer architects have consequently reached the stage to achieve the best performance by technical innovation. Pipelining in the microarchitecture allows the system to achieve a high clock frequency. Branch predictors and out-of-order execution lead the microprocessor to speculative execution and dynamic scheduling for performance improvement. However, the breakdown of Dennard scaling [26], and increasing the power per unit area in CMOS technology, prohibits a single processor core from being advanced. Due to the key effect of Dennard scaling, the power density is constant if transistors get smaller and supply voltage scaling has made it possible to use abundant transistors for the circuit design because threshold voltage is able to be scaled proportionally. However, the ability to scale the threshold voltage for reliable circuit operation has been limited by increasing current leakage to unacceptable levels, and reducing capacitance is the only way to maximize clock frequency. Without scaling capacitance, higher power/energy is consumed for higher performance computing. Thus, the trend of microarchitecture research has shifted from a single core processor to accelerators to save power and energy while achieving high performance.

The most promising option to achieve efficiency by high performance and low power consumption is to utilize customization which leads to reducing power waste when a general-purposed processor runs and accomplishes the same task with less work. In addition, the customized hardware for a specific application is able to improve performance with low power consumption, which allows the system to achieve greater energy efficiency. As a result, application-specific integrated circuits (ASICs) and using reconfigurable architecture for specific tasks are an effective way to maximize efficiency. This approach is called hardware accelerators to specialize specific workloads of tasks. Two types of accelerators are often used: reconfigurable accelerators that trade hardware complexity for generality; or fixed-function accelerators that minimize the energy envelope. First of all, reconfigurable architectures such as FPGAs have the following advantages:

- **Faster time-to-market:** Reconfigurable architecture does not require other manufacturing steps for design. Commercial FPGAs provide the ability to burn user-implemented HDL code. So, it makes target applications respond quickly in fast changing markets.
- **No Non-Recurring Expenses (NRE):** Reconfigurable architectures are typically cheaper than ASICs design because vendors of reconfigurable architecture provide tools, and no additional cost to design is expected.
- **Reusability:** Reusability of reconfigurable architecture is the main advantage. Prototype of the design is implemented on reconfigurable architectures and has verified the accuracy of tasks. In the case that the design has faults, the user can reimplement the HDL code and generate bitstream. This generated bitstream is programmed into reconfigurable architectures and can be used to test again.

On the other hand, fixed-function accelerators have the following advantages:

- **Cost:** Compared to reconfigurable architecture, fixed-function accelerators promise lower cost than reconfigurable architecture.
- **Performance:** Fixed-function accelerators give design flexibility. This gives enormous opportunity for speed optimization.
- **Low power:** Fixed-function accelerators can be optimized for required low power. There are several low power techniques such as power gating, clock gating, multi vt cell libraries, and pipelining are available to achieve the power target. This is where reconfigurable architecture fails.

Table 1.1 describes the comparison between reconfigurable architectures and fixed-function accelerators. In energy critical platforms such as mobile devices, fixed-function accelerators are widely adopted because of better overall benefit in energy. However, they still have a disadvantage in terms of design time compared to the software approach running on general-purpose processor. According to Rowan et al. [63], there is a major dilemma between hardware logic design and using a general-purpose processor. With the former, it is possible to create

	Reconfigurable architecture	**Fixed-function accelerators**
Faster time-to-market	Good	Worse
Design cost	High	Low
NRE	Low	High
Reusability	High	Poor
Performance	Low	High
Power	High	Low
Design time	Worse	Worse

Table 1.1: Comparison: Reconfigurable architecture vs. Fixed-function accelerators

smaller and faster than the software approach because of parallel execution in hardware, and the latter guarantees more flexible and easier implementation than hardware design.

Despite the huge potential of fixed-function accelerators, programming with hardware description languages (HDLs) to design accelerators is significantly harder than programming of software using high-level languages for the general-purpose processor. First, the characteristics of programming are totally different. While the software approach is based on sequential execution in a simple processor core, programming for accelerators using HDLs tries to exploit parallel execution. This difference makes users confused. Second, the semantic gap between software and HDLs is enormous and makes error-prone results, so users have to master two program languages. Finally, users must understand the low-level details of hardware design and deal with those details to map into hardware. Therefore, numerous reasons prevent fixed-function accelerators from being widely used and have an influence on increasing design time. Automatic compilation from the software written in high-level program languages such as C or C++ to HDLs, which is called High-Level Synthesis (HLS), is widely believed to be an alternative approach to create efficient fixed-function accelerators with a short amount of design time. In this thesis, I describe my efforts to understand and create HLS compilers. From the specific to general applications, I show how accelerators are generated and achieve efficiency. First, I describe the HLS compiler that supports specific application for network lookup. Then, HLS for generalized applications is described. I then briefly outline my contributions in this work and summarize this chapter with an overview of this thesis.

1.1 High-Level Synthesis

Implementing functionality in software differs significantly from deploying the same functionality as a hardware accelerator. In the former approach, algorithms are expressed in a high-level programming language which is compiled for the target platform. In the latter approach, the developer specifies an implementation of the functionality using a hardware description language (HDL) such as Verilog or VHDL. Such description is then synthesized to hardware. The two approaches present different trade-offs in several important aspects.

- **Performance:** An HDL implementation can achieve higher parallelism compared to a high-level programming language. Without multi-thread programming, a microprocessor can only execute sequentially. However, the logic in the HDL contains only the data path and simple control logic elements and these elements can be easily arranged to run in parallel for high throughput. Moreover, in hardware much functionality can be assembled to run in a limited number of clock cycles. For these reasons, an HDL design can be more than one hundred times faster than software running on a microprocessor [63].

- **Efficiency:** There are several reasons why the HDL approach can achieve superior efficiency in system design. Because hardware can perform the same computations in fewer cycles, FPGAs and ASICs can often run at a lower frequency than a microprocessor. Therefore, they tend to consume in tens of watts, while microprocessors consumes more. Moreover, an HDL design provides less architectural overhead, since – contrary to microprocessors – it does not include potentially unnecessary components [1]. According to [63], hardwired logic can be more than one hundred times smaller than an equivalent software implementation.

- **Design Time:** Directly designing using an HDL is notoriously complicated and painful. The designer must take into account both the high-level algorithm and low-level hardware considerations, resulting in a complex, error-prone implementation. According to Kathail, design and verification effort varies depending on the implementation approach [43]. Implementations must go through several steps, with the first being a reference software-only implementation and the final being the HDL logic. The HDL implementation alone consumes on average several engineering years. Conversely, software-based approaches allow rapid prototyping and optimization.

- **Debugging and Verification:** High-level programming languages enable rapid prototyping and debugging. Hardwired design with HDL does not provide easy debugging methods, so engineers rely on the output signal from simulators like Modelsim. This is one of the primary reasons why software is chosen over hardwired logic, even though the latter gives superior performance and efficiency.

High-level Synthesis (HLS) is a set of techniques for generating HDL from algorithms expressed in a high-level programming language (typically C or C++). The goal is to achieve the performance and efficiency of the hardware approach and the ease of design/verification of software.

1.2 Building Accelerators from Software

In this thesis, two major approaches to generate accelerators from the software are proposed. One is to seek to figure out how to automatically create hardware accelerators from network lookup programs for the specialized architecture, which is called Pipeline LookUp Grid (PLUG). It has the specialized programming model with provided APIs and its programming model is well-matched to be used for compilation input of the first proposed HLS. The other is to build the HLS tool for the normal application programs which is guaranteed generalization. Previous approaches are only adapted for network lookup application, hence generalized HLS techniques is required for conventional programs.

1.2.1 Building Accelerators for a Specific Application - Network Lookup

In this thesis, for the network lookup specialization, the HLS for the network lookup called SoftWare Synthesis for network Lookup (SWSL) is proposed to reduce time and effort for design. The programming model which is used for source code for SWSL follows that of PLUG. Basically, PLUG is the tile-based architecture and each tile has 32 simple in-order cores (μCore), 6 routers for communication with adjacent tiles, and SRAM for lookup data structure. The architecture of PLUG is based on Von-Neumann structure with programming of simple code block. Thus, it is closer to a general-purpose processor approach. However, it is not a fully general-purpose processor because it is an application-specific processor specialized for data lookup. Due to this reason, PLUG has multiple special instructions for lookup engine operations and its compiler has unique properties to support specialized applications with high performance and efficiency. Network algorithms or applications running on PLUG are based on data flow graph (DFG), which consists of logical pages with connecting edges between them. The logical page has partitioned workloads depending on lookup data structure and these partitioned workloads are called code block and APIs are provided for special purpose operations. Logical pages in DFG are assigned into PLUG tiles and the code block in the logical page is executed in the free μCore.

The main role of SWSL is to generate an accelerator hardware description by directly using PLUG code blocks as source code while retaining DFG properties of each network lookup application. From SWSL, each code block is represented as the hardware module and those modules organize top level module of the logical page in DFG. The module for the code block has the shape of pipelines. Thus, it allows lookup requests to be pipelined and achieve higher throughput without stall by the lack of free μCores in PLUG.

1.2.2 Building Accelerators from Generalized Applications

In SWSL, it only focuses on generating a specialized accelerator - network lookup. SWSL compiler framework is designed for PLUG code blocks as compilation source code and induces the lack of generality. In most cases, a host processor executes non-accelerating region and HLSs parallels accelerated region with multiple hardware devices that do not have data dependencies. Conventionally, HLS is proven to be capable of targeting dataflow with fixed latency scratchpad memory, but struggles managing control-flow and varied latency memory operations. The issues in interface are even more complicated; programmers have long been developing applications assuming a unified memory. Specialized memory hence requires a different management mechanism, which introduces complexity and inefficiency. I believe future fixed-function accelerators will need to be cache coherent with the host processor, which existing HLS approaches cannot handle. This thesis proposes such a cache-coherent High-Level Synthesis framework that offers performance improvement and power reduction. The Execution model is based on Decoupled Access/Execute (DAE) model, which partitions workload as computation and memory access and runs them in parallel, and generates two accelerator components: compute accelerator and memory access accelerator. Generated accelerators utilize data cache in the host processor and turn off the host processor's pipeline while accelerated regions are executed. In order to organize the DAE model, this thesis proposes the Extended Access/Execute Program Dependence Graph (E-AEPDG) for a novel HLS implementation. The E-AEPDG provides memory access with control dependencies as well as computation core insights of the accelerated region. Provided memory access and computation are transformed as accelerators and interaction between two accelerators is feasible by simple finite state machine (FSM). By falling into deep sleep of the host processor and accelerating regions from accelerators, the proposed HLS guarantees good performance improvement and power reduction.

1.3 Contributions

This thesis investigates and designs compilation techniques of new HLSs for hardware accelerator generation. Mainly, two approaches will be proposed, thus this thesis classifies contributions in terms of specialized and generalized target applications.

HLS for specialized applications SWSL consists of a lookup programming API and a specialized compiler middle layer that generates efficient lookup hardware logic from software. This approach improves the hardware/software development cycle in various ways. First, SWSL is architecture-neutral: lookup implementations are valid C++ that can be compiled for a conventional CPU, or fed to the SWSL compiler and deployed as FPGA or ASIC. The former approach retains flexibility, while the latter prioritizes performance. Thus SWSL eases code reuse. Moreover, software and hardware designs are consolidated: design and verification can be done efficiently in a high-level programming language, reducing the need for a separate hardware verification cycles. Power and area usage are limited, as the specialized logic implements exactly the functionality needed by the application.

The SWSL programming model (§ 2.1.3) is dataflow-based and naturally exploits pipelining opportunities present in lookup algorithms [24]. While most high-level synthesis compilers focus on latency improvement through loop acceleration, SWSL leverages the simple, acyclic nature of lookup programs to generate hardware logic for the entire lookup algorithm, performing aggressive pipelining to achieve high throughput. One of the main challenges in synthesizing hardware from software is that the latter is inherently sequential, offering limited opportunities for concurrent execution. SWSL employs optimizations to uncover concurrency at basic-block level (§ 3.1.2), and increases parallelism through the use of *variable lines*. Each line maintains an independent copy of the execution state of the program, allowing multiple executions to proceed independently in parallel.

HLS for generalized applications We believe future fixed-function accelerators will need to be cache coherent with the host processor, which existing HLS approaches cannot handle. This thesis develops such a cache-coherent High Level Synthesis (ccHLS) framework that offers performance improvement and power reduction.

- Proposing ccHLS, a cache-choerent High Level Synthesis flow for energy efficient fixed function accelerators.

- Designing a fixed-function memory access accelerator architecture based on Finite State Machines and compound address generator.

- Implementing a compiler producible fixed-function computation accelerator micro architecture based on various dataflow computing literature.

- Developing a ccHLS compiler that generates fixed-function memory and computation accelerators based on Decouple Access Execute model.

1.4 Thesis Organization

Chapter 2 discusses the motivation and background of building a specialized HLS for the network lookup and HLS for generic programs. In chapter 2, it addresses the compilation of PLUG, which is mainly used to create Software Synthesis for Network Lookup (SWSL) HLS and surveys other state-of-the-art HLS. Then, it describes the proposed HLS techniques for generic programs. In chapter 3, the major efforts to construct SWSL are presented. Chapter 4 describes compilation techniques to support generalized applications. In chapter 5 and 6, experiment setup and methodology are described. In addition, they show the evaluation of PLUG that is the baseline of SWSL compilation first; then, it evaluates network lookup accelerators generated from SWSL. Finally, accelerators from ccHLS to support generalized applications are evaluated. In the last chapter, I conclude and summarize this thesis.

2 MOTIVATION

This chapter discusses the background of proposed High-Level Synthesis (HLS). Two phases to support specialized and generic applications are described. In the first phase, the motivation for generating hardware logic from the program for a specialized architecture, pipeline lookup grid, called PLUG [47, 24]. It intends to lookup the data structure for a network lookup and its program and programming model are formed as the hardware accelerator from the proposed HLS. In section 2.1, the concept of lookup for network, the PLUG architecture and compilation, and its programming model are presented. In addition, the overview of a novel HLS for the specialized program, SoftWare Synthesis for Network Lookup (SWSL), is described. In the second phase, we discuss another proposed HLS for the generic program. It is designed to accelerate loops in the conventional source code. In section 2.2, the motivation for generating accelerators for the conventional loop and its execution model is shown.

2.1 Generating accelerators from the specialized program

In this section, I discuss the motivation of a novel HLS for the specialized program of network lookup. In computing, a lookup means to search a data structure for a certain piece of data. The lookup schemes vary depending on the purpose of applications and algorithms; the data for the lookup request can be structured by in many ways. Most commonly used in software area is hash. Using the hash key from the hash function, a program may find the data structure for its own objective. Due to its popular utilization, the hardware area is also delving into achieving high performance lookup engines. Lookups for network algorithms, which are the main topic of this document, require a high performance lookup engine attaining low latency and high throughput. It is because network algorithms such as Ethernet, IPv4, and IPv6, frequently lookup the data structure to find the next hop or destination. Each network algorithm of protocols has its own packet structure and a computing system constructed by a microprocessor or a specialized hardware design lookups a data structure to decide the next hop of a packet. To attain high packet rates, a network device such as a router requires a high performance lookup engine for low latency and high throughput. In addition to high performance, a lookup engine should ideally be flexible. Even if the lookup engine has high throughput, it may be inefficient to develop different hardware for various network algorithms. Therefore, architects designing efficient lookup engines have to consider not only the architectural flexibility, but also design efficiency to meet the requirements. Section 2.1.1 disscusses more details of network lookup.

2.1.1 Network Lookup

The fundamental task of switches and routers is to determine next-hop information (e.g. an outgoing port number) given some packet data, such as a layer 2 or 3 address, or the connection 5-tuple. Making a forwarding decision usually requires searching large data structures of various kinds, depending on the specific algorithm. For example, layer-3 routers use the IP destination address to search IP routing tables; OpenFlow [57] and its predecessor Ethane [19] look up per-flow rules in a tables index by layer-3/4 headers. The operation must be performed at line speed and for each incoming packet. As lookups are among the most expensive operations on packets' critical path in terms of latency and power, there is a large body of research, both in academia and industry, on implementing them efficiently.

Software approaches are problematic because network lookups are known to suffer from poor locality: multi-Gigabit routers process packets from tens of thousands of unrelated flows, limiting the effectiveness of caching. Algorithmic lookups approaches rely on specialized data structures, with the aim of minimizing the number of memory references per lookups and the forwarding table size. Examples include [14, 25] for IP lookups and [69, 77] for packet classification. However, because general-purpose CPUs are not optimized for lookup tasks, they cannot sustain throughput requirements of large routers.

Ternary content-addressable memory (TCAMs) is a popular hardware-based approach. TCAMs can concurrently compare a search key (including wildcards) with all the entries of a table, implementing hardware bit-level parallelism. Their high throughput comes at a cost: TCAMs are expensive, have low storage density and high power consumption. A different approach consists of implementing algorithmic approaches as fixed-function hardware (e.g. [35, 42, 48, 73]). Such hardware is based on inexpensive RAM and can achieve performance comparable to TCAMs with better power and area usage. However, deploying such designs as

ASICs require a long and expensive design and verification cycle, and is hardly flexible.

Significant research effort has focused on flexible lookup engines, with the goal of achieving throughput comparable to ASICs while retaining some programmability. Several proposals use GPUs as packet processing engines [37, 60]. Neither GPUs nor their programming models are optimized for the task; therefore, these approaches have seen little adoption outside academia.

Other lookup engines leverage the observation that –despite their heterogeneity – hardware lookup implementations do have common aspects [24] [23]. Proposals such as [12, 38, 47, 49] provide hardware implementation of functions typically used by lookup algorithms, arranged in a configurable architecture. Similar to GPUs, these engines use specialized programming models; lookup implementations are hardware-specific and cannot be used on different platforms.

We argue that the specialized engine approach does not avoid two significant pitfalls of ASICs: hardware verification costs and excessive software specialization. According to [4, 67], the main cost factors in hardware development are design/verification (∼40%) followed by developing the companion software (∼30%). Designing lookup engines still requires complex and costly hardware/software codesign; lookup algorithms developed for a platform are highly specialized and cannot be ported to different architectures.

2.1.2 Lookup Engine Designe Space

Before detailing the methodology of a system design for lookup engines, a design configuration is briefly described in this subsection. For many decades, the earlier lookup approaches have aimed either at flexibility or at efficiency for the specific algorithms. In general, these approaches are based on either software for general-purpose processors or hardwired designs. Figure 2.1 shows the design space in the perspective of the architectural flexibility and efficiency.

Software implementation with general-purpose processors gives high flexibility when functionality or algorithms are changed. It is easier and cheaper to modify software than hardwired logic design. Modification only requires to modify programs because the compiler regenerates the executable binary for general-purpose processors. While software on general-purpose processors retains high architectural flexibility, it does not guarantee hardware efficiency because this approach intends to execute a software algorithm under general-purposed processors,

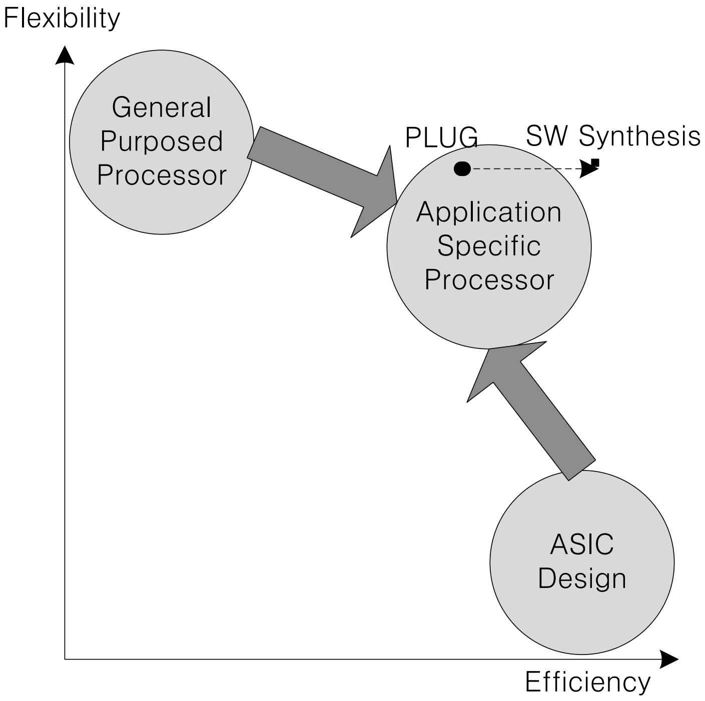


Figure 2.1: Design Space of Lookup Engine

which have unnecessary logic design without relation to pure software algorithm execution.

On the other hand, hardwired designs such as ASICs give more hardware efficiency. Because ASICs are hardwired and fabricate on silicon, parallelism, energy/power consumption, and performance will be taken into account from the initial design step. Thus, hardwired designs do not consider software implementation, but RTL-level design for its design efficiency achieves high performance and efficient design. However, hardwired designs limit the programming flexibility because its fixed design to operate given tasks does not allow engineers to modify the functionality.

A hybrid design that can overcome limitations of software-centric and hardwired designs is a specialized approach for application-specific execution. In terms of software engineers, application-specific processors are able to run software by modifying software to utilize specific designs to achieve more efficiency. Because general-purpose processors are not optimized for a specific application to support generic software, application-specific processors give optimized output result due to their specialized structure. This is made possible by specific compiler techniques such as extracting hot-spot in programs or partitioning the workload of applications to utilize specialized design. Hardware engineers, on the other hand, shift from hardwired design to application-specific hardware blocks. It is similar to application-specific processors, but specialized module designs are based on configurable blocks in hardware. Simple programming for its configuration is allowed to increase the flexibility.

Therefore, the trend of a system design is to integrate software and hardware approaches. So, we can consider lookup design in terms of these approaches. PLUG is closer to the software approach with general-purpose processors. It is a tiled structure which has Von-Neumann such as processors, so it is easy to achieve flexibility by implementing software. Moreover, the workload of applications is easily partitioned and assigned to architecture, so it also achieves hardware efficiency. Details of PLUG architectural properties and the compiler are described in appendix A as completed work.

2.1.3 SoftWare Synthesis for Network Lookup (SWSL)

PLUG is well-supported architecture for the network lookup and its programming model which includes the concept of data flow graph and code block. However, the conventional HLSs are not well-matched with PLUG applications for generating lookup accelerators. There are several reasons that they are improper. First, no profiler requires the generation of verilog code. As shown in compilers for fine-grained reconfigurable architecture, most HLSs need the profiler to find the kernel region in the program. PLUG applications have separate code blocks to be compiled and they can be directly compiled without profiling data. Second, applications which handle massive input streams are unsuitable for those compilers. Target applications in PLUG require the capability to run all input messages from a host processor. Because each input message runs on a μCore in PLUG, PLUG has 32 μCores per tile. These μCores lead to high throughput by assigning each task requested by each input message. However, most HLSs compilers only focus on the kernel region in program because they intend to improve performance in that region. Thus, they do not consider handling massive input streams. Third, the use of FSM to control hardware logic is not allowed. As aforementioned, PLUG applications must have the capability to handle massive input streams for network lookup. This property prohibits the HLSs from generating FSM because no input message can be proceed until control state changes to idle. SoftWare Synthesis for Network Lookup (SWSL) is a high-level synthesis compiler to compile code blocks in PLUG applications. Hence, SWSL should include following requirements:

- **Requirement 1.** Generated hardware logic must handle a lookup input message at every clock cycle to achieve equal throughput to system clock cycle.
- **Requirement 2.** Maximum parallelism must be exploited for high speed lookup.
- **Requirement 3.** Target source code for the SWSL is code blocks of PLUG network algorithm applications.

Requirement 1 is obvious because crucial role of network routing is to obtain high throughput and high throughput allows a router or switch to achieve data lookup of a number of packets. For this purpose, SWSL generates hardware logic output as pipelined structure to support coming input messages at every clock cycle. This allows PLUG applications to avoid a number of μCores. Requirement 2 is straightforward because the objective of using hardware design is to evade sequential execution of a microprocessor. Therefore, SWSL tries to eliminate data dependency by using many operators. In Requirement 3, PLUG code block with its framework is well-matched with compiler implementation for SWSL. Firstly, implementation of code block for lookup algorithm is based on a high-level programming language C++. Secondly, we have already implemented front-end compiler, it can be reused without any modification for back-end compiler of SWSL. Finally, network algorithms using lookup processing are partitioned as many code blocks in the data flow graph of applications, thus it is easy to modularize partitioned workloads as verilog code.

SWSL can reuse the front-end of a PLUG compiler because the code block source code itself is not modified. So, the major implementation part of software synthesis is the back-end compiler for generating hardware logic design.

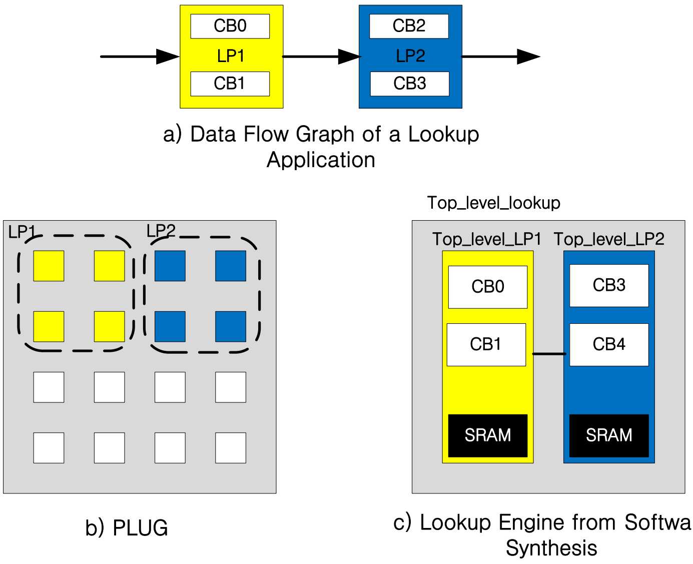


Figure 2.2: Overview of Lookup Engine Design using SWSL

2.1.4 Execution Model of Network Lookup Accelerator from SWSL

The target source code for SWSL is a PLUG application; generated logic design must follow the data flow graph and code block properties of PLUG application. Hence, the main role of SWSL is to generate verilog files for logic design executing the same property as that of PLUG applications. Figure 3.1 shows the overview of a lookup engine using SWSL from PLUG applications. Figure 3.1 a) represents the data flow graph of a simple lookup application. It is organized into two logical pages and each of which has 2 different code blocks. Figure 3.1 b) shows the assignment of logical pages in PLUG tiles. Each code block in the logical pages embeds in instruction memory and a free μCore runs code. In figure 3.1 c), the lookup engine design of the target application from SWSL is shown. From the compiler, modules of each code block are generated and they construct the top level of the logical page with a SRAM module. At the lookup engine top level, top level designs of logical pages are combined.

The structure of SWSL is quite similar to that of a PLUG logical page combined with multiple tiles. In PLUG architecture, compiled code blocks are loaded in instruction memory and each input message is assigned to a free μCore as a task. The input message runs the sequence of instruction code in an assigned μCore to execute its target code block. The lack of free μCores limits increased throughput, thus a number of μCores in the tile guarantee high throughput because they can handle input messages without any message stall. However, the structure of PLUG is an inefficient design due to the number of μCores in each tile and it degrades hardware efficiency while it gives flexibility and high throughput. Instead of using μCores on the tile, the proposed lookup engine intends to integrate the code block modules generated from SWSL and input messages are routed to the target code block module. From the requirements in section 2.1.3, each code block module from SWSL has a data path with pipeline structure for massive input streams. It means that it does not require a number of μCores and code block modules to handle all input messages because a code block module is able to operate current input messages, even if previous input messages are operating in the same code block module. From the design and operation of SWSL, two advantages of SWSL are predicted. One is that SWSL gives high hardware efficiency and performance because it eliminates μCores and integrated code block module design, which target applications needs. So, it massively increases the efficiency and performance compared to a Von-Neumann style application processor. Another is that the data flow graph and code block source code in PLUG applications are able to be directly used without any modification because application design decides the data path that it requires. It has a positive influence on pursuing high flexibility of lookup engine design.

2.2 Generating Accelerators for Loops in Generic Program

This section describes the way to generate accelerators in generic programs for loop acceleration. SWSL is a specialized HLS that generate the lookup accelerator from the specialized programming model. While it is useful for a special purpose for lookup with programming constraints, generic programs following conventional processor architecture are not feasible. In this section, thus, we discuss a new HLS to support conventional programs.

2.2.1 Motivation for HLS and Fixed-Function Accelerators

Table 2.1 shows the classification of accelerators. Custom-RTL, which is static specialization for acceleration, is efficient in terms of power and performance. However, its design cost is extremely high, it usually focuses on computation only, and since design time is also high, it is rarely used

Accelerator			**Specialization strategy**	**memory strategy**	**Benefits**	**Drawbacks**
Custom-RTL			Static	Specialized	All Power, Perf	Generality, Design Cost
DSP cores in SoC			Dynamic	Specialized	Power, Perf	Memory Generality
DySER			Dynamic	Coherent	Perf	Power
HLS	Soft-Core	C2H, Warp, LegUp, CGR	Static	Specialized	All Power, Perf	Memory Generality
	Non-soft-core	Trident, GARP	Static	Specialized	Power, Perf	Complexity
		C-Core	Static	Coherent	Power	Perf
ccHLS			**Static**	**Coherent**	**All Power, Perf, Memory generality**	

Table 2.1: Taxonomy: Classification of accelerators

as an accelerator. DSP cores achieve well-known acceleration with their VLIW architecture. They are dynamically programmed by compilation and efficient in streaming data for multimedia. However, a DSP core requires separate memory, like scratchpad for the execution (and cannot share the host processor's case), thus it reduces memory generality. DySER [34, 32] is a recently proposed accelerator for dynamic specialization. Due to sharing a data cache with the host processor, they maintain memory generality which shares the data cache with the host processor. In addition, they provide reasonable performance improvement and can be used for computing intensive acceleration. However, DySER requires support from the host processor to provide memory access, therefore it is always power hungry for acceleration. We elaborate on HLS techniques next.

According to [63], the design choice between hardware logic design and using a general-purpose processor is important to efficiently achieve the goal. Hardware design is good to meet the goal in terms of design objective while software approach using the general purpose processor gives more flexibility and easier implementation than hardware design. An alternative approach is HLS to utilize advantages of hardware and software. HLS approaches can target performance, power, or both, depending on that various sub-regions of code [18]. In general, a host processor is interfaced with an accelerator to execute non-accelerated code and explicit calls move memory from host address space to the accelerator's memory space. The main purpose of HLS is to increase instruction-level parallelism, thus, HLS schedules instructions that do not have any dependency to execute at the same cycle state as dataflow fashion and replicate it for running in parallel [21].

Recently, many HLS techniques are proposed, which are classified whether the integration with soft-core is provided by industries or not. It is whichever a configurable processor core by end-users is for their purposes. For a soft-core approach, Altera and Xilinx introduce Nios-II and Microblaze soft-cores that are customized by the end-users [7, 5]. To configure and execute synthesizable accelerator hardware logic, HLSs using soft-core eliminate instruction and data cache from the soft-core stack to avoid cache coherence operation. While it is easy to maintain memory generality between the accelerator and the host process from soft-core, it degrades the performance when shared memory between the accelerator and the host processor is accessed because reading or writing memory is based on on-chip memory shared by the host processor and accelerators. LegUp, CGPA, and Warp adapt this approach for acceleration [17, 53, 76, 54]. In case of cache configuration with soft-core, it definitely requires cache coherence mechanism and on-chip memory enables accelerator to make less efficient. C2H from Altera can alternatively configure with or without cache in the soft-core for the role of the host processor [6]. For a non-soft-core approach, it means that synthesizable accelerator is directly attached in the host processor and access to the same memory system. C-Core is tightly integrated with a host processor, however it shows poor performance because it allows each basic block to access memory only once for memory generality between the C-core accelerator and the host processor [79, 31, 64]. Trident HLS assigns all memory allocation during compilation and the deterministic memory address is referenced with spreading data to multi-bank memory and fixed latency when its output hardware accelerator accesses memory [75]. Garp is another HLS technique which shares a data cache with the host processor, but its design complexity is high when its configurable hardware accelerator is connected with the data cache because it requires memory queues and crossbar to communicate with the cache [39, 16]. In addition, both approaches duplicate hardware logic and have a negative influence on power/energy efficiency because they consume additional logic elements while they improve performance by achieving high ILP. In terms of HLS compiler implementation, organizing FSM is an issue because flow control, which is the rule of injecting data to dataflow, must be maintained. Furthermore, it is

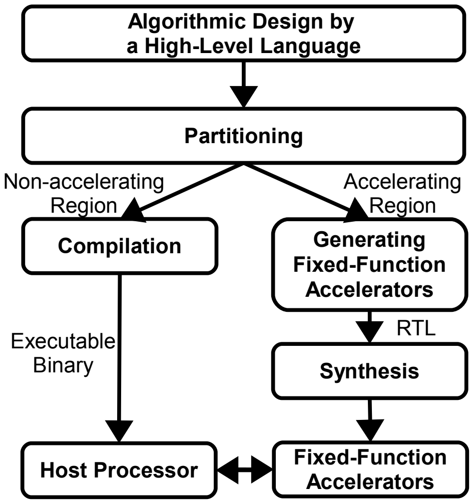

Algorithmic Design by a High-Level Language
Partitioning
Non-accelerating Region
Accelerating Region
Compilation
Generating Fixed-Function Accelerators
RTL
Executable Binary
Synthesis
Host Processor
Fixed-Function Accelerators

Figure 2.3: Hardware Design Flow

hard to schedule dataflow operation in case multiple clock cycle consumed operations exist in the dataflow, such as floating point instruction. For this reason, many HLS compilers have the constraint that they cannot support floating-point instructions [17, 6]. Somewhat tangential is the approach of King et al. [44] in which they use the specialized Bluespec language - they also sidestep the coherence issue and instead assume a bus to communicate with the processor.

To summarize, we need a new HLS to avoid the constraints described above. In this thesis, we improve the HLS by 1) decoupling the memory access and computation as in DAE for efficient cache utilization of the host processor and 2) designing a fixed function parallel access engine.

2.2.2 The Architecture of Fixed-Function Accelerators

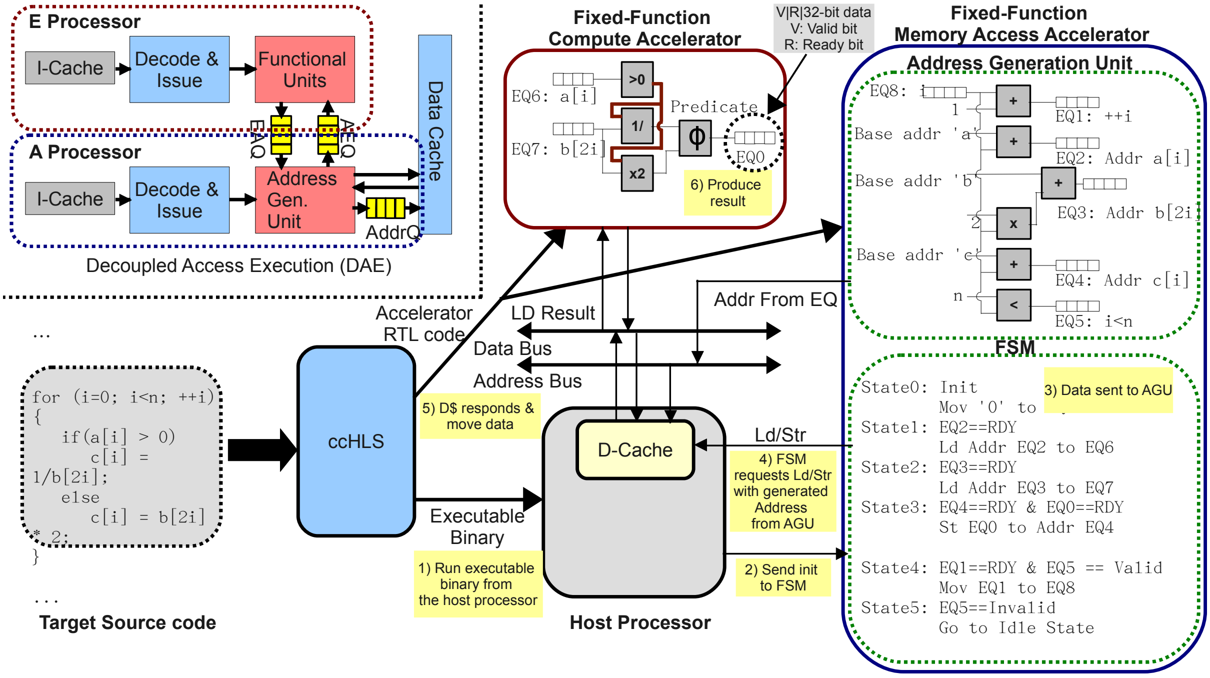
Figure 2.3 shows the proposed hardware design flow. First, the target applications are characterized and analyzed for frequent code regions. Second, we use the cache-coherent High Level Synthesis (ccHLS) to compile the application, replace the codes in the frequent regions with interface instructions, and generate fixed-function accelerators hardware for the removed codes. These Fixed-function accelerators are integrated into the host processor with a unified interface

Fixed-Function Accelerators Architecture Fixed-function accelerators are organized by the mixing of two components, a fixed-function compute accelerator and a fixed-function memory access accelerator. First, a fixed-function compute accelerator is the group of functional units (FU) to organize dataflow of computing intensive code region. Purely, it is concentrated on the computing acceleration because memory access operations, such as load and store, are invisible. To schedule operations by instructions, it does not require state transition of FMS to schedule a datapath because each FU has a control block to manage execution.

A fixed-function memory access accelerator has a role to generate a memory address to access and manage data movement among memory and accelerators. Unlike a fixed-function compute accelerator, it is organized by FSM with an address generation, unit which is the dataflow for address computation. Including load and store operations, FSM provides the following data movement capabilities between: 1) the fixed-function memory access accelerator and itself, 2) the fixed-function memory access accelerator and fixed-function compute accelerator, 3) memory and the fixed-function memory access accelerator, and 4) memory and the fixed-function compute accelerator. The first two cases are data movement to pass the computing result for address calculation or injecting local variables to fixed-function compute accelerator. The last two data movements are for load or store operation to move the computation result or variables. State transition of FSM is done by Boolean logic combination of extended header bits that show the status of result. Details of fixed-fuction accelerators are described in chapter 4.

Unlike other HLSs [17, 6], the fixed-function accelerators support all operations including floating point operations. This is feasible because FU's control logic dynamically schedules the operations in datapath. Moreover, memory access issues in the conventional HLSs are relaxed. Mostly, HLSs take a fixed latency for scheduling, but it is not a good approach in case the source code has massive memory accesses [80, 78, 58]. Fixed-function accelerators are tightly integrated data caches in the host processor and executed with a dynamic memory latency even though cache miss occurs. It is because FSM controls data movement by requesting from the address generation unit.

Cache-Coherent High Level Synthesis (ccHLS) ccHLS, which is proposed in this dissertation, is a new HLS to generate hardware logic from applications. The target code region is generated as a fixed-function compute accelerator and fixed-function memory access accelerator by parti-



```
...
for (i=0; i<n; ++i)
{
   if(a[i] > 0)
      c[i] =
1/b[2i];
   else
      c[i] = b[2i]
* 2;
}

...
```

```
State0: Init
        Mov '0' to
State1: EQ2==RDY
        Ld Addr EQ2 to EQ6
State2: EQ3==RDY
        Ld Addr EQ3 to EQ7
State3: EQ4==RDY & EQ0==RDY
        St EQ0 to Addr EQ4

State4: EQ1==RDY & EQ5 == Valid
        Mov EQ1 to EQ8
State5: EQ5==Invalid
        Go to Idle State
```

Figure 2.4: Execution Model: From DAE model, E processor and A processor are replaced with a fixed-function compute accelerator and a fixed-function memory access accelerator, respectively.

tioning region as a pure computation part and an access/movement part. It is motivated from the Program Dependence Graph (PDG), which describes the data and control dependencies between instructions [28]. It can be used for creating dataflow of fixed-function accelerators. More specifically, [34] proposes AEPDG which partitions PDG into an execution-PDG (EDPG) and access-PDG (APDG) for pure computation and interfacing with EPDG, respectively. EPDG is only a computation datapath, thus it can be easily transformed into a fixed-function compute accelerator. In addition, APDG is transformed to the fixed-function memory access accelerator because it provides memory operations to inject data into EDPG. Therefore, AEPDG concept drives the fundamental of ccHLS.

