Title	Code Optimization Methods for Configurable Processors
Author(s)	Hieda, Takuji
Citation	大阪大学, 2011, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/1622
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

Code Optimization Methods for Configurable Processors

January 2011

Takuji HIEDA

Code Optimization Methods for Configurable Processors

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2011

Takuji HIEDA

Publications

Journal Articles (Refereed)

[J1] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai: “Heuristic Instruction Scheduling Algorithm using Available Distance for Partial Forwarding Processor,” IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences, vol. E92, no.12, pp. 3258–3267, Dec., 2009.

International Conference Papers (Refereed)

[I1] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai: “Optimal Instruction Scheduling for Processors with Partial Forwarding using Integer Programming,” Proceedings of 13th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2006), pp. 274–279, Apr., 2006.

[I2] Hiroaki Tanaka, Yutaka Ota, Nobu Matsumoto, Takuji Hieda, Yoshinori Takeuchi, and Masaharu Imai: “A New Compilation Technique for SIMD Code Generation across Basic Block Boundaries,” Proceedings of 15th Asia and South Pacific Design Automation Conference (ASP-DAC) 2010, pp. 101–106, Jan., 2010.

International Conference Papers

[I3] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai: “Effectiveness of Partial Forwarding in Pipeline Processors,” Proceedings of Workshop

on Compiler Assisted SoC Assembly 2006 (CASA2006), Oct., 2006.

Domestic Conference Paper (Refereed)

[C1] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai, “Heuristic Instruction Scheduling Supporting Partial Forwarding Structure, ” Proceedings of 21st Circuits and System Workshop at Karuizawa, pp. 599–604, Apr., 2008 (in Japanese).

[C2] Keishi Sakanushi, Junya Okamoto, Takuji Hieda, Taichiro Imamura, Yoshinori Takeuchi, Junji Kitamichi and Masahari Imai: “Electronic Triage System for Disaster Medical Treatment, ” Proceedings of Embedded Systems Symposium (ESS2009), pp. 147–152, Oct., 2009.

Domestic Conference Paper

[D1] Taichiro Imamura, Keishi Sakanushi, Yasumasa Ode, Takuji Hieda, Junya Okamoto, Yoshinori Takeuchi, Masahari Imai, Hiroshi Tanaka and Teruo Higashino: “Evaluation of Electronic Triage Tag to obsurve injured person’s condition in Real Time, ” IPSJ Technical Report vol.2010-EMB-16, no. 5, pp. 1–8, Mar., 2010 (in Japanese).

[D2] Keishi Sakanushi, Akihito Hiromori, Taichiro Imamura, Junya Okamoto, Takuji Hieda, Yoshinori Takeuchi, Masaharu Imai, Junji Kitamichi, and Teruo Higashino: “Triage Device Slightly Injured Person in Disaster Medical Assistant Network, ” IEICE Technical Report VLD2009-37, Vol. 109, No. 201, pp. 45–50, Sep., 2009.

[D3] Hirofumi Iwato, Takuji Hieda, Hiroaki Tanaka, Jun Sato, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai: “A Highly Extensible Base Processor for Short-term ASIP Design, ” IPSJ Technical Report 2007-SLDM-132, vol. 2007, no. 114, pp. 133–138, Nov., 2007 (in Japanese).

[D4] Aiko Watanabe, Takuji Hieda, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai: “ Debug environment generation method for Software development in Hardware/Software

Co-design, ” Proceedings of DA symposium 2007, pp. 43–48, vol. 2007, No. 7, Aug., 2007 (in Japanese).

[D5] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai: “Heuristic Instruction Scheduling Method for Processors with Partial Forwarding Structure, ” IEICE Technical Report VLD2007-1, vol. 107, no. 31, pp. 7–12, May, 2007 (in Japanese).

[D6] Takuji Hieda, Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai: “Optimal Instruction Scheduling for Partial Forwarding Processors, ” Proceedings of DA symposium 2005, vol. 2005, No. 9, pp. 85–90, Aug., 2005 (in Japanese).

Summary

Configurable processor based development of embedded processor reduces cost of hardware design effort and development time of processors. However, development of software tools for configurable processors is still not an easy task. In order to achieve feasible time software development, programming in high level language is essential and compiler is one of indispensable tools for the target processor. In order to achieve performance of customized processors, compiler should generate assembly codes utilizing customized instructions of processors. When configurable processor is based on simple RISC processor, it is not so difficult to generate code for processors with basic instruction set. However, to achieve performance of customized processors, code generation technology to generate optimized codes is a very difficult task. Therefore, there is great need for code generation technique to generate optimized assembly code for customized configurable processors.

This thesis discusses two types of code optimization methods for configurable processors. The first one is a methodology for configurable processors with partial forwarding architectures. This thesis proposes two instruction scheduling methods for partial forwarding processors which has limited valuable forwarding paths for the target application. To utilize partial forwarding mechanism effectively, design space exploration of processors with forwarding structure is required, since optimal forwarding structure depends on the application of the target system. Fast optimized compiler which supports variable partial forwarding is required to find the optimal processor from all the candidates, because there are a lot of structures of forwarding datapaths on the target processor. This thesis proposes an optimal instruction scheduling algorithm and a fast heuristic instruction scheduling algorithm for partial forwarding processors by making use of the characteristics of the partial forwarding. Experimental results show that the proposed instruction schedulers generated efficient code for arbitrary partial forwarding

processors.

The second code optimization method is data order optimization for configurable processors for media applications. Programs of media application include a lot of data level parallelism and are executed on processors with SIMD instructions. Compilers for SIMD processor should have responsibility to exploit parallelism from sequential code without consideration of parallel data processing. In this thesis, media processors which have two types of instructions, SIMD instructions and data permutation instructions, are selected as target processors. SIMD instructions are instructions which operate on subword data in registers, and permutation instructions are instructions which reorder or repack data in registers. In this thesis, code optimization problem for media processors is discussed and optimization methods for data permutation instructions are proposed. Experimental results show the proposed methods reduces the number of permutation instructions compared to the conventional permutation instruction generation method.

Acknowledgements

I would like to deeply thank my supervisor, Prof. Masaharu Imai, for his guidance throughout my undergraduate and graduate researches. I am especially grateful for his useful advice which develops my expertise in the research area and academic mentality.

I would like to thank Prof. Kenichi Hagihara, Prof. Nagisa Ishiura and Prof. Yoshinori Takeuchi for a review of this thesis. They gave me useful discussion and comments on my study. Also, I would like to thank Prof. Onoye for giving useful comments in this thesis.

I would like to thank Prof. Yoshinori Takeuchi again. His comments greatly improved this thesis. Without his advice and support, I would not have continued my research activities. I am also grateful for his guidance, support and encouragement through everything in my student life.

I would like to thank Prof. Keishi Sakanushi for his guidance, giving comments on my research activities. He often gave me great inspirations about my researches in daily discussion.

I would like to thank Dr. Hiroaki Tanaka and Dr. Yuki Kobayashi for introducing me to this research area and guiding my early study. They gave me instruction in research activity, presentation skills, programming skills, etc.

I would like to thank Prof. Ittetsu Taniguchi and Mr. Takashi Hamabe. They also gave me useful advices for research activities and daily life in the laboratory.

I would like to thank Mr. Hirofumi Iwato, Mr. Takahiro Notsu, and Mr. Huynh Long Kim. They provided me good time through my days in Integrated System Design Laboratory in Osaka University. I always enjoyed all kinds of activities in the laboratory by them.

I thank Dr. Kyoko Ueda, Dr. Mohamed AbdElSalam Hassan, Mr. Takahiro Kumura, Ms. Yukako Nishikawa, Ms. Motoko Higashide, Ms. Asako Murakami, Mr. Noboru Yoneoka, Mr. Tatsuhiro Yoshimura, Ms. Aiko Watanabe, Mr. Taichiro Shiraishi, Mr. Junya Okamoto, and

other current and former members of Integrated System Design Laboratory in Osaka University. They gave me a lot of comments on my study. They also made my life enjoyable with their talks and acts.

I would like to thank collaborators through my doctoral research. I would like to thank Dr. Hiroaki Tanaka again, Mr. Nobu Matsumoto, Mr. Yutaka Ota and Mr. Hiroki Tagawa from Toshiba Corporation. They provided me discussion and comments on the study of code optimization for media processors. They also provided me tools and test codes for evaluation including compiler, simulator and so on. Corporation for giving discussion and comments on the study of the SIMD code optimization. They also helped me with implementation of the programs for evaluation.

Finally, I would like to deeply thank all my friends and family.

Contents

1 Introduction **1**

1.1 Configurable Processor 2

1.2 Compiler Support for Configurable Processors 3

1.2.1 Compiler Optimization for Configurable Processor 3

1.2.2 Data Path Customization 4

1.2.3 Architectures for Data Level Parallelism 5

1.3 Contribution of the Thesis 6

1.4 Overview of the Thesis 7

2 Configurable processor and Code Optimization **9**

2.1 Architecture Customization for Configurable Processor 9

2.1.1 Customizing Datapaths on Processor 10

2.2 Code Optimization for Configurable Processor 11

2.2.1 Parallelism Exploitation for Configurable Processor 12

3 Optimal Code Scheduling for Partial Forwarding Processors **15**

3.1 Partial Forwarding Architecture 15

3.1.1 Naive Pipeline Processor 15

3.1.2 Forwarding Processor and Partial Forwarding Processor 17

3.1.3 Characteristics of Partial Forwarding 19

3.2 Treatment of Partial Forwarding in Instruction Scheduler 21

3.2.1 Architecture Model of Partial Forwarding 21

3.2.2 Available Distance 23

3.2.3 Complexity of instruction scheduling for partial forwarding processor . 25
3.3 ILP Formulation of Instruction Scheduling Problem for Partial Forwarding Processor . . . 25
3.3.1 Preconditions . . . 25
3.3.2 Scheduling Method . . . 26
3.3.3 DDG Simplification . . . 28
3.3.4 Calculating Upper Bound of the Total Execution Cycles . . . 28
3.3.5 Solving ILP . . . 28
3.4 Experiments . . . 30
3.4.1 Environments . . . 30
3.4.2 Code Size . . . 33
3.4.3 Execution Cycles and Times . . . 33
3.5 Summary . . . 35

4 Heuristic Code Scheduling for Partial Forwarding Processors 39

4.1 Instruction Scheduling Algorithm for Partial Forwarding Processor . . . 39
4.1.1 Detail of the Scheduling Algorithm . . . 39
4.1.2 Complexity Order of the Scheduling Algorithm . . . 44
4.2 Experiments . . . 44
4.2.1 Specification of processors . . . 45
4.2.2 Evaluation of the proposed scheduling algorithm . . . 45
4.2.3 Comparison with ILP method . . . 46
4.2.4 Comparison with hazard detection unit . . . 48
4.3 Summary . . . 50

5 Code Optimization for SIMD Instruction-set Processors 57

5.1 SIMD Instructions in Embedded Processor . . . 57
5.2 Code Generation for SIMD Processor . . . 59
5.2.1 Grouping SIMD Operations . . . 60
5.2.2 Data Permutation Ordering . . . 61
5.2.3 Data Permutation Instruction Sequence Generation . . . 62

5.3 Data Permutation Optimization in SIMD Registers 63
5.3.1 Data Permutation Optimization Problem 64
5.3.2 Data Permutation Optimization Method 65
5.3.3 Data Order Propagation from SIMD Load Nodes and Store Nodes . . . 67
5.3.4 Data Order Propagation from Neighbor Nodes 69
5.4 Experiments 72
5.4.1 Target Processor Architecture 72
5.4.2 Experimental Results 75
5.5 Summary 77

6 Conclusion and Future Work 83

6.1 Conclusion 83
6.2 Future Work 84
6.2.1 Algorithm Expansion 84
6.2.2 Theorical analyzation of the heuristic algorithms 85
6.2.3 Compilation Techniques for Low Power 85

List of Figures

1.1 Advantage of ASIPs. 4

3.1 DLX processor which has five pipeline stages 17
3.2 DLX processor with full forwarding unit 18
3.3 DLX processor with partial forwarding from only the third stage 19
3.4 An example of partial forwarding DLX processor with a forwarding path for only one of the ALU input ports 21
3.5 Pipeline of the sample processor with partial forwarding 22
3.6 Sample code for sample processor 25
3.7 DDG of the sample code on the sample processor 37
3.8 DDG construction algorithm 38

4.1 An example DDG for the scheduling algorithm 42
4.2 Example for scheduling problem about partial forwarding 43
4.3 The proposed scheduling algorithm 52
4.4 Trade-off graph of sieve 53
4.5 Trade-off graph of mat1x3 53
4.6 Trade-off graph of inssort 54
4.7 Trade-off graph of crc16 54
4.8 Trade-off graph of fir2dim 55

5.1 Example of datapath of ADD2, a SIMD instruction in TI C62xx. 58
5.2 Example of dataflow of permutation instructions. 59
5.3 Automatic SIMD code generation flow. 59

5.4 Operation grouping 61
5.5 MDDs for { abcd } and { abcd,abdc } 63
5.6 An example of SIMD DFG 64
5.7 Examples of partial propagation 66
5.8 Example of spreading propagation 66
5.9 Composition of partial and spreading propagations 67
5.10 Operation ordering 67
5.11 Example of propagation from SIMD Load/Store Nodes 68
5.12 Propagation from Load_A 69
5.13 Propagation from Store_D 70
5.14 Result of the propagation from Load/Store nodes 71
5.15 Algorithm of data order propagation from SIMD Load/Store nodes 72
5.16 permutation adjustment on a junction node 73
5.17 An example of permutation adjustment on branch paths 74
5.18 Data permutation adjustment method inside of the loop 75
5.19 Algorithm of data order propagation from neighbor nodes 78
5.20 MeP processor architecture 79
5.21 SIMD coprocessor architecture 80
5.22 Patterns of fixed permutation for 4-parallel halfword operations 81
5.23 Arbitrary permutation instruction 81
5.24 Comparison of the number of execution cycles with two strategies 81

List of Tables

3.1 Available distance of DLXs 31
3.2 Specifications of DLXs 32
3.3 Improvement Ratio of code size compared ILP with list scheduler 34
3.4 Improvement Ratio of execution cycles compared ILP with list scheduler . . . 35
3.5 Execution time of the programs on each DLX processor 36
3.6 Scheduling time using ILP [sec.] 36

4.1 Total, average, and maximum sizes of basic block for each program in DSPstone 45
4.2 Comparison of execution cycles between proposed heuristic and ILP methods . 47
4.3 Comparison of execution time between proposed heuristic and ILP methods . . 48
4.4 Comparison of compilation time between proposed heuristic and ILP methods . 49
4.5 Execution time for each benchmark program with the proposed scheduler . . . 50
4.6 Execution cycle for each benchmark program with the proposed scheduler . . . 51
4.7 Execution cycle for each benchmark program with hazard detection unit 55
4.8 Differences of execution cycles on processor with hazard detection unit from the result of scheduled code 56

5.1 Comparison of the number of permutation instructions with two strategies . . . 76

Chapter 1

Introduction

In modern society, there are an enormous number of microprocessors in our daily life. Integration scale of microprocessors grows as processing technologies for semiconductor products evolved according to Moore's law [1]. This progress also contributes to shrink size of microprocessors and reduce market price of them. Nowadays microprocessors exists everywhere, not only large information appliances such as information infrastructure systems and personal computer but portable platforms such as cell phones and personal digital assistants. Moreover, microprocessors are also used in various fields where the people do not realize. Modern home appliances such as 3-D TV, air-conditioner, and video players have microprocessors to operate their functions. Up-to-date automobiles are equipped with electronic vehicle control systems implemented by microprocessors such as throttle and transmission controllers and collision avoidance systems. These kinds of electronic systems which implements dedicated functions by microprocessors inside of the products are called *embedded systems* and such microprocessors are called *embedded processors*. Embedded processors are expected to satisfy tight design constraints for the application dedicated purpose. For example, power consumption of the embedded systems must be low for mobile systems which depend on battery as power resource. Product cost and size of embedded processors for mass production are also an important aspect.

As modern embedded processors have improved their performance, complexity of the semiconductor systems explodes even though in embedded systems. Function requirements for embedded systems also become more complex so that the scale of embedded software systems which runs on embedded processors is also explodes. Ubiquitous computing is one of the huge

scale systems related to embedded processors. Hundreds or thousand of embedded processors orchestrate for the same dedicated application through wireless network such as information home appliances, intelligent transport systems for automobiles, and sensor area network. In contrast, product development cycle for embedded systems become shorter although size of the systems inflates. Although improvement of processing technology expands application area and functions for embedded systems, it shortens the life range of existence embedded systems and raises development cost of embedded systems. To satisfy both requirements, flexibility is key feature for development of embedded system.

1.1 Configurable Processor

Once processing technology was not mature so much, embedded system designers have two design options to develop functions for target specific application; one is software implementation which run on general purpose processor (GPP), the other is application specific integrated circuit (ASIC) which is integrated circuit dedicated for functions of the target specific application. If the functions do not require execution speed to satisfy timing deadline, software implementation on GPP is sufficient. This solution is sufficient since the cost of massively product GPP is reasonable. Besides, developer can implement the functions in high level language such as C so that the developed functions have portability, flexibility, and re-usability. If the execution speed by GPP is inadequate to satisfy severe timing constraint which is required for the functions, the designer had to choose ASIC, hardware implementation for the functions, in past days. ASIC accomplishes the functions with extremely faster speed than the solution by GPP. However, implementation cost of ASIC is extremely expensive, especially the planned amount of the production is not much, and the specification cannot be changed even though crucial problem is found. Also, reusing ASIC for another products is almost impossible since it is dedicated hardware for the functions of the target product. In recent embedded system design, these two design options is not sufficient due to huge scale of the applications, variety of the application domains, highly expensive manufacture cost of the semiconductor products, and rapid production cycle, as discussed at the beginning of this chapter.

Configurable processor is introduced to complement the gap between GPP and ASIC [2].

Configurable processor whose architecture is customizable is one of the solutions to improve design productivity. Configurable processor has some customizable parameters such as the bit width of data path, the number of general purpose registers, and additional functional units and instructions, based on basic instruction-set architecture. The designers implements additional functional units for dedicated application domain such as multimedia processing to satisfy performance requirements and design constraints. Configurable processor also has flexibility against modifying specification of the target systems since basic instruction-set is often constructed as simple general purpose processor and Such a customized processor for dedicated application domains are called application specific instruction set processor (ASIP.) Configurable processor is often employed to design ASIP. Designing embedded processor or ASIP based on configurable processor reduces cost of the design effort and development time since the designers can concentrate on architecture optimization. As a result, the customized processor can be desirable for the target application. Figure 1.1 shows the trade-off of design solutions between performance/power consumption and cost/flexibility. ASIPs stand on the intermediate position between ASICs and GPPs. Though ASIPs are considered desirable solution for embedded systems, appropriate design efforts, both hardware customization and software optimization, are required to achieve performance improvement of embedded systems.

1.2 Compiler Support for Configurable Processors

Configurable processors can be customized their hardware structure to suit for dedicated embedded systems. Compiler for configurable processors must understand customized architectures to generate correct and effective machine code. Electronic design automation (EDA) tools with configurable processors usually have compilers to adapt customized processors. Such compilers are called *Retargetable Compiler* [3, 4].

1.2.1 Compiler Optimization for Configurable Processor

Code optimization flow in compiler can be split into two parts; machine-independent optimization and machine-specific optimization. Machine-independent optimization methods are

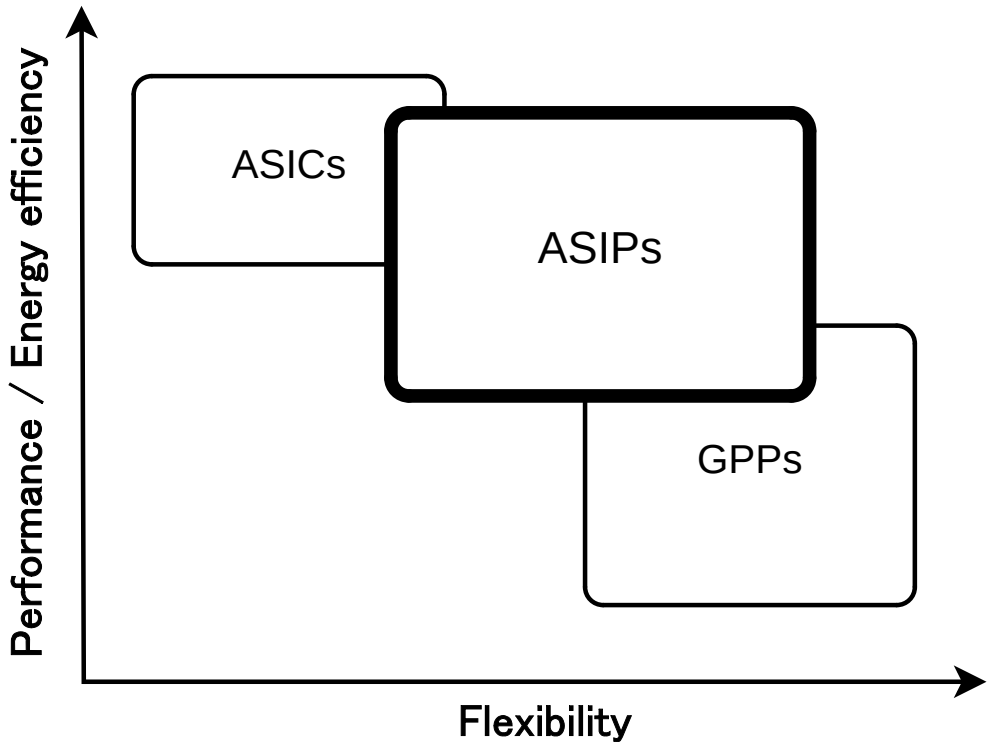


Figure 1.1: Advantage of ASIPs.

applied in the form of intermediate representation, which are used the inside of the compiler, before code generation.

In compiler back-end for the target processor, machine-specific optimization are applied. There are three major parts: code generation for the target processors, instruction scheduling, and register allocation. Retargetable compilers are required to adapt these optimization phases according to customization results. The next two subsections describe the target of the retargetable optimization problems in this thesis.

1.2.2 Data Path Customization

Even though a processor in target embedded system is a single issue RISC processor, the design cost of the processor have to be reduced if a processor violates design constraint of the system. In such a case, partial forwarding is a way to reduce datapaths in the processor to suppress size of the processor [5]. Partial forwarding can reduce cost of the processor and may improve execution time of the program by removing certain forwarding paths which are not inefficient.

Partial forwarding processor which has only valuable forwarding paths for the target application achieves both performance design constraints. However, design space exploration about forwarding structure is required to use partial forwarding since optimal forwarding structure is depend on the application of the system. Structure of forwarding datapaths on the target processor varies so fast optimization compiler which supports variable partial forwarding should be desirable to find the optimal processor from all the candidates.

Partial forwarding can decrease hardware cost and still keep execution performance to remove forwarding datapaths that are not used frequently. It is important to explore design space of forwarding for high performance system design, in particular, for embedded systems that have no margin for forwarding at all possible points between the functional units. However, suitable instruction scheduler which takes account of partial forwarding is required to execute application programs with high performance.

1.2.3 Architectures for Data Level Parallelism

Digital signal processing (DSP) is one of the most important task for embedded processors since embedded systems often engage in signal processing systems such as media encoding/decoding. There are many processors which are named *DSP processor*. DSP processor has special instruction set to process huge amounts of signal data in a short time. Single-Instruction Multi-Data (SIMD) instruction set is one of the major instructions for DSP SIMD architecture consists of parallel functional units so that SIMD instruction process several data at a time. If a processor has 4-way SIMD instructions, a program can be executed four times faster than a processor without the SIMD instructions in ideal. However, practical programs have sequential codes so that ideal performance cannot be obtained. To improve the opportunity of parallel execution, programmers have to consider parallelism during development with mature skills of parallel programming.

Recently, automated SIMD instruction generation methods are proposed [6, 7, 8]. These method generates SIMD instructions in assembly code by compilers so that programmers can develop their programs without consideration of parallelism. However, there are some optimization problems which are difficult to solve by compilers. Permutation optimization problem

is one of such problem. When SIMD instructions are executed, each operand register contains a number of data. According to the position of data in registers, SIMD instructions calculate their operations. If position of data in registers differs from the context of the program, it is required to change data order in registers to execute the program correctly.

1.3 Contribution of the Thesis

This thesis discusses code optimization method for configurable processors and application domain specific instruction set processors.

The main contributions of this thesis are as follows. The first contribution is on instruction scheduling method for partial forwarding processors.