2.2.3 Execution Model

We now discuss the execution model during runtime, the underlying architecture and interface, and compilation. The architecture of fixed-function accelerators from the ccHLS follows the Decoupled Access-Execution (DAE) model to relax on-chip memory issues. The DAE model was first proposed in [70] and figure 2.4 presents the structure of it. DAE was initially designed to run computation (Execution) and memory access (Access) separately for hiding the delay of memory communication due to address calculation. By decoupling, execution and access of workload are assigned in E-processor and A-processor, respectively. Normal instruction is executed in E-processor and its operands are passed to A-processor. With the computation result from E-processor, A-processor generates accessing memory address to resolve instructions such as load and store. The DAE model is able to expand to HLS to generate fixed-function accelerators and allows an accelerator to maintain memory consistency with the host processor. In the perspective of output accelerators of HLS, its hardware logic is mixed with a fixed-function compute accelerator and memory accelerators, which have the same role of E-processor and A-processor.

Figure 2.4 shows the generating and execution flow at runtime. Source code is compiled and generated as executable binary with instruction extension and accelerators. Accelerators are composed of fixed-function compute accelerator and memory accelerator to follow DAE model. First, for the execution process, the host processor executes the compiled application, which has interface by instruction extension. Second, the host processor sends initial operation to the fixed-function memory accelerator to enter the initial state. Third, the data sent to the event queue interface drives the FSMs in the fixed-function memory accelerators to move the initial values to address generation units. Fourth, after address generation units produce the address, the FSMs access cache to acquire data from memory. Fifth, the cache responds to the FSMs and it moves the retrieved data to the fixed-function compute accelerator. Finally, the compute accelerator performs computation, based on the dataflow, and produces results at the output.

From the execution model, we expect 1) power and energy efficient computation because the host processor enters sleep mode except the data cache, 2) cache coherence caused by inconsistent data sharing between the host processor and accelerators when acceleration is finished is ignored because data cache in the host processor is directly accessed for data movement operations, and 3) parallel execution from the datapath allows computation to be faster and guarantees high performance.

2.2.4 Contribution

The contribution of ccHLS can be thought in terms of performance and power. The benefits of ccHLS come from 1) the separate operations for computations and memory access and 2) the execution model including dataflow and FSM of fixed-function accelerators. In particular:

- **Performance:** The performance improvement is induced by the decoupled access/execute (DAE) model. DAE model is expected abundant instruction-level parallelism (ILP) and memory-level parallelism (MLP). Basically, ILP is exploited by two accelerator components, which is a fixed-function compute accelerator and a fixed-function memory accelerator. They are executed in parallel and generate output including memory access address and values for variables in advance. Consequently, its result influence on ILP improvement. Moreover, MLP, which is the ability to issue far away memory accesses, is achieved by precedently generated memory addresses from the fixed-function memory access accelerator and required memory operations proceed beforehand.

- **Power:** the main contribution of fixed-function accelerators in power reduction is that the pipeline of a host processor is turned off during acceleration, and it possibly gains the power benefit. Mainly, power reduction is classified by pipeline stages, register files, and controls. First, the code segments running on the host processor are assigned as the dataflow execution for computation and memory access and the need for pipeline operation such as fetch, decode, and issue is eliminated. Second, the event queues which are the path of temporary generated address or value of variables explicitly retain them instead of using register-renaming and a big register file in the out-of-order processor. Finally, no control signals and dependence/control stall are required because FSM manages those operations triggered by boolean combination of the event queues, hence all controls depend on the condition in the event queues.

2.3 Chapter Summary

This chapter presented the motivation for why high-level synthesis is the design goal of accelerators for specialized and general applications. SWSL is focused on specialized applications for network lookup and its design concept is borrowed from a specialized architecture PLUG. Due to the lack of generality, however, it is not well-supported by the general applications. Thus, we discussed another HLS approach with the decoupled access-execution model (DAE) to generate separate accelerators for memory access and computation. More details are described in chapter 3 and 4.

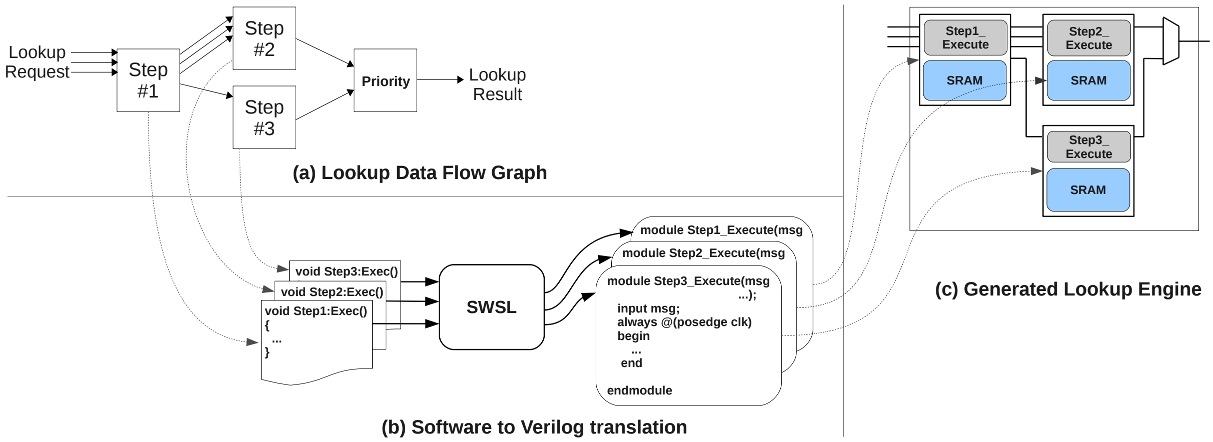
3 SWSL

This chapter discusses the compilation techniques to generate the accelerator using the program for the specialized architecture of network lookup. SoftWare Synthesis for network Lookup (SWSL) is designed to generate loop accelerators from the high-level program language for PLUG architecture and its code block source code is directly fed into SWSL. The programming model of PLUG code block is concentrated on lookup processing and its data flow graph (DFG) model for organizing network lookup is useful to create the datapath of lookup acceleration. In section 3.1, the overview of SWSL is discussed, including the programming model and compilation process. Then, details of the compilation process for SWSL are described in section 3.2. Finally, the summary of this chapter is shown in section 3.3.

3.1 Overview of SWSL

SWSL generates hardware logic descriptions from C++ programs using HLS techniques. It consists of two components: a hardware-agnostic dataflow-based programming model specialized for lookup algorithms, and a compiler middle layer that converts programs to verilog code.

SWSL is based on the observation that lookup algorithms can be decomposed in simple algorithmic steps, each accessing its own private state. The SWSL programming model enables the developer to specify the lookup algorithm as a pipeline of steps; the SWSL compiler generates



(a) Lookup Data Flow Graph

(b) Software to Verilog translation

(c) Generated Lookup Engine

Figure 3.1: Lookup Engine Generation using SWSL

verilog code for each step, and connects the steps to implement the full algorithm. The goal of the compiler is to exploit pipelining to handle a request per clock cycle, achieving throughput equal to system clock. At the same time, to reduce latency, SWSL increases parallelism by executing operations that are not data-dependent in parallel.

3.1.1 Programming Model

Previous work [24] observed that data structure lookups used by network applications tend to consist of simple steps, each accessing some private state. The SWSL programming model reflects this structure, allowing the programmer to express application as a Data Flow Graphs (DFG). This approach is inspired by the PLUG programming model described in [24]. Each node in a DFG represents an algorithmic step, and includes some simple computation and the associated data. This model is a particularly good fit for SWSL as the application workload is well-partitioned in simple steps.

Figure 3.1 shows an overview of how SWSL generates lookup hardware from applications expressed in the DFG model. Figure 3.1a depicts the DFG of a simple application. The first stage computes a hash that is then used to perform lookups in two secondary stages. If both stages return a result, one takes priority over the other. This simplified graph can, for example, model a router/firewall, where a high-priority table lists specific flows that must be dropped while a low-priority table holds more general forwarding rules. SWSL takes the functions executed at each stage and generate equivalent blocks of lookup hardware logic (Figure 3.1b). Finally, each generated logic block is associated with a SRAM module to hold the forwarding table, and all blocks are combined in a single lookup chain (Figure 3.1c). Blocks in the lookup chain are independent and communicate via on-chip messages.

This approach has two significant advantages. In comparison with a software-only approach, SWSL replaces Von-Neumann style cores with specialized logic, increasing efficiency and performance. Moreover, DFG-based applications express algorithms in a form that is convenient for hardware generation, yet architecture-independent. Indeed, we were able to use SWSL on applications written for the PLUG accelerator, which has a similar model, with no modification (Section 5). We also note that applications written for SWSL are in standard C++, and can be compiled and run fully in software for prototyping/debugging purposes.

From the point of view of the programmer, SWSL poses a series of constraints with the goal of keeping the algorithm hardware-friendly. First, to enable pipelining, SWSL requires programs to be loop-free (this condition could be easily relaxed to require all loops to be bounded, enabling static unrolling). To minimize the impact of these constraints, the SWSL programming model provides API primitives to perform common loop-based operations such as bit-counting. Then, SWSL requires accesses to the main lookup data structure to be performed with dedicated API calls, thus being clearly separated from accesses to temporary variables. This enables SWSL to discriminate temporary variables and store them in fast registers, avoiding related memory references at runtime. To prevent ambiguous memory references to such variables, SWSL does not allow pointer arithmetic. Finally, dynamic memory allocation is not available as this concept cannot be mapped to a static hardware implementation.

In general, we found that such constraints do not limit the expressiveness of the programming model significantly, and lookup algorithms can be naturally implemented with SWSL.

3.1.2 SWSL Compiler

Algorithmic steps (represented as nodes in the DFG of Figure 3.1 3.1a) are individually converted to hardware logic via the SWSL compiler. This component is implemented as a pass within the LLVM compiler toolkit [50]. LLVM provides the basic compilation infrastructure to parse C++ source code, translate it to intermediate representation and build the program control flow graph (CFG). In the CFG, the program is decomposed in basic blocks - straight lines of code with a single point of entry and one or more exits (e.g. a branch or a switch construct). Arcs represent the control flow, i.e. the possible paths the program can follow when executing. The first transformation performed by SWSL is to restructure the control flow to merge certain basic blocks that can be executed in parallel (specifically, basic blocks that compute multiple conditions evaluated by a `branch` instruction).

In general the structure of the dataflow graph, where each algorithmic step is represented as an independent node (Figure 3.1a), offers some opportunity for pipelining. However, the gain in throughput is limited as the graph coarsely subdivides the algorithm into a limited

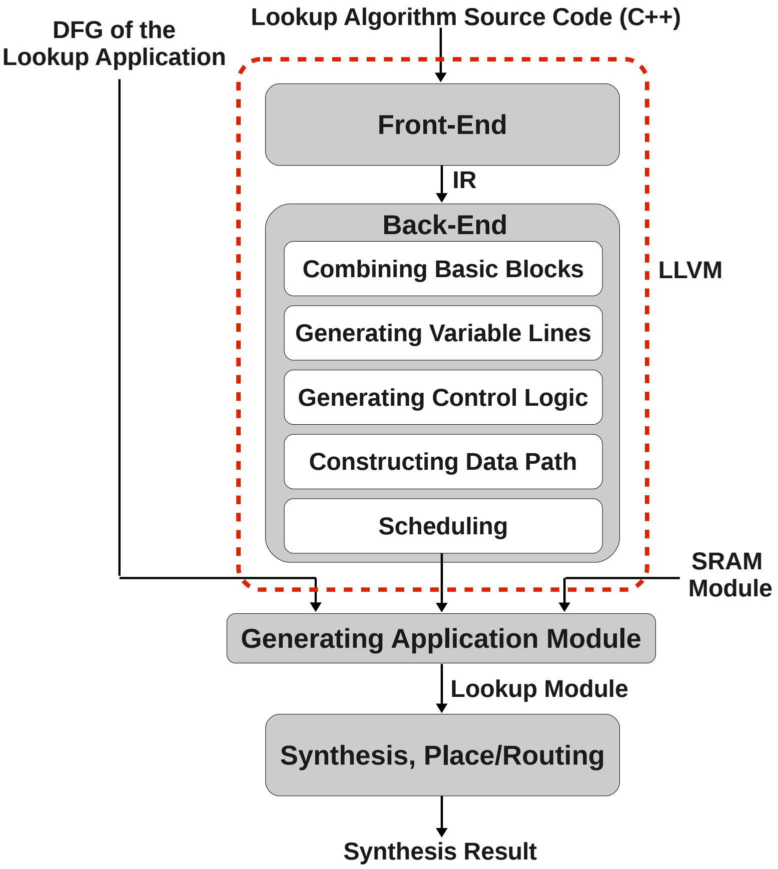


Figure 3.2: Compilation Process of SWSL

number of steps, each of which can take tens of cycles to execute. To improve throughput, SWSL further internally pipelines each step. In general, the structure of the computation may not naturally lend itself to pipelining. SWSL circumvents this problem by replicating the temporary state associated with the computation. SWSL instantiates multiple buffers called `variable lines`, each capable of holding inputs, intermediate results, and output used/generated by each step. By using different variable lines, multiple computations can proceed independently. This enables SWSL to pipeline the logic at the basic block level.

After generating variable lines, SWSL creates the control logic that determines which execution path is followed at runtime (such logic will decide, for example, which step must be activated after a branch condition). Then, it generates the actual hardware datapath implementing the functionality described by the software. In general, the resulting hardware will have multiple execution paths; the actual path followed during each execution will depend on the result of branch conditions. As SWSL organizes the logic in a pipeline, it must ensure that the number of steps is constant, regardless of which path is followed (e.g. the number of steps must not change depending on which side of a branch is taken). SWSL performs a scheduling step that inserts additional delay buffers to "pad" all paths to the same length.

Figure 3.2 summarizes the overall SWSL compilation process. After being parsed by LLVM, each step in the original algorithm (node in the DFG) goes through SWSL, which generates hardware logic. At the end, a SRAM block is added to the module to store the subset of the forwarding table associated with the computation. At the synthesis and place/routing step, the generated top level design is synthesized using a general synthesis tool (our implementation uses the Synopsys design compiler).

3.1.3 Compilation Process for SWSL

Figure 3.2 shows the compilation process for software synthesis. LLVM provides basic compilation infrastructure to generate logic design from code block source code written in C++. Each source code of code block passes front-end to generate IR and its output feeds into back-end to output verilog code. With the provided SRAM module, top level design creates the top level design of each logical page and lookup engine. One thing missing here is to let SWSL know the data flow graph of a target application. To do that, a configuration file is needed, which includes the shape of data flow graph of the application. It provides information about code blocks embedded in the logical page, required SRAM size, and other configuration data used in back-end compiler. Local constant value used in the PLUG framework is provided by this configuration file. At synthesis and place/routing step, the generated top level design is synthesized by a general synthesis tool. Synopsys design compiler provides synthesis capability for the generated logic design.

3.2 Back-end Compiler for SWSL

Back-end compiler is the most fundamental part in the compilation process of SWSL because it provides the capabilities required to achieve functionality of SWSL. A major concern in back-end

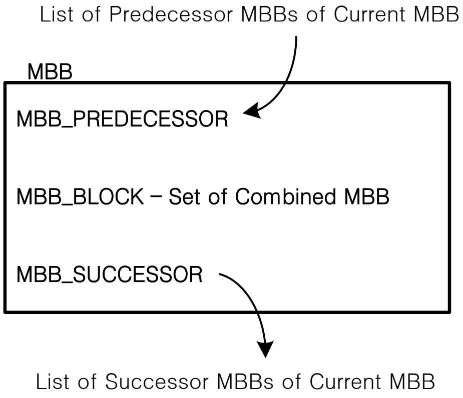

List of Predecessor MBBs of Current MBB
MBB
MBB_PREDECESSOR
MBB_BLOCK – Set of Combined MBB
MBB_SUCCESSOR
List of Successor MBBs of Current MBB

Figure 3.3: MBB Structure

compiler design is how to structure the pipeline for providing the capability to support massive input message streams. A common compiler technique for a conventional microprocessor is to create target machine assembly by following the control flow graph from the source code. In addition, the target machine assembly follows sequential code execution by program counter. Therefore, it is impossible to organize pipelined data processing for the network lookup engine. It follows that back-end compiler of SWSL needs a different compiler structure for pipelining. In this subsection, control flow and data flow graph of a target code block are separately considered in terms of generating control signals and exploiting parallelism, respectively. In this subsection, we will show the compilation techniques to achieve SWSL. First, combining basic blocks which degrade parallelism technique is shown. Second, a variable handling method will be described to achieve pipelining scheme for input message stream. Third, control logic from the control flow graph of MBB is shown. Finally, exploiting parallel design from the data flow graph of each basic block will be shown.

3.2.1 Combining Conditional Basic Blocks

In the control flow graph, there might be some basic blocks which only represent conditional expression. This type of basic blocks does not have any data dependency with predecessor blocks, hence these basic blocks are considered to be combined with their own predecessor and this combined block executes in parallel for parallel decision. To collect such basic blocks, the back-end compiler gives the concept called Merged Basic Block (MBB). The MBB is the set of basic 31 blocks maintaining features of the control flow graph given from the target source code. It includes the set of basic blocks with MBB successors and predecessors. Figure 3.3 shows the MBB structure, which includes $MBB\_PREDECESSOR$, $MBB\_SUCCESSOR$, and MBB_BLOCKS. The initial two represent the list of MBB which are connected to current MBB as predecessors and successors, and last one means the set of combined basic blocks organizing the current MBB. At the beginning of basic block combining, each basic block is encapsulated as an MBB and the set of MBB_BLOCKS has a target basic block itself. For an $MBB\_PREDECESSOR$ and an $MBB\_SUCCESSOR$, the current MBB points to the list which has the MBB including predecessor and successor basic blocks.

After the beginning of basic block combining, the next step is to find combined MBB. Combined MBBs have the following properties:

- A target MBB must have two successor MBBs.
- A target MBB must have only one predecessor MBB.
- A set of predecessor's successors of a target MBB must have a intersected MBB with a set of successors of a target MBB.

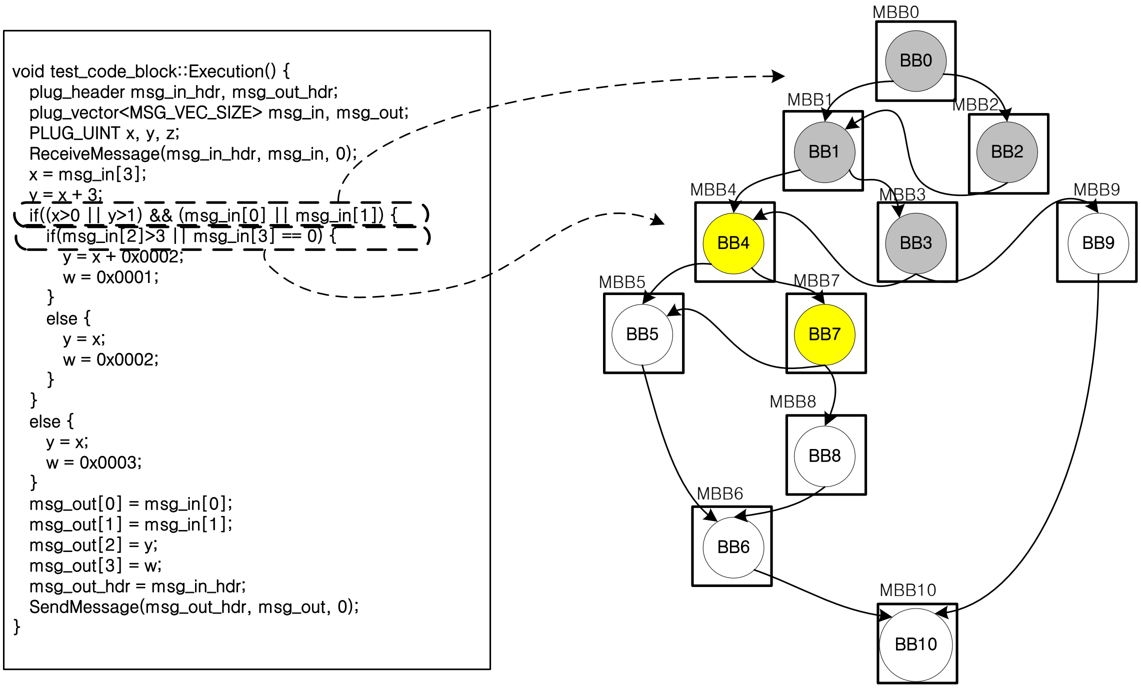
A target MBB can be merged with a group of predecessor's MBB. This target MBB is eliminated and its MBB_BLOCKS is included in that of predecessor's MBB. Algorithm 1 shows the way to combine conditional basic blocks.

An example code and its control flow graph are shown in figure 3.4. The code block itself is relatively simple, but its control flow graph has some basic blocks which can be combined as a group. Gray and Yellow basic blocks represent those having conditional expressions using the `branch` instruction. At the beginning of the combination process, each basic block is replaced as an MBB and its predecessors and successors are also changed to point MBBs.

In the code block example, it has two `if/else` statements in the code block. For the first `if/else` statement, the first condition expression `x > 0` is described in BB0 and the `branch` instruction in MBB0 points MBB1 or MBB2. The second expression `y > 1` is represented in MBB2, then it constructs `(x > 1 || y > 1)` expression with MBB0. The third and fourth conditional

Algorithm 1 Combining Conditional Blocks

```
$MBB\_SET \leftarrow Set\ of\ All\ MBBs\ in\ a\ Target\ Code\ Block$
$OLD\_MBB\_SET \leftarrow NULL$
while $MBB\_SET \neq OLD\_MBB\_SET$ do
    $OLD\_MBB\_SET \leftarrow MBB\_SET$
    while $MBB\_SET \neq \emptyset$ do
        $T \leftarrow Pick\ a\ MBB\ from\ MBB\_SET$
        $Eliminate\ T\ from\ MBB\_SET$
        $numOfSucc \leftarrow Get\ \#\ of\ T's\ Successor$
        $numOfPred \leftarrow Get\ \#\ of\ T's\ Predecessor$
        if $numOfPred == 1\ \&\&\ numOfSucc == 2$ then
            $PRED \leftarrow T's\ Predecessor$
            $A\_SET \leftarrow All\ Successors\ of\ PRED$
            $B\_SET \leftarrow All\ Successors\ of\ T$
            if $(A\_SET \cap B\_SET) \neq \emptyset$ then
                $Combine\ T\ with\ PRED$
                $Update\ successors\ of\ PRED\ with\ T's\ successor$
            end if
        end if
    end while
    $MBB\_SET \leftarrow All\ Combined\ MBBs$
end while
```



```
void test_code_block::Execution() {
   plug_header msg_in_hdr, msg_out_hdr;
   plug_vector<MSG_VEC_SIZE> msg_in, msg_out;
   PLUG_UINT x, y, z;
   ReceiveMessage(msg_in_hdr, msg_in, 0);
   x = msg_in[3];
   y = x + 3;
   if((x>0 || y>1) && (msg_in[0] || msg_in[1]) {
      if(msg_in[2]>3 || msg_in[3] == 0) {
         y = x + 0x0002;
         w = 0x0001;
      }
      else {
         y = x;
         w = 0x0002;
      }
   }
   else {
      y = x;
      w = 0x0003;
   }
   msg_out[0] = msg_in[0];
   msg_out[1] = msg_in[1];
   msg_out[2] = y;
   msg_out[3] = w;
   msg_out_hdr = msg_in_hdr;
   SendMessage(msg_out_hdr, msg_out, 0);
}
```

a) Code Block Example

b) Control Flow Graph

Figure 3.4: An Example Code and its Control Flow Graph

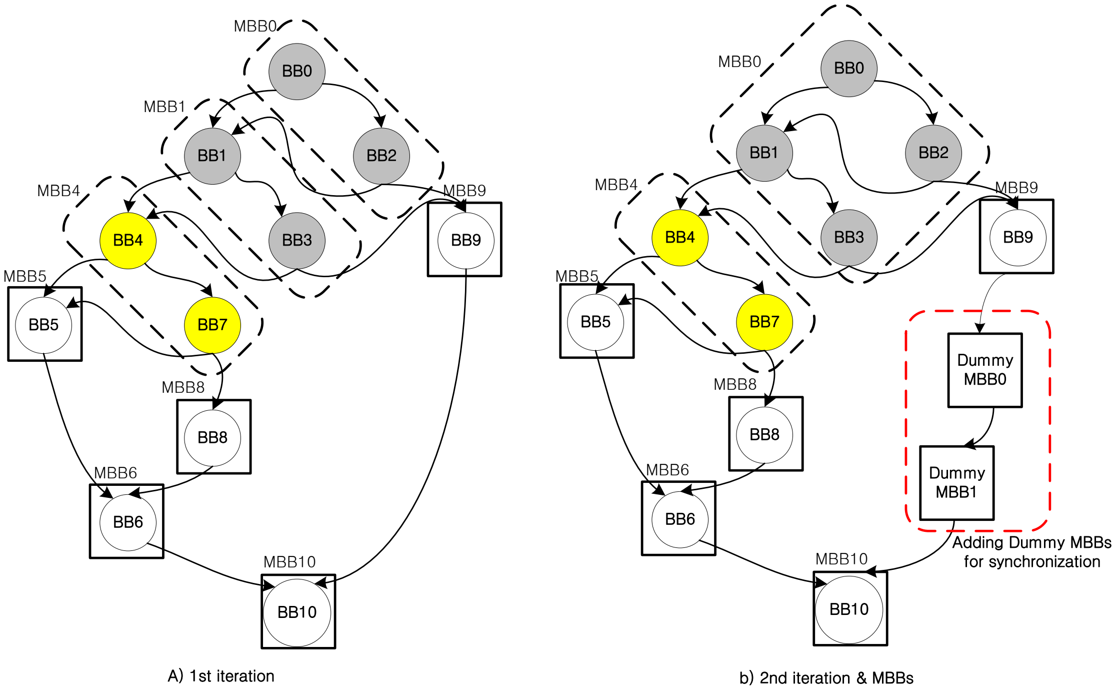
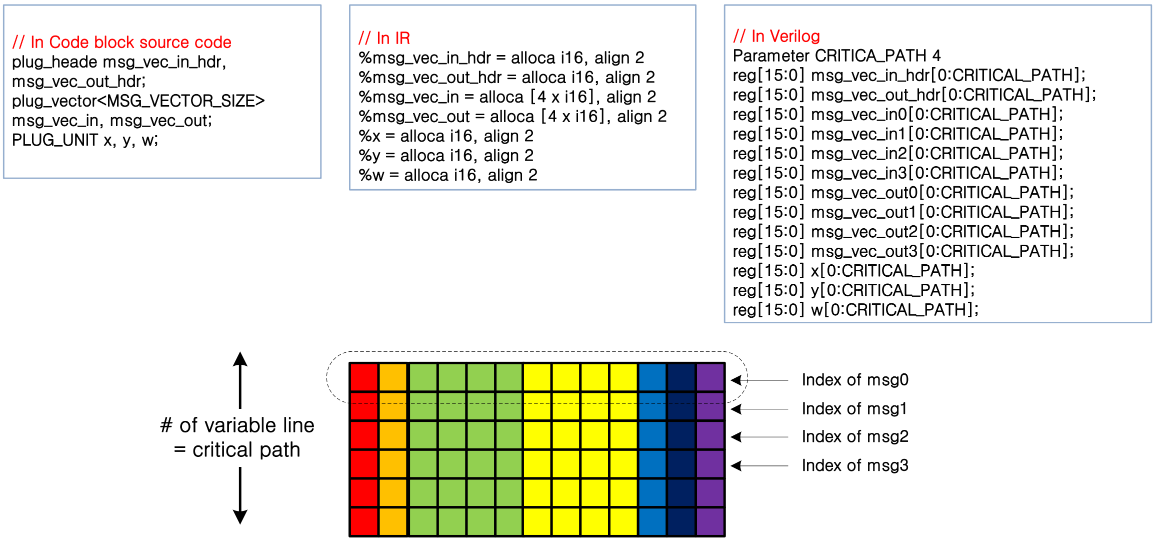


Figure 3.5: An Example Code and its Control Flow Graph

expressions `msg_in[0]` and `msg_in[1]` are expressed in MBB1 and MBB3, respectively. In these MBBs, MBB2 and MBB4 are the target to be combined because they follow combining properties. For the second `if/else` statement, the combining MBB target is MBB4 and MBB6 because they have conditional expressions `msg_in[2]` and `msg_in[3]`, respectively. In these MBBs, MBB6 are combined with MBB4 because it follows combining properties. After combining MBBs, the control flow graph is shown in Figure 3.5a). However, it is not the end of iteration because MBB1 in Figure 3.5a) follows the combining properties again. Therefore, a modified control flow graph should iterate again to combine. Figure 3.5b) shows the modified MBB control flow graph with no MBB following combining properties.

After combining MBBs, dummy MBBs should be embedded in the MBB control flow graph. A dummy MBB is padded when predecessors of a current MBB does not lie in the same level as the root MBB. It allows input messages to be synchronized. In figure 3.5b), MBB6 and MBB9 are not in the same level as MBB0 and their level distance is equal to 2. Therefore, two dummy MBBs are embedded between MBB9 and MBB10. In addition to relaxing level distance, the dummy MBB helps the back-end compiler to set the counter index for each MBB. Further details



```
// In Code block source code
plug_heade msg_vec_in_hdr,
msg_vec_out_hdr;
plug_vector<MSG_VECTOR_SIZE>
msg_vec_in, msg_vec_out;
PLUG_UNIT x, y, w;
```

```
// In IR
%msg_vec_in_hdr = alloca i16, align 2
%msg_vec_out_hdr = alloca i16, align 2
%msg_vec_in = alloca [4 x i16], align 2
%msg_vec_out = alloca [4 x i16], align 2
%x = alloca i16, align 2
%y = alloca i16, align 2
%w = alloca i16, align 2
```

```
// In Verilog
Parameter CRITICA_PATH 4
reg[15:0] msg_vec_in_hdr[0:CRITICAL_PATH];
reg[15:0] msg_vec_out_hdr[0:CRITICAL_PATH];
reg[15:0] msg_vec_in0[0:CRITICAL_PATH];
reg[15:0] msg_vec_in1[0:CRITICAL_PATH];
reg[15:0] msg_vec_in2[0:CRITICAL_PATH];
reg[15:0] msg_vec_in3[0:CRITICAL_PATH];
reg[15:0] msg_vec_out0[0:CRITICAL_PATH];
reg[15:0] msg_vec_out1[0:CRITICAL_PATH];
reg[15:0] msg_vec_out2[0:CRITICAL_PATH];
reg[15:0] msg_vec_out3[0:CRITICAL_PATH];
reg[15:0] x[0:CRITICAL_PATH];
reg[15:0] y[0:CRITICAL_PATH];
reg[15:0] w[0:CRITICAL_PATH];
```

Figure 3.6: Variables Used in a Code Block and its Expression

are shown in section 3.2.2.

3.2.2 Pipeline Structure for Massive Input Message Stream

In code block, variables are used for computation. In the perspective of Von-Neumann architecture, these variables are located in memory and `load/store` instruction which allows the program to access memory to load/store data from/to provided registers. It disrupts pipelined processing because an input message may access a variable while another input message tries to access the same variable. Each input message is for a different lookup execution, thus, this case brings up race condition for a variable and it may cause unwanted results. To avoid this obstacle, a back-end compiler describes variables as variable lines. Each variable line is reserved for each input message to avoid race condition and index counters to point to an appropriate variable line for each input message are provided. The length of variable lines is the same as that of the critical path of hardware logic generated from the back-end compiler. The length of critical path is described as the definition of 'parameter' of verilog primitive. Variables and arrays can be found in IR because used variables are described by LLVM instruction 'alloca' [3].

Figure 3.6 shows an example of variable lines. In the code block source code, variables and arrays are represented with their data type. In IR of source code from the front-end compiler, variables and arrays are represented by using LLVM instruction alloca. Finally, it can be described as verilog code with register assignment, and the variable index counter helps the input message point out the correct variable line for its execution. In Figure 3.6, red and orange boxes represent msg_vec_in_hdr and msg_vec_out, respectively. Green and yellow boxes are used for array msg_vec_in and msg_vec_out. The last box represent variables x, y, and w. When an input message arrives, the index counter is incremented to avoid conflict in accessing variables. Thus, an input message only accesses a variable line to compute and other input messages only access their variable line. It eliminates race condition, and execution for each input message can be achieved under race condition free. The reason that the number of variable lines is the same as that of the critical path is that a variable line is free after all processing is done within the critical path. For indexing a variable line, variables used in each variable line use an index counter of an input message. The index counter number is cascaded to MBB successor blocks and a branch instruction in an MBB block passes the current index counter number to successors. To index a variable line in a current MBB block, the index counter is set by its predecessor's counter because handling an input message at a current MBB block is preceded by its MBB predecessor block. One thing to be careful of is that MBBs which have the same MBB predecessor block increment the index counter at the same time because they are mutually exclusive.

The cascading index number follows. First, the back-end compiler collects all MBB blocks in the control flow graph except root and exit MBB blocks. Second, the message input counter to set input message data to appropriate array in source code is defined. Third, counters for root and exit MBB blocks are set, and the output counter to send message data to the next logical page is also defined. Fourth, root and exit MBB blocks fix their current and next counter. Finally, the back-end compiler sets all counters using iteration by looking at their successors and predecessors. In algorithm 2, the way to set the index counter for variable lines is described.

Figure 3.7 shows the counter set example from figure 3.4 and 3.5. The current counter of an MBB block is the same as the next counter of its predecessors. The dotted box describes a mutually exclusive case when MBBs have the same predecessors or successors and they have the same current and next index counter. When a message comes from the previous logical page, it uses MSG_CNT to set input message data to arrays and MSG_CNT is incremented whenever

Algorithm 2 Cascading Variable Line Index Counter

```
1. $MBB\_SET \leftarrow Set\ of\ MBBs\ in\ CFG\ except\ Root\ and$
                  $Exit\ MBBs$
2. $MC \leftarrow\ 'MSG\_CNT'\ for\ Input\ Message\ Data\ Counter$
3. $RC \leftarrow\ 'ROOT\_CNT'\ for\ Root\ MBB\ Counter$
4. $EC \leftarrow\ 'EXIT\_CNT'\ for\ Exit\ MBB\ Counter$
5. $OC \leftarrow\ 'OUTPUT\_CNT'\ for\ Send\ Message\ Counter$
6. $For\ Root\ MBB\ 'R'$
  $R \rightarrow CurrCnt\ = RC$
  $R \rightarrow NextCnt\ = new\ 'BB\_(i)\_CNT'$
  $i++$
7. $For\ Exit\ MBB\ 'E'$
  $E \rightarrow CurrCnt\ = EC$
  $E \rightarrow NextCnt\ = OC$
  $i++$
8. $B \leftarrow\ Pick\ asuccessor\ of\ Root\ MBB$
// $Pred(B)\ -\ B's\ predecessor\ MBB$
// $Succ(B)\ -\ B's\ successor\ MBB$
repeat
    $Erase\ B\ from\ MBB\_SET$
    if $Pred(B) \rightarrow NextCnt \neq NULL$ then
        $B \rightarrow CurrCnt\ =\ Pred(B) \rightarrow NextCnt$
    else
        $B \rightarrow CurrCnt\ =\ new\ 'BB\_(i)\_CNT'$
    end if
    $i++$
    if $Succ(B) \rightarrow CurrCnt \neq NULL$ then
        $B \rightarrow NextCnt\ =\ Succ(B) \rightarrow CurrCnt$
    else
        $B \rightarrow NextCnt\ =\ new\ 'BB\_(i)\_CNT'$
    end if
    $i++$
    $B \leftarrow\ Pick\ asuccessor\ of\ Root\ MBB$
until $MBB\_SET\ =\ \emptyset$
```

input messages come. This counter number of `MSG_CNT` is cascaded to `ROOT_CNT`. When the root MBB block is done, the counter number of `ROOT_CNT` is cascaded to the next MBB block counter. This cascading keeps going until it meets `EXIT_MBB`. `OUTPUT_CNT` is used when computed output data is sent to the next logical page when operation of `EXIT_MBB` is finished. Thus, the MBB uses the set MBB current counter to point an appropriate variable line, and the next counter of the current MBB is used for cascading current index counter to successors of the current MBB in the control logic.

In section 3.2.1, dummy MBBs are described to explain synchronization. Dummy MBBs also have another role in terms of variable line index counters. By embedding them, they allow hardware logic to avoid counter crashing which means that the current index counter is

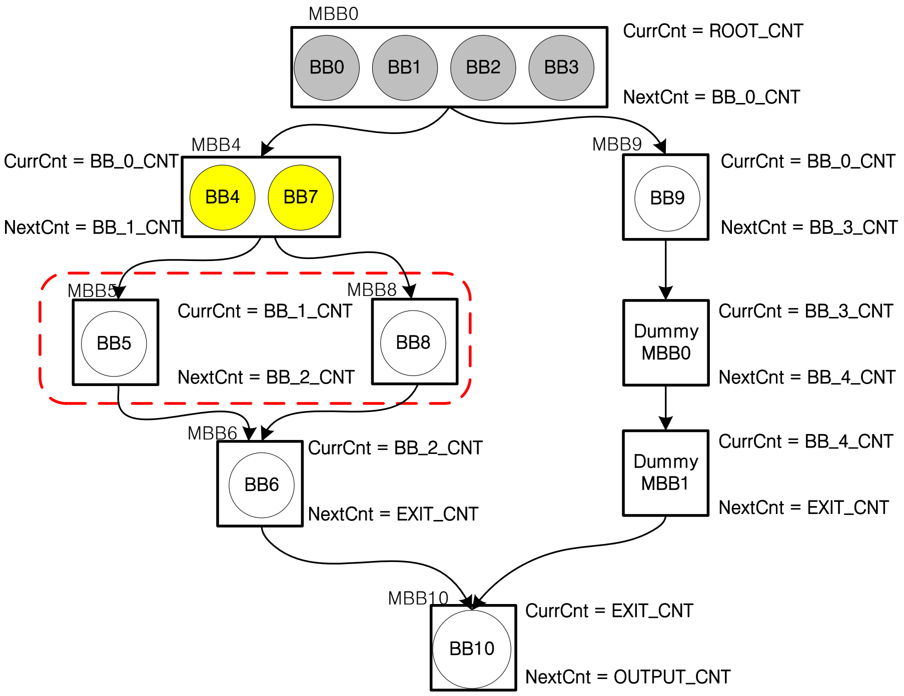


Figure 3.7: Cascading Variable Line Index Counter

mismatched with the next index counter of predecessors. By adding dummy MBBs, they relax the crash and the cascading index counter operates smoothly.