- The optimal instruction scheduling method for partial forwarding processors is proposed. Characteristics of partial forwarding from the view of compiler are studied. For given partial forwarding architecture, the proposed method solves scheduling problem converted into 0-1 integer linear programming problem which reflects the characteristics of the target partial forwarding processor.

- A heuristic instruction scheduling method for partial forwarding processors is proposed. For given partial forwarding architecture, the proposed heuristics scheduler optimizes the order of instructions to improves the rate of operation for each forwarding paths.

- With these instruction scheduling algorithms, effectiveness of design space exploration for partial forwarding is evaluated. Both the proposed scheduling methods supports arbitrary partial forwarding structures so that designer can configure forwarding datapaths to suit for the target application. Experimental results showed that the performances of some partial forwarding processors are superior to the existence full forwarding processor.

The second contribution is on data permutation optimization method for SIMD instructions.

- Data permutation optimization problem for SIMD registers is discussed and a heuristic data permutation optimization method for SIMD registers in proposed. The proposed

optimization method supports general input programs which are not considered data level parallelism.

- Experimental results show that the proposed optimization method improves the number of permutation instructions and execution cycles.

1.4 Overview of the Thesis

The rest of this thesis is organized as follows: chapter 2 discusses about configurable processors and compiler optimization methods. In chapter 3, partial forwarding architecture is summarized and optimal instruction scheduling method for partial forwarding processor is proposed. Then, chapter 4 proposes heuristic instruction scheduling method for partial forwarding processor. In chapter 5, data permutation optimization in SIMD registers for SIMD instructions is described. Finally, chapter 6 concludes this thesis and summarizes future works.

Chapter 2

Configurable processor and Code Optimization

This chapter discusses configurable processors and code optimization methods for configurable processors.

2.1 Architecture Customization for Configurable Processor

One of the advantage of configurable processors is that processor designer can customize to satisfy tight design constraints. Xtensa [2], a pioneer of configurable processor, is based on RISC processor and embedded system designer can change the number of registers and append customize instruction to write behavior of the instruction. Other properties, such as instruction bit width and pipeline stages, are fixed.

ASIP meister [9] is an EDA tool which generates processor HDL (Hardware Description Language) and software development tools for the processor. To describe micro operation description, HDL description of the processor is obtained. On the other hand, to describe behavior description, compiler for the processor based on DLX processor [10] can be obtained [11]. ASIP meister also provides Brownie [12], a RISC configurable base processor, with GCC compiler toolchain supports.

LISA [13] is an architecture description language to design processor and simulator. The processor designers can be developed VLIW or superscalar processors to devote much effort to the development. Processor Designer [14] provided by Coware cooperates with LISA processor. This tool generates compiler by nML [15].

2.1.1 Customizing Datapaths on Processor

Abnous and Bagherzadeh analyze performance effect of limited datapaths for VIPER VLIW processor [16] which has four ALUs and one global register file [17]. They show that limited interconnect datapaths among ALUs could achieve higher performance than fully-connected datapaths. They solved read-after-write (RAW) data hazards by stalling pipeline and did not focus on scheduling in their study, although they schedule instructions to reduce hazards. For similar architecture, Buss, et al. propose a bypass wiring method by which the most communicating interconnection datapaths were wired actually [18]. An arranged list scheduler of IMPACT [19] retargetable compiler was used for compaction in their work. Brown and Patt also examined partial forwarding for their adders that use redundant binary representation [20].

Sassone, et al. propose broadcast forwarding for out-of-order superscalar processor to forward the results in several cycles [21]. Although the number of forwarding paths can be reduced with decreasing instructions per cycle, performance improvement by increasing processor frequency is superior. In embedded systems with strict constraints, selecting superscalar processor may be difficult.

Ahuja, et al. study partial forwarding for MIPS-like single-issue pipelined processors [5]. They measured down rate of execution performance with two types of partial forwarding, symmetric and asymmetric forwarding. A processor with the latter have to care about the order of operands for each instruction because forwarding paths may be connected with a part of inputs of a functional unit, while the former do not have to. They concluded that partial forwarding could reduce hardware cost with just a few percent performance losses. They used safety scheduling method; an instruction will be scheduled only it can use the forwarded result at the current cycle.

M. Kudler, et al. develop instruction scheduling algorithm for partial forwarding [22].

FLASH, which is acronym for ”Foresighted Latency-Aware Scheduling Heuristic,” is improved version of what they have been used in the study of design space exploration for partial forwarding [23]. FLASH based on exhaustive scheduling which checks all candidates of scheduled instruction sequences. To reduce number of candidates, FLASH limits search depth and uses some assumption for simplification.

A. Shrivastava, et al. insist that instruction scheduler based on Operation Table [24], which represents instruction data flow in each stage, can achieve better scheduling for partial forwarding processor than that based on reservation table which is commonly used in many compilers to avoid structure hazards. Operation Table is also used for design space exploration for partial forwarding [25, 26, 27]. They did not focus on scheduling algorithm but hazard detection, and there are no discussions about optimization for partial forwarding. Jayaseelan, et al. propose forwarding customization method to improve usability of instruction set extensions [28].

Kimura, et al. propose Procyon processor which takes other approach similar to partial forwarding to optimize program [29]. Procyon has full forwarding paths which can be activated or inactivated by control instructions. Each forwarding path is activated or inactivated to use old register value after an instruction which will change the contents of the register is issued.

Shoji and Tian, et al. study for forwarding architecture implemented by bus interconnection on VLIW processor [30, 31]. All processing elements (PEs) are connected by bus for global forwarding to spread local forwarding value into other PEs. Compared to forwarding structure which connects for each pair of PEs, the number of forwarding datapaths is reduced and implementation facility is improved, instead of bus occupation constraint which means only one PE can broadcast forwarding value at a cycle. They also propose a power-consumption-aware instruction scheduling method for the bus forwarding processor to reduce the number of register accesses.

2.2 Code Optimization for Configurable Processor

Code optimization is one of the major compiler optimization problem so that many code generation and optimization methods for many kinds of processors have been studied [32, 33, 34, 35].

2.2.1 Parallelism Exploitation for Configurable Processor

There are several approaches to generate and optimize assembly code or programs [33, 36]. Target dependent transformation on intermediate representation of the input programs is major optimization method. There are many machine-independent optimization techniques based on intermediate representation so that this method is widely used in practical compiler. This approach is also applied for machine-specific optimization.

Another approach to optimize programs and assembly code is generation of high quality assembly code in assembly code generation phase [35]. As described so far, there are three major tasks in compiler back-end; code generation, instruction scheduling and register allocation. These tasks are the main focus of this thesis.

Like as automatic SIMD code generation, there are many studies on SIMD code optimization [37, 38, 7, 39], SIMD instructions have fixed length registers for their operands so that SIMD load/store operations also access registers according to data alignment of the SIMD register size, thus shifting is also important problem during optimization.

Leupers studies SIMD code generation problem in early stage [37]. In this study, SIMD code generation for TI 62xx processor is proposed by integer programming.

Franchetti, et al. proposes an efficient SIMD code generation technique for numeric operation [38]. This technique focuses on optimization for the output numerical kernels, including matrix arithmetic libraries and DSP algorithms such as FFT, generated by automatic performance tuning systems.

In [6], generation method of SIMD and *permutation* instructions is presented. SIMD instructions are generated by grouping operations in a basic block represented by Data Flow Graph (DFG). After the grouping, permutation instructions are inserted between SIMD instructions. Ren, et al. proposes an optimization technique to minimize permutation instructions in basic block [7]. By propagating permutations across statements and merging consecutive permutations, the number of permutations can be reduced. Suzuki, et al. also propose similar automatic SIMD instruction generation method [40] for COINS compiler infrastructure [41, 40].

[39] solves SIMD alignment problem using MIN-CUT/MAX-FLOW algorithm in some special cases. This paper insists that the cases supported by the algorithm often appears in practical

code. Both [7] and [39] show that the problems are reducted into the multiway cut problem which is an NP-hard problem [42].

Chapter 3

Optimal Code Scheduling for Partial Forwarding Processors

This chapter describes optimal instruction scheduling method for partial forwarding processors. This chapter is organized as follows. Partial Forwarding Architecture is summarized in section 3.1 and how to treat characteristics of the partial forwarding architecture in code scheduler is summarized in section 3.2. The way to convert scheduling problem into integer linear programming (ILP) problem is presented in section 3.3. Experimental results are described in section 3.4. Finally, this chapter is concluded in section 3.5.

3.1 Partial Forwarding Architecture

In this section, partial forwarding architecture [5] in processor pipeline is summarized.

3.1.1 Naive Pipeline Processor

In typical pipeline processor, a pipeline stage to read content from a register file is different from that to write result into a register file. Figure 3.1 shows pipeline structure of DLX processor [43]. IM and DM mean instruction memory and data memory, respectively, and Reg represents register file. This typical RISC processor has five pipeline stages. Role of each stage is summarized below;

- Instruction Fetch (IF) stage.
- Instruction Decode (ID) stage.
- instruction Execution (EX) stage.
- Memory Access (MA) stage.
- register Write Back (WB) stage.

ALU at EX stage in the DLX processor has two input ports and one output port. During decoding instruction in ID stage, the decoder resolves operand registers and read values from the registers according to input operands. execution result of ALU in EX stage and data loaded from data memory in MA stage are written in an output register at WB stage which is three stages after ID stage so that the result of an instruction become available four cycles after the instruction is fetched on this basic DLX processor. Program code which runs on this processor must satisfy latency constraint between instructions caused by the gap of register access timing between read and write, otherwise the processor will read an input value from the register in which EX or MA stages generate an output value, which changes semantics of the program code.

There are two solutions, hardware solution and software solution, to avoid violation of the latency constraint which is called hazard or data hazard [43]. The hardware solution is using hazard detection unit. Hazard detection unit is one of popular solution against data hazard to stall pipeline until execution result is written back to register file [43]. Hazard detection unit resolves the problem without modifying on the input instruction sequence in exchange for performance loss of the processor due to stalling which just consumes execution cycles. On the other hand, instruction scheduling which is the software solution resolves hazards. Instruction scheduling is one of the compiler back-end tasks to reorder instructions in input code [32]. Instruction scheduler changes the order of instructions in compilation time to optimize total execution cycles of the instructions. Between a couple of instructions which have dependence between them, instructions which are independent from them are inserted during this optimization process. If there are no or not enough independent instructions, No-OPeration (NOP) instructions are inserted for waiting. NOP does nothing but just consumes execution cycles,

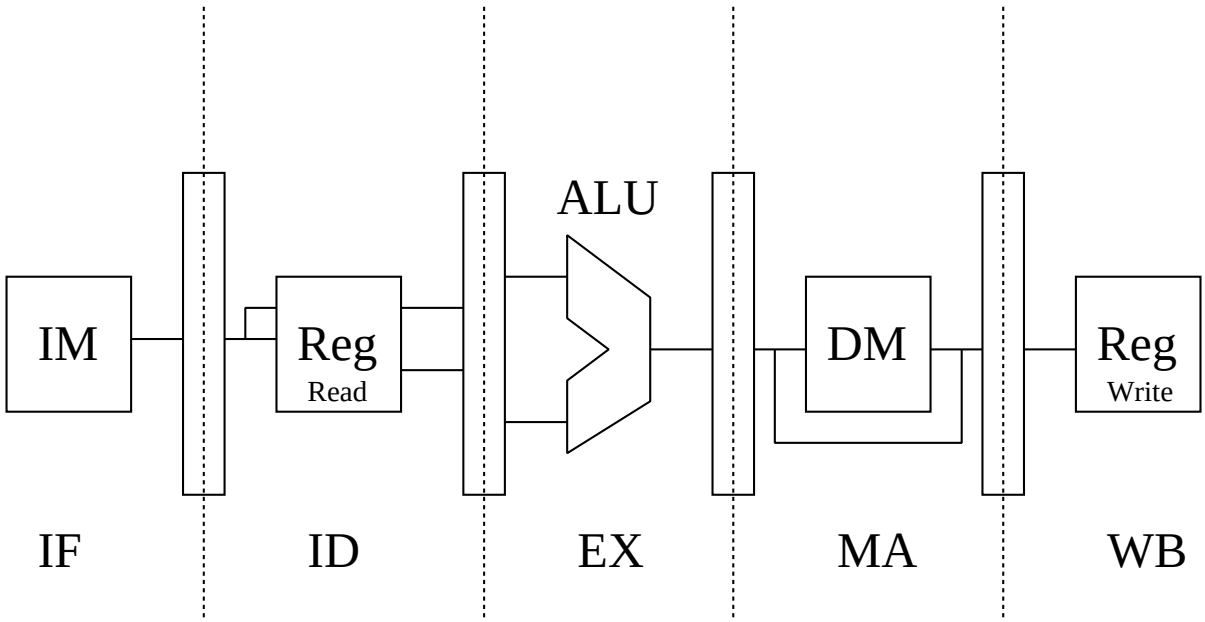


Figure 3.1: DLX processor which has five pipeline stages

one cycle in almost processors. Although it is the same result as the hardware solution if only NOP instructions are inserted, execution performance of the instruction sequence improves in most cases.

3.1.2 Forwarding Processor and Partial Forwarding Processor

As described above, naive pipeline processor has to wait for several cycles to finish writing the computation result of an instruction. This delay caused by data hazard increases execution cycles of the programs especially when the instruction sequence has long critical path. Forwarding, an processor architecture to dissolve this delay, is used to improve processor performance [43]. Forwarding unit consists of datapaths, multiplexer, and forwarding control logic. Each forwarding datapath with the processor pipeline sends computational result from a functional unit into pipeline stages to read register in which the result is written. In a read stage, the forwarding unit of the processor reads the result from one of the forwarding datapaths if the fetched instruction requires the result. Figure 3.2 shows DLX processor with forwarding unit. Each forwarding datapath is connected between ID stage and pipeline register after EX stages. Forwarding unit with decoder verifies whether input operands contain registers whose latest

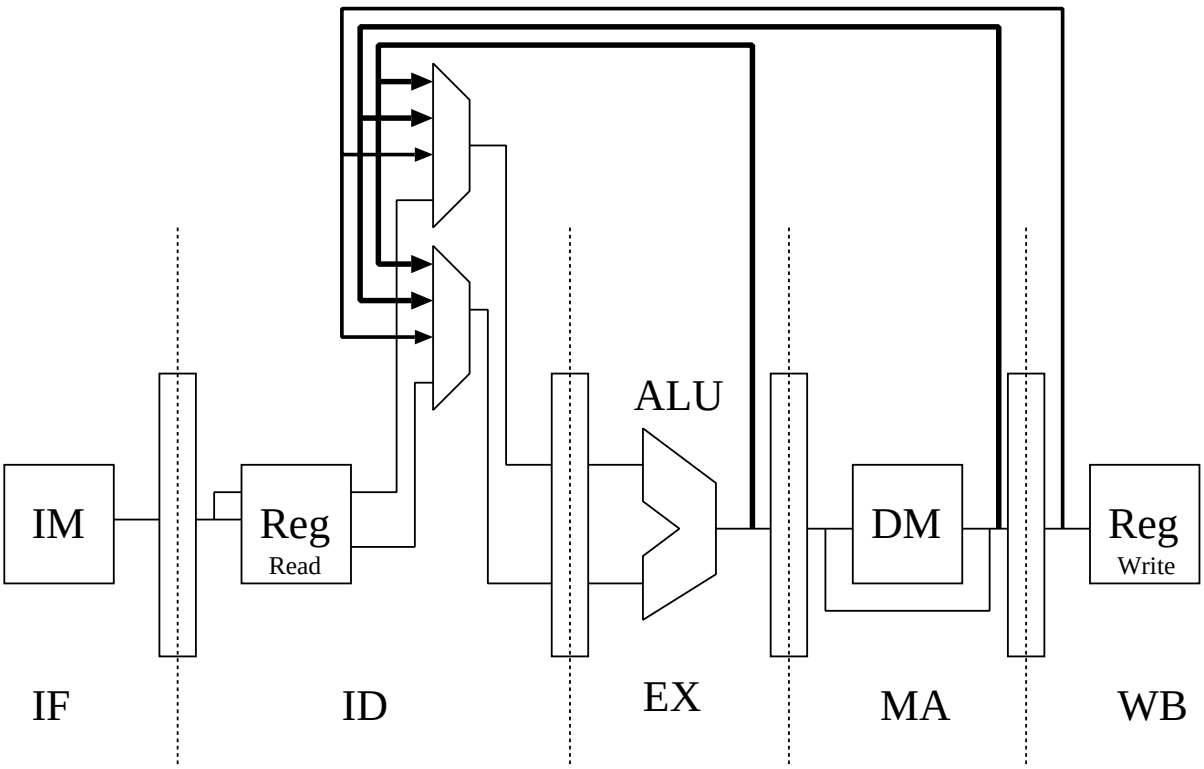


Figure 3.2: DLX processor with full forwarding unit

values are yet in pipeline registers. If the forwarding unit found the latest values, the decoder read input values from forwarding datapaths instead of the register file so that the decoded instruction make use of the latest value of the input registers. On the processor shown in Fig. 3.2, one cycle is enough to use results of the preceded instructions.

In general, processor area which is occupied by forwarding unit is directly proportional to the number of pipeline stages and quadratically proportional to the number of issue slots [5]. Embedded processor is also demanded to equip low power consumption and small processor area to satisfy design constraints by the systems. Furthermore, forwarding unit also may increase the critical path delay of the processor to extend the length of the critical path even though the processor is a single-issue RISC processor. *Partial forwarding*, also called incomplete bypassing [5] or partial bypassing [24], is a forwarding which datapaths for forwarding are not at all possible points in the processor. Figure 3.3 is an example of partial forwarding implemented on

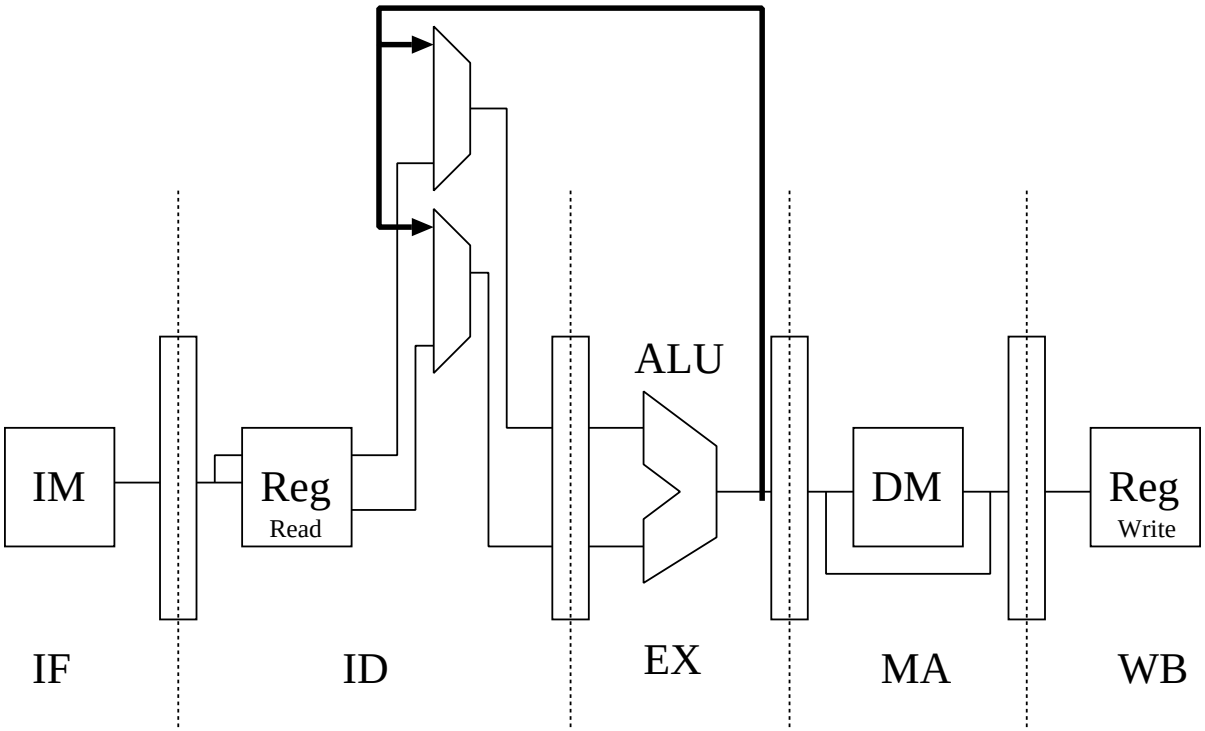


Figure 3.3: DLX processor with partial forwarding from only the third stage

DLX processor. Compared with DLX processor which has forwarding paths from all effective pipeline registers as Fig. 3.2, the forwarding datapath from the fourth stage to the second stage is removed on DLX processor in Fig. 3.3. In contrast, processor with forwarding unit which has all possible forwarding datapaths like Fig. 3.2 is called *full forwarding*. Partial forwarding decreases the number of forwarding datapaths and reduces hardware cost which arises from a multiplexer and the control logic.

3.1.3 Characteristics of Partial Forwarding

Partial forwarding processor has distinct property of issue timing of instruction from full forwarding processor; an execution result of an instruction may not be always available although it was available one cycle before. If an instruction is issued on a processor without forwarding like Fig. 3.1, the result of the instruction can be used when it is written back to a register. After that, it can be used on any following instructions until a new value is overwritten back to the register. In the case of full forwarding processor like Fig. 3.2, the result can be used after the

next cycle through forwarding datapaths. In both cases, there are common property that the computational result of the functional unit can be used until the register is overwritten when it become available at once. On the other hand, partial forwarding processor may not satisfy this property. Consider an instruction of which following three instructions use the result is issued on the DLX processor with partial forwarding datapaths illustrated in Fig. 3.3. The first instruction produces the result when it is processed at the third stage. At the next cycle, the second instruction is able to obtain the result of the previous instruction through the forwarding datapath, and it is executed correctly. Then, the third instruction also tries to get the latest execution result from the first instruction, but it cannot be used since the result is in the fourth stage and there are no forwarding paths. As a result, the functional unit reads the register which has old register value. For this reason, instruction sequence for a processor with full forwarding datapaths cannot be used on the processor which changes its partial forwarding datapaths. The last instruction executed at after one cycle is executed correctly because it can read the desired value from the forwarding path from the fifth stage.

3.1.3.1 Hazard Detection for Partial Forwarding Processor

Hazard detection unit enables to execute instructions without instruction scheduling even though on partial forwarding processor. However, hazard detection unit without instruction scheduling is not enough to dissolve the disadvantage of partial forwarding. When pipeline stall occurs, following instructions have to wait their issue until the pipeline resolves the hazard. Due to this delay, the total number of execution cycles becomes larger in almost cases. This problem is also discussed in [5].

3.1.3.2 Effect of the Operand Position by Partial Forwarding

The ALU in Fig. 3.3 has two input ports. In the case of Fig. 3.3, there are forwarding datapaths into the input ports if a pipeline register is connected with forwarding unit and vice versa; in other words, the form of forwarding datapath is *symmetrical*. It means that the performance of partial forwarding are independent from the ALU ports. This property is not required, although it is desired for application developers. Figure 3.4 shows an example of DLX processor with such a partial forwarding datapaths. In this processor, forwarding datapaths are connected

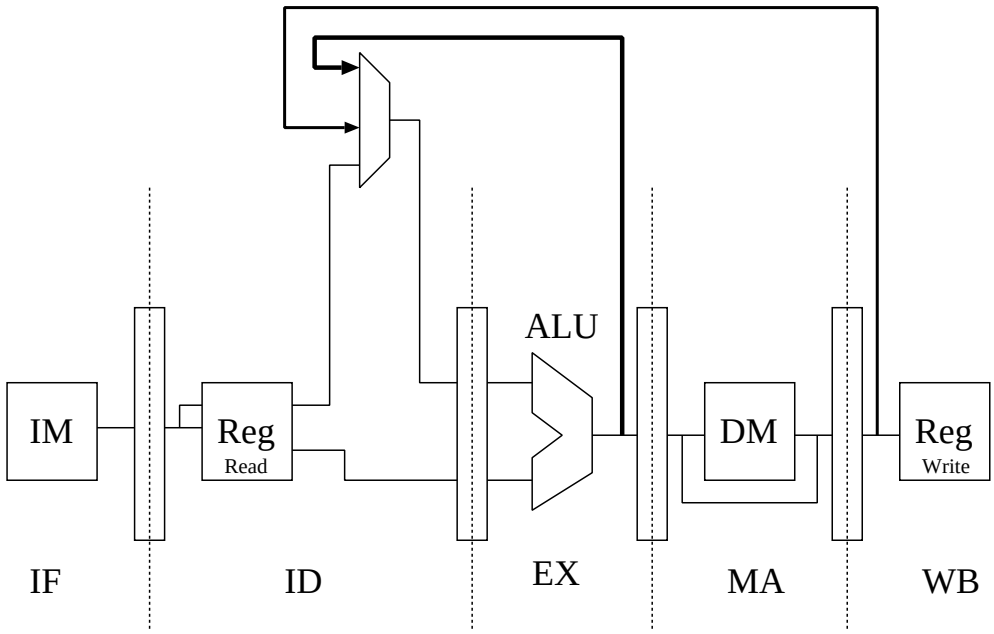


Figure 3.4: An example of partial forwarding DLX processor with a forwarding path for only one of the ALU input ports

into only one of input ports of the ALU. In other words, the form of forwarding datapath is *asymmetrical*. The decoder analyzes fetched instruction and assigns ALU input ports for input operands so that the performance of partial forwarding depends on the ALU ports. For application developers for the processor, this difference appears in the form of difference of the latency constraint of the instruction set architecture.