3.2.3 Generating Control Logic

The main role of control logic in SWSL is to let the generated verilog code decide which MBBs are executed. Thus, each MBB must know what MBBs are run by that MBB. It may seem to be straightforward, but it is not so simple, like what we expect because deciding enable signals for each MBB depends on the control flow graph of MBB. In the case of an MBB which has multiple successors, this MBB must enable one of the successors to execute. However, the MBB does not know what the enable signal is for the next MBB. Another case is an MBB which has multiple predecessors. It does not have any clue which predecessors are running the MBB. To overcome this obstacle, each MBB should have the set of current (`CurrEn`) and next (`NextEn`) enable signals. `CurrEn` and `NextEn` represent the signals that ignite the current next MBB. Thus, the data path in current MBB is running when the `CurrEn` signal is set by its predecessor's `NextEn` because the `NextEn` of the predecessor enables current enable signal. Likewise, the

Algorithm 3 Setting Enable Signal for Each MBB

```
$1:$ $cnt \leftarrow 0$
$2:$
for $i = 0$ to $\# \ of \ MBBs$ do
    if $\# \ of \ predecessor \ of \ MBB[i] \ < \ 2$ then
        $MBB[i].CurrEn \leftarrow \ mbb\_(cnt)\_en$
        $cnt++$
    end if
end for
$3:$
for $i = 0$ to $\# \ of \ MBBs$ do
    if $\# \ of \ successors \ of \ MBB[i] \ > \ 1$ then
        $MBB[i].NextEn$
        $\leftarrow \ CurrEn \ of \ MBB[i]'s \ successors$
    end if
end for
$4:$
for $i = 0$ to $\# \ of \ MBBs$ do
    if $MBB[i].CurrEn \neq \emptyset$ &&
            $MBB[i].NextEn \ = \ \emptyset$ then
        if $CurrEn \ of \ successor \neq \emptyset$ then
            $MBB[i].NextEn \leftarrow CurrEn \ of MBB[i]'s$
                                                    $successors$
        else
            $MBB[i].NextEn \leftarrow \ mbb\_(cnt)\_en$
            $cnt++$
        end if
    end if
end for
$5:$
for $i = 0$ to $\# \ of \ MBBs$ do
    if $\# \ of \ predecessor \ of \ MBB[i] \ > \ 2$&&
            $\# \ of \ successor \ = \ 1$ then
        $MBB[i].CurrEn \leftarrow \ NextEn \ of \ MBB[i]'s$
                                            $predecessors$
        $MBB[i].NextEn \leftarrow \ mbb\_(cnt)\_en$
        $cnt++$
    end if
end for
$6:$
for $i = 0$ to $\# \ of \ MBBs$ do
    if $MBB[i].CurrEn \ = \ \emptyset$ then
        $MBB[i].CurrEn \leftarrow \ NextEn \ of \ MBB[i]'s$
                                            $predecessors$
    end if
    if $MBB[i].NextEn \ = \ \emptyset$ then
        $MBB[i].NextEn \leftarrow \ CurrEn \ of \ MBB[i]'s$
                                            $successors$
    end if
end for
```

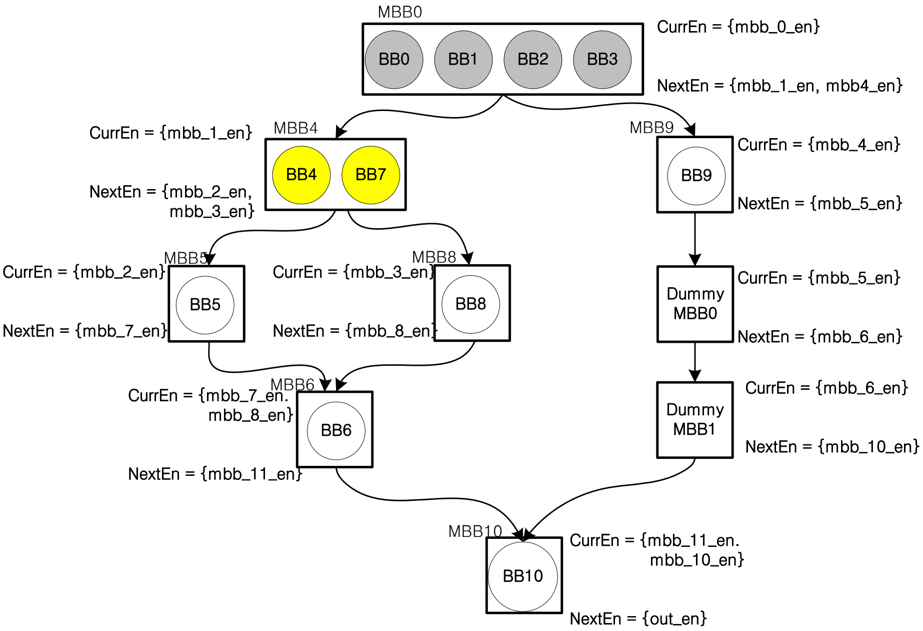


Figure 3.8: Enable Signal Set

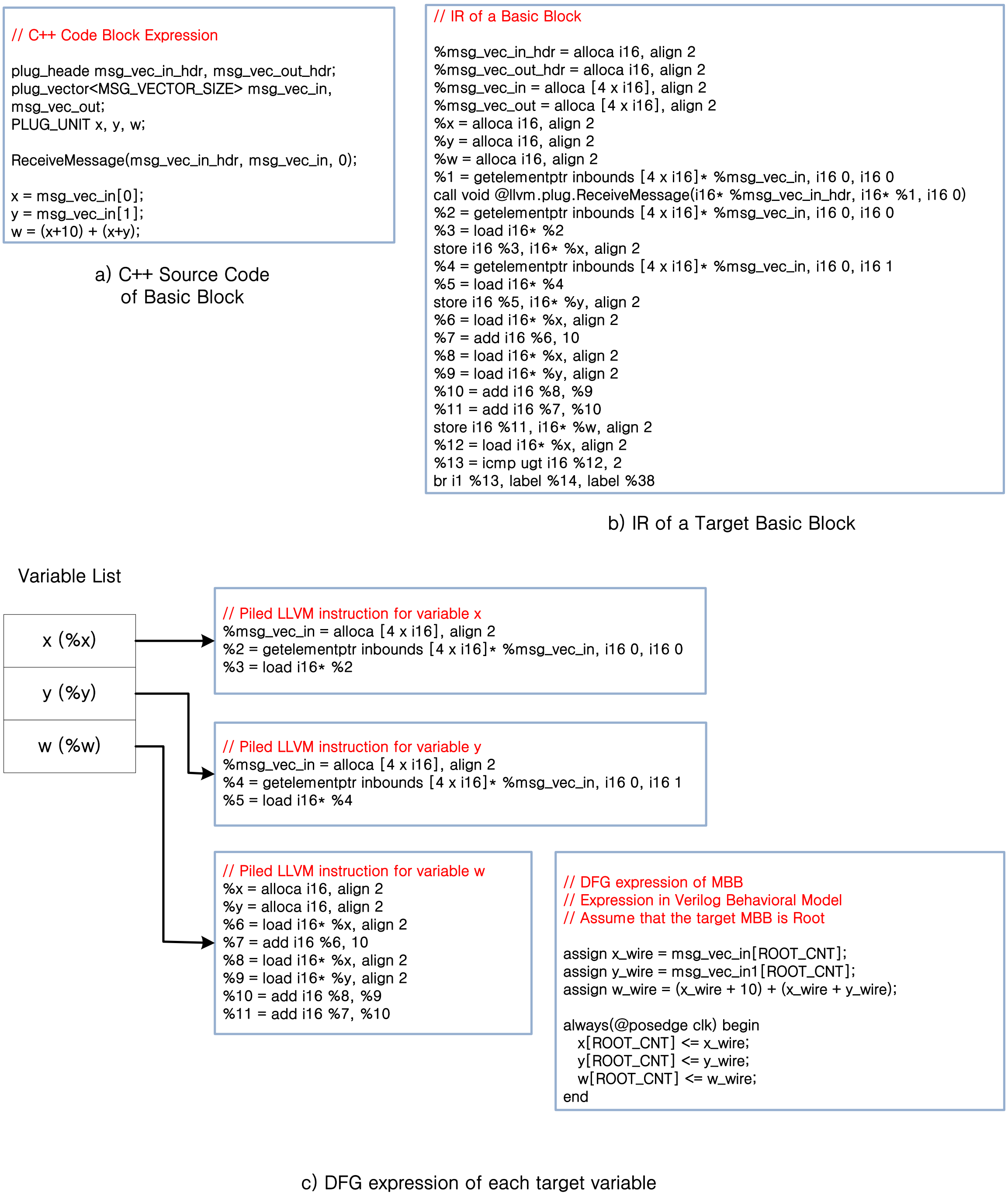
NextEn of the current MBB selects one of its successor to be executed. Therefore, the problem is to solve once CurrEn and NextEn are set for each MBB. To set the CurrEn and NextEn of each MBB, the MBB having multiple successors or predecessors must be carefully handled. To achieve assigning the CurrEn and NextEn of each MBB, algorithm 3 is proposed. First, CurrEn of MBBs which have one or non-predecessor is set. It is straightforward because these MBBs only concern themselves. Second, MBBs having two or more successors are considered, these MBBs are shown if/else statement or switch/case because these types of expression have a number of successors. Their NextEn are the CurrEn of their successors. Hence, they have multiple NextEn signals to select one of the successors and its decision is made by branch LLVM instruction in their basic blocks [3]. Third, MBBs, which are set CurrEn and not set NextEn, are considered. If the CurrEn of their successors is set, their NextEn are the CurrEn of their successors. Otherwise, they set new enable signals as their NextEn. A fourth case is an MBB which has more than two predecessors and one successor. This is a case in which one of predecessors was running and finished its execution. This MBB does not know which predecessor was running. So, the MBB must collect all NextEn signals of its predecessors and described as OR (||) operators in the verilog behavioral model. Finally, the last of the MBBs, which are not set their CurrEn or NextEn, set by using their predecessors and successors. Figure 3.8 shows the set of `CurrEn` and `NextEn` of each MBB from the example of figure 3.4 and 3.5. By following algorithm 3, each MBB has the set `CurrEn` and `NextEn`. From figure 3.8, MBBs having multiple successors or predecessors get multiple enable signals for `NextEn` or `CurrEn`.

3.2.4 Generating Data Path from the Data Flow Graph of Each MBB

In this subsection, constructing the data flow graph of each MBB and data path design are described. As shown in 3.2.1 and 3.2.2, the control flow graph of source code IR is reorganized as the MBB control flow graph. To generate a verilog behavioral model as an output of the SWSL compiler, it creates a data path by using the data flow graph of each MBB. The MBB has the set of basic blocks and each basic block is organized by the LLVM instruction sequence. From section 3.2.2, each MBB has a current and next index counter, so data dependency between MBBs is able to be ignored because each MBB references data from the variable line with the current variable index counter. Therefore, it is simple to organize the data path using that LLVM IR code sequence because it only considers the data flow graph of basic blocks within the MBB. However, parallel execution of the data flow graph should be carefully considered. Data dependency must be evaded if the LLVM instruction sequence has data dependency in it. Otherwise, variables having data dependency get an expected value after one cycle delay.

The first and second operands in LLVM `store` instruction represent the data flow graph to construct the data path and a variable to store computation value from the data path because the first and second operand describe value to be stored and the target variable, respectively. This target variable is stored in the variable list and variables in the variable list are described as `wire` fan out connection. The variable list points to a vector piled with LLVM instructions organizing the data flow graph for the target variable of LLVM `store` instruction. When LLVM instructions are piled in the variable list, the back-end compiler checks the data dependency and the first operand of LLVM `load` instruction treating variables should be evaluated by searching the variable list. If the operand of LLVM `load` instruction is found from the variable list, the data path is organized with the variable in the variable list described as `wire` connection.

Figure 3.9 shows an example to construct the data flow graph without data dependency.



```
// C++ Code Block Expression

plug_heade msg_vec_in_hdr, msg_vec_out_hdr;
plug_vector<MSG_VECTOR_SIZE> msg_vec_in,
msg_vec_out;
PLUG_UNIT x, y, w;

ReceiveMessage(msg_vec_in_hdr, msg_vec_in, 0);

x = msg_vec_in[0];
y = msg_vec_in[1];
w = (x+10) + (x+y);
```

a) C++ Source Code of Basic Block

```
// IR of a Basic Block

%msg_vec_in_hdr = alloca i16, align 2
%msg_vec_out_hdr = alloca i16, align 2
%msg_vec_in = alloca [4 x i16], align 2
%msg_vec_out = alloca [4 x i16], align 2
%x = alloca i16, align 2
%y = alloca i16, align 2
%w = alloca i16, align 2
%1 = getelementptr inbounds [4 x i16]* %msg_vec_in, i16 0, i16 0
call void @llvm.plug.ReceiveMessage(i16* %msg_vec_in_hdr, i16* %1, i16 0)
%2 = getelementptr inbounds [4 x i16]* %msg_vec_in, i16 0, i16 0
%3 = load i16* %2
store i16 %3, i16* %x, align 2
%4 = getelementptr inbounds [4 x i16]* %msg_vec_in, i16 0, i16 1
%5 = load i16* %4
store i16 %5, i16* %y, align 2
%6 = load i16* %x, align 2
%7 = add i16 %6, 10
%8 = load i16* %x, align 2
%9 = load i16* %y, align 2
%10 = add i16 %8, %9
%11 = add i16 %7, %10
store i16 %11, i16* %w, align 2
%12 = load i16* %x, align 2
%13 = icmp ugt i16 %12, 2
br i1 %13, label %14, label %38
```

b) IR of a Target Basic Block

```
// Piled LLVM instruction for variable x
%msg_vec_in = alloca [4 x i16], align 2
%2 = getelementptr inbounds [4 x i16]* %msg_vec_in, i16 0, i16 0
%3 = load i16* %2
```

```
// Piled LLVM instruction for variable y
%msg_vec_in = alloca [4 x i16], align 2
%4 = getelementptr inbounds [4 x i16]* %msg_vec_in, i16 0, i16 1
%5 = load i16* %4
```

```
// Piled LLVM instruction for variable w
%x = alloca i16, align 2
%y = alloca i16, align 2
%6 = load i16* %x, align 2
%7 = add i16 %6, 10
%8 = load i16* %x, align 2
%9 = load i16* %y, align 2
%10 = add i16 %8, %9
%11 = add i16 %7, %10
```

```
// DFG expression of MBB
// Expression in Verilog Behavioral Model
// Assume that the target MBB is Root

assign x_wire = msg_vec_in[ROOT_CNT];
assign y_wire = msg_vec_in1[ROOT_CNT];
assign w_wire = (x_wire + 10) + (x_wire + y_wire);

always(@posedge clk) begin
  x[ROOT_CNT] <= x_wire;
  y[ROOT_CNT] <= y_wire;
  w[ROOT_CNT] <= w_wire;
end
```

c) DFG expression of each target variable

Figure 3.9: An Example to Construct Data Flow Graph without Data Dependency

Figure 3.9a) shows a simple basic block written in C++. It does not have any combined basic block, hence it organizes a MBB. Figure 3.9b) described its IR. In this example, LLVM `store` instructions point to variables `x`, `y`, and `w`. They are assigned in the variable list shown in Figure 3.9c), and each variable points to piled LLVM instruction as a vector form. According to the variable list, wires for variables `x`, `y`, and `w` are organized and the back-end compiler knows that the wires for `x` and `y` are represented by LLVM `load` instructions as array `msg_vec_in0` and `msg_vec_in1`, respectively. In variable `w`, it references `x` and `y` variables following LLVM instructions for the `w` variable. Hence, `w` is constructed by the wires for `x` and `y` due to its dependency with variable `x` and `y`. Finally, the `always` block represents value in variables are synchronized by the system clock.

By putting code expression on a generated verilog behavioral model, the data path writer in the back-end compiler uses cascade index counter to collect correct value from the variable line. Each MBB has already known the current and next cascade index counter. Assuming that MBB of figuree 3.9a) is the root MBB, its expression in the verilog file is presented as shown in figuree 3.9c). The expression of this MBB is always performing the same computation in the different variable line whenever input messages come.

3.3 Chapter Summary

In this chapter, we discussed the network lookup engines with the high-level program language called SWSL. With the compilation techniques, it automatically generates the network lookup engines from rhe PLUG programming model. Hence, it follows conventional intermediate representation to realize hardware design. To achieve higher throughput, in addition, it follows pipeline structure with the data flow graph of PLUG applications. We propose algorithms that create the pipeline structures and control logic. However, it is only focused on the network lookup application with special APIs, so generalization is limited for other conventional applications. The next chapter will discuss another approach to support HLS for conventional applications.

4 CACHE-COHERENT HIGH-LEVEL SYNTHESIS FOR FIXED-FUNCTION ACCELERATORS

In this chapter, the compilation process to generate fixed-function accelerators is described for generic source code. As described in the previous chapter, major concerns to compile genic source code for HLSs are 1) long latency operations, 2) how to manage loop control and 3) how to handle memory access by the accelerator hardware. In the conventional HLS, first, they do not provide the long latency operation, such as floating points, due to complex scheduling. In a simple manner, delayed flip-flops are provided to support those operations. Second, managing loop control is not considered because they only focus on parallel execution of loop kernels. Finally, the third concern is leveraged by using scratchpad memory in the generated accelerator. Thus, the host processor is always turned on to manage both operations by itself. By the third concern, in addition, it requires a cache coherent mechanism between the host processor and generated accelerator because they have separate memory space and it degrades the hardware efficiency in terms of power. For this reason, generating the loop kernel in the source code is only emphasized in conventional HLSs.

The goal of the cache-coherent High-Level Synthesis (ccHLS) is to generate fixed-function accelerators with loop control and cache coherence free by following the DAE execution model. Thus, ccHLS generates two accelerator modules called the fixed-function compute accelerator and fixed-function memory access accelerator. During execution of accelerated region, the host processor is going into sleep except the data cache and execution of the loop kernel with all loop controls. Memory accesses are performed by generated fixed-function accelerators by directly accessing the data cache in the host processor. In order to organize the loop control and cache coherent free accelerators, this chapter develops an intermediate representation called the Extended-Access/Execute Program Dependence Graph (E-AEPDG) to generate separate fixed-function compute and memory access accelerators.

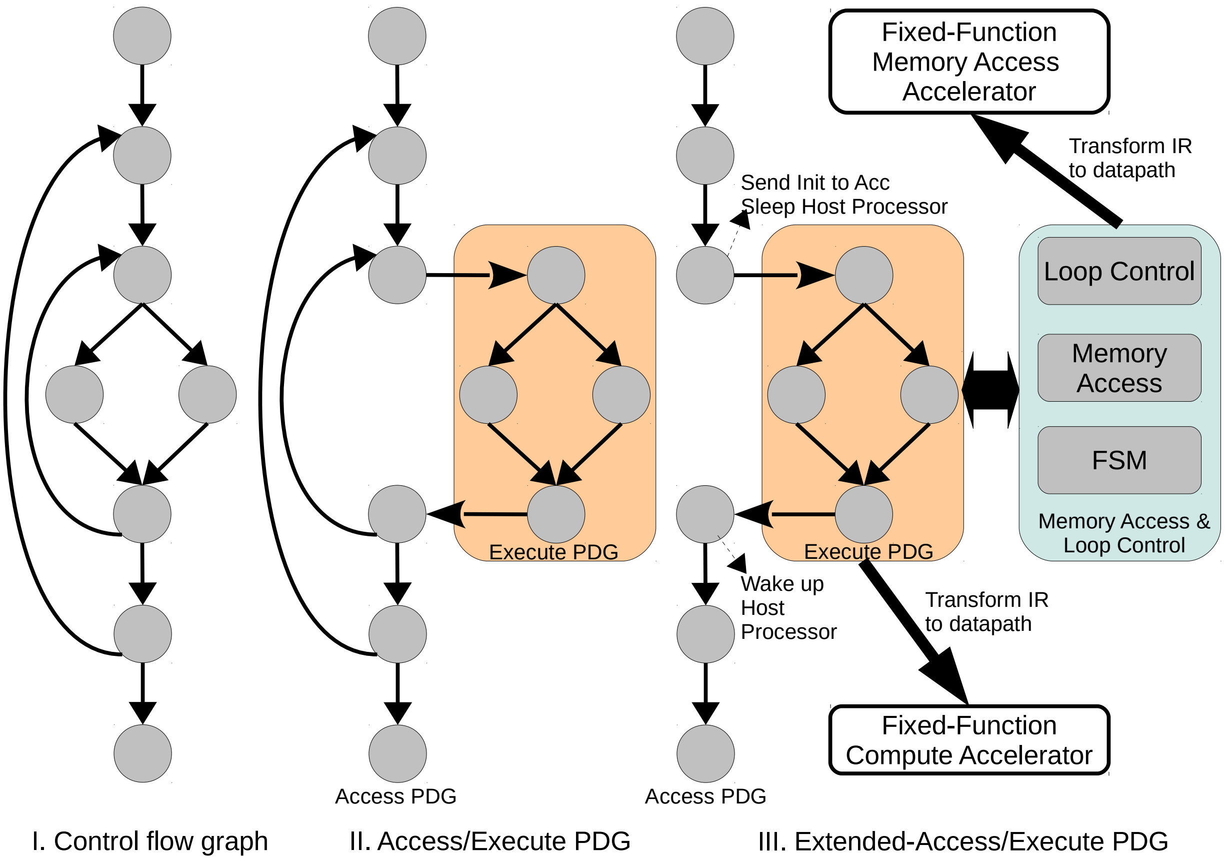


Figure 4.1: ccHLS Overview

4.1 Overview of the CcHLS

This section describes an overview of the ccHLS. The ccHLS targets for a generic program written in C/C++ and generates a loop accelerating region as fixed-function accelerators without utilizing any functionality except the data cache in the host processor. The host processor provides an initial loop induction variable to the fixed-function accelerators and is turned-off during execution of fixed-function accelerators. In the ccHLS, two major components are generated, which are the fixed-function compute accelerator and the fixed-function memory access accelerator. First of all, the fixed-function compute accelerator is the organization of the datapath of the loop execution kernel and only focuses on the computing kernel itself. The fixed-function memory access accelerator has a management role of loop control and memory access that are used in the fixed-function compute accelerator. Especially, the fixed-function memory access accelerator has two hardware modules called address generation units (AGUs) and finite state machine (FSM) for data movement. Figure 4.1 shows the high level compilation flow of the ccHLS. In step I, the ccHLS is the frontend to generate the intermediate representation (IR) from the source code as the control flow graph (CFG). Step II takes the IR as the input and generates an Access/Execute Program Dependence Graph (AEPDG), which is an intermediate representation to classify the IR to the computation kernel and memory accesses by slicing the program dependence graph into Execute-PDG (EPDG) and Access-PDG (APDG). In step III, the concept of AEPDG is expanded to provide loop control and induction variable injections, which is called Extended-AEPDG (E-AEPDG). From the E-AEPDG, each node and edge organizing E-AEPDG is transformed to hardware datapath for the fixed-function compute accelerator and the fixed-function memory access accelerator with I/O interfaces for data movement between fixed-function accelerators or among fixed-function accelerators and the data cache in the host processor.

The rest of this chapter is described as follows: Section 4.2 describe the Extended-Access/Execute Program Dependence Graph and how to construct the E-AEPDG from the CFG of input source code. Section 4.3 and 4.4 shows the transformation E-AEPDG to fixed-function accelerators and how to operate data movement between fixed-function accelerators or between fixed-function accelerators and the data cache in the host processor. Section 4.5 discussed the complex scenarios that may be possible in the program source code. In section 4.7, the implementation of the ccHLS is described. Section 4.8 concludes this chapter.

4.2 Extended-Access/Execute Program Dependence Graph (E-AEPDG)

The ccHLS requires several mechanisms to be expressed in the intermediate representation to generate the datapath of the fixed-function accelerators. From the initial idea that obeys the execution model in chapter 2, the ccHLS follows the decoupled access/execute execution model. Properties for this execution model should be presented in the intermediate representation. Therefore, the ccHLS provides i) pipelined datapaths for the fixed-function accelerators, ii) control capability in the datapath, iii) representing memory address calculation, and iv) loop control expression. The ext paragraphs detail mechanisms that should be represented in the intermediate representation for compilation.

Pipelined datapath for the fixed-function accelerators The intermediate representation for the ccHLS should provide a method to organize a specialized datapath. Except memory access operations, all computation is constructed as the chains of functional units to express dependencies and the intermediate representation needs to easily provide the dependencies among functional units.

Control capability in the data path The intermediate representation should support control instructions to perform select operation with a "valid" status bit. Accompanying the control flow graph (CFG), fixed-function accelerators are able to select the value using the phi-functional unit (ϕ-functional unit). Thus, ccHLS should define the control instruction in the datapath as predication form.

Memory address calculation The new intermediate representation should present memory address calculation for data access. In the conventional intermediate representation, it performs by using memory access instruction such as load/store and dependencies with memory address calculation. The ccHLS requires a new mechanism to transform those dependencies into hardware datapath.

Loop control expression The conventional intermediate representation describes loop control decision and induction variable as the form of control-flow graph and computation dependencies with the combination compare and branch instructions. The intermediate representation for ccHLS should easily represent this combination to create the datapath for loop control hardware datapath.

In order to express properties that ccHLS requires, we visit the Access/Execute Program Dependence Graph (AEPDG), which is proposed by Govindaraju thesis [33]. It is based on the Program Dependence Graph (PDG) and enhanced. This representation provides the capability

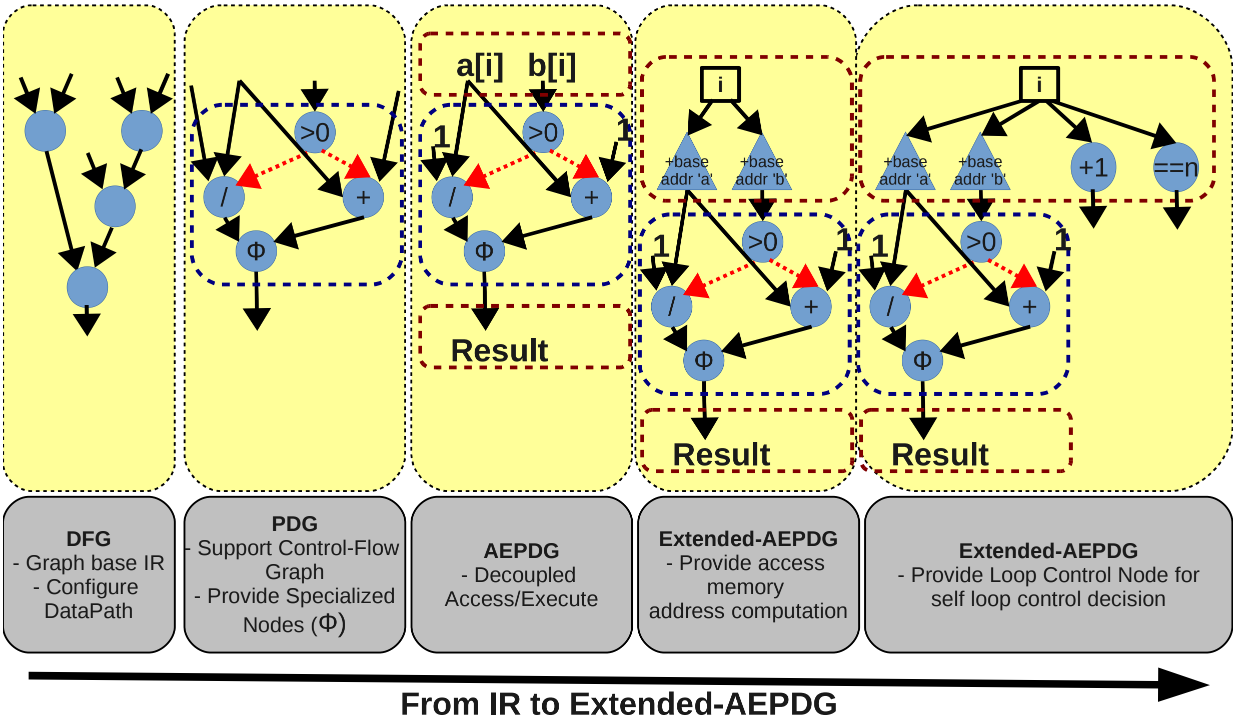

a[i] b[i]
>0
/
+
Φ
1
Result
i
+base addr 'a'
+base addr 'b'
+1
==n
DFG
- Graph base IR
- Configure DataPath
PDG
- Support Control-Flow Graph
- Provide Specialized Nodes (Φ)
AEPDG
- Decoupled Access/Execute
Extended-AEPDG
- Provide access memory address computation
Extended-AEPDG
- Provide Loop Control Node for self loop control decision
From IR to Extended-AEPDG

Figure 4.2: Conceptual Model from IR to Extended-AEPDG

to make explicit the data and control dependencies between instructions and partitioned into pure computation kernel and memory access/loop control decision. Thus, it can be easily mapped into the specialized accelerator called DySER for pure computation. Additionally, it executes memory access and loop control decision on the host processor. This separate operation enables the compiler to follow the DAE model. In addition, it provides the concept Access-PDG (APDG) and Execute-PDG (EPDG) to concrete the DAE model by assigning memory access as a node. By visualizing the memory access, hence, it easily deploys the correspondence between data movement between EPDG and APDG.

However, AEPDG does not provide memory address calculation and loop control expression. Since both operations are executed on the host processor, memory access operation is identified by the link of load/store node and memory address calculation is hidden in the AEPDG. Moreover, AEPDG does not have any clue when the acceleration is terminated and the host processor keeps providing data to be executed in the DySER. To relax such drawbacks, I develop the Extended-Access/Execute Program Dependence Graph (E-AEPDG), which visualizes the memory address calculation conjunction with memory operation such as load/store and expands the loop control decision in the AEPDG. Thus, E-AEPDG defines a new intermediate representation (IR) that provides a capability to exploit the data and control dependency of instructions and data movement among the accelerator and memory. Based on the E-AEPDG of the target source code from ccHLS, fixed-function accelerators are feasible.

Figure 4.2 shows the conceptual model of E-AEPDG from conventional IR of source code. First, it is possible to organzie the datapath from the dataflow graph (DFG) from IR. Then, the idea of PDG provide the specialized node to support control flow graph. AEPDG partitions PDG graph into execute and access subgraph. E-AEPDG is the extension to support memory address computation and loop control nodes to decide loop condition. The next two subsections describe how to construct E-AEPDG from the source code.

4.2.1 Access/Execute PDG Construction

From the source code, the target to be organized as fixed-function accelerators is the most frequently executed regions, especially loops. To identify the most frequently executed region in the source code, the compiler community has proposed many analysis techniques such as hyperblock [55], superblock [29] , or inner loop. For a simple way, pragrmas are inserted into the source manually. In the ccHLS, inserting pragmas into the source code is basically used, otherwise searching loop analysis techniques are used to collect loop headers. By the loop analysis of the compilation techniques, in case no pragma is provided in the source code, the outer-most loop header basic block can be found with the sequence of inner loop chains of the outer-most loop header. That is the target region to be constructed as fixed-function accelerators.

Once the region is identified, the ccHLS organizes the form of PDG by existing techniques [28] and AEPDG proposed by Govindaraju [33]. From the control flow graph (CFG), to construct the AEPDG, the ccHLS classifies instruction nodes in the CFG as memory access nodes, loop control nodes, and other nodes. Memory access nodes such as loads and stores instruction nodes are found and their relevantly dependent instruction nodes for memory address computation are assigned as the APDG. The loop control nodes are reserved to extend AEPDG for describing loop control dependencies in the E-AEPDG. The remaining instruction nodes, except the above

```
...
for(j=0; j<n; ++j) {
   for(i=0; i<m; ++i) {
      if(a[i]>0) {
         d[i] += 1/b[c[2*i]];
      }
      else {
         d[i] += b[c[2*i]]*2;
      }
   }
}
 ...
```

Source Code

```
entry
br label BB0
BB0:
%j = phi i32 [0, entry], [%j.next, BB5]
br label BB1

BB1:
%i = phi i32 [0, BB0], [%i.next, BB4]
%0 = getptr inbounds i32 *%a, %j
%1 = load i32 *%0
%2 = icmp sgt i32, %1, 0
%3 = getptr inbounds i32 *%d, %i
%4 = load i32 *%3
%5 = shl %i, 1
%6 = getptr inbounds i32 *%c, %5
%7 = load i32 *%6
%8 = getptr inbounds i32 *b, %7
%9 = load i32 *%8
br %2, label BB2, label BB3

BB2:
%10 = sdiv 1, %9
br label BB4

BB3:
%11 = shl %9, 1
br label BB4
```

Continued to next column

Continued from previous column

```
BB4:
%.pn = phi i32 [%10, BB2], [%11, BB3]
%strmerge = add i32 %.pn, %4
store i32, %strmerge, *%3
%i.next = add i32 %i, 1
%exitcond = icmp eq i32 %i.next, m
br %exitcond, label BB5, label BB1

BB5:
%j.next = add i32 %j, 1
%exitcond1 = icmp eq i32, %j.exit, n
br %exitcond1, label %exit, label BB0

exit:
ret void
```

Intermediate Representation by the Control Flow Graph

Figure 4.3: Target Source code and Intermediate Representation by Control Flow Graph

two classifications, form the EPDG. From the perspective of the host processor, memory access and loop control operations are performed in the host processor, hence the execution of the relevant instruction nodes for memory access and loop control is not depicted in the AEPDG.

In this chapter, the source code in figure 4.3 is used as a case study. The source code is enough to explain the complex cases including the nested loop and the form of indirect memory access. According to loop analysis, there are two loops in the intermediate representation by the control flow graph and the inner-most loop is the kernel to be represented as the AEPDG form. Figure 4.4 shows its control flow graph and the loop kernel to be generated as the fixed-function accelerators. The AEPDG represented in figure 4.4 consists of the EPDG for the loop kernel and APDG which is the fingerprint of memory access. The structure of AEPDG follows the pipelined datapath for data flow execution and has control capability by predication. However, it is insufficient to calculate a memory address for loads or stores and the loop control decision is not expanded in the AEPDG. Therefore, the AEPDG is to be extended to have identifiers

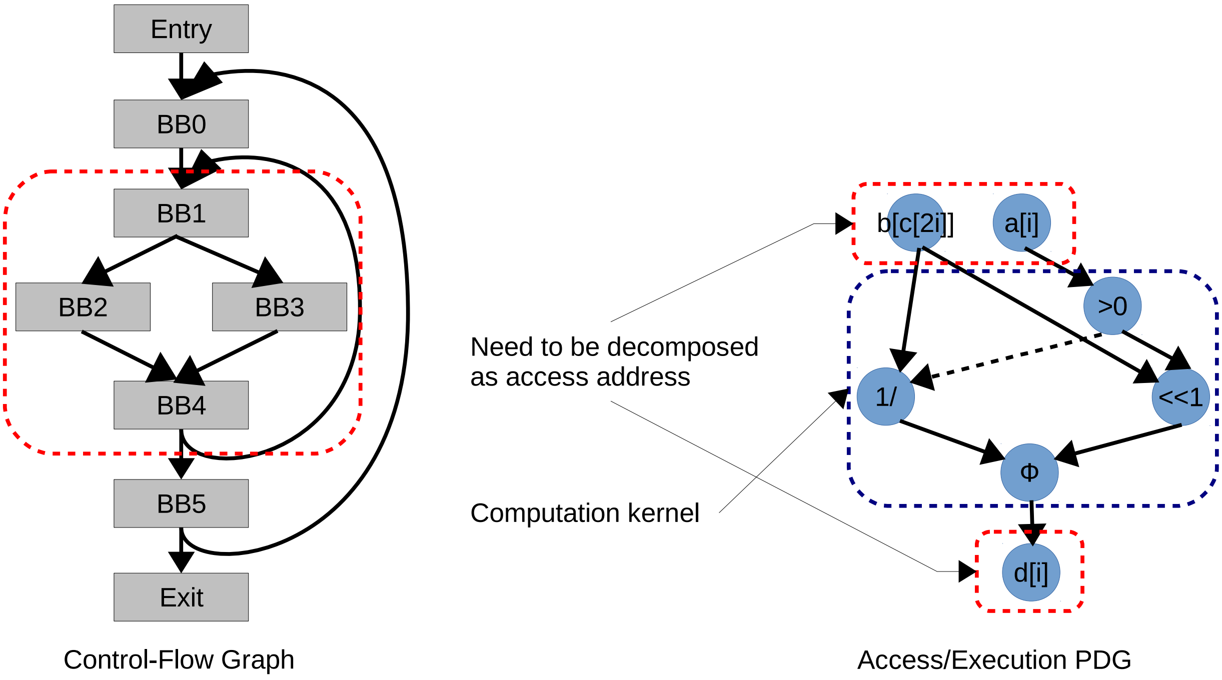


Figure 4.4: Control Flow Graph and Access/Execute PDG

describing memory address calculation and the decision path of loop control.

4.2.2 Extended-Access/Execute PDG Construction

The main role of APDG is to handle the access EPDG and data movement from/to memory. Hence, it only focuses on interfacing EPDG for data movement from memory by the instructions such as loads or stores. In addition, it has a role for data movement from variable or constant value to execute. The APDG is shown as the group of instructions to describe memory operations and data movement. In APDG, however, it is not enough to represent dataflow for memory access computation or data movement because it only gives hints as to what data moves into EPDG. Figure 4.4 shows that APDG only provides the traces of memory access or data movement as nodes. Thus, it requires the form of dataflow to calculate the memory address and its detail expansion should be mapped to transform source code to static hardware datapath. In addition, loop control expression of the target acceleration region is not presented in APDG. To relax these obstacles, the ccHLS extends the APDG with those properties. In APDG, its memory access nodes are decomposed to illustrate memory address calculation and loop control subgraph,

Instruction	Role
Arithmetic	- Induction variable: Arithmetic node for loop control subgraph to increment induction value of loop, connected to output EQ - Elsewhere: Arithmetic node of access or execution subgraph
Lod/Store	- Constructing output of address calculation node of access subgraph, connected to output EQ
Compare	- In loop exit BB: node for loop control subgraph, connected to output EQ - Elsewhere: compare node of access or execution subgraph
Branch	- In loop exit BB: Link edge of compare node for loop control subgraph to output EQ - Elsewhere: Edge for predication, edge to link phi(ϕ) for execute subgraph
Phi(ϕ)	- In loop head BB with induction variable: Input EQ to inject induction variable for loop control subgraph - Elsewhere: Node for phi(ϕ) to predicate for execute subgraph

Table 4.1: Classification by instruction type and position

including induction variable increment, and loop decision is conjoined with APDG.

ccHLS constructs the AEPDG into E-AEPDG based on instruction type and location in the control flow graph. Instructions are identified by the mixing of type and location and the APDG is transformed to the extended graph form based on identified instructions. Table 4.1 shows the type of instructions and the role based on the location in the control flow graph. The role of instructions, depending on the instruction type and location, is as follows:

Arithmetic instruction Arithmetic instructions mainly define the computing node and set the skeleton of subgraphs. In case that the instruction is an induction variable, it is used to increment induction value and assigned to loop control subgraph. When it is the induction variable, it has an edge to pass data outside of E-AEPDG.

Memory access instruction For memory access instructions, such as loads or stores, they create nodes to compute addresses with a base address and provide a memory access address to load or store data interacting with memory. The output edge of the load is connected with the node that has the dependency with that load instruction, while the output edge of the store just provides address and data to be stored.

Compare instruction Compare instructions have various roles depending on their location. In the case that the instruction is in the loop exit basic block, which is the basic block that exits loop in the control flow graph, the ccHLS replaces it with the node for the loop control subgraph and the node makes the decision for the case of loop exit. Otherwise, it is only used for the compare node for predication to provide control capability in the datapath depending on the condition.

Branch instruction Branch instruction, which is usually accompanied by the compare instruction, provides the edge to connect for comparison output either outside of the graph or selecting the valid output of predication result for control capability. The case that a branch instruction is located in the loop exit basic block means that the iterating loop is determined by the result of the condition instruction referenced by this branch instruction. The remained case of the branch instruction is for providing the edge to set predication and influence on value decision such as if-else statement. For an unconditional branch, it just sets a link edge for the connected basic block in the control flow graph.

Φ-instruction (ϕ-instruction) ϕ-instruction is the node to assemble output from previous nodes and set its output depending on validation. In case that the ϕ-instruction with an induction variable is located in the loop head basic block, which is the basic block that begins the loop, it has the role to inject either the initial value of loop iteration or the passing induction value to loop.