3.2 Treatment of Partial Forwarding in Instruction Scheduler

In this section, the way to treat partial forwarding in instruction scheduler is described.

3.2.1 Architecture Model of Partial Forwarding

Partial forwarding is implemented on pipeline processors. The proposed method assumes that the target processor has only one pipeline. The proposed method also assumes that register

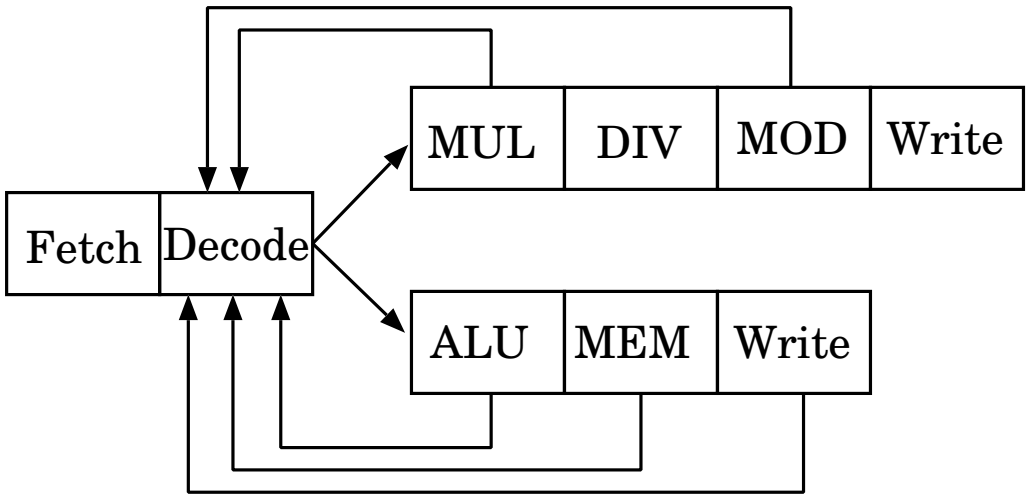


Figure 3.5: Pipeline of the sample processor with partial forwarding

value is read at the decode stage in early stage of the pipeline and written into the destination register at the write-back stage in the end stage of the pipeline like as DLX processors of Fig. 3.3.

However, the proposed method support for processor with more than two pipelines if instruction scheduler can regard pipeline structure of the target processor as one pipeline. The processor shown in Fig. 3.5 has two pipelines; The upper pipeline has forwarding paths from stages 3 and 5 and the lower pipeline has forwarding paths from stages 3, 4, and 5. The processor decodes fetched instruction and reads register file at the second stage. The decoder chooses the destination pipeline according to the type of the fetched instructions. Only instructions for multiplication and division are passed into the upper pipeline in Fig. 3.5. In this case, the proposed method is applicable.

The proposed method schedules instructions in compiler so that the order of the input instruction sequence is unchangeable. Therefore, the proposed method assumes the target processor is an in-order issue processor since out-of-order processor may change the execution order of the fetched instructions.

Avoiding pipeline hazard is one of the important role in instruction scheduler for pipeline processor. In general, there are three type of hazards in instruction sequence; read after write (RAW), write after read (WAR), and write after write (WAW) hazards. RAW hazard occurs

when an instruction use the result of a former instruction before it is written into the destination register. RAW hazard is unavoidable hazard in any pipeline processor so that dependence between a pair of instructions with RAW hazard is called true dependence. WAR hazard occurs when an instruction writes the result into the destination register before the register is accessed to read the old value by former instructions. WAR hazard can be avoided to change the destination register of the writing instruction so that dependence between a pair of instructions with WAR hazard is called false dependence. WAW hazard is similar to WAR hazard; it occurs when an instruction writes the result into the destination register before the register is accessed to write the old result by former instructions. WAW hazard also can be avoided to change the destination register of the writing instruction as WAR hazard and dependence between a pair of instructions with WAW hazard is called output dependence. However, after register allocation, both WAR and WAW hazards are also real hazards so that instruction scheduler must avoid them.

3.2.2 Available Distance

When program runs on partial forwarding processor without hazard detection unit, instruction sequence of the programs must be scheduled by compiler to avoid any hazard to drive the program correctly, otherwise the functional units which use the result of preceding instruction read old and wrong value from register file described in section 3.1. Due to the characteristics of partial forwarding as described in section 3.2, conventional scheduling algorithms which do not consider partial forwarding cannot optimize programs for partial forwarding processor. The property of partial forwarding demands consideration of partial forwarding from instruction scheduler.

A conventional instruction scheduler cannot utilize partial forwarding datapaths enough since it uses a single value, instruction latency, to represent the number of cycles which are required that the execution result becomes available. To solve instruction scheduling problem, analyzing dependence among instructions in an instruction sequence is required. The analyzing result is represented by means of a Data Dependence Graph (DDG) [36]. DDG is the base of instruction scheduling. In the case of Fig. 3.3, an instruction scheduler must avoid generating

scheduling result which contains an instruction which uses the result of an instruction scheduler at two cycles before. To satisfy this constraint, a single latency value is inadequate. *Available distance* is defined to adapt partial forwarding to instruction scheduler. An available distance is represented as (D, l). D is a set of natural numbers. Each element in D represents the distance at which the execution result of the predecessor is available. The successor can use the execution result of the predecessor if the instruction distance between them equals to an element in D. On the other hand, l is a natural number which is bigger than any elements in D. l denotes the minimum distance that the following instruction can use the preceding instruction's result without data hazard. The successor can use the execution result of the predecessor if the instruction distance between them is equals to or bigger than l.

For example, Fig. 3.7 shows the DDG of the instruction sequence represented in Fig. 3.6 on the processor in Fig. 3.5. LOAD, ADD, ADDI, and MULI in Fig. 3.6 mean load memory data into a register, add second and third operand registers into first operand register, add second register and third immediate into first operand register, and multiply decoder register and third immediate into first operand register, respectively. If a decoded instruction uses multiplier or divider, the processor calculates it in the upper pipeline in Fig. 3.5. Other instructions, such an ALU instruction or a memory access instruction, are processed in the lower pipeline in Fig. 3.5. Only MULI is processed in the upper pipeline among the instructions in Fig. 3.6. The result of MULI becomes available through forwarding paths from MUL and MOD stages, and it is written back to the register file at write stage in the upper pipeline. The decoder in ID stage can use the result after one, three, and five cycles and after. Therefore, the available distance on the edge $4 \rightarrow 6$ is $(\{1, 3\}, 5)$. Since LOAD instruction is processed in MEM stage which is one stage after ALU stage, The edges from LOAD have $(\phi, 2)$ while The edges from ADD have $(\phi, 1)$. The proposed method generates a DDG that has one source and one sink to insert virtual start and terminal nodes for the following scheduling phases. The start and terminal nodes are represented as s and t in Fig. 3.7.

```
1: LI    %R0,  $1          ; R0 = 1;
2: LD    $100, %R1         ; R1 = Mem[100];
3: LI    %R2,  $3          ; R2 = 3;
4: ADD   %R0,  %R1,  %R3   ; R3 = R0 + R1;
5: ADDI  %R0,  $1,   %R0   ; R0 = R0 + 1;
6: MUL   %R1,  %R2,  %R4   ; R4 = R1 + R2;
7: SUB   %R3,  %R4,  %R5   ; R5 = R3 - R4;
```

Figure 3.6: Sample code for sample processor

3.2.3 Complexity of instruction scheduling for partial forwarding processor

Hennessy and Gross show that instruction scheduling with arbitrary latencies on one single pipeline in-order issue processor is a NP-hard problem [44]. Obviously the proposed architecture model for partial forwarding includes conventional pipeline processor model. This fact implies that instruction scheduling method for arbitrary partial forwarding is also NP-hard. In general, only some restricted instruction scheduling problems are solvable in polynomal time [45].

3.3 ILP Formulation of Instruction Scheduling Problem for Partial Forwarding Processor

In this section, optimal instruction scheduling method for partial forwarding processors is described.

3.3.1 Preconditions

ILP scheduling method described in this chapter is based on some preconditions. The target processor on which ILP scheduling method is applied must satisfy following conditions:

- RISC processor.

CISC processor has complex instruction set architecture (ISA) such as variable length instructions. Required cycles for each instruction also varies in wide range so that instructions not flow in the processor pipeline smoothly. Partial forwarding contributes for simple ISA and pipeline control logic processor such as RISC.

- Single-issue processor.

  Multi-issue processor can fetch several instructions at a cycle so that the cost of forwarding increases in order of the square of the number of issue slots [5]. There are many previous work to reduce forwarding cost like clustered VLIW [17, 46]. When partial forwarding is implemented on clustered VLIW processor the scheduling method for partial forwarding can be applied to treat for each issue slot as a pipeline of single-issue processor, after assigning instructions for each issue slot.

- In-order issue processor.

  Out-of-order issue processor shuffles the order of fetched instructions in their instruction buffer to execute instructions in parallel. In the proposed, the order of instructions is optimized by compiler and must not be changed in the target processor.

- Symmetric partial forwarding processor.

  On asymmetric partial forwarding processor, instruction scheduler requires to consider order of the operands for each instruction. In the proposed method, the orders of the input operands for each instruction are ignored. Note that full forwarding processor and no forwarding processor are also symmetric.

3.3.2 Scheduling Method

Input of the instruction scheduling method is an instruction sequence for each basic block, and ISA information which includes available distance. The scheduling method consists of following four steps:

- DDG construction
- DDG simplification

- Calculate upper bound of the total execution cycles

- Solving ILP

3.3.2.1 DDG Construction

Data dependence occurs when an instruction uses a value in a register or a memory before precedence instructions define [32]. Figure 3.8 is DDG construction algorithm. V and E mean a set of nodes and edges, respectively. An element in E is represented as (n_{from}, n_{to}). n_{from} and n_{to} are precedence and succession nodes of the edge. R_k represents storage resource such as register and memory.

This algorithm constructs DDG with only one source node and one sink node. These nodes are called *START* and *TERMINAL* node, respectively. The instructions which are indicated by these nodes are called $START$ and $TERMINAL$ pseudo instructions, or simply $START$ and $TERMINAL$. $START$ and $TERMINAL$ are removed after instruction scheduling.

3.3.2.2 ILP Formulation for Partial Forwarding Processor

Nodes, edges, and weights of the edges in DDG constructed by the previous step represent instructions, dependences, and available distances, respectively. An available distance is represented as (D, l). l denotes the minimum distance that the following instruction can use the preceding instruction's result without a data hazard. $D = \{d_1, d_2, d_3, \ldots\}$ is a set of distances. All elements in D are smaller than l. To execute a program without hazards, all pairs of instructions that have dependencies must keep their distances more than l or a distance in D.

The proposed scheduling method generates a DDG that has one source and one sink to insert virtual start node and terminal node for the following phases. The start node and terminal node are represented as s and t in Fig. 3.7.

3.3.3 DDG Simplification

In this step, DDG is simplified to reduce computation complexity to partition DDG and to remove redundant edges in DDG [47, 48].

Partition node is a node on any path from the source node to the sink node. In other words, there are no paths from the source to the sink without partition node. The source and the sink are also partition nodes since output DDG generated by Fig. 3.8 has only one source node and one sink node. These facts insist that the scheduler regards partition node as barrier node which prevents dependences between preceding nodes and succeeding nodes. All instructions in preceding nodes must be executed before the instruction of the partition node. In contrast, all instructions in succeeding nodes must be executed after the instruction of the partition node. Therefore, the scheduler obtains the optimal solution to solve for each instruction chunk divided by partition nodes. For example, nodes s, 3, and t are partition nodes in Fig. 3.7.

Redundant edges are removable edges when solving scheduling problem. Let e be an edge from nodes a to b. If there is another path between the nodes without e and the minimum length of the path is larger than l of e, b can use the result of a since the dependence of e is dissolved while that of the other path is dissolved. Therefore, e is a redundant edge and the scheduler can ignore this edge when solving scheduling problem. Since the number of edges in the DDG affects the size of the ILP problem, deleting redundant edges in DDG contributes the problem size reduction. For example, the edge between 3-6 is redundant in Fig. 3.7.

3.3.4 Calculating Upper Bound of the Total Execution Cycles

Before solving the ILP problem, Instructions are scheduled by list scheduler to obtain an execution cycle which may be smaller than the original value. The obtained execution cycle is used as the upper bound of the objective function.

3.3.5 Solving ILP

In the last phase, the scheduler solves ILP problem converted from scheduling problem for partial forwarding processors. This problem is formulated as 0-1 ILP problem [47, 49]. Variable x_i^j denotes whether instruction i is scheduled at cycle j. If instruction i is scheduled at cycle j,

x_i^j is 1. If it is not scheduled, x_i^j is 0. The range of i is $[0, n-1]$ where the number of nodes in DDG is n. The range of j is $[0, m+1]$ where the execution cycle of list-scheduled instructions is m, not $[0, m-1]$ since the virtual start and end nodes are inserted in the DDG.

To solve the scheduling problem by ILP, a valid instruction sequence must satisfy the follows constraints:

Constraint 1 Each instruction must be executed exactly once.

Constraint 2 No more than two instructions can be executed in the same cycle.

Constraint 3 A following instruction that uses a result of the preceding instruction must not be executed at an unavailable distance.

Constraint 1 means that exactly one variable among the variables related to i holds 1; in other words, the summation of the variables equals 1, which can be represented as Eq. (3.1):

$$\forall i \quad \sum_{j=1}^{m} x_i^j \quad = \quad 1. \tag{3.1}$$

Constraint 2 means that at most one variable among the variables about j holds 1 and can be represented as Eq. (3.2):

$$\forall j \quad \sum_{i=1}^{n} x_i^j \quad \leq \quad 1. \tag{3.2}$$

Whether D is empty or not for constraint 3 is the most important constraint for partial forwarding processor. Assume that instruction i depends on instruction k. If D is empty, the distance between k and i must be more than L_{ki}, which means l about k and i. In this case, constraint 3 can be represented as Eq. (3.3):

$$\sum_{j=1}^{m} j \times x_k^j + L_{ki} \quad \leq \quad \sum_{j=1}^{m} j \times x_i^j. \tag{3.3}$$

If D is not empty, constraint 3 is represented by two equations. Let $d_{\min}$ be the minimum value of D_{ki}, and N_{ki} be the set of unavailable distances:

$$N_{ki} = \{n | n < L_{ki} \wedge n \notin D_{ki}\}. \tag{3.4}$$

In this case, constraint 3 means that the distance between k and i is more than $d_{\min}$ and not $l \in N_{ki}$. Therefore, it can be represented as Eqs. (3.5) and (3.6),

$$\sum_{j=1}^{m} j \times x_k^j + d_{\min} \quad \leq \quad \sum_{j=1}^{m} j \times x_i^j \tag{3.5}$$

$$x_k^j + x_i^{j+l} \quad \leq \quad 1 \quad \text{for } l \in N_{ki}, \tag{3.6}$$

where j in Eq.(3.6) takes $0, 1, 2, \ldots, m + 1 - l$.

The objective function of the problem is the execution cycle of the terminal node's instruction since it equals all input instructions. The objective function can be represented as Eq. (3.7) because Eq. (3.1) binds the variables in Eq. (3.7) where exactly one variable holds 1:

$$\min \sum_{j=1}^{m} j \times x_n^j. \tag{3.7}$$

3.4 Experiments

In this section, the experimental results are presented.

3.4.1 Environments

To perform and evaluate the proposed method, I constructed a C compiler using a CoSy compiler kit [50] and an opbdp 0-1 ILP solver [51] to solve the scheduling problem's formulation. I compared execution cycles of the outputs by the proposed ILP scheduler with a list scheduler and calculated how many execution cycles are reduced. Target processors in the experiments have various partial forwarding paths based on DLX with five-stage pipeline. The DLX processors were generated by ASIP Meister [9] in form of VHDL description. Partial forwarding paths of the processors are configured on ASIP Meister. For simplification, the DLX processors were implemented only integer instructions. In experiments, ModelSim was used for cycle accurate simulation. Table 3.1 shows available distance for each processor. The first column shows the processors. All processors names start with "DLX_". What follows the prefix

Table 3.1: Available distance of DLXs

Processor Name	Available Distance
DLX_no	$(\{\}, 4)$
DLX_X	$(\{1\}, 4)$
DLX_M	$(\{2\}, 4)$
DLX_W	$(\{\}, 3)$
DLX_XM	$(\{1, 2\}, 4)$
DLX_XW	$(\{1\}, 3)$
DLX_MW	$(\{\}, 2)$
DLX_XMW	$(\{\}, 1)$

means the stages where a forwarding path exists between the input of the EX stage and specified stages. X, M, and W represent EX, MA, and WB stages, respectively. All target DLX processors have no hazard detection units so that the compiler has to insert NOP instructions.

Table 3.2 shows the specifications of the DLXs. We obtained these estimation values for each processor by logic synthesis. The frequency of the processors increases in inverse proportion to the number of forwarding paths. Also, if the numbers of the forwarding paths are the same between two processors, e.g. DLX_X and DLX_M, the processor which has a forwarding path on an earlier stage runs at lower frequency instead of efficient execution. This result is inconsistent with an intuition that reducing forwarding path on a later stage improves frequency of the processor higher. The delay of the critical path is the reason. DLX_X have the critical path through forwarding paths and ALU in EX stage so that the maximum delay of DLX_X become longer than DLX_M and DLX_W. Delay due to control unit for forwarding also affects the frequency so that the frequency of DLX_M is longer than that of DLX_W. For the same reason, the processors which have forwarding paths from two stages, DLX_XM, DLX_XW, and DLX_MW is slower than DLX_M and DLX_W. However, DLX_X is slower than DLX_MW because DLX_MW does not have forwarding paths through ALU which is the largest functional unit in the processor.

Although execution cycles of the programs increases in proportion to the number of for-

Table 3.2: Specifications of DLXs

	program		
processor	Frequency(MHz)	Area(Gate)	Power(mW)
DLX_no	141.64	34961	7.92
DLX_X	124.38	37241	8.11
DLX_M	128.87	38188	8.46
DLX_W	130.72	38853	8.77
DLX_XM	120.34	40383	8.59
DLX_XW	119.76	41083	8.90
DLX_MW	127.39	42071	9.24
DLX_XMW	119.05	44077	9.36

warding paths, however, the frequency of the processors increases in inverse proportion to the number of forwarding paths. If programs are scheduled optimally for the processor with less forwarding paths, the total execution time may faster than the processor with more forwarding paths. For example, DLX_M may be faster than DLX_XMW if the proposed scheduler solves the better solution.

The following programs were used in this experiment:

- sieve: perform the Sieve of Eratosthenes
- mat1x3: calculate product of 3x3 matrix with 1x3 matrix
- crc16: calculate 16bit-CRC
- inssort: perform insertion sort
- fir2dim: perform FIR-filtering for 4x4 matrix

Sieve is a program that searches for prime numbers between $(0, 100)$. Fir2dim and mat1x3 are from DSPStone [52]. The rest of programs in DSPStone are not chosen because not all of them can be compiled using ILP on all DLX processors in reasonable time. Crc16 and inssort were made from scratch.

All programs in this experiment have no basic blocks which size is more than 20; This characteristic means that the programs may be solved in practical time even though instruction scheduling for partial forwarding is an NP-hard problem.

3.4.2 Code Size

Table 3.3 shows the improvement ratios of code size for each program by the proposed method against the list scheduler. An improved program ratio of is calculated as $1 - C_{ILP}/C_{list}$, where C_{ILP} and C_{list} denote the output code size of ILP and the list scheduler. The list scheduler is used as a comparison target since it is a well-known scheduling algorithm. The proposed algorithm reduces code size up to 4.0% on average, and maximum improvement is 13.5%. Note that the improvement ratio tends to be proportional to the program size. In the result, fir2dim shows the most improvement by the proposed method. This result suggested that more improvement is possible if the program size is large. In this experiment, target program sizes were relatively small, and thus more improvements was expected for practical programs. The improvement ratio tends to be inverse proportional to the number of forwarding paths, except DLX_no. Note that all programs scored the highest ratio on DLX_X. This result implies that the result of an instruction is often used by the next instruction, and the list scheduling cannot fully exploit the partial forwarding path of the DLX_X.

3.4.3 Execution Cycles and Times

Table 3.4 shows the execution cycles of the programs with the proposed method and list scheduler and improvement ratio. Although each improvement ratio of execution cycles differs from that of code size because of the number of the execution count for each basic block, the total tendency of the results are similar to the result of code size. This result seems to show the trivial result that full forwarding processor, DLX_XMW, is the best solution for all programs. However, experimental results of execution time deny the intuition.

Table 3.5 shows the execution times of the programs by the proposed method and the list scheduler for each DLX processor. The fastest time for each program is marked in Table 3.5.

Table 3.3: Improvement Ratio of code size compared ILP with list scheduler

	program														
	sieve			mat1x3			inssort			crc16			fir2dim		
processor	List	ILP	Ratio[%]	List	ILP	Ratio[%]	List	ILP	Ratio[%]	List	ILP	Ratio[%]	List	ILP	Ratio[%]
DLX_no	177	175	1.1	131	125	4.6	277	269	2.9	379	367	3.2	559	524	6.3
DLX_X	140	127	9.3	111	103	7.2	218	203	6.9	301	277	8.0	483	418	13.5
DLX_M	143	137	4.2	111	105	5.4	214	209	2.3	304	294	3.3	490	435	11.2
DLX_W	154	153	0.6	117	111	5.1	241	233	3.3	333	324	2.7	496	464	6.5
DLX_XM	128	121	5.5	100	99	1.0	193	186	3.6	270	258	4.4	425	408	4.0
DLX_XW	128	123	3.9	104	101	2.9	197	190	3.6	272	265	2.6	446	412	7.6
DLX_MW	134	132	1.5	105	103	1.9	205	200	2.4	290	283	2.4	451	430	4.7
DLX_XMW	122	121	0.8	99	99	0.0	182	181	0.5	257	257	0.0	408	408	0.0

All programs become faster with the proposed method; in case sieve, DLX_XW is the fastest processor if the program is scheduled by list scheduler. However, the proposed method shows that DLX_X is the fastest processor.

Except for fir2dim, the actually fastest processor for each program scheduled by the proposed method differs from that scheduled by the list scheduler. This result insists that the proposed method is useful for partial forwarding.

Table 3.6 shows the results of compilation time using the proposed method. All programs except one are compiled within practical time. Since the proposed scheduler aims to calculate the optimal solution, compilation time in this experiment is almost acceptable. However, inssort consumed over 24 hours for DLX_no processor. The reason for this overtime apparently is that, for one of the basic blocks, DDG has a complex structure, and thus combinational explosions occurred at scheduling. The same reason may exists in the rest of the DSPStone programs so that shortening compilation time is one of the important future work.