Creating E-AEPDG from ccHLS is represented in algorithm 4. From the initial PDG, it collects elements of APDG including edges and nodes. Loading and storing nodes are related to constructing APDG and the collection of those nodes is used as the skeleton of APDG nodes. Except those nodes, the remaining nodes are used for nodes for EPDG. Then, APDG is proliferated to describe extension including memory address computation and loop control depending on the instruction type and position in the control flow graph. At the same time, IO edges are collected to identify the IO connection. Table 4.2 illustrates edges and nodes used to describe E-AEPDG. For edge, the black edge stands for the normal dependency edge which represents that data dependency is existed between connected nodes. The remaining edges are used for identification when E-AEPDGE is transformed to fixed-function accelerators and provide input/output interface between fixed-function accelerators or between a fixed-function accelerator and memory. The input and output edges, which are red and blue edges, tell that it

Algorithm 4 E-AEPDG

```
$worklist \leftarrow \emptyset$
$apdg \leftarrow \emptyset$
$extension \leftarrow \emptyset$
$LoopExitBB \leftarrow$ {Loop Exit Basic Blocks From Control Flow Graph}
$LoopHeadBB \leftarrow$ {Loop HEAD Basic Blocks From Control Flow Graph}
{Initialize worklist for slicing: Loads, Stores}
for all $node \in pdg$ such that $node \in$ LOADS($pdg$) do
    $worklist \leftarrow worklist \cup node$
end for
for all $node \in pdg$ such that $node \in$ STORES($pdg$) do
    $apdg \leftarrow apdg \cup node$
    $worklist \leftarrow worklist \cup$ GET-ADDRESS($node$)
end for
{Add branches that are latches to loops to $apdg$}
for all $node \in pdg$ such that $node \in$ IS-LATCH($pdg$) do
    $apdg \leftarrow apdg \cup node$
end for
{Slice PDG}
while HAS-ELEMENT($worklist$) do
    $node \leftarrow$ POP($worklist$)
    $apdg \leftarrow apdg \cup node$
    for all $op \in$ GET-OPERANDS($node$) such that $op \notin apdg$ do
        $worklist \leftarrow worklist \cup op$
    end for
end while
$epdg \leftarrow pdg - apdg$
{Initalize IO Edges}
$i\_edges \leftarrow \emptyset$
$o\_edges \leftarrow \emptyset$
{Extend APDG}
for all $CMPnode \in apdg$ such that $node \in LoopExitBB$ do
    $extension \leftarrow extension \cup CMPnode$
    $o\_edges \leftarrow o\_edges \cup outputedgeofCMPnode$
end for
for all $Branchnode \in apdg$ such that $node \in LoopExitBB$ do
    $extension \leftarrow extension \cup Branchnode$
    $o\_edges \leftarrow o\_edges \cup outputedgeofBranchnode$
end for
for all $PHInode \in apdg$ such that $node \in LoopHeadBB$ do
    $i\_edges \leftarrow i\_edges \cup inputedgeofPHInode$
end for
{IO Edge of epdg}
for all $node \in epdg$ do
    for all $op \in$ GET-OPERANDS($node$) such that $op \notin epdg$ do
        $i\_edges \leftarrow i\_edges \cup inputedgeofnode$
    end for
    for all $use \in$ GET-USES($node$) such that $use \notin epdg$ do
        $o\_edges \leftarrow o\_edges \cup outputedgeofnode$
    end for
end for
$e - aepdg \leftarrow (apdg, epdg, extension, i\_edges, o\_edges)$
return $e - aepdg$
```

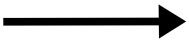
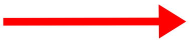
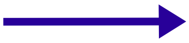
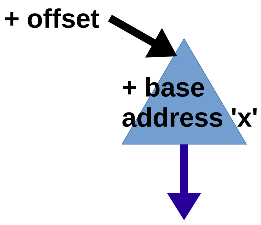
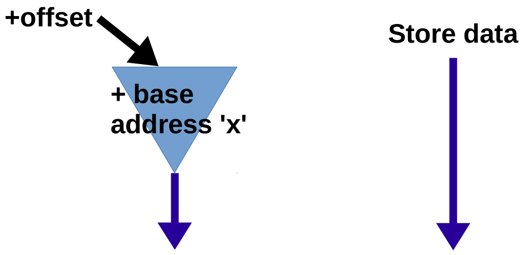
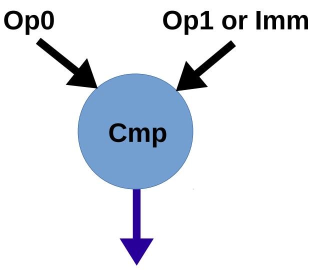
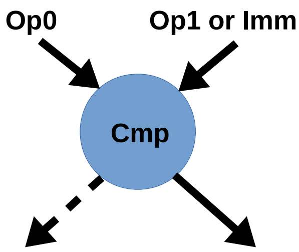
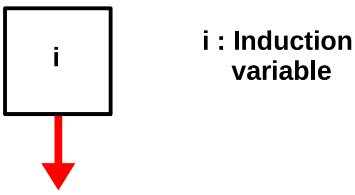
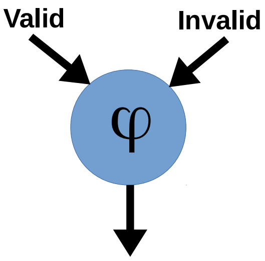
Edge					
Normal Dependency Edge		Input Link Edge		Output Link Edge	
Node					
Load					
Store					
Branch	In Loop Exit BB 		Elsewhere 		
Phi(ϕ)	In Loop Head BB 		Elsewhere 		

Table 4.2: Symbolic Representation in Extended AEPDG

is used as input and output interfaces for data movement and defines the pathway for entering or withdrawing data. Related to edges, the ccHLS organizes E-AEPDG with modified nodes for describing nodes depending on purposes. The property of node is as follows:

Load node Load instruction is deployed by the load node, which is illustrated as an upper-triangle. In the load node, it consists of the base address to be accessed and the offset to compute the accurate memory address. The load node has the dependency with the offset for address computation, hence it is coupled with a type of edges. To pass the computed memory address, its output is linked with the output edge and the output departs from the generated fixed-function accelerators based on E-AEPDG.

Store node Similar to the load node, it computes the memory address to store data that has the data dependency with this store node. Thus, the store node illustrated as a lower-triangle has the base address and offset to present memory address calculation. Unlike the load node, it should define the data to be stored in the computed memory address. The output edge for storing data is connected from the node that has the data dependency with the store instruction. Thus, computed address and storing data should be outgoing from E-AEPDG.

Compare node The compare node always accompanies the branch instruction. Branch instruction just links by the edge depending on the location. In the case that a branch instruction is in the loop exit basic block, it identifies that loop condition decision depending on the compare node. Thus, a branch instruction is transformed to an output link edge for the loop control subgraph and used for loop condition decision. In the remaining cases, however, the branch instruction with a related compare node is used for predication representation.

Φ-node (ϕ-node) The ϕ-node is able to be used for two cases. In the loop head basic block, it is an identifier that injects either initial induction value or incremented induction value. ϕ-instruction to organize the node notifies where the induction value comes. In the case that induction value comes from the outside loop, it provides the initial induction value. Otherwise, the incremented induction value is provided to the node. In any case that the ϕ-node is not

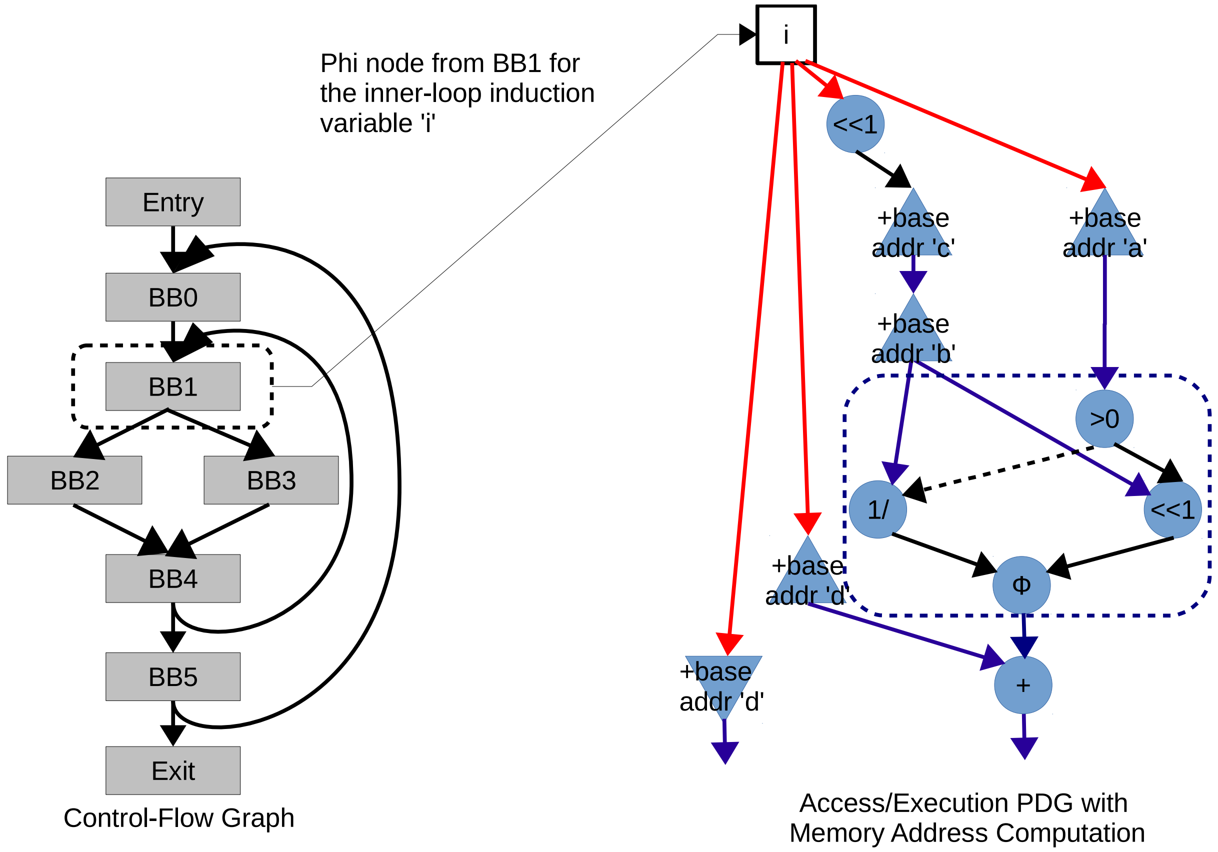


Figure 4.5: Access/Execute PDG with Memory Computation Nodes

located in the loop head basic block, the pure ϕ-node is organized in the E-AEPDG to decide predication.

4.2.3 E-AEPDG by Example

Based on edges and nodes from table 4.2, the ccHLS is able to decompose memory access nodes into the memory address with computation nodes and describe loop conditions. From figure 4.3 and 4.4, details of organizing E-AEPDG is illustrated. First, figure 4.4 only shows the AEPDG form of the source code in figure 4.3. It has two loads (b[c[$2 \times i$]] and a[i]) and one store (d[i]) and they should be decomposed with memory address computation. Indirection memory access in b[c[$2 \times i$]] is given as the load link between b and c, and c[$2 \times i$]] is used for the offset of b. In addition, c[$2 \times i$]] requires offset computation for $2 \times i$ and this operation is transformed to 2 bit left shift. All the dependencies in b[c[$2 \times i$]] are presented as the appropriate nodes and edges.

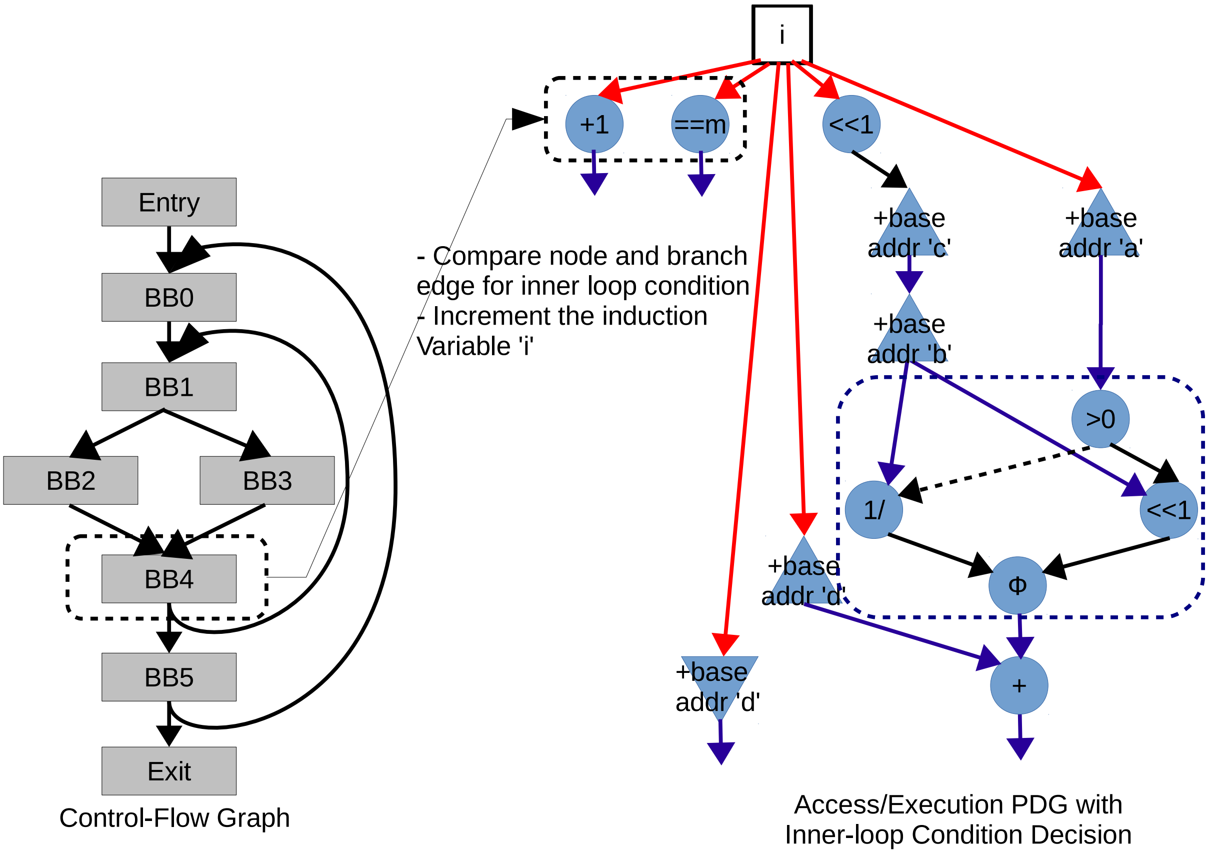

Entry
BB0
BB1
BB2
BB3
BB4
BB5
Exit
Control-Flow Graph
i
+1
==m
<<1
+base addr 'c'
+base addr 'a'
+base addr 'b'
>0
1/
<<1
Φ
+base addr 'd'
+base addr 'd'
+
- Compare node and branch edge for inner loop condition
- Increment the induction Variable 'i'
Access/Execution PDG with Inner-loop Condition Decision

Figure 4.6: Access/Execute PDG with Inner-loop Condition

Another load operation, a[i], is relatively simple and deployed as the load node and offset i. Operation d[i] should be expanded to the load and store because of d[i] += 1/b[c[$2 \times i$]] or d[i] += b[c[$2 \times i$]] $\times 2$. Thus, one load node to load data from d[i] and one store for computation result from the EPDG are orchestrated in the E-AEPDG. In the AEPDG, all loads and stores require i as the initial offset to compute memory address and it is the incremental induction variable for inner loop. Basic block 1 has the ϕ-instruction for this induction variable, hence this ϕ-instruction is illustrated as the induction node and all relevant load and store nodes are linked with this node. Figure 4.5 shows the decomposition of memory address computation in the AEPDG.

Next, the ccHLS extends AEPDG to represent loop control for the acceleration. In figure 4.4, it has the form of nested loops and should be organized for creating both loops (outer and inner loops). Figure 4.6 shows the AEPDG with the inner loop condition. For the inner loop,

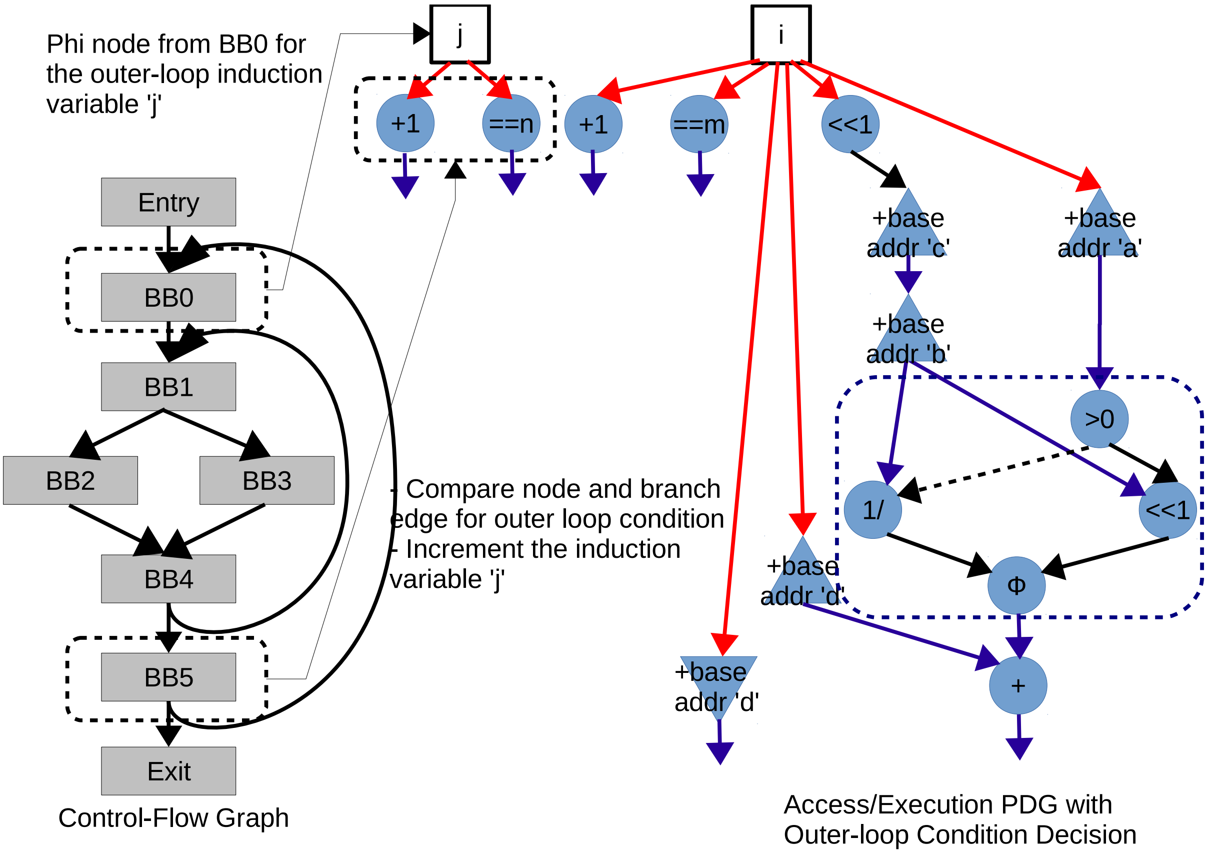


Figure 4.7: Extened Access/Execute PDG

the ccHLS analyzes instructions in the control flow graph and targets compare and branch instructions. In addition, it searches the ϕ-instruction in the loop head basic block to find the incremental induction value. In the basic block 4, it has branch and compare instructions to add the output edge and compare nodes. They are dependent on the incremental induction value. From the ϕ-instruction dependency, the ccHLS is aware that the induction value is $++i$ and it has the output edge for inject induction value into the induction node since the ϕ-node from the ϕ-instruction and the node for the incremental induction value have the relationship of data dependency.

Figure 4.6 emphasizes the inner loop condition decision. However, it still has another loop which encapsulates the inner loop. This follows in a similar manner what is described in figure 4.6 and organizes the loop control subgraph for the outer loop. From the control flow graph, it searches the ϕ-instruction in the loop head basic block of the outer loop and relevant compare and branch instruction in the loop exit basic block of the outer loop. In the basic block 0, which is the loop head basic block of the outer loop, ϕ-instruction is found and allows it to be presented as the induction variable node for `j`. In addition, the basic block 5, which is the loop exit basic block of the outer loop, provides compare and branch instructions and they are assigned as the node and edge for the loop control condition. Figure 4.7 finally shows the E-AEPDG generated from the ccHLS.

4.3 Transformation E-AEPDG to Fixed-Function Accelerators

In this section, generating fixed-function accelerators from E-AEPDG is explained. Two separate accelerators, the fixed-function compute accelerator and fixed-function memory access accelerators are the group of fixed-function accelerators. In E-AEPDG, the EPDG is transformed into the fixed-function compute accelerator. The fixed-function memory access accelerator has two major components including the address generation unit (AGU) and finite state machine (FSM) for data movement. The remaining nodes and edges in E-AEPDG, except the EPDG, are utilized for AGU and data movement operations are assigned for FSM. First, hardware components providing functionality and IO interface to communicate are discussed in section 4.3.1. Then, transforming from E-AEPDG to the fixed-function accelerators is described in section 4.3.2. In addition, the role of FSM in the fixed-function memory access accelerator is presented in this section.

4.3.1 Event Queues (EQs) and Functional Units (FUs)

The fixed-function accelerators are the chains made by several functional nodes to construct the datapath from E-AEPDG and executes the computing oriented code region. In addition, it requires the way to provide IO interface in input/output edge for data movement. To realize both properties, the ccHLS exchanges the functional nodes and input/output edges for function units (`FUs`) and Event Queues (`EQs`).

First of all, its input/output is connected to a buffer called an event queue (`EQ`) and passes data to/from the fixed-function compute accelerator. It has the role of temporary storage when data from the memory or other nodes arrives and stays until one consumes the data in the EQs. The ccHLS replaces input/output edges with EQs for data movement. Replaced EQs of input and output are called input EQs (iEQs) and output EQs (oEQs). From input EQs, data required for computation flows into the fixed-function accelerator and executes it in a dataflow fashion. FUs construct datapaths and operate various operations instead of instructions in the executable binary of the host processor. To avoid control dependency, predication is allowed and ϕ-function is provided as an FU to acquire the result. In the case that the target code region has control dependency, predication is adapted in the dataflow and decides which output from the FU is valid. Thus, the edge connected between FUs has two meta-bits, valid (V) and ready (R), with data bits and EQs also havving those bits for data movement. Figure 4.8 shows the structure of event queues. In iEQ, this is the form of FIFO with a meta-bit (V) in the input data. As soon as a Request is set from the connected node to get the data, a decoder examines the valid bit in the data. Once it is valid, iEQ sends the data to the requesting node. Otherwise, no action is required in iEQ because valid data has not arrived in iEQ yet. oEQ has a similar role to iEQ. However, oEQ itself requests data from the previously connected node. Depending on whether FIFO is full/busy, it schedules the request signal to get the data. If it gets data from the previous node, meta-data has arrived with the pure data. Meta-data (R and V bits) is used to decide the state in FSM for data movement.

To obviate congestion of the datapath, FUs have a flow control mechanism with a small finite state machine, and figure 4.8 expands the sequence of FU. An FU requests operands from previous connections and it is stalled until two operands are ready for flow control (req_OpA and req_OpB signal requesting operands to previous connections). Once they are ready (OpA_rdy and OpB_rdy notify a requested operand is ready), the functional unit passes two operands into the function pipeline which has various pipeline stages, depending on instruction (mul, div, and various floating point instructions have a deep pipeline stage, otherwise only one pipeline stage exists for execution.). When computation is done and the next connection requests output as one of the operands, output data moves one of operand registers for dataflow execution. In the case that next connection holds data in the target register and waits for another operand, the hold signal is set and lets the FU stall until the next functional unit requests data. To support

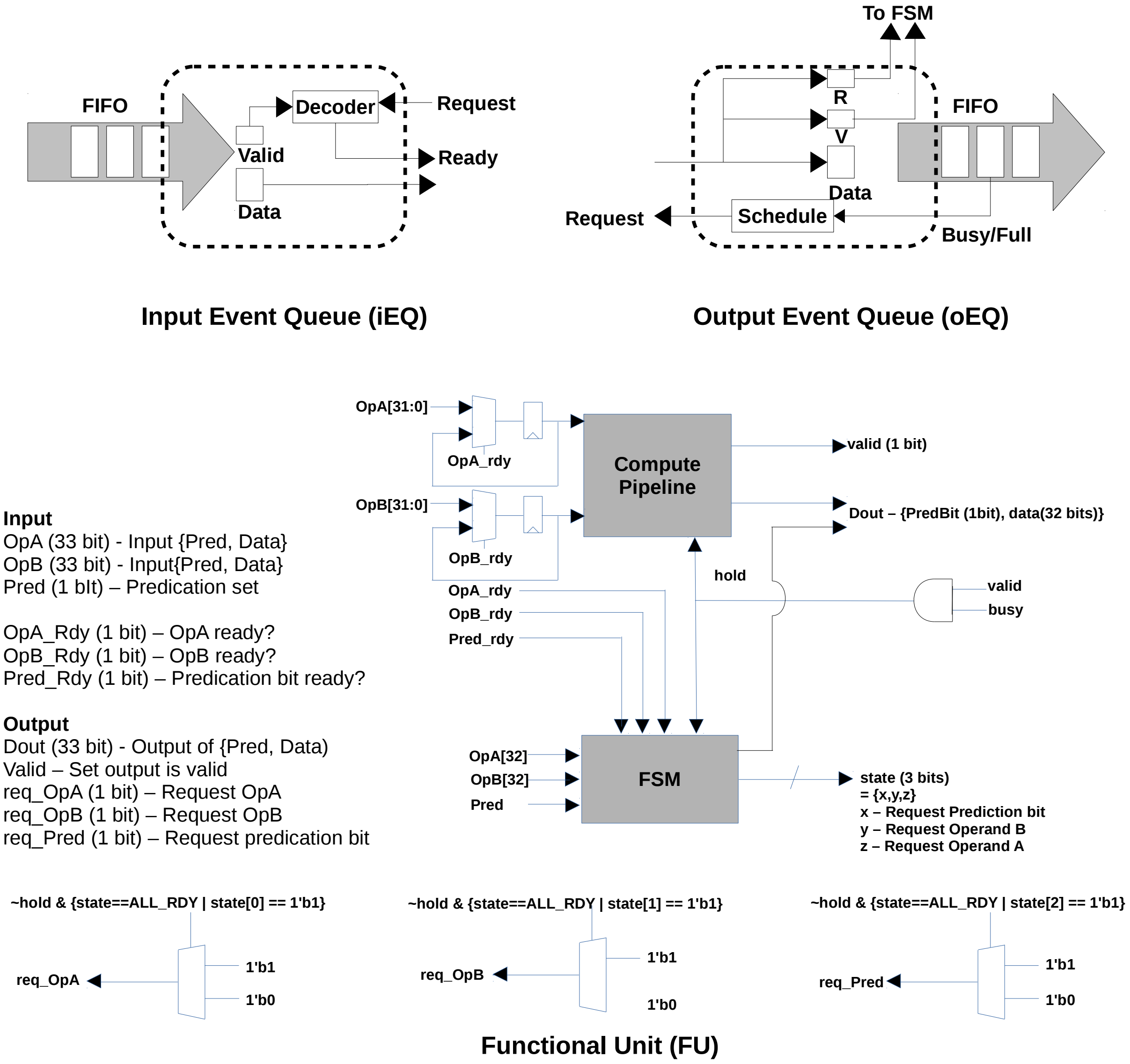

To FSM
FIFO
Decoder
Request
Valid
Ready
Data
R
V
Data
FIFO
Request
Schedule
Busy/Full
Input Event Queue (iEQ)
Output Event Queue (oEQ)
OpA[31:0]
OpA_rdy
Compute Pipeline
valid (1 bit)
OpB[31:0]
Dout – {PredBit (1bit), data(32 bits)}
OpB_rdy
hold
Input
OpA (33 bit) - Input {Pred, Data}
OpB (33 bit) - Input{Pred, Data}
Pred (1 bIt) – Predication set
OpA_rdy
OpB_rdy
Pred_rdy
valid
busy
OpA_Rdy (1 bit) – OpA ready?
OpB_Rdy (1 bit) – OpB ready?
Pred_Rdy (1 bit) – Predication bit ready?
Output
Dout (33 bit) - Output of {Pred, Data)
Valid – Set output is valid
req_OpA (1 bit) – Request OpA
req_OpB (1 bit) – Request OpB
req_Pred (1 bit) – Request predication bit
OpA[32]
OpB[32]
Pred
FSM
state (3 bits)
= {x,y,z}
x – Request Prediction bit
y – Request Operand B
z – Request Operand A
~hold & {state==ALL_RDY | state[0] == 1'b1}
~hold & {state==ALL_RDY | state[1] == 1'b1}
~hold & {state==ALL_RDY | state[2] == 1'b1}
req_OpA
1'b1
1'b0
req_OpB
1'b1
1'b0
req_Pred
1'b1
1'b0
Functional Unit (FU)

Figure 4.8: Event Queues and Functional Unit

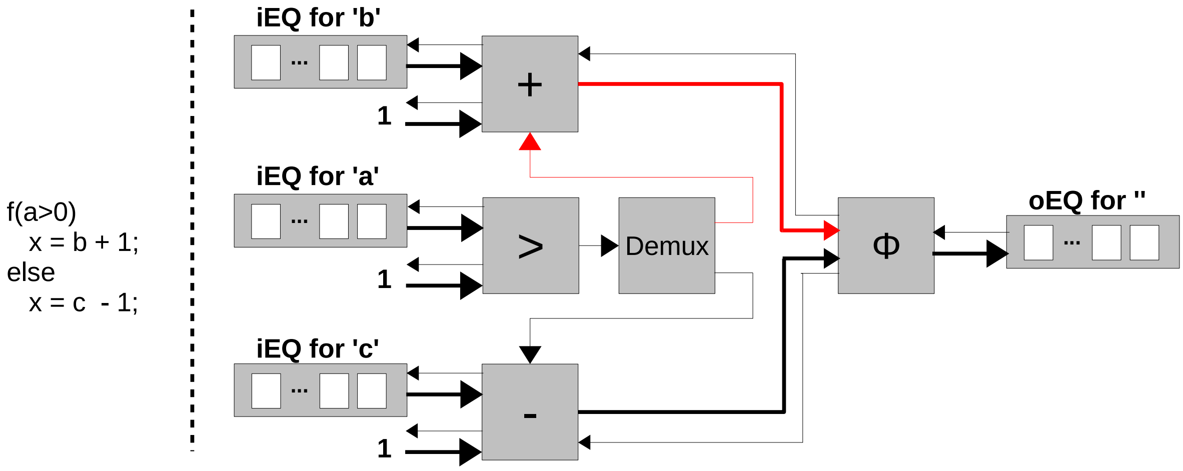


Figure 4.9: Constructing Datapath using EQs and FUs

predication, `Pred` and `Pred_rdy` are also provided and notifies that its output is selected for predication valid. Overall, this resembles a coarse-grained reconfigurable architecture-like paradigm. Almost, any conventional HLS technique when repurposed to handle control-flow and variable memory latency will likely resemble the substrate we have proposed.

To show the construction of the datapath using `FU`s and `EQ`s, figure 4.9 provides a simple `if/else` statement. It is consist of 3 `iEQ`s and 1 `oEQ`. Each `iEQ` depends on the `request` signal from a connected functional unit and data is sent to that functional unit once the data is valid in `iEQ`. For control dependency, a `compare` functional unit and `phi`(ϕ) functional unit are provided. For decision of control, demux is padded and linked with a compare functional unit. So, its output is only influences one of the functional units for computation (`b+1` or `c-1`), and the ϕ-function selects based on the predication decision. In `oEQ` for `y`, it keeps requesting data to the `phi`(ϕ) functional unit and data is passed to `oEQ` when the `phi`(ϕ) functional unit finishes execution. In case that `oEQ` is full or busy, data requesting is reset and data is not passed to `oEQ`.

The number of states in the FSM depends on code optimization in IR. If memory access operations or data movements are reduced in the target source code, the number of states in the FSM is decreased. Current ccHLS version does not provide a backend pass to optimize for decreasing load/store operations and data movement. Investigation on the complexity and optimality of our algorithm in terms of whether it selects the optimal number of states is future work.

4.3.2 Fixed-Function Accelerators from E-AEPDG

E-AEPDG is transformed to organize the datapath of fixed-function accelerators. Each edge link among nodes provides the wire connectivity among nodes and each node is replaced with the appropriate `FU`. For IO connectivity, input and output `EQs` are installed to interface with the inside and outside of fixed-function accelerators instead of input and output edges. By the meta-bits in `EQs`, valid (V) and ready (R), Boolean logic of two configuration bits of IO edges from flow data ingenerates and triggers state transition for data movement. E-AEPDG has memory operations such as load and store and the loop control also has data movement from an output edge to other input edges and those operations provide data movement action. A V bit gives a clue that data in the `EQ` is valid so that a specific operation is available. It is generally set by a `compare` instruction node to notify whether the loop condition is true or false. An R bit stands for ready data delivered from the instruction node connected to this `EQ`. The combination of Boolean logic of a bunch of IO `EQs` provides a set of action functionality and it is used to construct FSM to control the fixed-function accelerators interfacing with the memory in the host processor.

First, edges in E-AEPDG are transformed to construct the datapath of the fixed-function accelerators. Since the normal dependency edges provide the connectivity among the nodes, it is left without any modification to show dependencies. However, input and output edges are carefully transformed because they are evidences to communicate outside the fixed-function accelerators and give IO interface. The following paragraphs tell how to transform those edges.

Input edge Basically, an input edge is transformed to the input `EQ` (iEQ) for providing the capability of passing input data into the fixed-function accelerators. In the case that an input edge is derived from the induction variable node, that node is replaced with iEQ and the input edge connected with that node simply provides connectivity to the node that is linked with this input edge.

Output edge An output edge is considered in the two perspectives of connectivity, the non-connecting edge and the connecting edge with the node. In the case of the non-connecting edge, it is transformed to the normal dependency edge connected to output EQ (oEQ). Hence, the data passing through this edge arrives to oEQ and waits for data movement action by FSM. The connecting edge with the node, on the other hand, normally is used to provide the connectivity from load operations and informs the data dependency describing data load from the memory access address. Hence, it simultaneously requires iEQ and oEQ. In oEQ, it provides the memory access address and load data is injected into iEQ. Thus, the originated node of output edge is connected to oEQ and the node pointed by this edge has iEQ to get the data from the memory.

Second, the node is easily transformed by replacing it with an appropriate FU. In addition, the ϕ-FU is provided to map the native control and allows the ccHLS to support for control instructions. However, E-AEPDG has special nodes for load and store. The ccHLS replaces such nodes with the combination of computation for memory access address. These nodes are connected through output edges for providing memory access address and the combination of memory address computation is directly connected oEQ to after all.

Finally, the ccHLS needs to create the finite state machine (FSM) for data movement operations between the fixed-function accelerators and memory or between the fixed-function compute accelerator and the fixed-function memory access accelerator. The main role of the FSM is i) to provide the initial induction value, ii) to manage data movement in the fixed-function accelerators, and iii) loop control.

Initial induction value Initially, FSM provides the initial induction value for loop acceleration when the fixed-function accelerators are executed. Injecting the initial induction value lets the fixed-function accelerators know that acceleration begins and performs its properties.

Manage data movement FSM provides the operations for data movement. Basically, the fixed-function accelerators allow three operations, such as move, load, and store. First, the move operation is simply to move data in oEQ to iEQ. When the R bit in oEQ is set, data in oEQ is passed to iEQ for the fixed-function accelerator execution. Second, the load operation requests to load data from the memory address in oEQ to iEQ. As we discussed in the previous section, an output edge is decomposed into oEQ and iEQ for notifying memory address and input data injection. Thus, the address in oEQ is passed to the memory and loads the data into iEQ when the R bit in oEQ is set. Finally, the store operation requests to store data in $oEQ0$ to the memory address in $oEQ1$ because the memory access address and data are located in separate $oEQs$. Thus, the store operation is done when both the R bits in $oEQ0$ and $oEQ1$ are set.

Loop control The major invocation of loop is to compare conditions and increment the induction value. Thus, the fixed-function accelerators should always check that the comparison condition is valid and incremental induction value is ready. Thus, the V bit of $oEQ0$ for condition and the R bit in $oEQ1$ for the induction value set the state transition in FSM. In case that the source code has nested loops, the ccHLS should check the loop dependency and this dependency influences how to define the state. So, it is sensitive to the loop exit condition from the dependent loop and the loop exit condition of dependent loop organizes the Boolean logic to execute the loop.

4.3.3 Transformation E-AEPDG to Fixed-Function Accelerators by Example

Using the E-AEPDG from figure 4.7, we discuss the transformation E-AEPDG to the fixed-function accelerators. First, the induction variable node for j and i are replaced with $iEQ0$ and $iEQ1$. Then, output edges for loop control are also transformed to $oEQ0$, $oEQ1$, $oEQ2$, and $oEQ3$. The ccHLS analyzes that the source code has loop dependency, nested loops, and their relationship is reflected in the FSM. The FSM sets the initial induction value as state0 and the control for the inner loop (comparing and incrementing induction variable for the inner loop) is enlisted at state1 and state4. In state2 and state3, they notify the control for the outer loop (finishing the fixed-function accelerators and incrementing induction variable for the outer loop). Finishing acceleration depends on the decision by $oEQ1 == Valid$. Figure 4.10 illustrates the transformation of loop control in the E-AEPDG.

In the source code, it has indirection memory access `b[c[2×i]]` and is shown as the chain of load nodes. Between the load nodes for c and b, they are tightly connected by the output

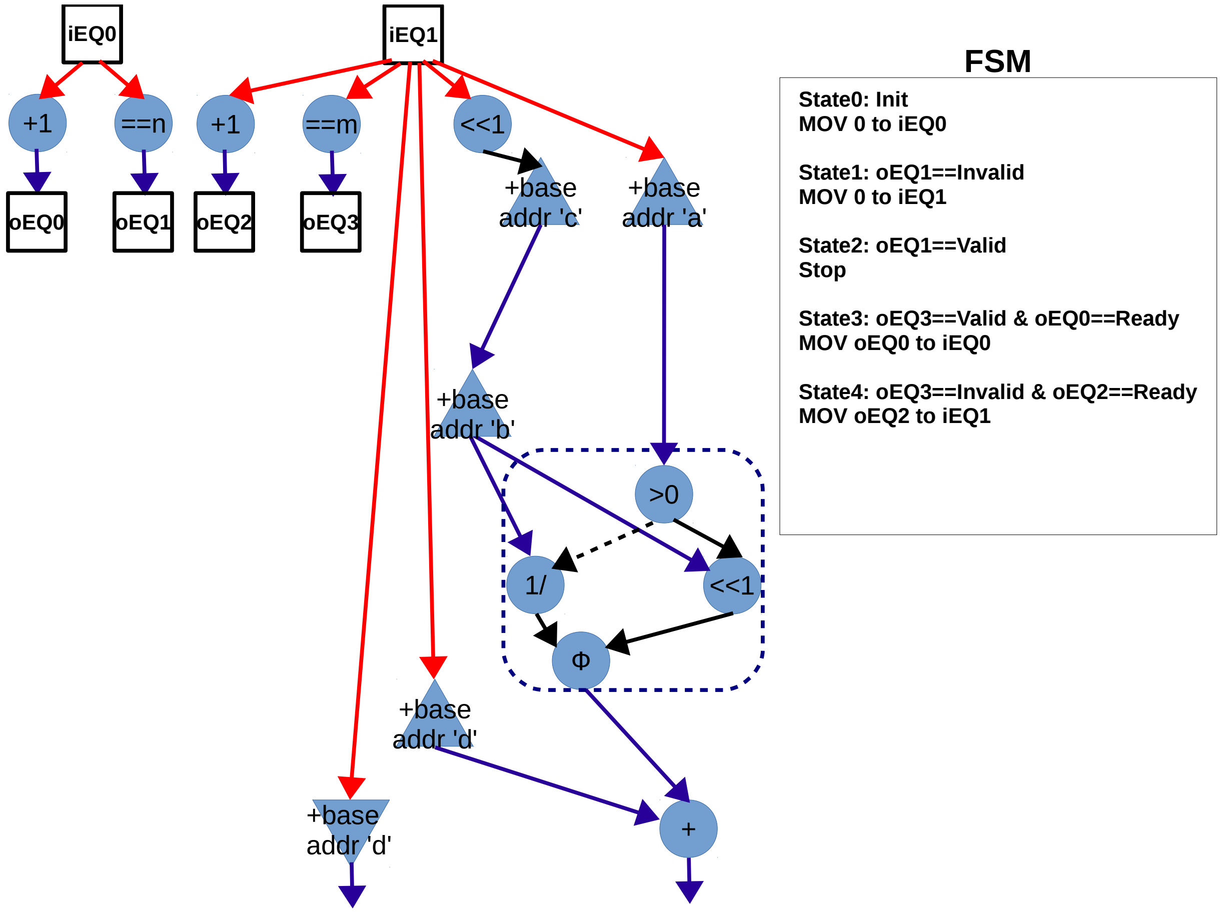


Figure 4.10: Transformation of Loop Control

edge to inject the load data of memory address $c\times\texttt{i}$ and this edge is transformed to EQs, $iEQ2$ and $oEQ4$ to provide the requested data into the datapath of fixed-function accelerators. FSM applies this operation to FSM, and `state5` for $oEQ4 == Ready$ manages the data movement in case the memory access address for $c \times i$ is ready. Requested data is loaded into the `iEQ2` and the fixed-function accelerators are executed in the dataflow fashion. Figure 4.11 presents the operation for indirection memory access in the fixed-function accelerators.