Table 3.4: Improvement Ratio of execution cycles compared ILP with list scheduler

	program								
	sieve			mat1x3			inssort		
processor	List	ILP	Ratio[%]	List	ILP	Ratio[%]	List	ILP	Ratio[%]
DLX_no	15295	15193	0.7	979	926	5.4	21899	21768	0.6
DLX_X	11817	10586	10.4	897	859	4.2	18387	18144	1.3
DLX_M	12155	11749	3.3	899	865	3.8	18531	18472	0.3
DLX_W	13313	13212	0.8	917	883	3.7	20131	20000	0.7
DLX_XM	10999	10380	5.6	864	855	1.0	17770	17466	1.7
DLX_XW	10793	10382	3.8	864	857	0.8	17756	17539	1.2
DLX_MW	11534	11332	1.8	873	861	1.4	18363	18235	0.7
DLX_XMW	10481	10380	1.0	855	855	0.0	17201	17113	0.5

	program					
	crc16			fir2dim		
processor	List	ILP	Ratio[%]	List	ILP	Ratio[%]
DLX_no	12597	12173	3.4	15517	15204	2.0
DLX_X	9219	8335	9.6	14643	13361	8.6
DLX_M	9470	9436	0.4	14581	13954	4.3
DLX_W	10995	10574	3.8	14683	14268	2.8
DLX_XM	8316	7868	5.4	13835	13777	0.4
DLX_XW	7906	7887	0.2	14138	13781	2.5
DLX_MW	9396	8977	4.5	14131	13928	1.4
DLX_XMW	7855	7855	0.0	13777	13777	0.0

3.5 Summary

In this chapter, an optimal scheduling method for partial forwarding processors solved by converting ILP problem is proposed. In experiments, the proposed instruction scheduler generated more efficient code than the simple list scheduler which supports partial forwarding. Experimental results also showed that the proposed scheduler extracted the advantage of partial forwarding processor. These results proved that the proposed scheduling method can achieve to optimize embedded systems with configurable processors.

Table 3.5: Execution time of the programs on each DLX processor

	sieve		matrix1x3		inssort		crc16		fir2dim	
processor	list[μs]	ILP[μs]	list[μs]	ILP[μs]	list[μs]	ILP[μs]	list[μs]	ILP[μs]	list[μs]	ILP[μs]
DLX_no	108.0	107.3	6.91	*6.54*	154.6	153.7	88.9	85.9	*109.6*	*107.3*
DLX_X	95.0	*85.1*	7.21	6.91	147.8	145.9	74.1	67.0	117.7	107.4
DLX_M	94.3	91.2	6.98	6.71	*143.8*	143.3	73.5	73.2	113.1	108.3
DLX_W	101.8	101.1	7.02	6.75	154.0	153.0	84.1	80.9	112.3	109.2
DLX_XM	91.4	86.3	7.18	7.11	147.7	145.1	69.1	*65.4*	115.0	114.5
DLX_XW	*90.1*	86.7	7.21	7.16	148.3	146.5	*66.0*	65.9	118.1	115.1
DLX_MW	90.5	89.0	*6.85*	6.76	144.1	*143.1*	73.8	70.5	110.9	109.3
DLX_XMW	88.0	87.2	7.18	7.18	144.5	143.7	66.0	66.0	115.7	115.7

Table 3.6: Scheduling time using ILP [sec.]

	program				
processor	sieve	mat1x3	inssort	crc16	fir2dim
DLX_no	0.87	85.00	100680.0	1.87	8.8
DLX_X	0.69	318.00	3.8	1.00	1100.0
DLX_M	0.68	129.00	1.3	1.00	230.0
DLX_W	0.53	30.00	58.2	1.13	23.9
DLX_XM	0.31	4.13	1.2	0.66	21.8
DLX_XW	0.27	0.16	1.2	0.33	61.0
DLX_MW	0.47	59.00	0.7	0.76	107.0
DLX_XMW	0.10	0.07	0.3	0.13	0.2

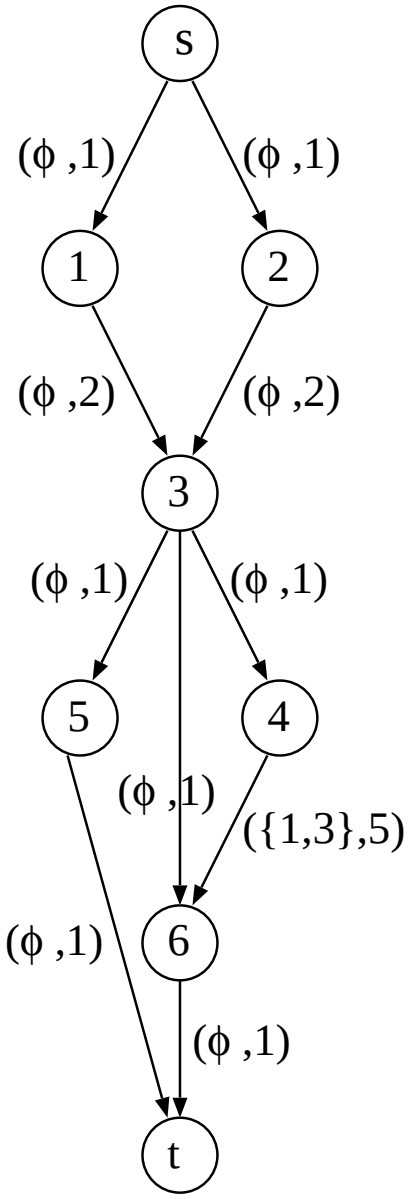


Figure 3.7: DDG of the sample code on the sample processor

Input: Instruction sequence $I = \{i_1, i_2, \ldots, i_{n-1}\}$

Output: Data dependence graph $DDG(V, E)$

begin

$n_s = STARTnode, n_e = TERMINALnode, V = \{n_s\}, E = \phi$;

$\forall R_k(USE[R_k] = \{n_s\}, DEF[R_k] = n_s)$;

for (k = 1 ; k $<$ n ; k++) {

n_k = instruction of node i_k, $V = V \cup \{n_k\}$;

for all R_t (i_k uses R_t) {

$L = Latencyweightofedge(DEF[R_t], n_k)$;

$E = E \cup (DEF[R_t], n_k, L)$;

$USE[r_t] = USE[R_t] \cup \{n_k\}$;

}

for all R_t (i_k defines R_t) {

for all n_j in $USE[R_t]$ {

$L = Latencyweightofedge(n_j, n_k)$;

$E = E \cup (n_j, n_k, L)$;

}

$USE[R_t] = \phi$, $DEF[R_t] = n_k$;

}

}

$V = V \cup \{n_e\}$;

for all $n_t \in V$ (n_t is sink node) {

$L = Latencyweightofedge(n_t, n_e)$;

$E = E \cup (n_t, n_e, L)$;

}

return DDG;

end

Figure 3.8: DDG construction algorithm

Chapter 4

Heuristic Code Scheduling for Partial Forwarding Processors

This chapter describes heuristic instruction scheduling method for partial forwarding processors, which differs from integer linear programming (ILP) method described in chapter 3. This chapter is organized as follows. Instruction scheduling algorithm for partial forwarding processors is described in section 4.1. Experimental results are described in section 4.2. Finally, this chapter is concluded in section 4.3.

4.1 Instruction Scheduling Algorithm for Partial Forwarding Processor

In this section, the proposed scheduling algorithm for partial forwarding processor is described.

4.1.1 Detail of the Scheduling Algorithm

The proposed heuristic scheduling algorithm for partial forwarding based on list scheduler which takes the length of the longest path in the input DDG as priority function. Figure 4.3 shows the proposed algorithm. $SchCycle$ represents current cycle. The scheduler increments $SchCycle$ when an instruction is scheduled at $SchCycle$. $MAXPATH_i$ is the length of the longest path from v_i to the other nodes in the graph. The scheduler uses $MAXPATH_i$ as

priority to sort instructions in $NodeList$ by descending order. $ExecutableList$ is a subset of $NodeList$, which has only executable instructions at $SchCycle$. The order of instructions in $ExecutableList$ is the same that in $NodeList$.

When the target processor has partial forwarding structure, it is important to determine priority of each node. The proposed algorithm chooses the instruction with the highest priority based on these policies:

- If there are enough executable instructions at the current cycle so that the processor can finish processing all instructions in the pipeline, the scheduler issues the instruction with the highest priority.
- If more than two instructions with the highest priority, the scheduler issues the instruction which uses the result from partial forwarding circuit.
- If the instruction with highest priority will prevent other instruction which uses the result from partial forwarding, the scheduler does not issue the highest instruction and try to issue the second highest instruction.

The first heuristic aims to ignore partial forwarding paths when there are enough instructions to be issued. Though an instruction requires certain cycles to write the execution result into a register, the processor can execute other instructions which do not require the result through forwarding datapaths. In this algorithm, if there are enough executable instructions, the scheduler issues instructions to wait results of the instructions in the pipeline are written back to the destination registers.

The scheduler takes second and third heuristics when the number of executable instructions is less than the number of the longest latency for which the target processor has to wait. These policies aim to raise use rate of partial forwarding paths. In some cases for partial forwarding processor, the optimal instruction sequence may have no operation cycle even if issuable instruction exists to reduce total execution cycle. For this reason, the proposed scheduler changes the instruction to issue from the highest instruction to the other instructions with lower priorities when a state satisfies these conditions during scheduling:

- The instruction i, the scheduler tries to issue, only has one instruction which uses the result of i.

- The following instruction j uses the result of i and other preceding instruction i'.
- Both dependencies $i \rightarrow j, i' \rightarrow j$ have D which is not empty set.
- D of $i' \rightarrow j$ has the shortest available distance value than D of $i \rightarrow j$.

Figure 4.2 shows an example of this case. In this case, node 1 should be scheduled at 1 cycle after the fastest executable cycle since node 3 become use the results of nodes 1 and 2 through partial forwarding paths. If node 1 is scheduled at the fastest executable cycle, the result of node 2 cannot be used.

Outline of the proposed scheduling algorithm described in Fig. 4.3 is summarized below;

1. Calculate maximum path length for each nodes which are not scheduled. According to the maximum path length as priority, choose the node with the highest priority as scheduling candidate node .
2. Compare each l of the succeeding instruction from the candidate instruction. If the number of current schedulable instructions is more than l, the scheduler schedules the candidate instruction at the current cycle, otherwise calculate minimum scheduling cycle for each instruction which succeeds the candidate instruction directly.
3. If the scheduler decides that the candidate instruction should be scheduled at the current cycle to execute the succeeding instruction at the earliest cycle, otherwise scheduler reject the candidate instruction and choose new candidate instruction with the second highest priority.
4. Repeat 2 and 3 until the candidate instruction is scheduled. If all schedulable instructions are rejected, NOP is scheduled.
5. Repeat 1-4 until all instructions are scheduled.

Step 1 aims to schedule instructions on the critical path prior to the others. Note that the weight of edge is available distance so that the scheduler has to consider how to calculate path length. In the proposed method, summation of all minimum element of available distance is adopted as path length. This means that the scheduler regards L, if D is empty set, or the

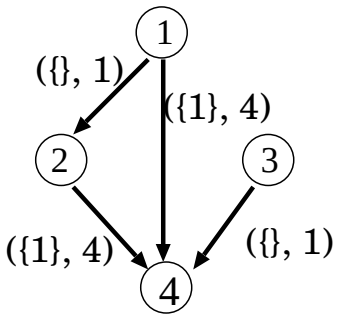


Figure 4.1: An example DDG for the scheduling algorithm

minimum element in D as the weight of each edge. The scheduler searches the critical path for schedulable instruction. If found, it is chosen as the candidate instruction and the scheduler advances to the next step. Otherwise, the scheduler searches candidate instruction on the second longest path. This search continues until candidate instruction is found.

In step 2, the scheduler counts all schedulable instructions except the candidate instruction. The scheduler decides whether the candidate instruction should be scheduled on the current cycle or should not be, according to L of the directly succeeding instruction on the critical path. If the number of schedulable instructions is greater than this threshold value, the candidate instruction is scheduled on the current cycle. This decision policy stands on an intuition that the result of the candidate instruction become available while other instructions are executed.

If the number of schedulable instructions is less than the threshold value, the scheduler calculates the earliest schedulable cycle of the directly succeeding instruction on the critical path. This strategy assumes that improvement of the issue cycle of the instructions on the critical path contributes the total execution cycle of whole instruction sequence. Let t, i, and i_d be the current cycle, the candidate instruction, and the directly succeeding instruction on the critical path, respectively. The scheduler searches the earliest schedulable cycle of i_d for each $t, t+1, t+2, ..., t+l_i-1$ of the scheduled cycle of i. where l_i means l of available distance of i. If the best scheduling cycle is estimated on the current cycle, t, the candidate instruction is scheduled on t. Otherwise, the scheduler rejects the candidate instruction from the schedulable instructions on the current cycle and searches the other instructions until an instruction is scheduled on the current cycle. If there are no schedulable instructions or all schedulable

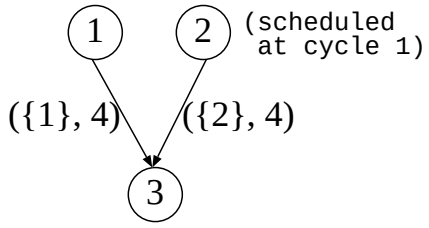


Figure 4.2: Example for scheduling problem about partial forwarding

instructions are rejected, NOP is scheduled on the current cycle.