Next, the ccHLS transforms all the output edges that are linked with nodes for memory access. Target nodes are loads with base addresses a, b and d. Load node for a has the output edge connected to the 'compare' node, so it is transformed to $oEQ5$ and $iEQ3$ and this load operation is applied to `state6`. For b, it is also the load operation to inject the data from `b[c[2×i]]` to

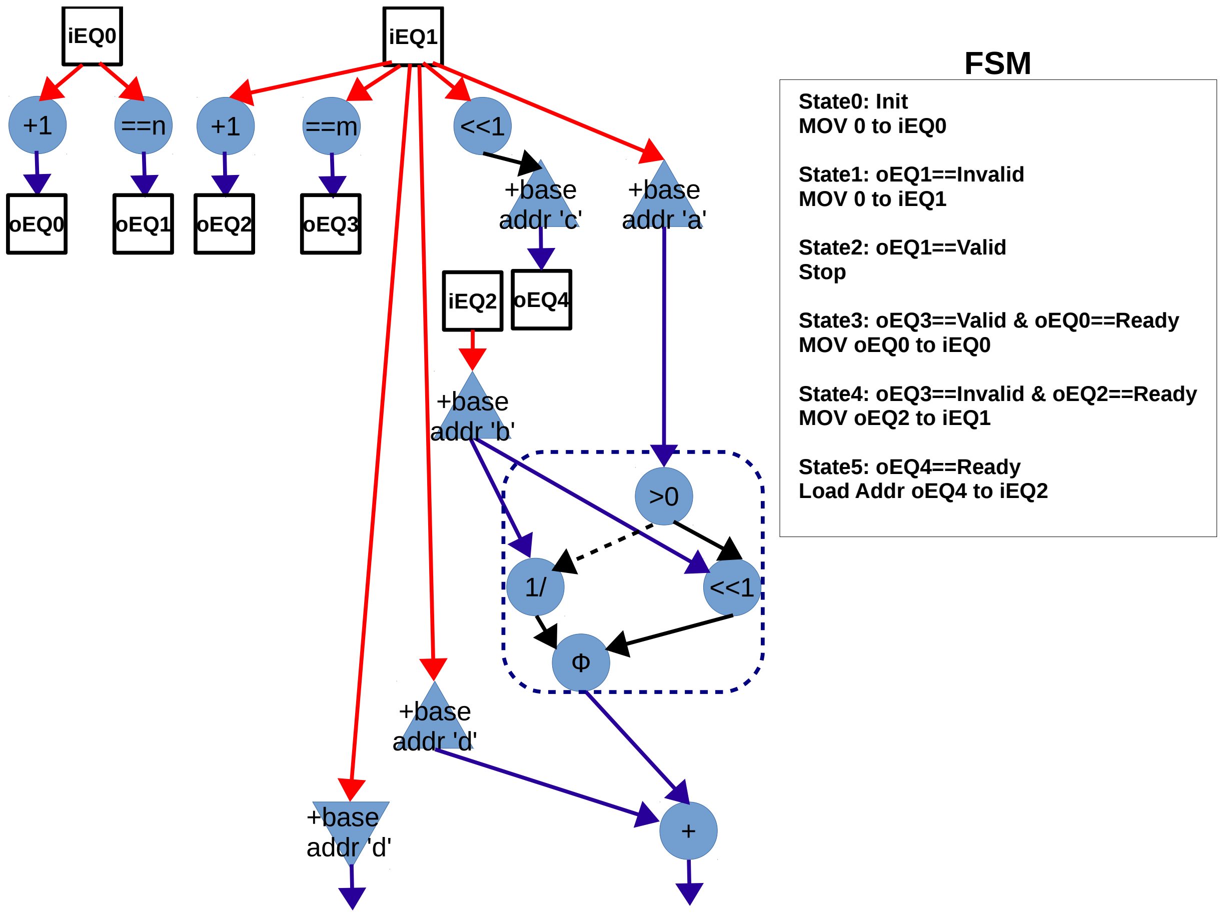
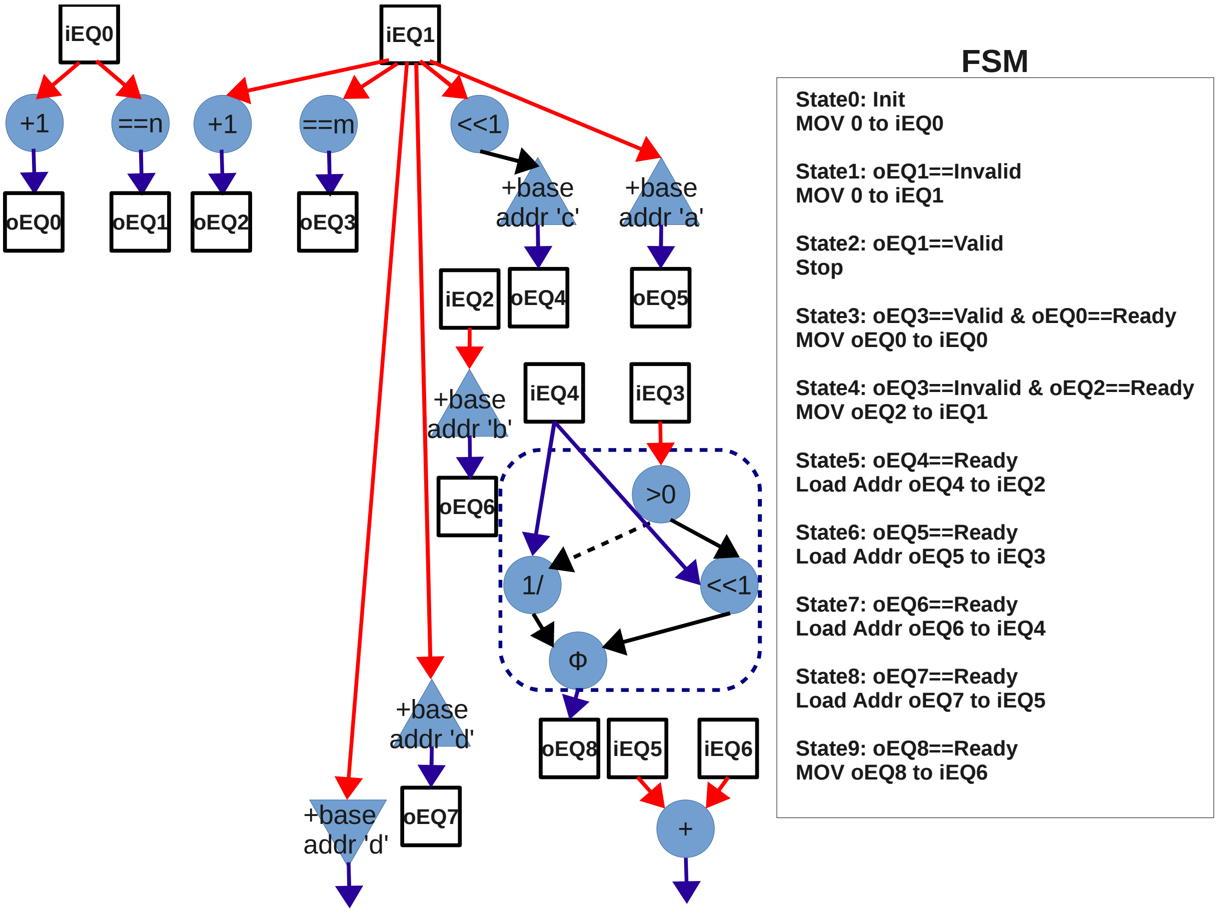


Figure 4.11: Transformation for Indirection Memory Access

EPDG (blue-dotted box). This output edge is decomposed to $oEQ6$ and $iEQ4$ and FSM reflects this load operation in the state7. The last load operation for d has the same characteristic with the loads for a and b, and it uses $oEQ7$ and $iEQ5$ and has state8. Unlike load nodes, the output edge connected between 'phi(ϕ)' node and 'add' node just represents the data dependency from the EPDG. Since the EPDG is used to construct the fixed-function compute accelerator, the output edge connects outside EPDG. Except memory access between these nodes, it is changed to $oEQ8$ and $iEQ6$ and its operation is to move data from $oEQ8$ to $iEQ6$ described in state9. The transformation for the output edges is shown in figure 4.12.

The remaining output edges are to store data to memory. To store data, the memory access address and the target data to store should be identified in the transformation. They are



State0: Init
MOV 0 to iEQ0

State1: oEQ1==Invalid
MOV 0 to iEQ1

State2: oEQ1==Valid
Stop

State3: oEQ3==Valid & oEQ0==Ready
MOV oEQ0 to iEQ0

State4: oEQ3==Invalid & oEQ2==Ready
MOV oEQ2 to iEQ1

State5: oEQ4==Ready
Load Addr oEQ4 to iEQ2

State6: oEQ5==Ready
Load Addr oEQ5 to iEQ3

State7: oEQ6==Ready
Load Addr oEQ6 to iEQ4

State8: oEQ7==Ready
Load Addr oEQ7 to iEQ5

State9: oEQ8==Ready
MOV oEQ8 to iEQ6

Figure 4.12: Transformation for Output Edges

represented as output edges and also have the dependencies for the store operation. Therefore, these output edges are replaced with oEQs and FSM should apply this data movement in itself. Figure 4.13 shows the organization for data store operation. $oEQ9$ and $oEQ10$ are used to provide memory access address and store data, respectively. In addition, state10 has this property to store data from $oEQ10$ to memory access address from $oEQ9$.

4.4 Execution

This section discusses the execution of the fixed-function accelerator. By transforming E-AEPDG to the fixed-function accelerators, the ccHLS is able to provide the hardware characteristics to accelerate the target region. The EPDG organizes the fixed-function compute accelerator and

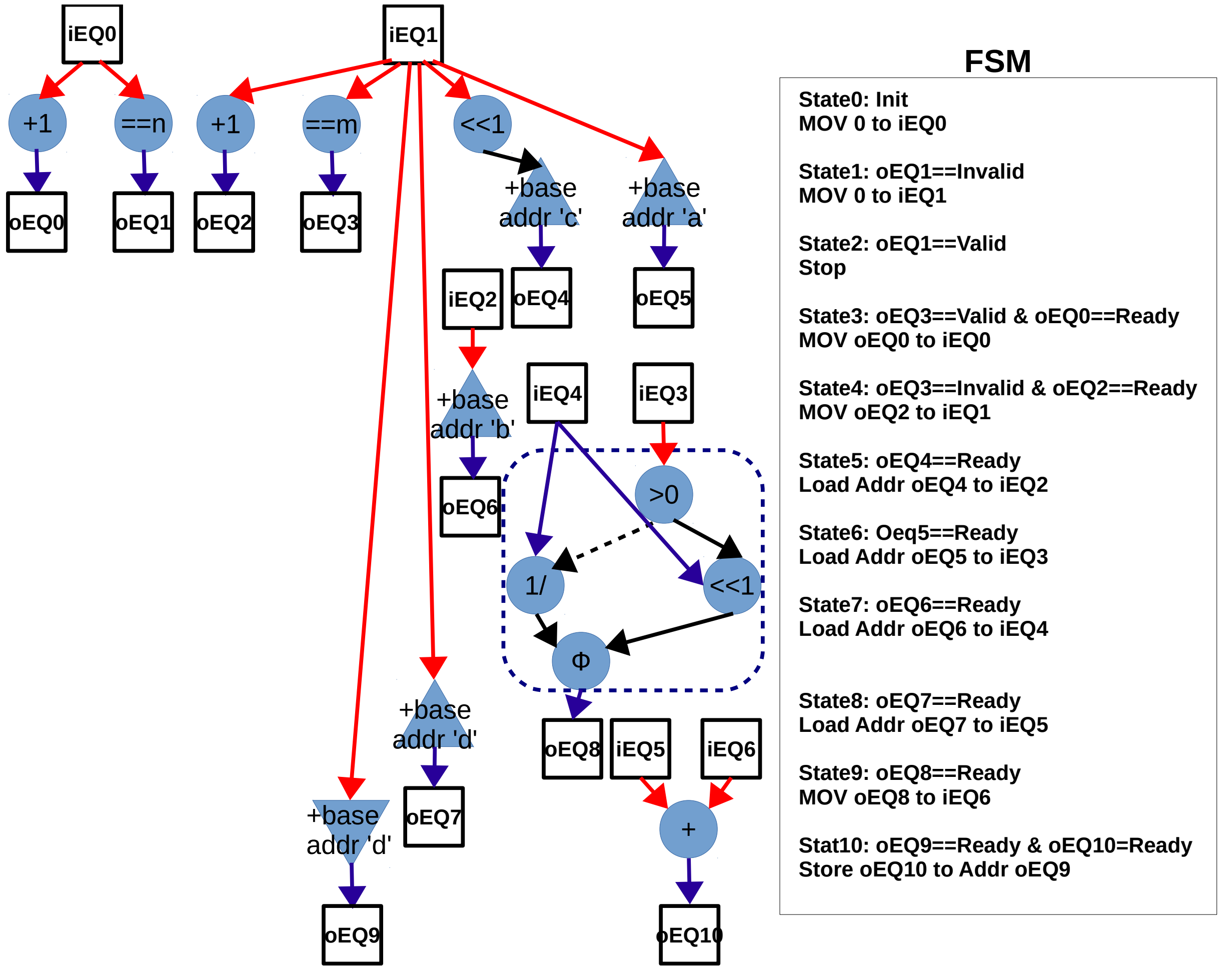


Figure 4.13: Transformation for Store Nodes

the remaining components with FSM in the transformation and has a role of the fixed-function memory access accelerator.

Figure 4.14 shows the fixed-function accelerators from the source code in figure 4.3. This example has a nested loop (inner-loop) in outer-loop. We consider 4 cases of execution: 1) outer-loop start, 2) inner-loop execution, 3) inner-loop stop, and 4) outer-loop stop. First, inner-loop starts following steps below:

- When the host processor faces the accelerating region, it sends initialization signal to FSM.
- FSM (`state0`) sends the initial induction value 0 to $iEQ0$ to begin outer-loop execution.
- Data from $iEQ0$ flows the datapath and results are resided in $oEQ0$ and $oEQ1$. $oEQ0$ has

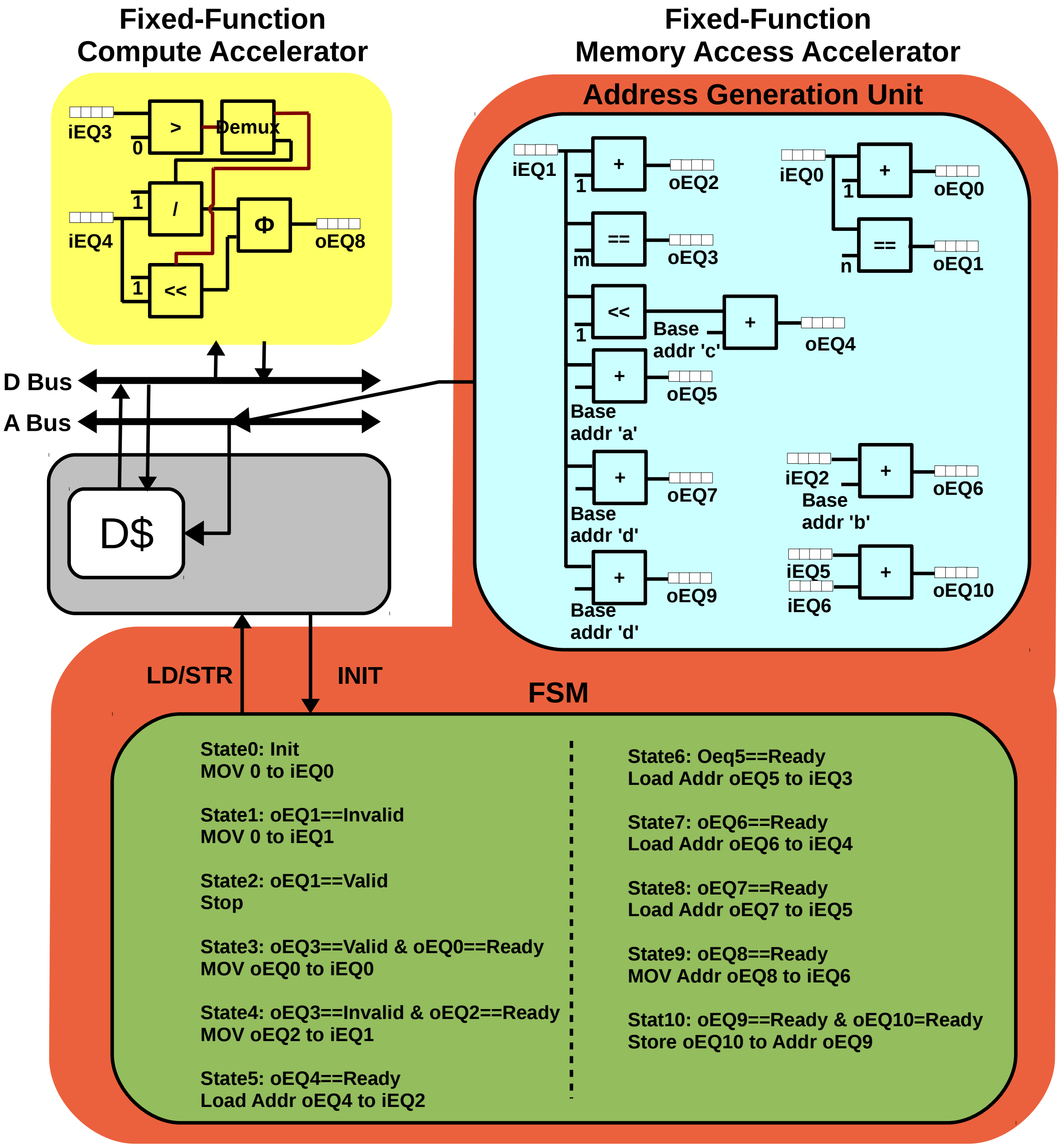


Figure 4.14: Execution

incremented induction value for outer-loop iteration and $oEQ1$ has a role to keep iterating of outer loop or not. In case that the V bit in $oEQ1$ is invalid, FSM (state1) sends the initial induction value 0 for the inner-loop to $iEQ1$.

- Outer-loop is hold until the execution of inner-loop is done (FSM state3).

Second, inner-loop execution is accelerated depending on the condition of outer-loop in $oEQ1$. When V bit in $oEQ1$ is invalid, initial induction value for the inner-loop is provided. Inner-loop is executed in the fixed-function accelerators by following:

- By FSM (state1), initial induction value for the inner-loop is passed to $iEQ1$.
- Induction value flows to dataflow and each output result resides in $oEQ2$, $oEQ3$, $oEQ4$, $oEQ5$, $oEQ7$ and $oEQ9$.
- $oEQ2$ and $oEQ3$ are related to inner-loop control for incremented induction value and inner-loop execution condition, respectively.
- Induction value for the inner-loop is provided from $oEQ2$ to $iEQ1$ until the execution of inner-loop is finished (FSM state4).
- When R bit in $oEQ4$ is ready, it loads data from memory for $c[i \times 2]$ to $iEQ2$ for indirect memory access (FSM state5).
- When R bit in $oEQ5$ is ready, it loads data from memory for $a[i]$ to $iEQ3$ which is input for fixed-function compute accelerator (FSM state6).
- When R bit in $oEQ6$ is ready, it loads data from memory for $b[c[2 \times i]]$ to $iEQ4$ which is input for fixed-function compute accelerator (FSM state7).
- When R bit in $oEQ7$ is ready, it loads data from memory for $d[i]]$ to $iEQ4$ which is input for fixed-function compute accelerator (FSM state8).
- When fixed-function compute accelerator generates computation result in $oEQ8$, R bit in $oEQ8$ is ready. This result move into the $iEQ6$ by following FSM state9.

- To store computation result to memory, $oEQ9$ and $oEQ10$ are used. When `R` bits of both of $oEQs$ are ready (FSM `state10`), computation result is stored in $d[i]$.

Third, fixed-function accelerators operate following steps when inner-loop execution steps:

- When `R` bit in $oEQ3$ is valid, inner-loop iteration is finished.
- Outer-loop needs to provide outer-loop induction value into the fixed-function memory access accelerator. So, induction value for outer-loop in $oEQ0$ is passed to $iEQ0$ (FSM `state3`).

Finally, execution of fixed-function accelerators stop when the iteration condition of outer-loop occurs by following steps:

- When `V` bit in $oEQ1$ is valid, outer-loop iteration is finished.
- Acceleration stops (FSM `state2`).

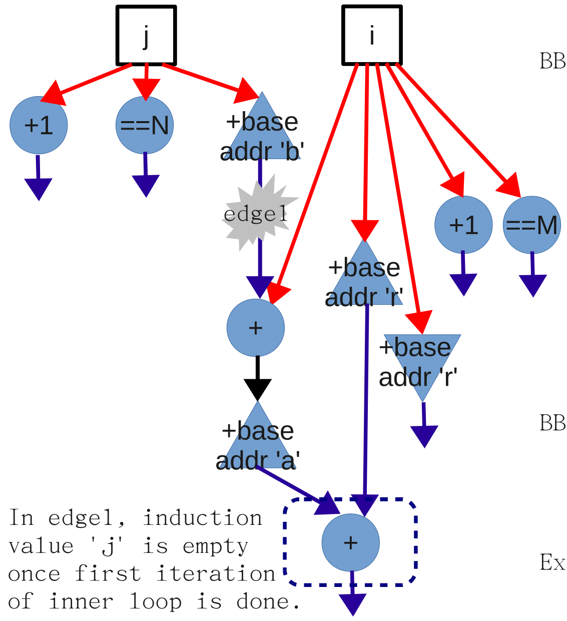
4.5 Complex Scenarios

In this section, we discuss complex scenarios that may be shown in the source code. First, we consider the source exhausted cases due to dataflow execution. It may happen when the value is not provided to the input event queues for some reasons. Then, we think about memory disambiguation which is frequently met in conventional source code.

The execution of fixed-function accelerators from ccHLS is based on dataflow operation. Hence, required values to generate address and computation in fixed-function accelerators are flowed through IO event eques and do computation when values in the input event queues are valid. Due to the nature of the dataflow machine, however, it prevents fixed-function accelerators from executing when required data is not ready or exhausted by other operations. Generally, nested-loops have such a case when an inner loop consumes data which has dependency on an outer loop. Induction values face this condition because they flow into the fixed-function accelerators and input event queues for them are empty when they flow. In that case, ccHLS places a special register called Induction-related Value Register (IVR) to hold required when

```
for(int j=0; j<N; ++j) {
  x = b[j];
  for(int i=0; i<M; ++i) {
    r[i] += a[x+i];
  }
}
```

a) Source code



c) E-AEPDG

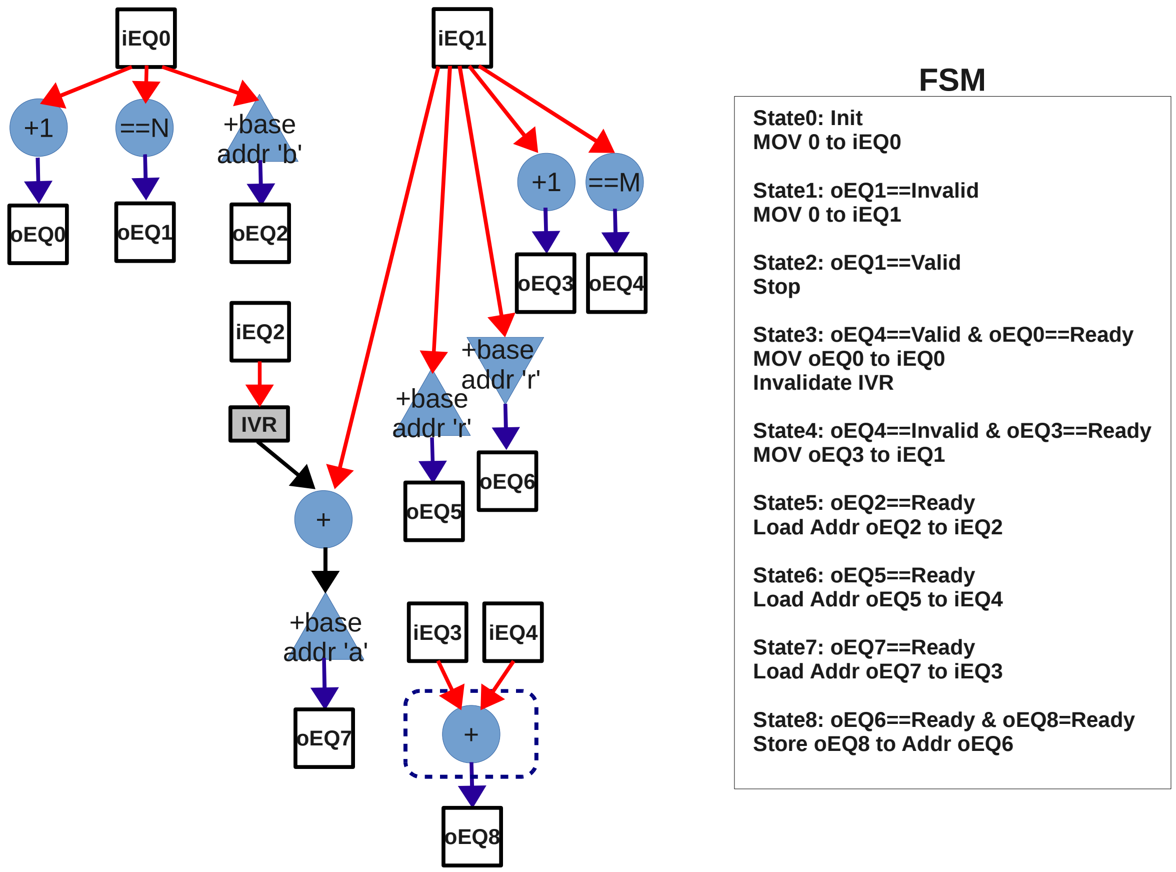
```
Entry:
  br label BB0

BB0:
  %j = phi i32 [ 0, Entry ], [ %j.next, BB2 ]
  %tmp1 = getelementptr inbounds i32* %b, i32 %j
  %tmp2 = load i32* %tmp1, align 4, !tbaa !1
  br label BB1

BB1:
  %i = phi i32 [ 0, BB0 ], [ %i.next, BB1 ]
  %tmp3 = getelementptr inbounds i8* %r, i32 %i
  %tmp4 = load i32* %tmp3, align 4, !tbaa !1
  %tmp5 = add nsw i32 %i, %tmp2
  %tmp6 = getelementptr inbounds i32* %a, i32 %tmp5
  %tmp7 = load i32* %tmp6, align 4, !tbaa !1
  %tmp8 = add nsw i32 %tmp7, %tmp4
  store i32 %tmp8, i32* %tmp3, align 4, !tbaa !1
  %i.next = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.next, M
  br i1 %exitcond2, label BB2, label BB1

BB2:
  %j.next = add i32 %j, 1
  %exitcond = icmp eq i32 %j.next, N
  br i1 %exitcond, label Exit, label BB0

Exit:
  ret void
```

b) Intermediate Representation

Figure 4.15: Inner Loop Dependency by Outer Loop Value

E-AEPDG is transformed. This special register is to be valid when data arrives and invalid when FSM in the fixed-function decides that is not required any more in the execution.

In an out-of-order processor, the issue of memory disambiguation arises when the source code has memory dependencies such as true-, anti-, and output-dependency. Hence, the load-store unit (LSU) is provided in the out-of-order processor to solve memory access ordering when it accesses memory and new LSU designs are proposed [71, 66] . However, fixed-function accelerators do not use LSU in the design and require a memory access ordering mechanism. ccHLS provide a new queue called the disambiguation queue (DQ) to relax the memory access ordering issue. FSM in the fixed-function memory access accelerator sees this queue and decides the state to access memory.



FSM

State0: Init
MOV 0 to iEQ0

State1: oEQ1==Invalid
MOV 0 to iEQ1

State2: oEQ1==Valid
Stop

State3: oEQ4==Valid & oEQ0==Ready
MOV oEQ0 to iEQ0
Invalidate IVR

State4: oEQ4==Invalid & oEQ3==Ready
MOV oEQ3 to iEQ1

State5: oEQ2==Ready
Load Addr oEQ2 to iEQ2

State6: oEQ5==Ready
Load Addr oEQ5 to iEQ4

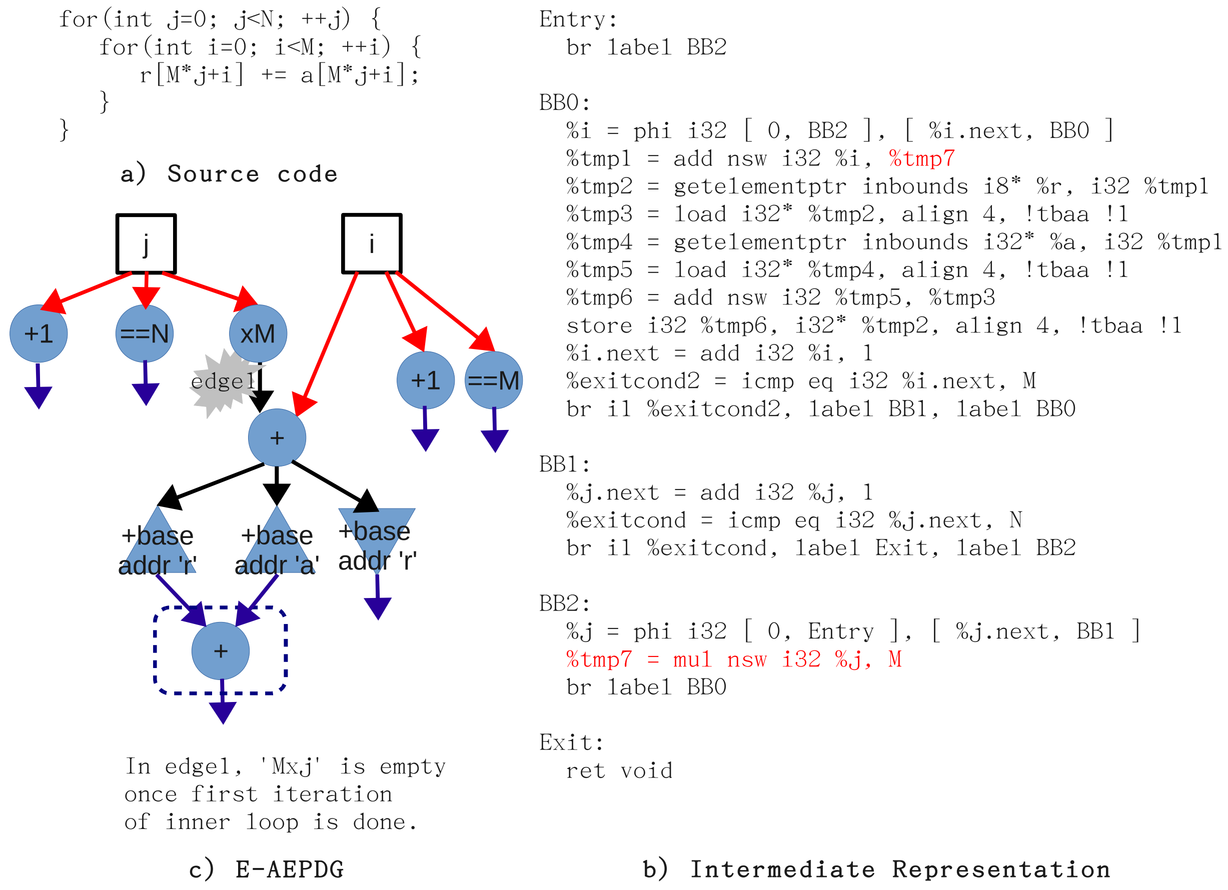
State7: oEQ7==Ready
Load Addr oEQ7 to iEQ3

State8: oEQ6==Ready & oEQ8=Ready
Store oEQ8 to Addr oEQ6

Figure 4.16: Transformation of Loop Dependency

Dependency by Load Figure 4.15 shows an example that data is empty when it executes. In figure 4.15a), nested loops are shown and the inner loop has data dependency by `x=b[j]` which is related to induction value `j`. It is consumed in the inner loop to compute the memory access address and finally used for address `a[x+i]`. The inner loop keeps accelerating the code region and requires value `x` when it iterates. `j` is empty until the inner loop is done and fixed-function accelerators arise deadlock because value `x` is not delivered in the fixed-function accelerators. From the intermediate representation, in figure 4.15b), the induction value `j` has the dependency to load the data (`%tmp2`) and loaded data has dependency on the inner loop. Figure 4.15c) presents the E-AEPDG and the edge that is dependent on the inner loop.

To avoid deadlock, IVR is inserted when E-AEPDG is transformed to generate fixed-function accelerators. The place where IVR is added is the node that has dependency on the inner loop. `%tmp2` is the target to add IVR because the load %tmp2 node is used with dependency within



```
for(int j=0; j<N; ++j) {
   for(int i=0; i<M; ++i) {
      r[M*j+i] += a[M*j+i];
   }
}
```

a) Source code

In edge1, 'Mxj' is empty once first iteration of inner loop is done.

c) E-AEPDG

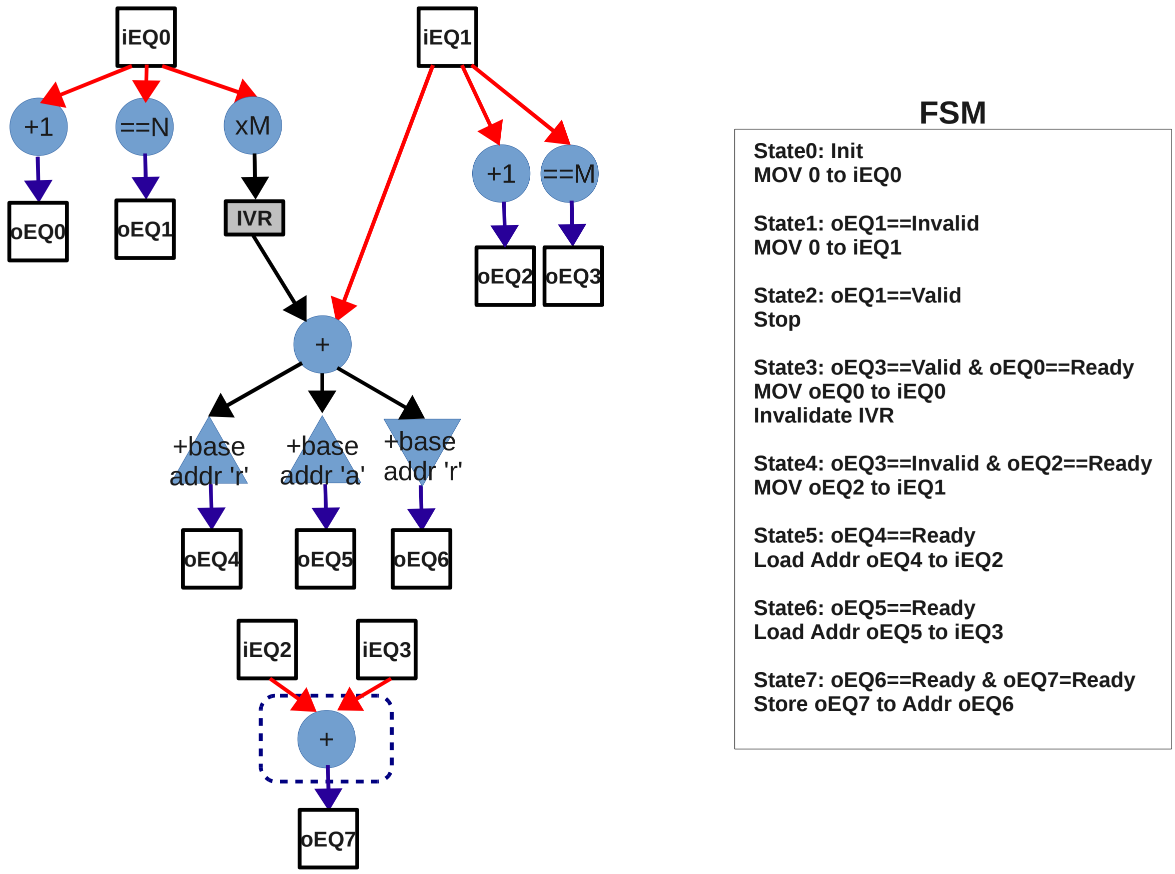
```
Entry:
  br label BB2

BB0:
  %i = phi i32 [ 0, BB2 ], [ %i.next, BB0 ]
  %tmp1 = add nsw i32 %i, %tmp7
  %tmp2 = getelementptr inbounds i8* %r, i32 %tmp1
  %tmp3 = load i32* %tmp2, align 4, !tbaa !1
  %tmp4 = getelementptr inbounds i32* %a, i32 %tmp1
  %tmp5 = load i32* %tmp4, align 4, !tbaa !1
  %tmp6 = add nsw i32 %tmp5, %tmp3
  store i32 %tmp6, i32* %tmp2, align 4, !tbaa !1
  %i.next = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.next, M
  br i1 %exitcond2, label BB1, label BB0

BB1:
  %j.next = add i32 %j, 1
  %exitcond = icmp eq i32 %j.next, N
  br i1 %exitcond, label Exit, label BB2

BB2:
  %j = phi i32 [ 0, Entry ], [ %j.next, BB1 ]
  %tmp7 = mul nsw i32 %j, M
  br label BB0

Exit:
  ret void
```

b) Intermediate Representation

Figure 4.17: Loop Dependency by Outer Loop Induction Value

the inner loop. Figure 4.16 shows the transformation with IVR. From FSM, State5 loads data for b[j] to iEQ2. When iEQ2 gets data, it flows to IVR and keeps providing b[j]. When the inner loop is done, State3 increments induction variable j and invalidates IVR simultaneously. Based on transformation, ccHLS replaces each node and edge as the formal form to generate fixed-function accelerators.

Dependency by induction value Figure 4.17 shows the different phases of the load dependency. In figure 4.17a), the inner loop uses the induction value of j for computation. In the general purpose processor, this value is in the register and gets the register number in the decode stage. However, the dataflow in the fixed-function accelerators consume j and it is empty after the first iteration of the inner loop. It also causes the deadlock and data does not flow into the fixed-function accelerators. In figure 4.17b), induction value j has a data dependency on %tmp7



State0: Init
MOV 0 to iEQ0

State1: oEQ1==Invalid
MOV 0 to iEQ1

State2: oEQ1==Valid
Stop

State3: oEQ3==Valid & oEQ0==Ready
MOV oEQ0 to iEQ0
Invalidate IVR

State4: oEQ3==Invalid & oEQ2==Ready
MOV oEQ2 to iEQ1

State5: oEQ4==Ready
Load Addr oEQ4 to iEQ2

State6: oEQ5==Ready
Load Addr oEQ5 to iEQ3

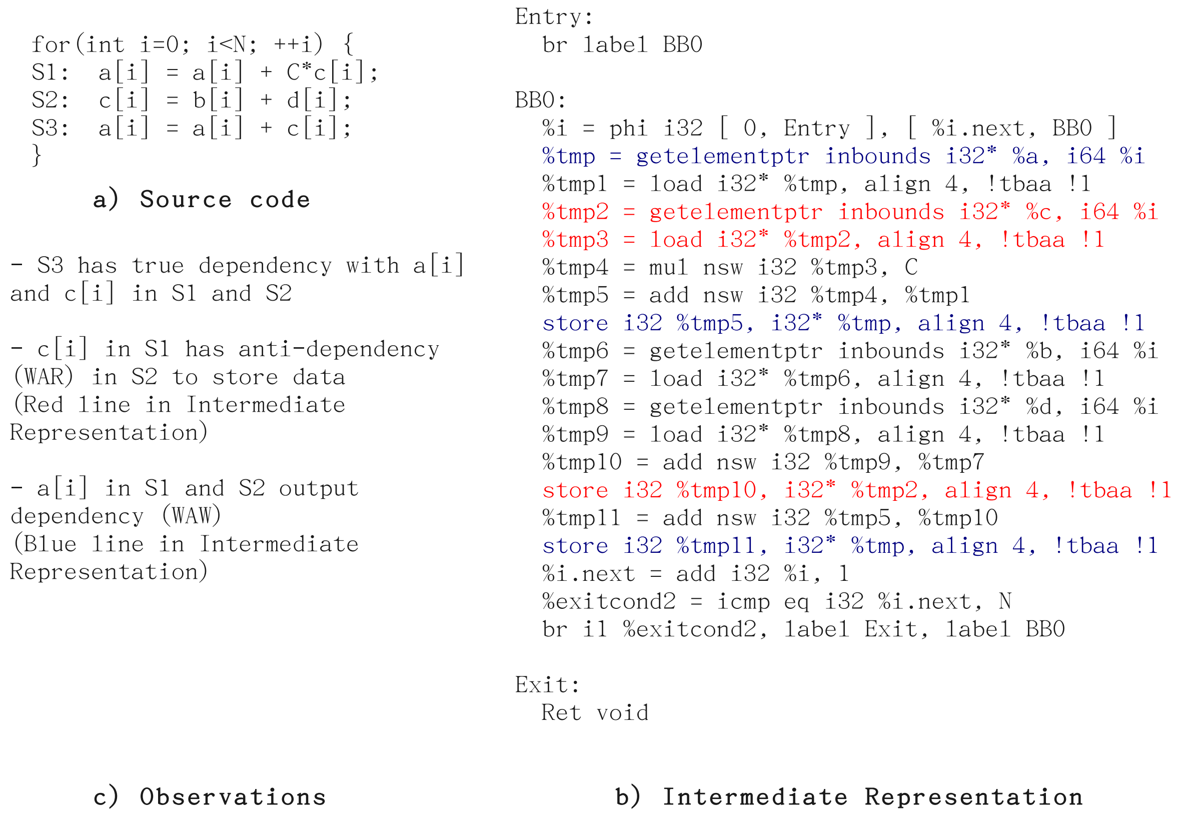
State7: oEQ6==Ready & oEQ7=Ready
Store oEQ7 to Addr oEQ6

Figure 4.18: Transformation of Loop Dependency by Outer Loop Induction Value

(in basic block BB2) and `%tmp7` is used in the inner loop (in basic block BB0) with dependency. In figure 4.17c), the edge that has dependency within the inner loop is a potential reason for deadlock.

In this case, IVR is also added into the node that has dependency on the inner loop and `%tmp7` is the target to place IVR to avoid deadlock. Figure 4.18 presents the transformation and IVR is linked by the node `%tmp7`. The induction value `j` arrives at IVR when the operation of node `%tmp7` and its value is valid until the inner loop is done in `State3`.

Memory Disambiguation Figure 4.19 shows code snippets that have memory dependencies. In figure 4.19a), it has all three memory dependency problems, which are true-dependency (RAW), anti-dependency (WAR), and output-dependency (WAW). In figure 4.19b) and c), dependencies in IR and the observation from IR are presented. First, `S3` has true-dependency with



```
for(int i=0; i<N; ++i) {
S1:  a[i] = a[i] + C*c[i];
S2:  c[i] = b[i] + d[i];
S3:  a[i] = a[i] + c[i];
}
```

a) Source code

```
Entry:
  br label BB0

BB0:
  %i = phi i32 [ 0, Entry ], [ %i.next, BB0 ]
  %tmp = getelementptr inbounds i32* %a, i64 %i
  %tmp1 = load i32* %tmp, align 4, !tbaa !1
  %tmp2 = getelementptr inbounds i32* %c, i64 %i
  %tmp3 = load i32* %tmp2, align 4, !tbaa !1
  %tmp4 = mul nsw i32 %tmp3, C
  %tmp5 = add nsw i32 %tmp4, %tmp1
  store i32 %tmp5, i32* %tmp, align 4, !tbaa !1
  %tmp6 = getelementptr inbounds i32* %b, i64 %i
  %tmp7 = load i32* %tmp6, align 4, !tbaa !1
  %tmp8 = getelementptr inbounds i32* %d, i64 %i
  %tmp9 = load i32* %tmp8, align 4, !tbaa !1
  %tmp10 = add nsw i32 %tmp9, %tmp7
  store i32 %tmp10, i32* %tmp2, align 4, !tbaa !1
  %tmp11 = add nsw i32 %tmp5, %tmp10
  store i32 %tmp11, i32* %tmp, align 4, !tbaa !1
  %i.next = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.next, N
  br i1 %exitcond2, label Exit, label BB0

Exit:
  Ret void
```

- S3 has true dependency with a[i] and c[i] in S1 and S2

- c[i] in S1 has anti-dependency (WAR) in S2 to store data (Red line in Intermediate Representation)

- a[i] in S1 and S2 output dependency (WAW) (Blue line in Intermediate Representation)

c) Observations

b) Intermediate Representation

Figure 4.19: Source Code with Memory Disambiguation

a[i] and b[i] and its memory access is held until the dependency is resolved. Second, c[i] in S2 has anti-dependency with that in S1 and memory access c[i] in S2 should not violate memory access order with that in S1. Finally, a[i] in S1 and S3 has output dependency. While LSU manages the order of memory access operations in the normal out-of-order processor, fixed-function accelerators do not use LSU for memory access and that should be solved within the E-AEPDG.

Figure 4.20 shows the organization of E-AEPDG with memory dependency. It is expanded from IR and deployed with edges and nodes. The evidence of memory dependency are in IR and reflected in the edges of E-AEPDG. edge1 is dependent on edge4 (output-dependent) and memory access by edge1 is older than edge4. In addition, edge2 is anti-dependency on edge3 and it is older than edge3. To provide memory access order to FSM in the fixed-function accelerator, DQs for those edges are attached to provide the capability of order decision. DQ is a single bit queues and used in the FSM to decide the state. For true-dependency, it does not

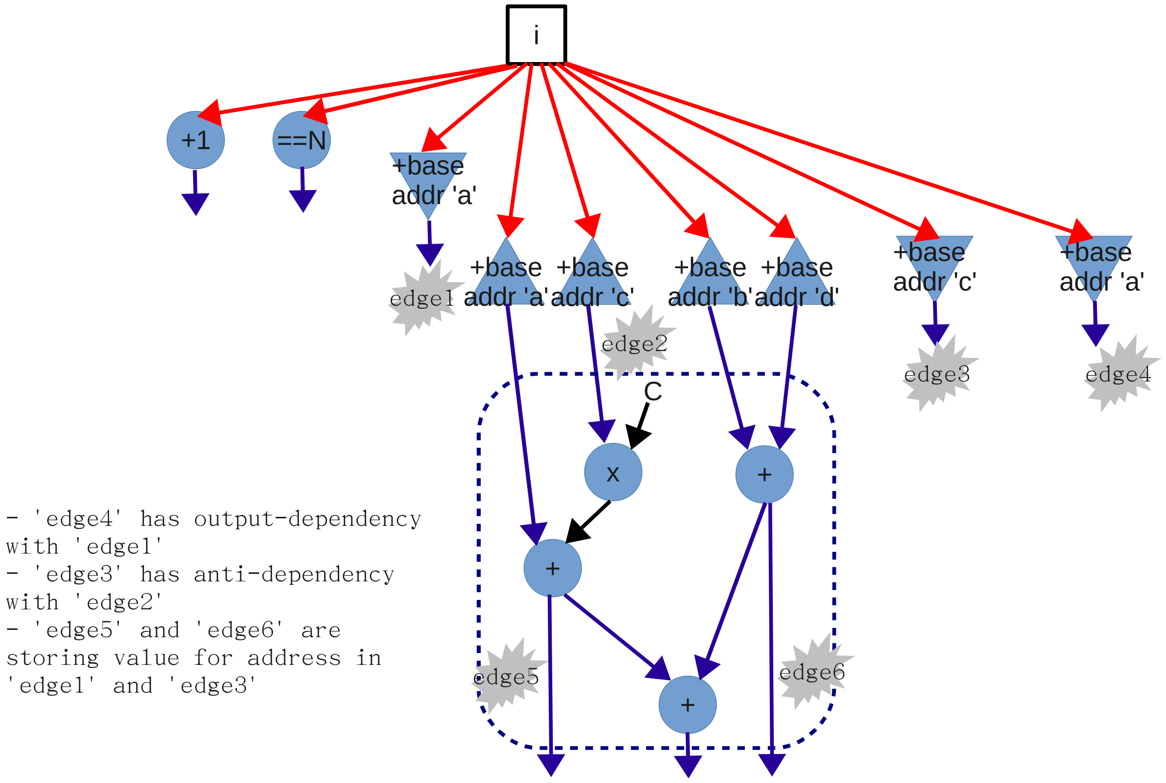
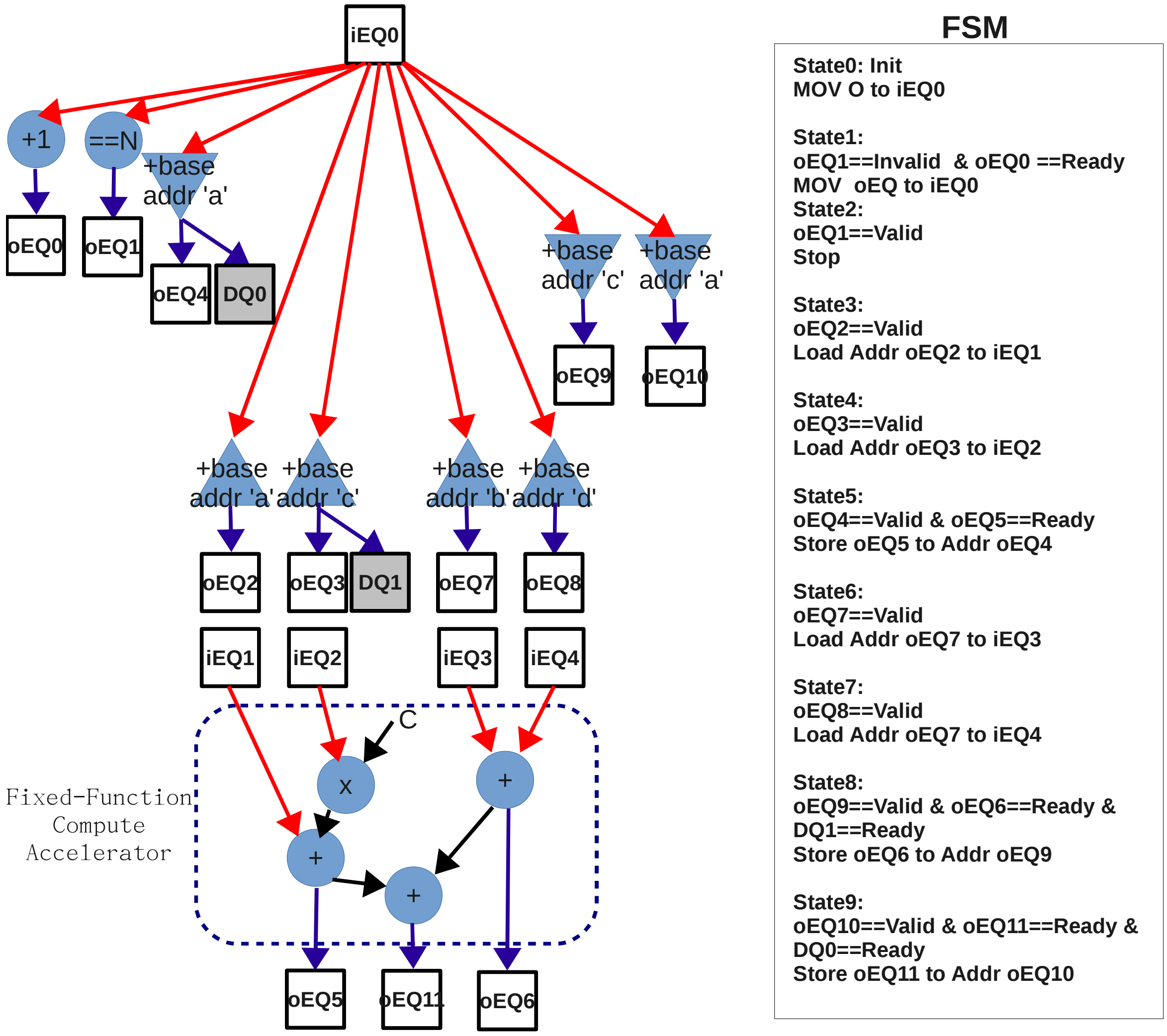

i
+1
==N
+base addr 'a'
edge1
+base addr 'a'
+base addr 'c'
+base addr 'b'
+base addr 'd'
+base addr 'c'
+base addr 'a'
edge2
edge3
edge4
C
x
+
+
+
edge5
edge6
- 'edge4' has output-dependency with 'edge1'
- 'edge3' has anti-dependency with 'edge2'
- 'edge5' and 'edge6' are storing value for address in 'edge1' and 'edge3'

Figure 4.20: E-AEPDG with Memory Disambiguation

require DQ because its value has already been ready and the output edge has data. In addition, edge5 and edge6 present output edges to export storing data for output- and anti-dependencies and they should be stored in the memory by using edge1 and edge6 as storing addresses In that case, attaching an additional output EQ can solve the issue and they are used for store value and data movement ports for a dependent node.

Figure 4.21 presents the transformation of A-EPDG with memory disambiguation and its FSM. For anti- and output-dependencies, DQs (DQ0 and DQ1) are attached on the edges (edge1 and edge2) and State8 and State9 are executed when DQs are ready. For true-dependency, edges are directly linked to the add node and storing data for source code line S1 and S2 are exported through oEQ5 and oEQ6, respectively.

Algorithm 5 describes the mechanism that avoids the complex scenario. For the case that dependecy shown by load, it collects target link edges which are the output of load nodes to insert IVR from all load nodes (LoadNode. When the target link edge has the dependency on



State0: Init
MOV O to iEQ0

State1:
oEQ1==Invalid & oEQ0 ==Ready
MOV oEQ to iEQ0
State2:
oEQ1==Valid
Stop

State3:
oEQ2==Valid
Load Addr oEQ2 to iEQ1

State4:
oEQ3==Valid
Load Addr oEQ3 to iEQ2

State5:
oEQ4==Valid & oEQ5==Ready
Store oEQ5 to Addr oEQ4

State6:
oEQ7==Valid
Load Addr oEQ7 to iEQ3

State7:
oEQ8==Valid
Load Addr oEQ7 to iEQ4

State8:
oEQ9==Valid & oEQ6==Ready &
DQ1==Ready
Store oEQ6 to Addr oEQ9

State9:
oEQ10==Valid & oEQ11==Ready &
DQ0==Ready
Store oEQ11 to Addr oEQ10

Figure 4.21: Transformation with Memory Disambiguation

inner-loop node, this target link edge requires IVR and the collection of input and output edges should be updated. For the dependency by induction variable, it is similar to the dependecy by load except choosing target link edges. In this case, target link edges get from the node having dependecy with input edges. For memory disambiguation, it checks all load and store node with base addresses provided by CFG. If one of store or load nodes with base address has younger node that access that address, this node is the target to requires DQ.

Algorithm 5 RESOLVING COMPLEX SCENARIO

```
$LoadNode \leftarrow$ {Load Node in CFG with Order}
$StoreNode \leftarrow$ {Store Node in CFG with Order}
$i\_edges \leftarrow$ {From e-aepdg}
$o\_edges \leftarrow$ {From e-aepdg}
$BaseAddr \leftarrow$ {Base Address in e-aepdg}
$IVR \leftarrow \emptyset$
$DQ \leftarrow \emptyset$
{Dependency by Load}
for all $LNode \in LoadNode$ do
    $target\_link\_edge \leftarrow$ OUTPUT_LINK_EDGE($LNode$)
    for all $node \in InnerLoop$ do
        for all $target\_link\_edge \in$ GET-OPERANDS($node$) do
            $IVR \leftarrow IVR \cup target\_link\_edge$
            $i\_edges \leftarrow i\_edges \cup target\_link\_edge$
            $o\_edges \leftarrow o\_edges \cup target\_link\_edge$
        end for
    end for
end for
{Dependency by Induction Value}
$ANode \leftarrow$ {All Node having Dependency with i_edge}
$target\_link\_edge \leftarrow$ OUTPUT_LINK_EDGE($ANode$)
for all $node \in InnerLoop$ do
    for all $target\_link\_edge \in$ GET-OPERANDS($node$) do
        $IVR \leftarrow IVR \cup target\_link\_edge$
        $i\_edges \leftarrow i\_edges \cup target\_link\_edge$
        $o\_edges \leftarrow o\_edges \cup target\_link\_edge$
    end for
end for
{Memory Disambiguation}
for all $Addr \in BaseAddr$ do
    for all $SNode \in StoreNode$ do
        for all $SNode(Addr)$ SUCH THAT {having younger SNode(Addr)} do
            $DQ \leftarrow DQ \cup$ OUTPUT_LINK_EDGE($SNode(Addr)$)
        end for
    end for
    for all $LNode \in LoadNode$ do
        for all $LNode(Addr)$ SUCH THAT {having younger SNode(Addr)} do
            $DQ \leftarrow DQ \cup$ OUTPUT_LINK_EDGE($LNode(Addr)$)
        end for
    end for
end for
```

Supported	Unsupported
All arithmetic computations Array Pointer Struct	function call Dynamic memory allocation Vectorization (SIMD, MMX, etc.)

Table 4.3: Supported and unsupported coding style

4.6 ccHLS Support

ccHLS is designed to generate fixed-function accelerators from target source code written in high-level program languages. Specifically, it is aimed to support all program languages that are transformed to intermediate representation (IR) because ccHLS is a back-end compiler based on IR. In this subsection, ANSI C is chosen to explain support and unsupport coding in the source code for synthesis to fixed-function accelerators. However, not all subsets of ANSI C are supported in ccHLS because it focuses on algorithmic loop acceleration. Hence, code segments that prohibit from constructing E-AEPDG is not supported by ccHLS.

Basically, all arithmetic operations including integer and floating point are supported by ccHLS because they construct the skeleton of E-AEPDG. Those operations are transformed to functional units for fixed-function compute and memory access accelerators. In addition, arrays and pointers that are frequently used in the code segment are supported, which are transformed as data movement and memory access by load and store operations through `iEQs` and `oEQs`.

However, some subsets of ANSI C are unsupported by ccHLS. Generally, they prevent the source code from organizing E-AEPDG, and function calls, dynamic memory allocation, and vectorization are examples of subsets that are not supported. First, function calls in the source code are not allowed by ccHLS because it cannot be built as a form of the E-AEPDG. The target source code that has function calls should be modified in the callee function. Second, dynamic memory allocation is not supported because its virtual memory address is not fixed in compile time, hence it is not possible to let E-AEPDG know the base address of dynamic memory allocation for load or store operations. Finally, vectorization such as SIMD instructions is not supported because it requires pairs of functional units for SIMD instructions and is wasteful when ccHLS transforms them as functional units. Thus, ccHLS transforms vectorized operations as sequential operations by providing data into a functional unit assigned for vectorized instructions. Table 4.3

shows the classification of supported and unsupported coding style in the code segment.

4.7 Implementation

To construct extended AEPDG, LLVM compiler framework is used [50]. From the front-end compiler of LLVM, it represents the source code written in high-level program language as an intermediate representation code for the control flow graph, which is machine-independent code described as a static single assignment form (SSA). After passing various code analysis and optimization passes, LLVM finally translates machine-dependent assembly code by the back-end compiler. ccHLS utilizes IR and analysis passes to generate E-AEPDG then provides fixed-function accelerators from the back-end.

ccHLS first optimizes IR and collects information used for creating E-AEPDG. Especially, unnecessary memory access, which accesses the same memory address to load or store temporal data, from load and store instructions is wasteful for creating the fixed-function accelerator because it consumes resources to organize it. Therefore, ccHLS removes all redundant instructions by reusing replaceable instructions in IR and makes compact fixed-function accelerators when it is generated as output. After optimizing IR, ccHLS collects information to organize extended AEPDG. To figure out what Instructions are the elements for execute, access, or loop control subgraph, ccHLS scrutinizes the target loop region and finds subloops in the target loop, loop head and exit basic block (BB) of each subloop. It is important because the location of instructions is a clue to classify instructions to construct subgraph of E-AEPDG. Then, I developed the LLVM passes to create E-AEPDG and transform E-AEPDG into the fixed-function accelerators.

4.8 Chapter Summary

In this chapter, we discussed the ccHLS design using a compiler intermediate representation called Extended Access/Execute Program Dependence Graph (E-AEPDG) to easily generate the fixed-function accelerators. Based on LLVM infrastructure, we showed how to convert the conventional intermediate representation to E-AEPDG and transform E-AEPDG to the fixed-function accelerators. Following the DAE model, the generated accelerator is classified to the fixed-function compute accelerator and the fixed-function memory access accelerator. In the fixed-function memory access accelerator, it parallelized loop invocations with the dataflow fashion and fed data generated from memory address computation into the fixed-function compute accelerator as the dataflow.

5 EVALUATION OF SWSL

This chapter discusses the evaluation methodology and results of generated accelerators from the SoftWare Synthesis (SWSL). SWSL utilizes the PLUG programming model and its code block source code are fed into SWSL for generating accelerators of specialized network loop algorithms. The data flow graph (DFG) model of the applications are customized as accelerators. It is the stand-alone accelerator and the host processor only that feeds the lookup request by injecting lookup request messages into the accelerator. Hence, the evaluation of the accelerator from SWSL only focuses on the generated accelerator itself. For the evaluation metrics, SWSL evaluates the latency, power and area size compared with PLUG and LEAP when SWSL generates the network lookup accelerator.

In section 5.1, we describe the overview of the evaluation methodology for SWSL. Section 5.2 discusses the benchmarks to evaluate and details of evaluation metrics are presented. Finally, the evaluation result is shown in section 5.3.

5.1 Evaluation Overview

In this section, the overviews of evaluation of SWSL are detailed. SWSL generates the lookup accelerator and the host processor feeds the lookup request message into the lookup accelerator. Due to the difficulty for comparison with other network lookup accelerators, the network lookup accelerator from SWSL is only concentrated on the generated accelerator and it compares with other network lookup accelerators that follow the same execution model feeding lookup requests by the host processor. The comparison accelerators have the same execution in the host processor and we are able to pay attention to the network lookup accelerators.

The accelerator from SWSL assumes a line card that interacts a host processor with the accelerators. Figure 5.1 shows the architectural target model of SWSL. To evaluate line card style execution as terms of comparison, we use PLUG and another specialized lookup accelerator, LEAP [38]. The choice of comparing with PLUG and LEAP is motivated by the fact that they represent opposite points within the software/hardware design tradeoff. PLUG adopts a software approach, with its computation engines being conventional Von-Neumann cores. LEAP instead

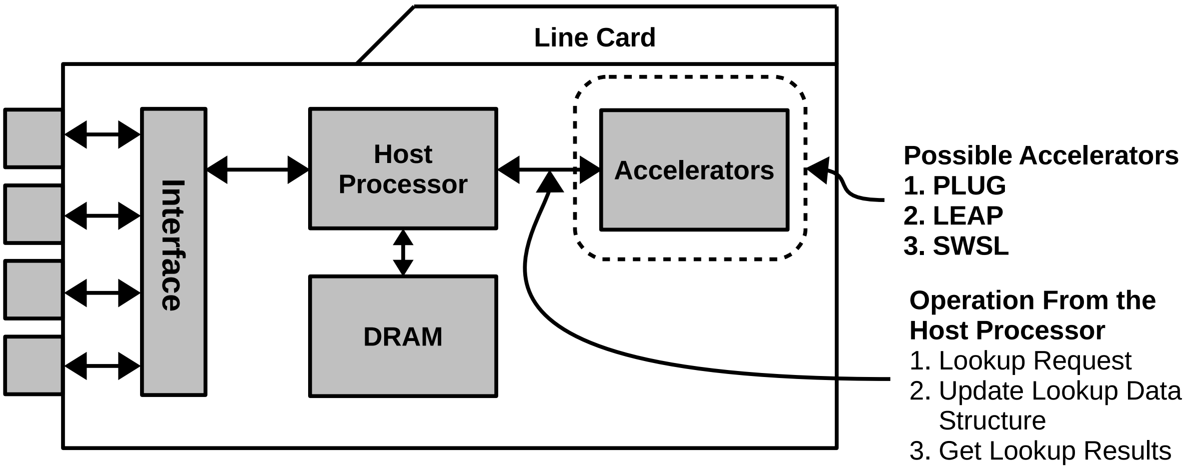


Figure 5.1: SWSL Target Model: Network Line Card

uses arrays of fixed-function hardware blocks. PLUG offers software-like programmability; LEAP has a constrained programming model but achieves near-ASIC performance. However, they are well-matched with the design of line card and the operation of the host processor is exactly the same in terms of lookup, update, and getting the result. Hence, we are able to ignore the operation in the host processor and only focus on the comparison of the accelerators.

As the target source code of SWSL, we used a PLUG programming model that includes the APIs [47], and a specialized lookup accelerator. Its API exports a DFG-based programming model, and provides infrastructure for simulating and verifying lookup algorithms in software. We then modified the PLUG toolchain to use the SWSL compiler as a backend. The SWSL compiler itself was implemented as a series of passes mentioned in chapter 3 for the LLVM compiler toolkit.

SWSL generates a datapath as RTL modules for code blocks in the DFG for network lookup applications and top model design to combine code block datapath modules. This thesis focuses on the analysis of pure network lookup accelerators and architectural trade-off. Hence, generated RTL modules and top model design are not integrated into the host processor, but serves as a stand-alone lookup request model. For this reason, we implement the testbench to request lookup operations and mimic such requests provided by the conventional host processor. Its performance is simulated using Synopsys VCS simulator with the testbenches.

To evaluate the performance of SWSL designs, we used the Synopsys design compiler with a 55nm design library and 1 GHz clock frequency. To evaluate the SRAM memory installed to each logical page, we leveraged the CACTI modeling simulator [74] with SRAM organized by four 64 KB memory banks for proper comparison to PLUG and LEAP (which use the same memory design).

5.2 Benchmarks

To evaluate, the SWSL uses different sets of benchmarks because SWSL does not have a matched benchmark set since it follows the specialized programming model of PLUG architecture. Consequently, PLUG applications are targeted for the comparison instead of conventional benchmarks. The main reasons for using PLUG applications are i) they are modeled for the comparison architectures (PLUG and LEAP) and ii) it is effective to show the differences among software (PLUG), software-hardware codesign (LEAP), and hardware intensive design (SWSL). As target lookup applications, we choose a suite of three standard lookup algorithms (Ethernet forwarding, IPv4, and IPv6), and one research protocol - Ethane, which is an academic precursor of OpenFlow. To evaluate the flexibility of SWSL we also included DFA-based regexp matching (a widespread primitive used in intrusion detection systems). We note here that the DFA application implements only a lookup in a compressed transition table, not the complete DFA. As SWSL shares PLUG programming API, we use a version of these applications previously implemented for PLUG. To generate SWSL designs, the applications were fed to the SWSL compiler, implementing the passes listed in Figure 3.2. For further details of each application the reader is referred to [47, 24]. We reuse these applications from the PLUG source code and code blocks with DFG configuration of applications are directly injected into the SWSL compilation process shown in Figure 3.2. Details of DFGs for each application can be found in figure 5.2 and table 5.1.

SWSL, which only concentrates on generating the network lookup accelerator, intends to build an accelerator which can feasibly execute a lookup per clock. Hence, we measure latency when a lookup operation is executed instead of performance improvement by reducing execution time. For power and area analysis, we estimate a generated accelerator RTL using a Synopsys

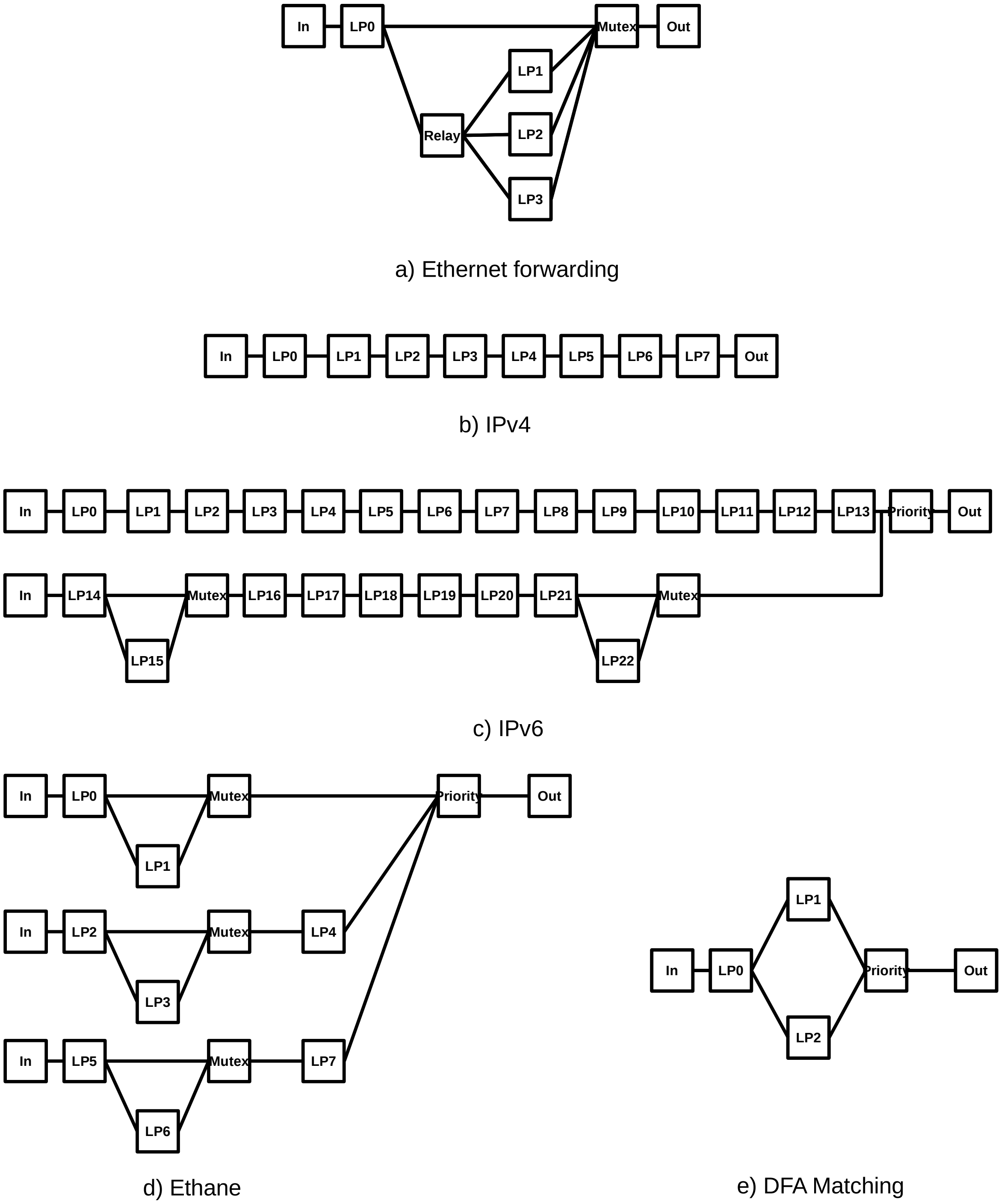


Figure 5.2: Data Flow Graph for Applications

LP Number	Number of code-blocks	Input Network	Output Network
Ethernet Forwarding			
0	3	0, 1	0, 1, 2
1, 2, 3	3	0, 1	1
IPv4			
0	5	0, 1, 2	0, 1, 2
1	5	0, 1, 2	0, 1, 2
2	8	0, 1, 2	0, 1, 2
3	5	0, 1, 2	0, 1, 2
4	5	0, 1, 2	0, 1, 2
5	5	0, 1, 2	0, 1, 2
6	6	0, 1, 2	0, 1
7	5	0, 1, 2	0, 1
IPv6			
0	5	0, 1, 2	0, 1, 2
1	5	0, 1, 2	0, 1, 2
2	8	0, 1, 2	0, 1, 2
3, 6, 9, 16	5	0, 1, 2	0, 1, 2
4, 7, 10, 17	5	0, 1, 2	0, 1, 2
5, 8, 11, 18	5	0, 1, 2	0, 1, 2
12, 19	6	0, 1, 2	0, 1
13, 20	5	0, 1, 2	0, 1
14	3	0, 1, 2	0, 1, 2, 3, 4
15	3	0, 1, 2	0, 1, 2
21	4	0, 1, 2	0, 1, 2, 3
22	4	0, 1, 2	0, 1
Ethane			
0	4	0, 1, 2	0, 2, 3, 4
1	4	0, 1, 2	0
2, 5	5	0, 1, 2	0, 2, 3
3, 6	5	0, 1	0, 1
4, 7	3	0, 1	0, 1
DFA Matching			
0	2	0, 1, 2	0, 1, 2, 3
1	2	0, 1, 2	0
2	2	0	0

Table 5.1: Details of Network Lookpu Application

design compiler.

5.3 Evaluation for Network Lookup with SWSL

In this section, we evaluate network lookup modules from SWSL. It includes the effectiveness, throughput, latency, power, and area analysis. Details of designs are shown in the paragraphs below:

Effectiveness/Ease of programming: To evaluate the SWSL compiler, we selected a set of 5 applications originally written for the PLUG lookup accelerator. We emphasize that basing the SWSL programming model on the PLUG API is purely a matter of convenience; the SWSL compiler could be adapted to a different API with minimal tweaks (as long as the programming model enforces the constraints listed in section 3.1.1).

SWSL was able to generate functionally correct lookup hardware for all the applications in our set; none of the applications required changes. According to [47], developing/verifying the PLUG hardware took 6 person-months and developing the applications took 18 person-months. Developing the PLUG-specific compiler took another 6 person-months. In this context, the SWSL approach would have made the hardware development cycle largely unnecessary, potentially reducing design time by up to 20%.

Throughput Analysis: Both PLUG and LEAP can achieve a throughput of 1 lookup per cycle with a 1 GHz clock frequency. We verified that SWSL is capable of achieving the same throughput.

Latency Analysis: Table 5.2 presents the latency of each SWSL-generated application (in ns) in the "Total Latency" column. The latency is further decomposed into the component due to the length of the (hardware or software) critical path, and the latency introduced by the scheduler for synchronization purposes (in section 3.1.2). The synchronization overhead is relatively high, contributing up to 1/3 of the overall latency. However synchronization is crucial, as it guarantees conflict-free execution, enabling pipelining.

Application	Computation Critical Path	Synchronization	Total Latency
Ethernet forwarding	8	3	11
IPv4	93	35	128
IPv6	175	63	238
Ethane	29	2	31
DFA	22	7	29

Table 5.2: SWSL Application Latency (ns)

Table 5.3 further compares the latency of SWSL-generated applications with the same applications deployed on PLUG and LEAP. SWSL achieves a significant latency reduction in comparison with PLUG, for three reasons.

First, SWSL exploits more parallelism than PLUG. SWSL generates merged basic block, computing multiple branch conditions in a single 1-cycle pass. As branch conditions tend to be on the computation critical path, parallelizing their computation directly decreases the overall application latency. Conversely, PLUG is based on the conventional Von-Neumann architecture, and executes algorithms in software. In this context, branch conditions have to be evaluated sequentially.

Second, the PLUG API offers specialized high-level operation, each of which is translated to several atomic instructions. For example, the PLUG *SendMsg* call is used to forward intermediate results between algorithmic steps. At compile-time, a single *SendMsg* is converted to multiple instructions that copy values to special-purpose network registers, construct the message header and send the message on the on-chip network – requiring 5 cycles. Instead, SWSL can implement the operation directly as efficient hardware and execute all data movements in a single cycle.

Finally, there is no communication overhead in SWSL. PLUG cores are organized in a matrix; the on-chip network allows arbitrary communication patterns between cores. This requires all communication to be mediated by on-chip routers. PLUG employs point-point links and XY routing, making communication latency dependent on the distance between cores. As SWSL specializes the hardware for a single algorithm, communication does not need to be flexible. Stages are connected directly via wires (Figure 3.1c), making communication latency negligible.

SWSL has still higher latency than LEAP. LEAP can achieve minimal latency because of its optimized programming model. A LEAP computation engine is an array of specialized hardware

Application	SWSL	PLUG	LEAP
Ethernet forwarding	11	55	6
IPv4	128	264	24
IPv6	238	524	42
Ethane	31	100	6
DFA	29	59	6

Table 5.3: Latency Comparison (ns)

Application	SWSL	PLUG	LEAP
Ethernet forwarding	0.582	0.504	0.392
IPv4	1.753	0.504	0.392
IPv6	5.473	2.331	1.813
Ethane	1.200	0.504	0.392
DFA	0.373	0.756	0.588

Table 5.4: Power Estimation (W)

units connected via a crossbar, resulting in near-ASIC performance. However, such performance comes at a price in terms of ease of development and generality. Programming LEAP involves routing bits and configuration values between units to implement the desired algorithm. LEAP exports a specialized Python API that alleviates the complexity of this programming model, however applications developed for LEAP are highly specific and cannot be easily supported on other architectures. For example, a LEAP programmer must explicitly ensure that the format and bit width of inputs and intermediate results match the specifications of each functional unit. Conversely, SWSL uses a conventional programming model which allows developers to leverage their programming expertise and express applications in a more intuitive form. Moreover, SWSL code is generic and platform-independent form, facilitating code reuse.

Power: For PLUG and LEAP, we estimate power by multiplying the power of a single tile with the number of active tiles configured for a target lookup application. While IPv6 is only configured as 8×8, other lookup applications are configured as 4×4 [47]. From tile configuration, Ethernet forwarding, IPv4, IPv6, Ethane and DFA have 8, 8, 37, 8, and 12 active tiles, respectively. The power consumption of a single tile of PLUG and LEAP is 63mW and 49mW [38]. We adopt this methodology to be consistent with results used in the LEAP work, since we compare to both LEAP and PLUG[1].

For SWSL, we collect the power number from the Synopsys power compiler with RTL code from SWSL with a default activity factor. Table 5.4 shows total power for each lookup engine approach. Results are mixed, with SWSL consuming slightly more than PLUG and LEAP in most cases, with the exception of the DFA application. This is motivated by the different complexities of each application. DFA consists of a simple algorithm that leads SWSL to instantiate a small amount of logic, leading to low power consumption. IPv4, IPv6 and Ethane are more complex and more deeply pipelined. In this case, the generality of PLUG and LEAP lead to smaller power consumption, as the same functionality can be re-used multiple times. Instead, SWSL instantiates dedicated logic for each pipeline stage, resulting in greater power consumption for complex applications. However, it is still a surprising and counter-intuitive result that an application-specific implementation consumes more power than a general-purpose engine. The reason is that our Verilog code generator backend is not as mature as the code-generation for engines like PLUG and LEAP which can re-use decades of research in instruction-level code generation. Also, there is one known source of significant inefficiency in our compiler. We are currently not using known hyperblock and predication [55] technology to handle control-flow. As a result, we have excessively long control-flow paths, which introduce unnecessarily large numbers of variable lines - each of which consumes significant power. We believe the underlying ideas in SWSL will provide power efficiency after these further engineering challenges are solved.

Area: We collect area for each network application from the synthesis result. Table 5.5 presents the area of each network application. From table 5.5, PLUG and LEAP have relatively larger areas than SWSL.

As PLUG and LEAP have a tiled configuration, with tiles arranged in 4×4 or 8×8 squares, they require significant area. However, the area taken by the computational engines is small; most of each tile's area is used by on-chip memory (PLUG - 64%, LEAP -95%). The computation

[1] In the PLUG papers, per-application power was reported, by considering only the tiles that are active based on individual lookup patterns and code-block activated. In contrast, here we are reporting PLUG power as power consumed by a tile multiplied by number of activated tiles.

Application	SWSL	PLUG	LEAP
Ethernet forwarding	16.897	51.2	33.6
Compute	0.937	18.432	1.68
Memory	15.96	32.768	32.92
IPv4	19.419	51.2	33.6
Compute	3.459	18.432	1.68
Memory	15.96	32.768	32.92
IPv6	56.062	204.8	134.4
Compute	10.177	73.728	6.72
Memory	45.885	131.072	127.68
Ethane	18.384	51.2	33.6
Compute	2.424	18.432	1.68
Memory	15.96	32.768	31.92
DFA	16.354	51.2	33.6
Compute	0.394	18.432	1.68
Memory	15.96	32.768	31.92

Table 5.5: Area Estimation (mm^2)

to memory ratio is also small for SWSL; we found that IPv6 computation area for SWSL is approximately the same as 3 PLUG tiles. An important difference is that SWSL constructs the chip design depending on the structure and computation of the input applications. Instead, PLUG and LEAP designs have a fixed structure, as the same design must support multiple applications (the only degree of freedom is the size – in terms of number of tiles – of the design).

SWSL has a high power-area ratio, which means that power consumption of an SWSL-generated design can be high even though its area is small. This can be explained as follows. The central idea of SWSL is to translate software functionality into hardware logic. Each block generated by SWSL implements exactly the action performed by the corresponding software; at run-time, all blocks will be busy performing their respective functions. Instead, PLUG and LEAP provide an array of cores (or, in the case of LEAP, specialized computational engines) to which functionality is assigned; the number of cores is overprovisioned to support computing-intensive applications. Therefore, in general, not all cores will be active at the same time, leading to less power consumption per unit of area.

5.4 Chapter Summary

This chapter investigates SWSL to generate hardware accelerators for network lookup. SWSL is the compilation techniques to generate hardware for a specific purpose, network lookup. Our evaluation shows that its operation latency is better than the specialized network lookup architecture (PLUG) and it gives reasonable power efficiency. The fixed function design from the code written in high-level program language provides reasonable result numbers.

6 EVALUATION OF CCHLS

In this chapter, we quantitatively evaluate the cache-coherent High-Level Synthesis (ccHLS). Generic program source code is the target of the ccHLS to generate the loop accelerator in the source code. Accelerators share the data cache in the host processor. Mixed architectural models, including the host processor and accelerators, are evaluated.

We evaluate the overall performance, power and energy efficiency of fixed-function accelerators by comparing with 2-issue and 4-issue out-of-order processors. In addition, comparison between the fixed-function accelerators from ccHLS and another research HLS compiler called C-Core are evaluated in this section. ccHLS accelerates the loop intensive region in the conventional source code and its output accelerator is a part of execution. Thus, we delve into its performance and efficiency compared to generic processor architectures and one HLS accelerator for comparison.

This chapter is organized as follows: Section 6.1 presents the overview of evaluation methodology for ccHLS. Section 6.2 introduces the benchmarks to evaluate and evaluation metrics. Section 6.3 presents the evaluation of the fixed-function accelerators generating from ccHLS with generic benchmark programs comparing with general purpose processor. Finally, Section 6.4 examines the comparison with C-Cores.

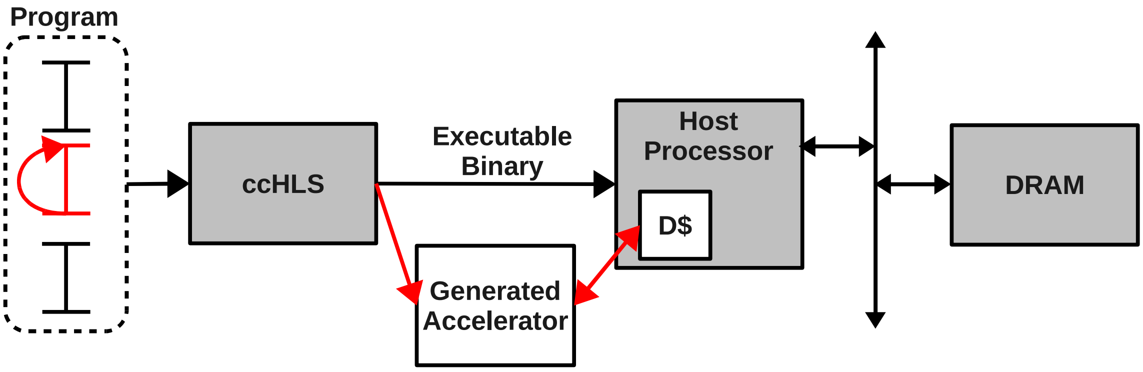


Figure 6.1: ccHLS Target Model: D-Cache utilization during acceleration

Parameters	Dual-issue OoO	4-issue OoO
Fetch, Decode, Issue, and Writeback Width	2	4
ROB Size	40	168
Scheduler (issue queue)	32	54
Register File (int/fp)	56/56	160/144
LSQ (ld/st)	10/16	64/36
DCache Ports	1(r/w)	2(r/w)
L1 Caches	I-Cache: 32 KB, 2 way, 64B lines D-Cache: 64 KB, 2 way, 64B lines	
L2 Caches	2 MB, 8-way unified, 64B lines	

Table 6.1: General purpose host processor models

6.1 Modeling for the Fixed-Function Accelerator from ccHLS

The fixed-function accelerators from the ccHLS share the data cache in the host processor. Thus, the execution model of the fixed-function accelerators assumes a general purpose processor exists. Figure 6.1 depicts the target model of the ccHLS. The host processor, except the data cache, sleeps during execution and awakes when the acceleration is finished. Hence, it is reasonable to compare the efficiency of the accelerator from the ccHLS with that of the general purpose processor.

To evaluate the fixed-function accelerators, x86 out-of-order architectures are modeled as the baseline in this evaluation and the host processor of the fixed-function accelerators in gem5 [13], which is a cycle-accurate simulator. For a more detailed evaluation, 2-issue out-of-order and 4-issue out-of-order processors are modeled in this evaluation for the comparison of low power cases and high performance cases, respectively. Basically, the pipeline of x86 out-of-order processor model in gem5 has fetch, decode, rename, issue, execute, writeback and commit stages. In addition, a branch predictor, a reorder buffer, and issue queue, a load-store queue for memory dependence predictor using store set in the queue are modeled in the gem5 simulator. Table 6.1 details the configuration of microarchitecture model.

Along with the baseline out-of-order processors, modeling of the fixed-function accelerators is based on RTL simulation, not integrated with the host processor. gem5 is configured for performance modeling and incorporated with RTL of the generated fixed-function accelerators for VCS simulation. To evaluate power, Synopsys Design Compiler and McPAT power model

Benchmarks	Application	Description
cutcp	3D Grid and Point Calc.	Small kernel with control flow
fft	Fast Fourier Transform	Regular memory access
kmeans	K-Means clustering	Regular memory access
lbm	FLuid Dynamics	Large computation kernel with control flow
mm	Dense Matrix Mult.	Small kernel
mri-q	Margnetic Resonence Imaging	Regular memory access
needle	Dynamic Programming	Loop carried dependency
nnw	Neural Networks	Indirect memory access
stencil	3D Matrix Jacobi	Small computation/memory ratio
spmv	Spare Matrix Vector Mult.	Indirect memory access
sad	Sum of Absolute Difference	High Computation/memory ration
tpacf	Angular Correlation	Irregular memory access

Table 6.2: Parboil Benchmark Description

are used to collect power simulation result with the fixed-function accelerators [52]. To power modeling simulation, TSMC 55nm design library and 1 GHz clock frequency are used in the evaluation.

6.2 Benchmarks

For the evaluation of fixed-function accelerators from the ccHLS, this dissertation chooses the Paboil benchmark [8]. An ideal application to evaluate the fixed-function accelerators is to have sufficient computation executed in the fixed-function compute accelerator and abundant memory access, such as arrays or pointer operation, managed by the fixed-function memory access accelerator. In addition, it has complex loop execution and indirect memory access. Thus, the Parboil benchmark suite that is generic source code written in a high-level program language (C and C++) is enough to 1) shows kernel operations in complex loops by the fixed-function compute accelerator and 2) discusses various memory address computations to directly access the data cache by the fixed-function memory access accelerator. Table 6.2 lists and describes the details of the Parboil benchmark.

For the fixed-function accelerators from the ccHLS, we evaluate the overall performance by comparing execution time by the fixed-function accelerator with the 2-issue and 4-issue out-of-order baseline architectural models from gem5. The execution time of the fixed-function accelerators is measured by the configuration integrating the fixed-function accelerators with the 2-issue out-of-order processor as a host processor. Thus, performance improvement is represented as:

$$\frac{Baseline\ Execution\ Time}{Fixed - FunctionAccelerators\ Execution\ Time}$$

The power consumption of the fixed-function accelerators with the host processor is measured by combining the McPAT modeling and the power report from the Synopsys design compiler. The power consumption of the host processor is estimated by using McPAT, taking microarchitectural parameters and events from gem5 simulation. For RTL simulation for the fixed-function accelerators, the design compiler provides its power modeling report. Using both results, it is reasonable to model power consumption. To evaluate energy efficiency, the power delay-product (Power × Execution time) is used in this thesis. Thus, the overall energy efficiency is described as:

$$\frac{Baseline\ Execution\ Energy}{Fixed - FunctionAccelerators\ Execution\ Energy}$$

Both metrics are used in evaluation to analyze modeled accelerator execution.

6.3 Fixed-Function Accelerators with the General Purpose Processor

In this section, we analyze performance and energy improvement of the fixed-function accelerators by comparing with a 2-issue out-of-order processor (2-OOO) and a 4-issue out-of-order processor (4-OOO) to verify the efficiency of fixed-function accelerators when they are adapted instead of a general purpose processor. First, we need to understand the ability of fixed-function accelerators, what characteristics impact performance and energy efficiency. In the fixed-function accelerators, the workload is transformed into functional unit execution while the general purpose processor uses instructions as the workload. Thus, analyzing both total number of instructions and functional unit execution in the general purpose processor and fixed-function accelerators provides meaningful information to predict the overall performance improvement. In addition, we measure the number of functional units used to organize dataflow

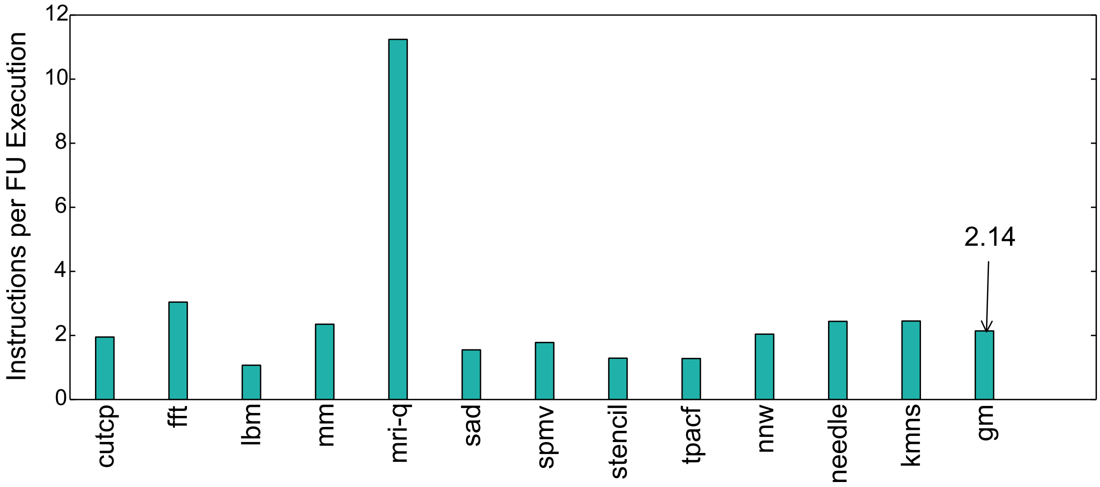


Figure 6.2: Ratio Insts/FU Execution

for fixed-function accelerators and it roughly provides the energy efficiency trend by removing the load on the pipeline in the general purpose processor.

To predict performance and energy efficiency, we analyze workload characterization with i) the ratio of instructions per a functional unit execution and ii) the number of functional units used for creating dataflow of fixed-function accelerators.

The ratio of instructions per a functional unit execution Fixed-function accelerators from ccHLS run with the combination of functional units that creates dataflow. On the contrary, however, the execution in the general purpose processor is based on pipeline with instructions. While the execution times of a general purpose processor is proportional to the number of instructions, that of fixed-function accelerators is correspondent to the number of functional units that execute for acceleration. Compared to the number of functional units executing for acceleration in fixed-function accelerators and instructions in the general purpose processor, we predict the overall performance improvement of fixed-function accelerators because it roughly represents reducing the amount of workload that the general purpose processor must run to execute a whole program. It tells us that fixed-function accelerators save execution time by executing fewer functional units instead of instructions in the processor pipeline. Figure 6.2 shows the ratio of instructions per a functional unit execution when it runs benchmarks and

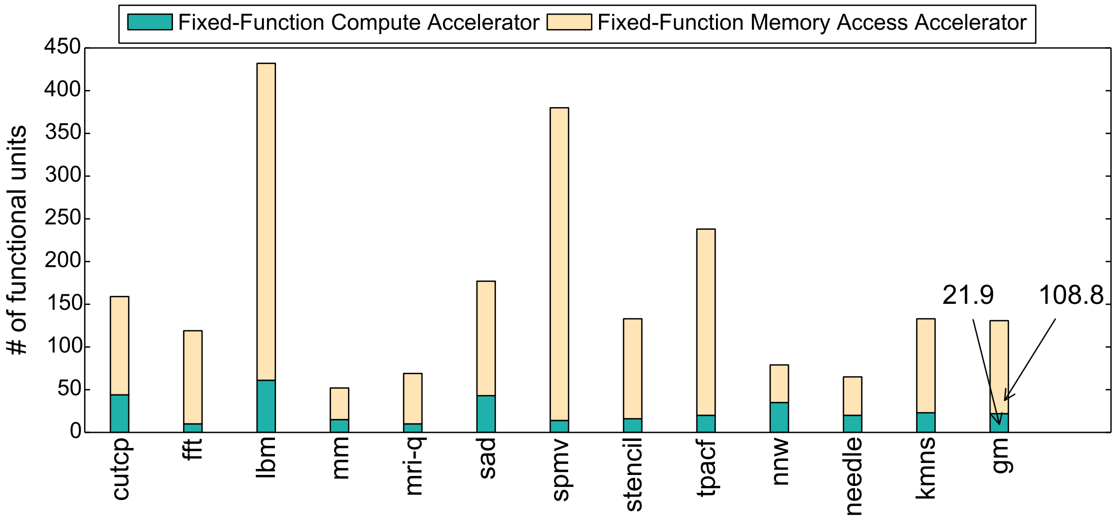


Figure 6.3: Functional unit usages

they are represented as:

$$\frac{Instructions\ executed\ in\ the\ general\ purpose\ processor}{Executions\ in\ the\ fixed-function\ accelerators}$$

Geometric mean of the ratio is 2.14 and it reports that one functional unit execution has the same amount of 2.14 instructions. Especially, `mri-q` shows higher a ratio than any other benchmarks because it is possible to execute a whole accelerating region with few functional unit executions in the fixed-function accelerators while the general purpose processor should consume numerous instructions in its pipeline. We refer to thesis ratios when the overall speedup is explained in section 6.3.1.

Functional units for dataflow Figure 6.3 summarizes the number of functional units that are required for dataflow construction for both the fixed-function compute accelerator and the fixed-function memory access accelerator. Instead of executing instruction on the pipeline of the general purpose processor, fixed-function accelerators can exploit from 52 to 432 functional units. Shown in the figure, most of the functional units are required to construct the dataflow of fixed-function memory access accelerators because memory address calculation and loop condition decision are common compared to computation kernel in benchmarks. Also, those

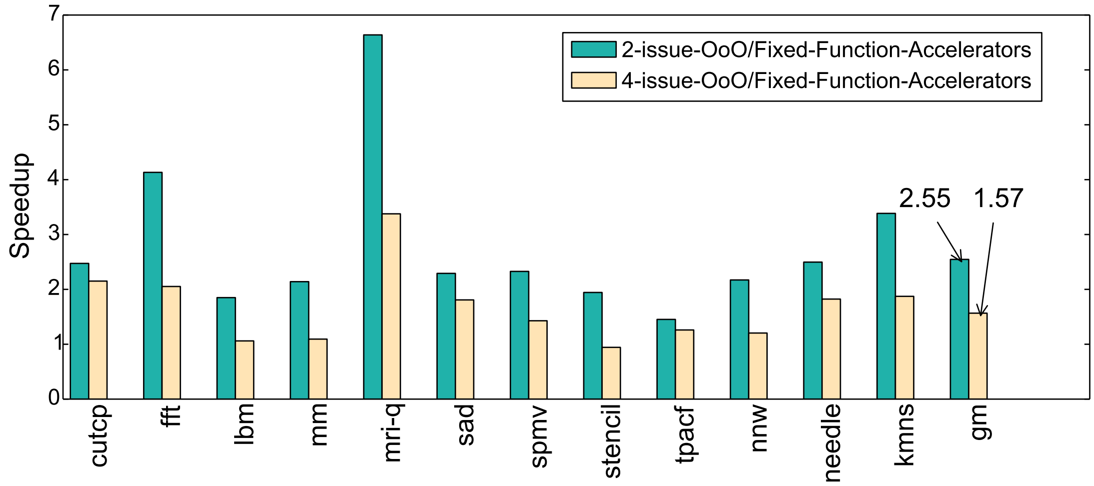


Figure 6.4: Speedup over 2-OOO and 4-OOO

behaviors, including memory address calculation and loop condition decision, are expected to give a negative influence on performance improvement because they may not quickly load data from memory to the fixed-function compute accelerator or store data from the output of the fixed-function compute accelerator to memory. Furthermore, it may have many functional units for loop condition decision in the fixed-function memory access accelerators. To put it another way, it may limit the performance improvement due to nested loops in the accelerating region.

In terms of energy efficiency, the above results suggest that fixed-function accelerators improve energy efficiency by removing instructions loaded on the general purpose processor. Instructions that run on the general purpose processor are eliminated and constructed as fixed-function accelerators with those number of functional units. All operations to execute the functional units supersede the pipeline execution on the general purpose processor and it reduces power consumption in the pipeline.

6.3.1 Performance and Energy Evaluation

This subsection presents the quantitative results of performance improvement and energy reduction achieved with fixed-function accelerators. For performance, we use relative speedup when we use fixed-function accelerators compared to the 2-OOO and 4-OOO general purpose processor. For energy, we use relative energy efficiency by multiplying the speedup factor and the power reduction number when comparing fixed-function accelerators with the general processors that are used for performance evaluation.

Figure 6.4 shows the speedup from fixed-function accelerators when compared to 2-OOO and 4-OOO. It roughly speedups from 1.45× to 6.65× and from 1.20× to 3.38× compared to 2-OOO and 4-OOO, respectively. ccHLS does not support vectorization which operates multiple data in a single instruction and leads to limitation of performance improvement. The fixed-function accelerators handle such vector data as a sequential input of a functional unit. Section 6.3.4 discusses the comparison of the fixed-function accelerators with OOO. The outline of the graph in figure 6.4 follows the trend of the ratio of instruction per a functional unit execution in figure 6.2. Consequently, the benchmarks that use fewer functional unit executions have higher speedup. In other words, fixed-function accelerators achieve speedup by reducing the number of instructions executed in the processor pipeline and executing functional units in parallel in the dataflow of the fixed-function compute and memory access accelerator. For example, `mri-q` only performs 41584 functional unit executions in the fixed-function accelerators while 2-OOO executes 467340 instructions in the pipeline. We can explain each benchmark based on the observation from figure 6.4 and 6.3. In particular:

- `mri-q` requires fewer functional units to organize fixed-function accelerators and it relatively performs fewer functional unit executions compared to instructions loaded on the pipeline of 2-OOO and 4-OOO.

- `lbm` has heavy nested loops with very large computation kernel in the accelerating region. Thus, it requires a number of functional units for the construction of fixed-function accelerators. The nature of `lbm` is that it has heavy nested loops in the source code. Due to the property of fixed-function accelerators, inner-most loop is well accelerated while loops that encapsulate the inner-most loop are held until the execution of accelerating inner-loop is done. The impact of fixed-function memory access accelerator is rare because injecting induction values for loops, except the inner-most loop, are waiting until their nested loops are done.

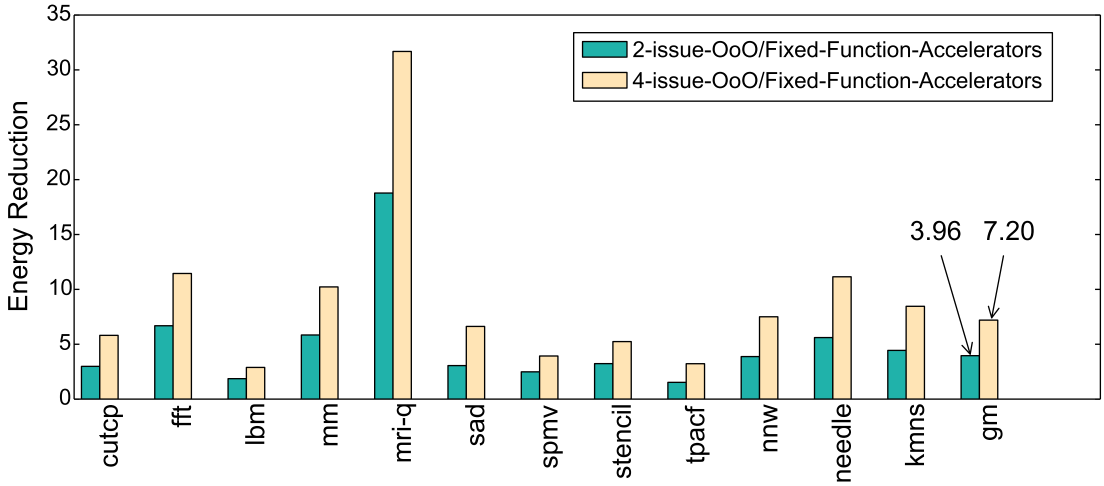


Figure 6.5: Energy reduction over 2-OOO and 4-OOO

- tpacf also has a heavy nested loop, but it has a relatively small computation kernel region. Thus, its speedup is limited by the loops embracing computation kernel loop.

- nnw has many indirect memory access and the true data dependency between functional units to compute memory access address limits overall performance.

- spmv has heavy loops with fewer computation in the computation kernel. Thus, the number of functional unit executions in the fixed-function compute accelerators is few, whereas many indirect memory accesses are shown in this benchmark and the true data dependency between functional units limits the performance improvement.

- sad and needle have sequential loops that have loop-carry dependency. It limits performance because the outer-most loop is held until the last loop in loop sequences is finished accelerating.

Performance improvement mainly contributes to energy reduction. Figure 6.5 shows the energy reduction of fixed-function accelerators compared to 2-OOO and 4-OOO. On geometric mean, fixed-function accelerators reduce the overall energy by 3.96× and 7.20× for 2-OOO and 4-OOO, respectively. Based on observation, unlike speedup, fixed-function accelerators are more efficient than 4-OOO, although its speedup is less than 4-OOO. In figure 6.4, fixed-function

Benchmark	**Area(mm^2)**	**Code lines**	**# of regions**
cutcp	4.541	98	2
fft	2.284	143	1
lbm	7.018	253	4
mm	1.965	48	1
mri-q	2.521	91	1
sad	2.572	85	3
spmv	4.847	110	1
stencil	2.535	74	1
tpacf	3.275	235	1
nnw	3.345	142	3
needle	1.127	63	1
kmeans	3.121	100	1

Table 6.3: Area Size of the Fixed-Function Accelerators

accelerators give better speedup when compared to 2-OOO than that when compared to 4-OOO. However, energy reduction of fixed-function accelerators compared to 2-OOO is worse than that of 4-OOO. It gives a clue that the power consumption of fixed-function accelerators is relatively low in comparison to 4-OOO. It is true because all units in the host processor fall sleep, except the data cache, and it has a positive influence on dynamic power saving. More details are elaborated in the next subsection.

6.3.2 Area Analysis

In this subsection, the area size of fixed-function accelerators is discussed. For area measurement, we use the area report of Synopsys design compiler. Basically, the size of area is the sum of fixed-function compute accelerator, AGU, and FSM. In case that ccHLS generates multiple fixed-function accelerators for multiple regions, the sum of area of fixed-function accelerators for each region is presented.

Table 6.3 shows the area, code lines, and the number of regions to be accelerated. Generally, the size of area is proportional to code lines targeted to generated as the fixed-function accelerators. For example, `lbm` has the longest code lines to be fixed-function accelerators and its area is larger than any other benchmarks. `mm` and `needle` have relatively short code lines and their area is small. The number of code regions, which is generated as separate fixed-function accelerators, also are related to the size of area. `cutcp`, `lbm`, `sad`, and `nnw` have multiple regions to be transformed to fixed-function accelerators and their areas are relatively large. In particular,

- `lbm` has longer code lines and four regions. Hence, it has the largest area size (7.018mm^2).
- `cutcp` has two regions to be generated as fixed-function accelerators while the total code lines for two accelerated regions are relatively short (98 lines for two regions).
- `tpacf` has small area size though it has long code lines (235 lines). The reason is that it has small computation region and most of code lines reuse the fixed-function compute accelerator for loop unrolling.
- While `sad` has three regions that are the target of ccHLS, the size of area is not so large because code lines are distributed to those regions.
- `mm` and `needle` have smaller area size with short code lines and a region.

6.3.3 Architectural Analysis

The fixed-function accelerators from ccHLS achieve their execution in dataflow fashion to improve performance and energy efficiency. To perform parallel execution, access and execute components following the DAE model are decoupled and their sequential code phases are broken into dataflow graphs to be represented as fixed-function compute and memory access accelerators. Finally, they extract parallelism from concurrent execution of those accelerators managed by the finite-state machine. From the architectural model, we expected benefits in terms of i) performance by exploiting parallelism and less execution than the general purpose processor and ii) power efficiency by using few functional units and eliminating pipeline front-end, issue, execute and write back stages. This subsection discusses the capability of fixed-function accelerators to acquire expected benefits. We firstly examine dataflow parallelism in the fixed-function accelerators, then we observe the power efficiency of the fixed-function accelerators with power breakdown.

Dataflow Parallelism in the fixed-function accelerators Figure 6.6 plots the instruction/operation level parallelism for 2-OOO, 4-OOO, and fixed-function accelerators. For the general-purpose processor, including 2-OOO and 4-OOO, we evaluate instructions per cycle (IPC) for parallelism. However, fixed-function accelerators are not able to use IPC to indicate parallelism

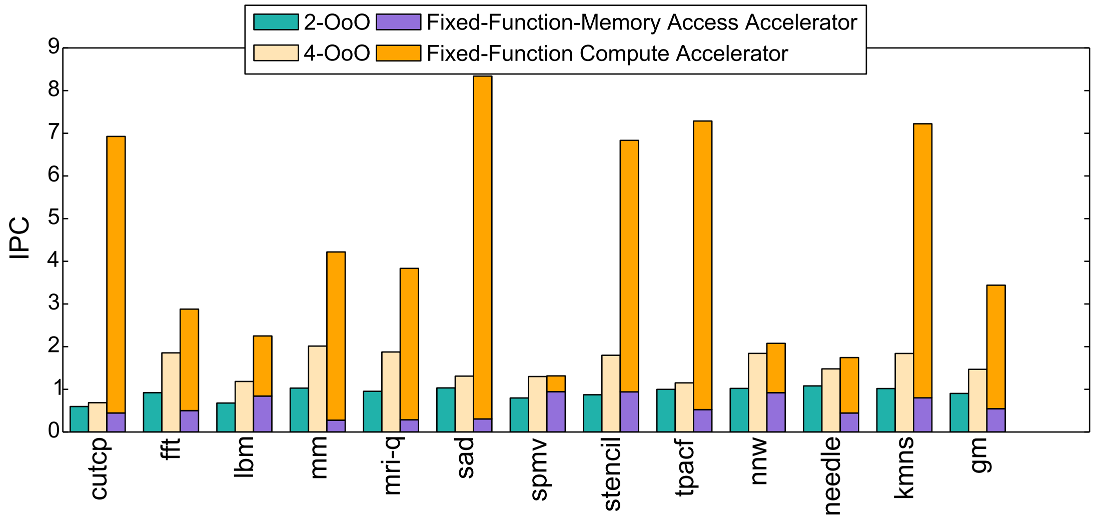


Figure 6.6: Parallelism in fixed-function accelerators

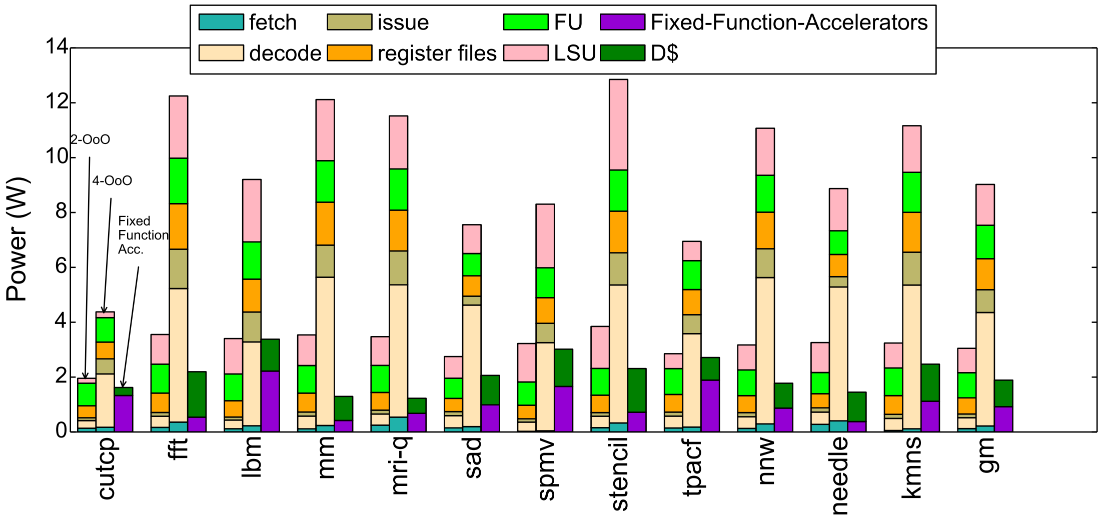


Figure 6.7: Power Breakwodn

and we analyze it with the calculation of the operations per cycle. Its computation is shown below:

$$OPC = \frac{\#\ of\ active\ operations\ of\ the\ functional\ unit\ execution}{cycles}$$

We observe that fixed-function accelerators from ccHLS are aggressively executed in parallel to compute memory address and condition decision in the fixed-function memory access accelerator. At the same time, the fixed-function compute accelerator is performing the computation for the computation kernel. `OPC` is the number of operations to execute the functional unit, so this number does not guarantee that it is the same as the number of executions of functional unit. For example, the functional unit `mul` requires 6 cycles to output the result and the `OPC` is 6.0 when the six `mul` functional units are executed to generate output in parallel.

Power Breakdown Figure 6.7 shows the power breakdown in terms of pipeline stages including 1) fetch, 2) decode and dispatch, 3) issue, 4) register file, 5) functional units in the execute stage, and 6) load store unit including data cache. For the fixed-function accelerators, dynamic power of the data cache is presented because the fixed-function accelerators do not require load and store queues to buffer and retire such instructions. The finite-state machine in the fixed-function accelerators manages for accessing data. Overall, fixed-function accelerators reduce the power because the pipeline stages in the general purpose processor are replaced with the role of fixed-function accelerators. We summarize the power reduction compared to the pipeline stages. Based on the results, we observe:

- For the same amount of workloads, 2-OOO and 4-OOO consume more power than the fixed-function accelerator. In figure 6.7, the sum of fetch, decode, issue, register file, functional units, and LSU exceeds the sum of fixed-function accelerators and data cache.
- In the general purpose processor (2-OOO and 4-OOO), the register file and pipeline front-end including fetch, decode, and issue are major components that consume more power than fixed-function accelerators. Fixed-function accelerators eliminate such pipeline stages by providing fixed-function logic and the register file and provide power efficiency.
- Fixed-function accelerators consume more power in the data cache than LSU, including data cache in 2-OOO because fixed-function accelerators frequently access the data cache instead of using LSQ to forward data. However, the power gap of data cache between 2-OOO and fixed-function accelerators is not so huge.
- Cache coherence by using a unified data cache in fixed-function accelerators is efficiently accelerating with small power consumption because it does not require additional memory for the accelerator and protocols.

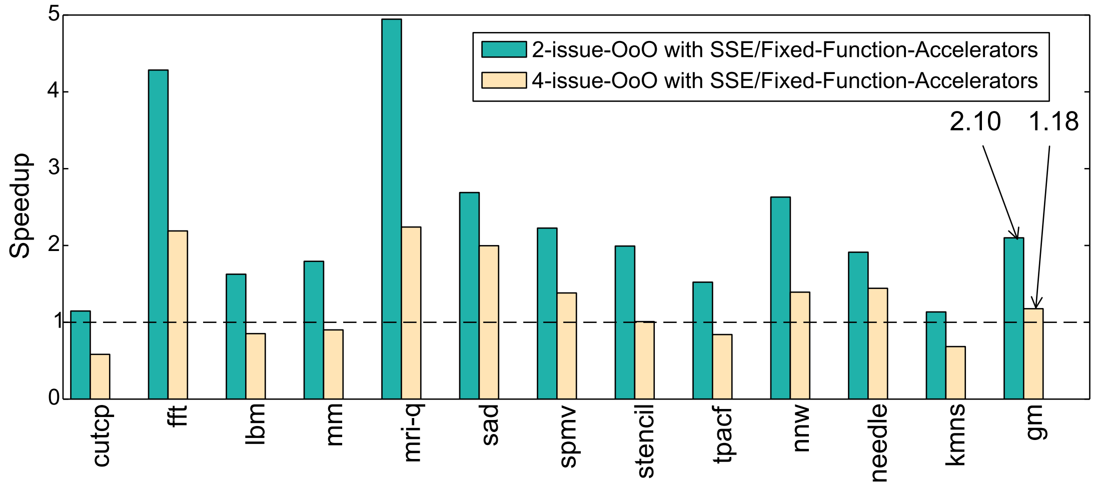


Figure 6.8: Speedup over 2-OOO/SSE and 4-OOO/SSE

6.3.4 Fixed-Function Accelerator with SSE

We also compare the fixed-function accelerators to general purpose processors including SSE [2]. Figure 6.8 outlines the speedup of fixed-function accelerators from ccHLS compared to 2-OOO/SSE and 4-OOO/SSE. It shows that fixed-function accelerators outperform 2-OOO though it includes SSE units. It tells us that parallel execution of functional units in the fixed-function accelerators surpasses that of 2-OO/SSE.

By comparing figure 6.4 and 6.8, we observed that OOO/SSE is efficient when vectorization is effective in the source code. Particularly, in figure 6.8, `kmean` and `cutcp` running on 2-OOO/SSE shows similar speedup numbers compared to fixed-function accelerators while those of 2-OOO give an outperformed number. For 4-OOO/SSE, it gives an outperformed speedup number compared to fixed-function accelerators. It means that non-vectorized processing in ccHLS limits performance improvement.

Figure 6.9 outlines the energy reduction of fixed-function accelerators compared to 2-OOO/SSE and 4-OOO/SSE. By mixing power reduction and performance improvement, it shows that ccHLS creates fixed-function accelerators with lower energy consumption than OOO/SSE.

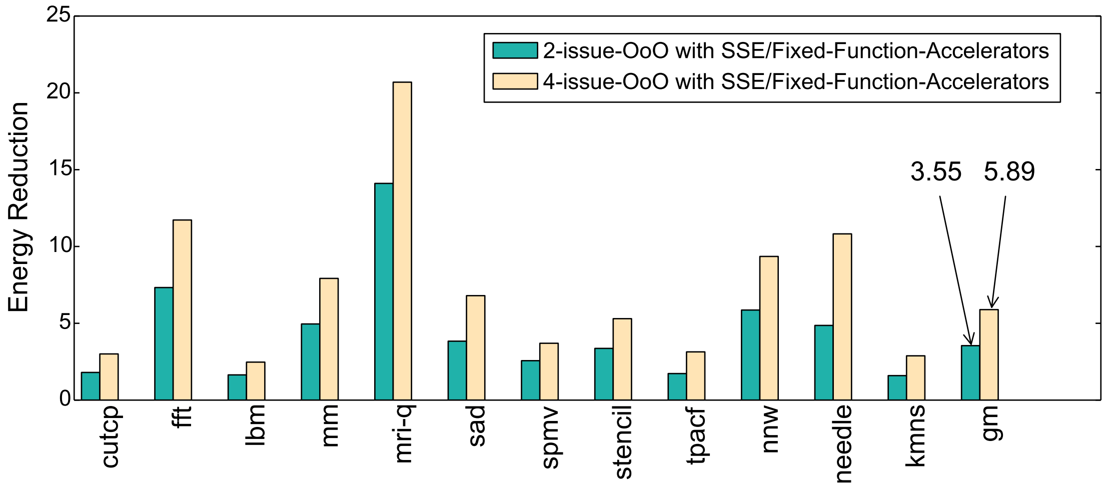


Figure 6.9: Energy reduction over 2-OOO/SSE and 4-OOO/SSE

6.4 Fixed-Function Accelerators with the C-Cores

In this section, we compare fixed-function accelerators to other HLSs. We evaluate the fixed-function accelerators compared to conservation cores (C-Cores), which are proposed by Venkatesh et al. [79]. It is a configurable block where the target accelerating region is embedded and they provide the compiler to generate the accelerators of hot region, and accelerating region, from the source code. The specific design point of C-Cores is integrated with the in-order core to save power because they intend to use it for mobile devices that need to maximize energy efficiency. Particularly, C-Cores limit the memory access only once in the basic block and sequentially access by the order of memory access pattern. For the experimental method, we use the same configuration, except ROB, and other out-of-order specific structures.

6.4.1 Speedup compared to the C-Cores

Figure 6.10 plots the speedup of the fixed-function accelerators and C-Cores. They are normalized with 2-OOO as baseline and shows a geometric mean of speedup 2.55× and 0.88×, respectively. In the results, most benchmarks in the C-Cores show worse performance while fixed-function accelerators outperform 2-OOO. We summarize the following result:

- The performance of C-Cores is limited due to its in-order core design. It means that all

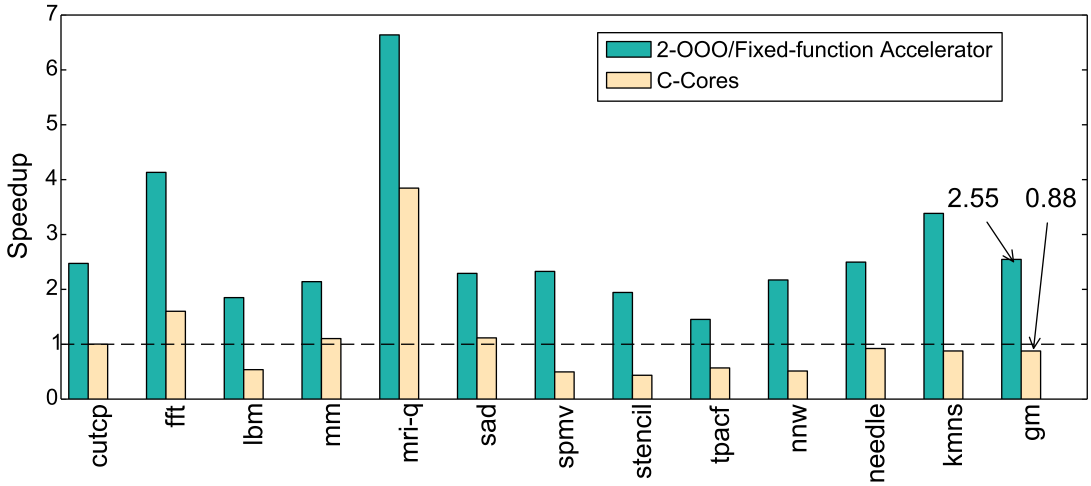


Figure 6.10: Speedup vs C-Cores

generated accelerating region is sequentially executed and always show less performance improvement than fixed-function accelerators.

- Fixed-function accelerators have the components executed in parallel (fixed-function compute accelerator and fixed-function memory accelerator) while C-Cores are always access memory sequentially by the memory access order.

Compared to other HLSs, in terms of performance improvement, ccHLS shows better performance. It is because fixed-function accelerators exploit parallelism by executing memory access and computation separately.

6.4.2 Energy compared to C-Cores

Figure 6.11 plots the energy reduction of the fixed-function accelerators and C-Cores compared to 2-OOO. Both accelerators show better energy reduction phases than 2-OOO. Due to the goal of C-Cores that intends to improve energy efficiency for mobile devices, C-Cores give their energy reduction 2.54×. However, fixed-function accelerators result in higher energy reduction than C-Core (3.96×). Basically, energy reduction of fixed-function accelerators follows the trend of speedup. As shown in figure 6.10, fixed-function accelerators perform better than C-Cores because of its parallelism. In particular:

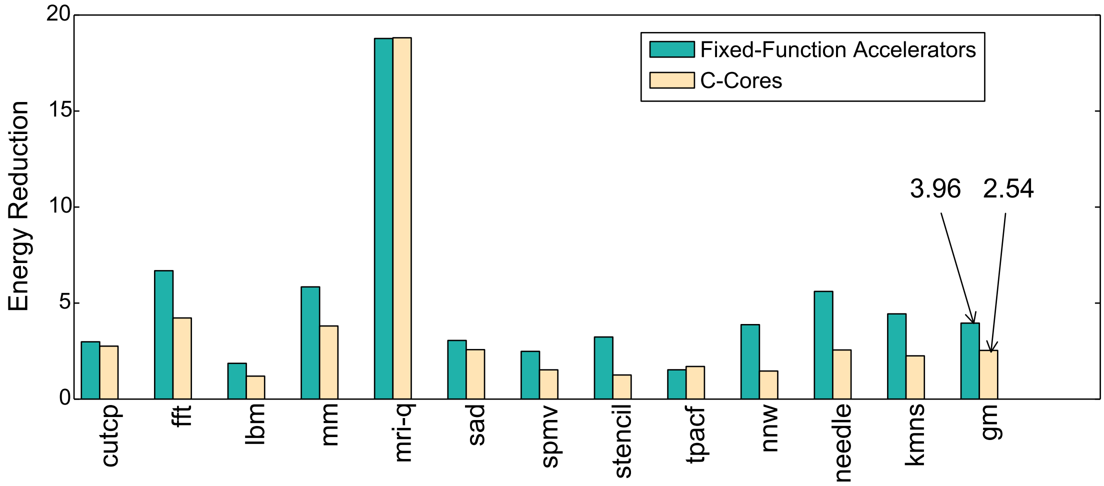


Figure 6.11: Energy Reduction Comapred to C-Cores

- We observe that the power reduction of C-Cores is good enough to save energy for mobile devices because it is integrated with simple in-order core.
- However, fixed-function accelerators have better energy reduction by eliminating power consumption in the pipeline front-end, register file and etc. In addition, the performance improvement of fixed-function accelerators exceed that of C-Core.

Both factors give the result that fixed-function accelerators from ccHLS win against the C-Cores in terms of energy efficiency.

6.5 Chapter Summary

The ccHLS generates fixed-function accelerators for the accelerating region. The generated fixed-function accelerators give better performance and energy reduction compared to 2-OOO and 4-OOO. Our analysis reports that reducing operations instead of executing heavy instructions and parallel execution in terms of fixed-function compute and memory access accelerators allows ccHLS to show better performance and energy reduction. We also compare the fixed-function accelerators to a different HLS (C-Cores). The performance and energy reduction of fixed-function accelerators from ccHLS win against those of C-Cores.

7 RELATED WORK

Various hardware specialization approaches have been proposed to improve performance and power efficiency, which potentially achieve energy efficiency. The architectural models and HLSs are highly focused on achieving efficient hardware specialization and SWSL and ccHLS are also categorized in that field. In this section, comparison of SWSL and ccHLS with other hardware specialization, including architectures and HLSs, are discussed. Section 7.1 talks about SWSL and other hardware design for network lookup and section 7.2 discusses the fixed-function accelerators from ccHLS related to other approaches.

7.1 SWSL and Other Specialization Hardware

Generating lookup hardware datapath from SWSL is inspired by the Pipeline LookUp Grid (PLUG) [47]. PLUG partitions the lookup workloads depending on hierarchy of tries and constructs the data flow graph based on logical pages with memory for lookup data. Those logical pages are mapped into PLUG tiles. In each logical page, there are code functionalities used to look up and they are transformed into the sequence of execution. Tightly connected tiles communicate with their neighbors and are used for network lookup. Similar to PLUG, SWSL organizes the data flow graph for pipeline operation for network lookup following the PLUG programming model and the generated network lookup datapath performs pipeline to achieve higher throughput. Major two differences of SWSL arise compared to PLUG. First, SWSL generates the datapath for a specific network lookup from a single algorithm while the PLUG compiler intends to generate executable binaries that run on cores in the tile; Due to the difference, the PLUG compiler requires additional operation for scheduling to synchronize operation between tiles. However, SWSL evades such synchronization issues and resolves its compiler internally by synchronized delays. Second, no additional hardware blocks are required in SWSL because it directly generates the datapath using wire for appropriate operations. However, PLUG requires additional substrates in the tile to communicate with neighbor tiles.

Latency-Energy-Area-optimized Lookup Pipeline (LEAP) is another approach enhancing network lookup [38]. It basically resembles PLUG in terms of a tile-based architecture and providing semi- programmability for flexibility. However, it is a native dataflow design configuring operation engines as computation. Thus, it is placed intermediately between PLUG (programmable cores) and SWSL (ASIC-like hardware design). In addition to PLUG and LEAP, we briefly discuss other specialization architectures in relation to fixed-function lookup engines from SWSL.

HLSs HLSs SWSL concentrates on improving throughput rather than performance improvement. The main reason that SWSL differentiates the interest from other HLSs is that the main characteristic of workloads for SWSL is not the same as that for other HLSs. In the network, flooding of lookup requests enable the lookup engine to demand the capability for managing all requests that execute the same workload continually. It allows SWSL to be fit for the network lookup while other HLSs cannot provide such capability because they are originally designed to reduce execution time for a certain amount of workloads.

Memory Generated lookup modules from SWSL are based on using SRAMs in the connection of logical pages. Normally, network lookups use Ternary content addressable memories (TCAMs) to achieve hardware parallelism to match the search key against all entries [68, 56]. Due to exploiting parallelism by activating all entries in parallel, it consumes a large amount of power; this is widely known as the problem. Although selective activation of TCAM blocks improves power efficiency [81], using SRAM-based algorithmic lookup is a better solution than TCAMs for supporting large tables. SWSL is categorized in this approach.

Functionality The SWSL programming model is motivated by the SDF model [51] and the Click dataflow model [45]. Unlike SDF and Click that focus on the entire functionality, SWSL only focuses on automatically generating lookup hardware designs. Moreover, the goal of SWSL is to generate the lookup hardware, not to run on the general-purpose processor.

7.2 Other Specialization Approaches related to ccHLS

Unlike SWSL, ccHLS targets specific tasks, regions, or phases in the generic programs running on the general-purpose processor to generate fixed-function accelerators. Generated fixed-function accelerators are classified as fixed-function compute accelerators for accelerating the loop kernel and fixed-function memory access accelerators for computing access memory address and loop condition decisions. In detail, the fixed-function memory accelerator consists of the combination of the address generation unit (AGU) and the finite-state machine (FSM). Those separate models follow the DAE model and simultaneously executes for computation and memory access management. This section discusses the fixed-function accelerators compared to other accelerators and HLSs in terms of execution model, memory utilization, scheduling, and programmability.

Execution model Fixed-function accelerators from ccHLS combines the DAE and dataflow model. The DAE model was proposed in the 1980's [70, 30] and adapted in recent designs [46, 40, 9]. They consider separate computation and access accelerating units and give the motivation that individual computation and access accelerators are integrated. The dataflow execution of fixed-function accelerators from ccHLS is borrowed from the classic dataflow machines [27, 36, 41, 11, 61] and more recent designs [65, 72, 15]. It also aims to target specific regions as dataflow computation. Recently, dataflow machines combined action/events were proposed and Trigger Instructions (TI) [62] and Memory Access Dataflow (MAD) [20] follow this model. MAD is amore evolved design in terms of using low-level dataflow for memory access. Fixed-function accelerators from ccHLS resemble MAD which also follows the DAE model and implements it using events and relevant actions. It is tightly integrated with various accelerators for execution and enhances accelerating. For data movement, MAD hardware engine is organized as computation, event, and action blocks and triggers action on the table in the action block by the event based on computation results motivated by TI. In terms of fixed-function accelerators, MAD has similarities with the fixed-function memory access accelerator because a series of works by MAD is exactly matched with that of the fixed-function memory access accelerator - Computation for memory address is done by AGU and triggering events and related actions are managed by FSM. The fixed-function compute accelerator is derived from

DySER, which is an in-core accelerator based on dataflow of the target region [32]. Instead of configuration, ccHLS directly generates the datapath of the fixed-function compute accelerator. Like the form of the DAE and dataflow models, ccHLS is an HLS to generate a static hardware for acceleration while the execution is similar to MAD.

Memory utilization For memory utilization, fixed-function accelerators directly access the data cache in the host processor. Major HLSs remove the data cache in the host processor or adapt scratch-pad memory for data sharing with the host processor [6, 17, 54]. Basically, they assume that no cache coherency mechanism is required to interact because the host processor and accelerators see the same memory space that is shared. In the case that the host processor has a data cache, the data cache is flushed when it starts acceleration to avoid operations for cache coherency [7]. However, fixed-function accelerators access memory from the data cache in the host processor; they do not have any cache conflict issues and the host processor does not need to flush the data cache when acceleration starts because the computation result is reflected in the data cache. Thus, ccHLS follows the concept of unified memory space.

Scheduling Scheduling is a less important issue for fixed-function accelerators. Many HLSs define fixed latency for memory access [80, 58] for execution scheduling and that is one of the reasons to limit generated hardware performance. While they assume fixed latency for memory access, ccHLS does not have such a constraint and generated fixed-function accelerators are free from the constraint because FSM in the fixed-function memory accelerator manages cache access and sets up the data when the accessed address is valid. Including the structure of the functional unit for relaxed scheduling, it allows fixed-function accelerators to schedule free.

Programmability Bluespec [10] has been proposed to relax the complexity of hardware design. Also, few designs with Bluespec have been presented [44, 23, 22]. Basically, it is a high-level functional hardware description programming language which was essentially Haskell-extended to handle chip design to support hardware design automation and write the program that is similar to high level program languages to ease the painful hardware design. Although it uses a novel program language, it is required to learn how to perform using Bluespec language because it is different from the conventional program languages like C or C++. ccHLS easily reuses the code without any heavy modification or rewriting it.

7.3 Chapter Summary

In this chapter, we discuss the related work to the proposed high-level synthesis approaches, SWSL and ccHLS. SWSL was initially invented to provide high throughput for network lookups with hardware design. Due to its specialized properties for network lookup, fixed-function accelerators from ccHLS are proposed and provide efficient memory access and parallel execution - computation and memory access, separately.

8 CONCLUSION

The interest of the computer architecture community has slightly shifted from microarchitectural strategies to accelerators to gain performance and energy efficiency. This dissertation has described novel compilation approaches to generate accelerators depending on the properties of target applications from high-level synthesis (HLS) that uses high-level program languages such as C/C++, which significantly reduce design complexity when accelerators are designed. In this chapter, we discuss the contributions.

8.1 Contributions

The HLSs are promising in that they create the accelerator for a specific application implemented with the use of high-level programming languages. We have proposed two approaches - SWSL for network loops and a ccHLS for accelerating of generic code region. In this section, we summarize the contributions of the proposed compilation techniques.

8.1.1 SWSL

SWSL is similar to PLUG which is the specialized architecture for network lookup. Thus, SWSL is a dataflow graph (DFG)-based application programming model that is mapped onto the PLUG tile to execute. However, the code blocks in the DFG are transformed into a lookup hardware module by SWSL compilation and its execution model is different from PLUG and other network loop architectures. In particular:

The Execution Model The lookup module from SWSL does not obey von-Neumann architectures that get instructions from memory to follow pipeline structures. The lookup hardware from SWSL completely seeks the nature of dataflow machine and each code block functionality is organized as the hardware datapath. The output from one logical page of DFGs is passed to next logical page and chooses the hardware module for it. The code block functionalities in DFG consist of the chain of datapaths. In that manner, SWSL achieves high throughput in that every lookup request is processed in the datapath of the lookup hardware.

Compilation The compilation SWSL compiler directly uses conventional intermediate representation (IR) without any modification in the front-end compiler. The back-end, however, modifies the control flow graph (CFG) by combining basic blocks and provides control logic for dataflow operation when it generates the lookup hardware. Each lookup request is independent and scheduled to avoid the confliction.

8.1.2 ccHLS

ccHLS has three different keys compared to other HLSs. In particular:

E-AEPDG in Compiler Unlike other HLSs that are originated from CFG IR, ccHLS proposes a novel IR called E-AEPDG, which partitions the computation kernel and data movement computing memory access address. It leads the program written in high-level program language to the hardware datapath for acceleration. Hence, E-AEPDG transforms the conventional CFG from to the datapath of accelerator execution and provides the performance improvement by exploiting parallelism and energy reduction by eliminating the energy consumed in the pipeline front-end.

The Execution Model Unlike other HLSs, the fixed-function accelerators from ccHLS borrow their execution from the decoupled access/execute model (DAE). Many HLSs combine generation memory access and computation kernel and follow a sequential order for calculating memory access and computing kernel. Based on the DAE model in the fixed-function accelerators, however, it calculates memory address and computation in parallel and improves performance. To manage execution, FSM is provided by the ccHLS, states are set based on the condition of the output event queue. It shares data cache in the Host Processor. The fixed-function accelerators from the ccHLS directly access the data cache in the host processor when it loads or stores data. While other conventional HLSs provide the memory space for the accelerator, the fixed-function accelerators share the data cache in the host processor. Therefore, it does not require any cache coherence mechanism to maintain coherency between the host processor and fixed-function accelerators.

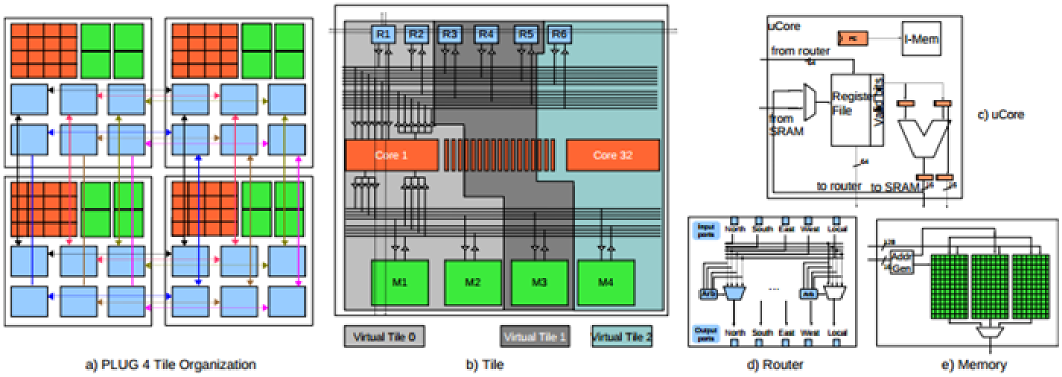
8.2 Closing Remarks

We have shown how accelerators are automatically generated from high-level languages using proposed compilation techniques for the HLS. Recently, the specialized accelerators are important for achieving both performance and energy to proliferate high performance computing and mobile devices. In addition, saving design time is a major factor in time-to market when the architects and engineers in the industries consider designing the hardware. During the design of SWSL and the ccHLS, we considered how we provide easy and effective ways to build the accelerators and concluded that HLS is the best way to do this. From the specific (network lookup) to generic applications, we have developed compilation techniques and tried to grow throughput and performance with energy efficiency. We hope that this work improves the development process of the future accelerators and strongly support the efficient hardware accelerator design.

A PIPELINE LOOKUP GRID (PLUG)

Figure A.1 shows the architectural overview of PLUG [47]. PLUG is a tile-based architecture, and each tile has 32 μCores, 6 routers, and 4 64KB SRAMs. Network algorithms executing in the PLUG are drawn as a form of data-flow graph with an amount of logical pages. Each logical page has programmed code called code block, which is compiled to be executed on the μCore. Hence, PLUG executes small pieces of workloads which construct the whole workload for the network algorithms. Because PLUG is only a high speed lookup engine, it requires a host processor to feed packet information in the form of input messages to the PLUG tile to lookup. Routers in each tile can be used to communicate with its neighbor tiles and output from a tile goes into its neighbor tile as an input message depending on the edge of data-flow graph by network algorithms. Up to 80-bits data can be transmitted in a message. A μCore in the tile is a single-issue in-order core with shared instruction memory. Partitioned workload of a network algorithm is compiled and loaded in the shared instruction memory. Hence, the input message at every single cycle is assigned to an idle μCore to lookup data structure and it allows PLUG to have high throughput by achieving thread-level parallelism because each input message is a different task. Each μCore has 32 registers to compute, so it must be carefully used. SRAMs in each tile provide total 256KB memory space, which uses as local data structure storage for data lookup. PLUG is scalable, so it is possible to modify parameters of PLUG tiles.

Figure A.1 shows the architectural overview of PLUG [47]. PLUG is a tile-based architecture,



a) PLUG 4 Tile Organization b) Tile c) uCore d) Router e) Memory

Figure A.1: Architectural Overview of PLUG

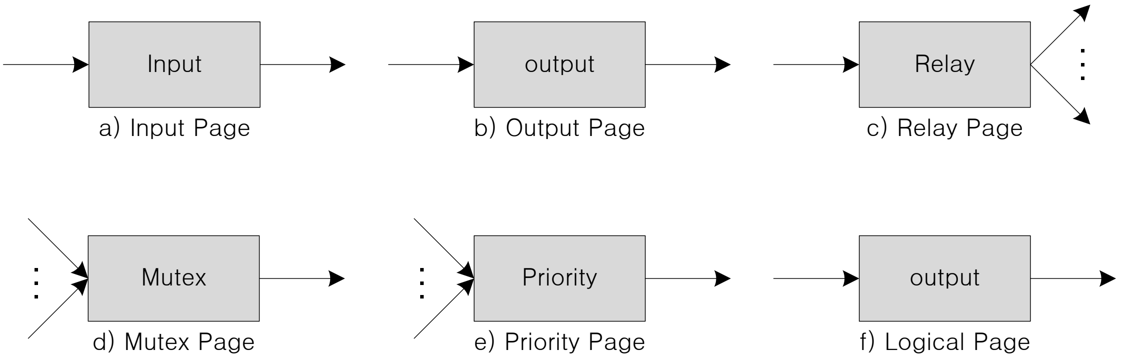


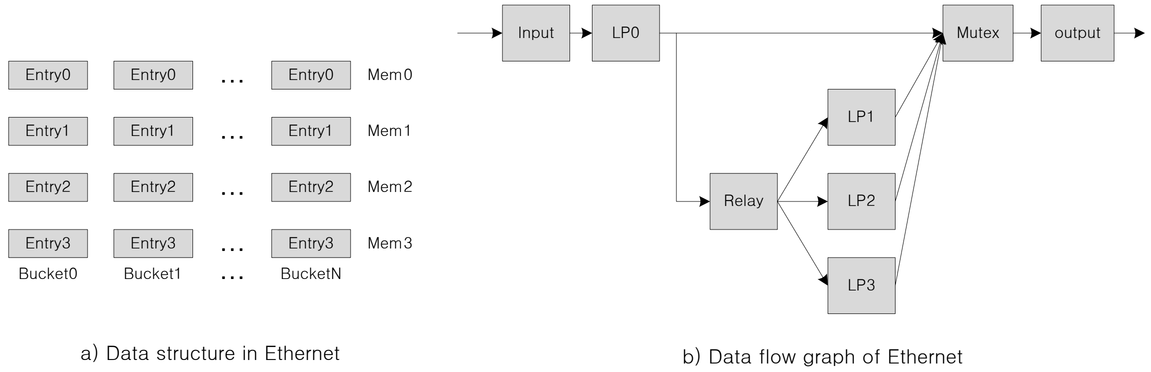
Figure A.2: Logical Page Class

and each tile has 32 μCores, 6 routers, and 4 64KB SRAMs. Network algorithms executing in the PLUG are drawn as a form of data-flow graph with amount of logical pages. Each logical page has programmed code called code block, which is compiled to be executed on the μCore. Hence, PLUG executes small pieces of workloads which construct whole workload for the network algorithms. Because PLUG is only a high speed lookup engine, it requires a host processor to feed packet information in the form of input message to the PLUG tile to lookup. Routers in each tile can be used to communicate with its neighbor tiles and output from a tile goes into its neighbor tile as an input message depending on the edge of data-flow graph by network algorithms. Up to 80-bits data can be transmitted in a message. A μCore in the tile is a single-issue in-order core with shared instruction memory. Partitioned workload of a network algorithm is compiled and loaded in the shared instruction memory. Hence, input message at every single cycle is assigned to idle μCore to lookup data structure and it allows PLUG to have high throughput by achieving thread-level parallelism because each input message is a different task. Each μCore has 32 registers to compute, so it must be carefully used. SRAMs in each tile provide total 256KB memory space, which uses as local data structure storage for data lookup. PLUG is scalable, so it is possible to modify parameters of PLUG tiles.

Lookup applications of the PLUG organize the logical structure of the data with associated computation for data lookup or update. For the utilization of the PLUG, lookup algorithms must be drawn as a data-flow graph (DFG). DFG describes the group of partitioned data structures of the algorithm, and each data structure in the group can be utilized for computation by its purposes such as lookup, update, and output for connected data structure. A series of data structures is called logical pages, which consist of a whole DFG. A logical page must be connected with other logical pages through edges depending on the matching data structure. Except a special purpose logical page, all the logical pages can be designed by general purposes. Figure A.2 a), b), c), d), and e) show the class of logical pages. As a rule of designing code block, directions of designing DFG are defined. First, an input page must be connected at the beginning of DFG to represent that DFG starts by receiving messages from the host processor. On the contrary, an output page notifies that whole lookup processing is finished and its output messages return to the host processor. Second, a relay page broadcasts input message to logical pages connected to the relay page. Thus, a DFG requiring broadcast messages from a logical page to other multiple logical pages can be designed with the relay page. Third, DFG requires a mutex page when multiple logical pages are connected to the same destination logical page. One output from them is used for the input messages of the destination logical page through bypassing the mutex page. Fourth, a priority page represents the output of one of the logical pages connected to the destination logical page which has priority. In a normal case, the priority page just bypasses its input messages to the connected logical pages, but input messages through high priority edges have a high priority to be passed to the destination logical page. Finally, a normal logical page is shown in Figure A.2 f) and is the main computation block for lookup. It is programmed by a programmer to lookup. The programmed code for lookup is called code block. Code block is implemented by C++ with the PLUG software framework. The PLUG framework provides special-purposed programming APIs such as send/receive message, memory utilization, and bit manipulation to utilize PLUG.

This programming structure increases flexibility to support newly developed network algorithms and only software upgrade or porting allows PLUG to execute other algorithms. Each logical page is able to have multiple code blocks which use the same data structure and code block decision can be done by the input message header. The message header has reserved 4 bits for code block decision of a logical page. Up to 16 code blocks are loaded in the logical page.

To describe the programming on the PLUG, Ethernet and IPv4 are shown as examples [24]. Figure A.3 shows the data structure for Ethernet and its DFG model. The main role of Ethernet



a) Data structure in Ethernet

b) Data flow graph of Ethernet

Figure A.3: Data Flow Graph of Ethernet

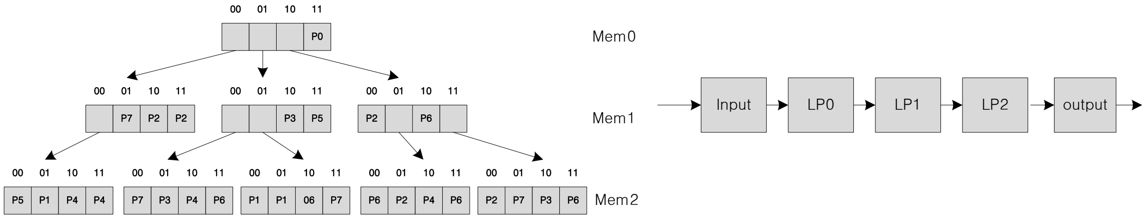
is to look up the port number of the packet that should arrive. Ethernet consists of 48-bits MAC address space, so the lookup engine compares the destination MAC address in the packet with that in the memory to find the destination port of the input packet. The data structure is organized as a 48-bit MAC address and the destination port number is stored in the logical page with a hash function. Figure A.3 a) shows simple 4 entries with an N bucket hash table, and the input packet broadcasts to 4 entries. Only one entry is enabled when the entry is matched with the hash key. If the MAC address in the data structure of the target entry is identical with that from the input packet, the destination port number in the data structure is sent outside the PLUG. Figure A.3 b) shows the data flow graph for Ethernet lookup. Each entry is assigned from logical page0 to logical page3, respectively. There are three code blocks, such as lookup, update and read for each logical page. Code block lookup tries to lookup the data structure, so it will send the destination port number as output if it has it. If the output is NULL, the host processor requests an update to store the non-matched MAC address. PLUG executes these 4 logical pages in parallel, so PLUG guarantees low latency and high throughput for searching the destination port number of an input packet. For full operation, each logical page has two code blocks - writing the MAC address and destination port number to the target entry and lookup the destination port number. Figure A.4 shows the code block for lookup in LP0 of figure A.3. In line 8, code block receives message from the input page and passes this message to the relay page for executing lookup in other logical pages. From line 10 to 21, it broadcasts the input message to other logical pages. In line 23 and 24, lookup code block reads data in memory using

```
void ethfwtable_head_lookup_code_block::Execute(){

  PLUG_UINT code_addr;
  plug_vector<MSG_VECTOR_SIZE> msg_vec_in, msg_vec_out, msg_vec_relay;
  plug_header msg_vec_in_hdr, msg_vec_out_hdr, msg_vec_relay_hdr;
  plug_vector<4> mem_vec;

  ReceiveMessage(msg_vec_in_hdr, msg_vec_in, (PLUG_UINT)0);

  msg_vec_relay[0] =  msg_vec_in[0];
  msg_vec_relay[1] =  msg_vec_in[1];
  msg_vec_relay[2] =  msg_vec_in[2];
  msg_vec_relay[3] =  msg_vec_in[3];

  // Set codeblock of message
  SetMessageCodeblock(msg_vec_relay_hdr, GetMessageCodeblock(
      msg_vec_in_hdr));

  // Set Destination of message
  SetMessageDestination(msg_vec_relay_hdr, PLUG_DESTINATION_IMPLICIT);
  // Send message to relay page through edge 1
  SendMessage(msg_vec_relay_hdr, msg_vec_relay, 1);

  code_addr = msg_vec_in[0];          // Get Hash key
  LoadWord<4>(mem_vec, code_addr);    // Get data in memory

  /** If 48 bit address in message and memory is the same,
      data should be sent to output.**/
  if( (msg_vec_in[1] == mem_vec[0]) &&
      (msg_vec_in[2] == mem_vec[1]) &&
      (msg_vec_in[3] == mem_vec[2]) )
  {
    msg_vec_out[0] = GetLocalConstant(0);
    msg_vec_out[1] = mem_vec[3];

    SetMessageCodeblock(msg_vec_out_hdr, 0);
    SetMessageDestination(msg_vec_out_hdr, PLUG_DESTINATION_IMPLICIT);
    SendMessage(msg_vec_out_hdr, msg_vec_out, 0);
  }
}
```

Figure A.4: Lookup code block in LP0

```
void ethfwtable_head_write_code_block::Execute(){
  PLUG_UINT bucket_index, dst;
  plug_vector<MSG_VECTOR_SIZE> msg_vec_in, msg_vec_relay;
  plug_header msg_vec_in_hdr, msg_vec_relay_hdr;
  plug_vector<4> mem_vec, msg_vec_store;

  ReceiveMessage(msg_vec_in_hdr, msg_vec_in, (PLUG_UINT)0);
  dst = GetMessageDestination(msg_vec_in_hdr);
  if(dst == GetLocalConstant(0)){
    bucket_index = msg_vec_in[0];
    // Edge 1: Address (48bits)|Port (12bits)| Timestamp(4bits)
    ReceiveMessage(msg_vec_in_hdr, msg_vec_in, (PLUG_UINT)1);

    // Store vector in memory
    msg_vec_store[0] = msg_vec_in[0];
    msg_vec_store[1] = msg_vec_in[1];
    msg_vec_store[2] = msg_vec_in[2];
    msg_vec_store[3] = msg_vec_in[3];

    StoreWord<4>(msg_vec_store, bucket_index);
  }
  else{
    msg_vec_relay[0] = msg_vec_in[0];
    msg_vec_relay[1] = msg_vec_in[1];
    msg_vec_relay[2] = msg_vec_in[2];
    msg_vec_relay[3] = msg_vec_in[3];

    SetMessageCodeblock(msg_vec_relay_hdr, GetMessageCodeblock(msg_vec_in_hdr));
    SetMessageDestination(msg_vec_relay_hdr, dst);
    SendMessage(msg_vec_relay_hdr, msg_vec_relay, 1);

    ReceiveMessage(msg_vec_in_hdr, msg_vec_in, (PLUG_UINT)1);
    msg_vec_relay[0] = msg_vec_in[0];
    msg_vec_relay[1] = msg_vec_in[1];
    msg_vec_relay[2] = msg_vec_in[2];
    msg_vec_relay[3] = msg_vec_in[3];

    SetMessageCodeblock(msg_vec_relay_hdr, GetMessageCodeblock(msg_vec_in_hdr));
    SetMessageDestination(msg_vec_relay_hdr, dst);
    SendMessage(msg_vec_relay_hdr, msg_vec_relay, 2);
  }
}
```

Figure A.5: Code block for updating MAC address in LP0



A) Data structure in IPv4

b) Data flow graph of IPv4

Figure A.6: Data Flow Graph of IPv4

the hash key. The MAC address is compared with the 48 bit address and results are passed to the mutex page if it finds a matching MAC address. In the case that a matching MAC address is not found in each logical page, the requested lookup MAC address and its destination port are updated. Figure A.5 presents the code block updating the MAC address and destination port number in the target entry. Two message edges are used to provide 16-bit bucket index, 48-bit MAC address, and 12-bit destination port number. Because each message edge passes up to 64-bits data to the next logical page, except the 16-bit message header, two edges are used to provide those data into the next logical page. From edge 0, in line 7, code block for updating receives the bucket index used for storing word location in memory and edge 1, shown in line 12 or 32, provides the MAC address and destination port number. If the destination logical page number is identical with the target destination number from the message header, the MAC address and destination port number are stored in the memory by the bucket index. Otherwise, data in two input edges are broadcasted to next logical pages connected by the relay page.

Figure A.6 shows the data structure and data flow graph of a simplified version of IPv4. IPv4 uses the longest prefix matching the IP address. To operate the longest prefix matching, the data structure is constructed as multibit tries shown in Figure A.6 a). Each data structure in each node has the port number of the next hop or an index pointer of the next node, so output will be generated if the pointed index has the next port number. Otherwise, it will point to the next node. At the beginning of operation, IPv4 algorithm traverses from root to leaves with two bits of IP address. These two bits of IP address are the index key of the current node. Figure A.6 b) shows the data flow graph of the simplified IPv4 algorithm. According to the data structure shown in Figure A.6 a), logical page 0 has the first memory structure (Mem0) in Figure A.6 a), and the data structure of Mem 1 and 2 are assigned to logical page 1 and 2, respectively. If the next port number is found at the logical page1, it will be sent outside PLUG through the logical page link as the destination port number. In that case, the connected logical pages bypass the port number without any operation Otherwise, the message will be sent to the next logical page with the point index in the current data structure to find the longest prefixed matched result.

[1] "Comparing and contrasting fpga and microprocessor system design and development https://www.xilinx.com/support/documentation/white _papers/wp213.pdf."

[2] "Intel Streaming SIMD Extensions 4 (SSE4)," http://http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.html, accessed: 2014-08-14.

[3] "Llvm language reference manual http://llvm.org/docs/langref.html."

[4] "Marc Horowitz. Why design must change (slides), http://www.synopsys.com/Community/UniversityProgram/CapsuleModule/Why-Design-Must-Change.ppt."

[5] *MicroBlaze Processor Reference Guide*. Xilinx Corp.

[6] *Nios II C2H Compiler User Guide*. Altera Corp.

[7] *Nios II Processor Reference Handbook*. Altera Corp.

[8] *Parboil Benchmark Suite*. http://impact.crhc.illinois.edu/parboil.php.

[9] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting mobile gpu performance with a decoupled access/execute fragment processor," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 84–93. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337169

[10] Arvind, "Bluespec: A language for hardware design, simulation, synthesis and verification invited talk," in *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, ser. MEMOCODE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 249–. [Online]. Available: http://dl.acm.org/citation.cfm?id=823453.823860

[11] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 300–318, Mar. 1990. [Online]. Available: http://dx.doi.org/10.1109/12.48862

[12] F. Baboescu, D. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *ISCA '05*.

[13] 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, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[14] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve ip lookups,," in *INFOCOM '03*.

[15] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 14–26. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024396

[16] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000. [Online]. Available: http://dx.doi.org/10.1109/2.839323

[17] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2514740

[18] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*, 1st ed. Springer Publishing Company, Incorporated, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1458033

[19] M. Casado, M. J. Freedman, J. Pettit, J. anying Luo, N. McKeown, and S. Shenker, “Ethane: taking control of the enterprise,” in *SIGCOMM '07*.

[20] K. S. Chen-Han Ho, Sung Jin Kim, “Memory access dataflow,” University of Wisconsin Computer Sciences Technical Report CS-TR-2014-1802, Mar 2007.

[21] P. Coussy, M. Meredith, D. D. Gajaski, , and A. Takach, “An introduction to high-level synthesis,” *IEEE Trans. Design & Test of Computers*, vol. 26, pp. 8–17, 2009.

[22] N. Dave, “Designing a reorder buffer in bluespec,” in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 93–102.

[23] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan, “Hardware acceleration of matrix multiplication on a xilinx fpga,” in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–100. [Online]. Available: http://dx.doi.org/10.1109/MEMCOD.2007.371239

[24] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, “Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers,” in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, ser. SIGCOMM '09, 2009, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592593

[25] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small forwarding tables for fast routing lookups,” in *SIGCOMM '97*. [Online]. Available: citeseer.ist.psu.edu/degermark97small.html

[26] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet's with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, October 1974.

[27] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*,

ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[29] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, no. 7, pp. 478–490, Jul. 1981. [Online]. Available: http://dx.doi.org/10.1109/TC.1981.1675827

[30] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "Pipe: A vlsi decoupled architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ser. ISCA '85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 20–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=327010.327117

[31] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1109/MM.2011.18

[32] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 503–514.

[33] V. Govindaraju, "Energy Efficient Computing Through Compiler Assisted Dynamic Specialization," PhD Dissertation, Unversity of Wisconsin-Madison, 2014.

[34] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles: Liberating accelerators by exposing flexible microarchitectural mechanisms," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques "'(PACT)"'*, 2013.

[35] P. Gupta, S. Lin, and N. Mckeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM '98*.

[36] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.2468

[37] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a GPU-accelerated software router," in *SIGCOMM '10*.

[38] E. N. Harris, S. L. Wasmundt, L. De Carli, K. Sankaralingam, and C. Estan, "Leap: Latency- energy- and area-optimized lookup pipeline," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396595

[39] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997, pp. 16–18.

[40] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 168–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_14

[41] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 131–140. [Online]. Available: http://dl.acm.org/citation.cfm?id=52400.52416

[42] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMs: an SRAM-Based parallel multi-pipeline architecture for terabit IP lookup," in *INFOCOM '08*.

[43] V. Kathail, "Creating power-efficient application engines for soc design," *Synfira Inc. Soc Central*, 2005.

[44] M. King, N. Dave, and Arvind, “Automatic generation of hardware/software interfaces,” in *ASPLOS*, 2012, pp. 325–336.

[45] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[46] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, “Towards more efficient execution: A decoupled access-execute approach,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS ’13. New York, NY, USA: ACM, 2013, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465012

[47] A. Kumar, L. De Carli, S. J. Kim, M. de Kruijf, K. Sankaralingam, C. Estan, and S. Jha, “Design and implementation of the plug architecture for programmable and efficient network lookups,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT ’10, 2010, pp. 331–342. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854316

[48] S. Kumar, M. Becchi, P. Crowley, and J. Turner, “CAMP: fast and efficient IP lookup architecture,” in *ANCS ’06*.

[49] S. Kumar and B. Lynch, “Smart memory for high performance network packet forwarding,” in *HotChips*, Aug. 2010.

[50] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO ’04*, pp. 75–88.

[51] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 7, no. 9, pp. 1235–1245, September 1987.

[52] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,”

in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009, pp. 469–480.

[53] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "Cgpa: Coarse-grained pipelined accelerators," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 78:1–78:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593105

[54] R. Lysecky and F. Vahid, "Design and implementation of a microblaze-based warp processor," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 3, pp. 22:1–22:22, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1509288.1509294

[55] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 45–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=144953.144998

[56] A. J. Mcauley and P. Francis, "Fast routing table lookup using cams," in *IEEE INFOCOM*, 1993, pp. 1382–1391.

[57] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[58] G. Mittal, D. C. Zaretsky, X. Tang, and P. Banerjee, "Automatic translation of software binaries onto fpgas," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 389–394. [Online]. Available: http://doi.acm.org/10.1145/996566.996678

[59] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[60] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. Deng, and S. Zhang, “IP routing processing with graphic processors,” in *DATE ’10*.

[61] G. M. Papadopoulos and D. E. Culler, “Monsoon: An explicit token-store architecture,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA ’90. New York, NY, USA: ACM, 1990, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/325164.325117

[62] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered instructions: A control paradigm for spatially-programmed architectures,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 142–153. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485935

[63] C. Rowen and S. Leibson, “Flexible architectures for engineering successful SOCs,” in *DAC ’04*.

[64] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. Taylor, “Efficient complex operators for irregular codes,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 491–502.

[65] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore, “Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture,” in *ISCA ’03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 422–433.

[66] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, “Scalable hardware memory disambiguation for high ilp processors,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 399–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956553

[67] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, 2010.

[68] D. Shah and P. Gupta, "Fast updating algorithms for tcams," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1109/40.903060

[69] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *SIGCOMM '03*.

[70] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 112–119. [Online]. Available: http://dl.acm.org/citation.cfm?id=800048.801719

[71] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 273–284. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.26

[72] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 291–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956546

[73] D. Taylor, J. Turner, J. Lockwood, T. Sproull, and D. Parlour, "Scalable ip lookup for internet routers," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 4, pp. 522 – 534, may 2003.

[74] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Labs, Tech. Rep. HPL-2008-20.

[75] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[76] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to fpga circuits," *Computer*, vol. 41, no. 7, pp. 40–46, July 2008.

[77] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," in *SIGCOMM '10*.

[78] G. Venkataramani, T. Chelcea, and S. C. Goldstein, "HLS support for unconstrained memory accesses," in *IEEE 14th International Workshop on Logic Synthesis (IWLS)*, Lake Arrowhead, CA, June 2005. [Online]. Available: http://www.cs.cmu.edu/~seth/papers/venkataramani-iwls05.pdf

[79] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 205–218. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736044

[80] W. Wang, T. K. Tan, J. Luo, Y. Fei, L. Shang, K. S. Vallerio, L. Zhong, A. Raghunathan, and N. K. Jha, "A comprehensive high-level synthesis system for control-flow intensive behaviors," in *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '03. New York, NY, USA: ACM, 2003, pp. 11–14. [Online]. Available: http://doi.acm.org/10.1145/764808.764812

[81] F. Zane, G. Narlikar, and A. Basu, "Coolcams: power-efficient tcams for forwarding engines," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 1, March 2003, pp. 42–52 vol.1.