For example, scheduling steps on DDG of Fig. 4.1 is as follows:

1. The current cycle is 1 and insns. 1 and 3 are schedulable in initial state.

2. The current critical path is instruction chain 1-2-4 so that insn. 1 is chose as the scheduling candidate.

3. The scheduler decides to schedule insn. 1 at cycle 1.

4. The current cycle is 2 and insns. 2 and 3 are schedulable. The current critical path is instruction chain 2-4 so that insn. 2 is chose.

5. The scheduler decides that insn. 2 should be scheduled at cycle 4 rather than cycle 2, at the current cycle, to schedule insn. 4 earlier. Insn. 2 is rejected.

6. Insn. 3 is the only schedulable instruction except insn. 2 so that it is chosen.

7. The scheduler decides to schedule insn. 3 at cycle 2.

8. Although insn. 2 is the only schedulable instruction at cycle 3, the scheduler still decides that insn. 2 should be scheduled at cycle 4 so that NOP instruction is scheduled.

9. Insn. 2 is scheduled at cycle 4.

10. At cycle 5, insn. 4 is schedulable. The scheduler decides that insn. 4 should be scheduled at the current cycle. Insn. 4 is scheduled and finish scheduling.

4.1.2 Complexity Order of the Scheduling Algorithm

In this section, the order of the algorithm according to line numbers of Fig. 4.3 are discussed. Lines 03-04 are initialize part. Line 04 can be implemented to use Dijkstra's algorithm [53], and the order of the initialize part is $O(E+N)$, where E is the number of edges and N is the number of nodes in DDG. Target processor is a RISC processor by the assumption, numbers of inputs of instruction are fixed. In this case, the order of these lines can be reduced $O(E)$ as $O(N)$. $ExecutableNodes$ has to be sorted by descending order so that the initialize part requires at least order of $O(N \log N)$. Therefore, the order of the initialize part if $O(E + N + N \log N) = O(N \log N)$.

In the main loop, at most one instruction is issued for each loop by the assumption. It means that the order of the main loop can be calculated as $O(N)$ times the number of loop iteration. The number of loop iteration increases only if the scheduler decides to suspend an instruction to use partial forwarding following the heuristics. Each line from line 09 to line 16 can be calculated by constant order since the number of input of node is restricted due to the constraint of RISC processor. Therefore, the order of the algorithm depends on the number of suspension. At a certain cycle during scheduling, the number of suspension is at most the half of number of waiting instructions due to the suspending condition described above, where waiting instruction means instructions in $ExecutableList$ and instructions which wait until the result of preceding instructions become available. Therefore, the number of suspension is $O(\log N)$ since the number of waiting instructions is at most $O(\log N)$ due to the form of DAG. Moreover, even though the scheduler continues not to issue instructions, the number of suspension does not grow at a certain number of the longest latency of the processor since the result of preceding instructions become available when the processor drives the pipeline. Therefore, the order of the main loop is $O(N \times \log N)$. Consequently, the order of whole algorithm is $O(N \log N)$.

4.2 Experiments

In this section, the experimental results are presented.

Table 4.1: Total, average, and maximum sizes of basic block for each program in DSPstone

	complex_ multiply	complex_ update	con- volution	dot_ product	fft	fir	fir2dim	iir_biquad_ N_sections
total	93	130	86	73	299	101	285	195
average	7	13	7	6	7	7	6	10
max	39	77	23	27	55	25	23	96

	iir_biquad_ one_section	lms	matrix1	matrix2	matrix1x3	n_complex_ updates	n_real_ updates	real_ update
total	120	140	150	173	69	216	125	63
average	7	9	5	6	6	15	8	5
max	62	25	23	27	21	100	37	20

4.2.1 Specification of processors

In experiments, execution time of testbench programs compiled with the proposed method were compared with that of optimal scheduling results obtained by the ILP scheduler described in chapter 3. The same DLX processors shown in table 3.2 are used in this experiments so that available distances of the processors also the same as table 3.1.

4.2.2 Evaluation of the proposed scheduling algorithm

I compared the execution time of the testbench programs whether the proposed scheduler generates optimal or suboptimal result within short time. Following result shows experimental results of comparison. I executed DSPStone Kernel Benchmarks [52] on each DLX processor in Table 3.1. Table 4.1 shows total, average, and maximum sizes of basic block size for each program in DSPstone. The program is easy to schedule if average size of the program is relatively small to the total size of that, and vice versa. In table 4.1, `fir2dim` is easy to schedule and `n_complex_updates` is difficult to schedule.

4.2.3 Comparison with ILP method

Although ILP scheduler solves the optimal solution, it requires long time to solve and only two programs, `mat1x3` and `fir2dim`, were solved within practical time among DSPstone. Therefore, I used these two programs and three additional programs, `sieve`, `inssort`, and `crc16`, used in chapter 3 for comparison of compilation and execution time. Table 4.2 shows the execution cycles with the proposed heuristic method and ILP method in chapter 3. I assumed that each processor drives at maximum frequency since partial forwarding is used to raise frequency to reduce amount of forwarding circuit. Increasing rates in Table 4.2 represent gaps of execution cycles with the proposed heuristic scheduler from that with the optimal ILP scheduler. Table 4.2 shows that the proposed heuristic method solves scheduling problems optimally for many pairs of processor and program. Also, increasing ratio of execution cycles from the optimal solution was at most 5% and only five pairs of processor and program shows more than 2%. Table 4.3 shows the execution time with the proposed heuristic method and ILP method in chapter 3. Each increasing rate in Table 4.3 was similar to that of Table 4.3. These results insist that the optimal or suboptimal solution can be obtained with the proposed scheduler.

At the point of compilation time, shown in Tables 4.4, the proposed scheduler finished compilation within 0.001s in contrast the ILP scheduler may consume several minutes. This rapid compilation time is enough to explore design space for partial forwarding.

At the point of design space exploration for partial forwarding, fast design space exploration method is important since a huge amounts of processors have to be evaluated. Figures 4.4-4.8 show trade-off graph for each test program. In each graph, points on solid line represent Pareto optimal solution with the proposed heuristic method and points on dotted line is Pareto optimal solution with ILP method. With `mat1x3`, `crc16`, and `fir2dim`, the proposed heuristic method obtained the same Pareto optimal solutions with ILP method. Although there were some suboptimal solutions, the proposed method could be used for fast design space exploration for partial forwarding. On the other hand, the proposed heuristic method missed the best Pareto optimal solution with `sieve` and `inssort`. However, the difference from the Pareto

Table 4.2: Comparison of execution cycles between proposed heuristic and ILP methods

	program								
	sieve			mat1x3			inssort		
processor	Heuristics	ILP	Ratio[%]	Heuristics	ILP	Ratio[%]	Heuristics	ILP	Ratio[%]
DLX_no	15193	15193	0.00	2181	2181	0.00	21802	21768	0.16
DLX_X	10995	10586	3.86	2003	1977	1.32	18303	18144	0.88
DLX_M	11989	11749	2.04	1933	1933	0.00	18490	18472	0.10
DLX_W	13212	13212	0.00	1974	1974	0.00	20059	20000	0.30
DLX_XM	10380	10380	0.00	1974	1974	0.00	17508	17466	0.24
DLX_XW	10744	10382	3.49	1976	1940	1.86	17612	17539	0.42
DLX_MW	11338	11332	0.05	1936	1936	0.00	18235	18235	0.00
DLX_XMW	10380	10380	0.00	1869	1869	0.00	17113	17113	0.00

	program					
	crc16			fir2dim		
processor	Heuristics	ILP	Ratio[%]	Heuristics	ILP	Ratio[%]
DLX_no	12173	12173	0.00	37719	37719	0.00
DLX_X	8731	8335	4.76	33869	33595	0.82
DLX_M	9436	9436	0.00	33339	33054	0.86
DLX_W	10688	10574	1.08	33360	33360	0.00
DLX_XM	7978	7868	1.40	33755	33225	1.60
DLX_XW	7892	7887	0.07	33617	33317	0.90
DLX_MW	9022	8977	0.50	33465	33465	0.00
DLX_XMW	7855	7855	0.00	30501	30501	0.00

optimal solutions with ILP method was small with both programs due to enough approximation ratio by the proposed heuristic method.

4.2.3.1 Results for other benchmarks

Table 4.5 shows the execution time for each pair of DLX processors and benchmark programs. `fft` in table 4.5 uses FFT INPUT SCALED C function in DSPStone and takes 16-bit input

Table 4.3: Comparison of execution time between proposed heuristic and ILP methods

	program								
	sieve			mat1x3			inssort		
processor	Heuristics [μs]	ILP [μs]	Ratio[%]	Heuristics [μs]	ILP [μs]	Ratio[%]	Heuristics [μs]	ILP [μs]	Ratio[%]
DLX_no	107.3	107.3	0.00	15.4	15.4	0.00	153.9	153.7	0.14
DLX_X	88.4	85.1	3.88	16.1	15.9	1.26	147.2	145.9	0.86
DLX_M	93.0	91.2	2.01	15.0	15.0	0.00	143.5	143.3	0.13
DLX_W	101.1	101.1	0.00	15.1	15.1	0.00	153.5	153.0	0.30
DLX_XM	86.3	86.3	0.00	16.4	16.4	0.00	145.5	145.1	0.27
DLX_XW	89.7	86.7	3.47	16.5	16.2	1.85	147.1	146.5	0.38
DLX_MW	89.0	89.0	0.00	15.2	15.2	0.00	143.1	143.1	0.03
DLX_XMW	87.2	87.2	0.00	15.7	15.7	0.00	143.7	143.7	0.03

	program					
	crc16			fir2dim		
processor	Heuristics [μs]	ILP [μs]	Ratio[%]	Heuristics [μs]	ILP [μs]	Ratio[%]
DLX_no	85.9	85.9	0.05	266.3	266.3	0.00
DLX_X	70.2	67.0	4.78	270.1	272.3	0.81
DLX_M	73.2	73.2	0.03	256.5	258.7	0.86
DLX_W	81.8	80.9	1.07	255.2	255.2	0.01
DLX_XM	66.3	65.4	1.38	276.1	280.5	1.59
DLX_XW	65.9	65.9	0.00	278.2	280.7	0.90
DLX_MW	70.8	70.5	0.46	262.7	262.7	0.01
DLX_XMW	66.0	66.0	0.00	256.2	256.2	0.00

data. In some cases, DLX_XMW which has forwarding circuits from all pipeline stages was not the fastest processor since these processors run at the maximum frequency. These results insist that partial forwarding processor can drive some benchmarks faster than DLX_XMW which has the full forwarding structure.

4.2.4 Comparison with hazard detection unit

As described in chapter 2, hazard detection unit can be used even though in partial forwarding processor. I compared the execution cycle of benchmark programs between the scheduled code for partial forwarding and the code for full forwarding processor on partial forwarding processor with pipeline stall function.

Table 4.4: Comparison of compilation time between proposed heuristic and ILP methods

	sieve		mat1x3		inssort		crc16		fir2dim	
Processor Name	Heuristics [s]	ILP [s]	Heuristics [s]	ILP [s]	Heuristics [s]	ILP [s]	Heuristics [s]	ILP [s]	Heuristics [s]	ILP [s]
DLX_no	0.001	42.0	0.001	3.5	0.001	140.5	0.001	1.2	0.001	2.8
DLX_X	0.001	157.2	0.001	45.1	0.001	22.2	0.001	3.0	0.001	267.0
DLX_M	0.001	13.6	0.001	2.2	0.001	6.2	0.001	3.2	0.001	4.8
DLX_W	0.001	23.0	0.001	1.3	0.001	11.3	0.001	4.0	0.001	9.0
DLX_XM	0.001	0.8	0.001	0.5	0.001	0.9	0.001	1.2	0.001	0.6
DLX_XW	0.001	1.2	0.001	11.2	0.001	32.1	0.001	0.6	0.001	58.0
DLX_MW	0.001	52.0	0.001	0.2	0.001	2.6	0.001	0.4	0.001	1.8
DLX_XMW	0.001	0.7	0.001	0.1	0.001	0.1	0.001	0.3	0.001	0.1

Table 4.7 shows the result of execution cycles on partial forwarding processors which have hazard detection unit. Each program is compiled for DLX_XMW and any data hazard is solved by hazard detection and pipeline stall. Table 4.8 illustrates differences of execution cycle between processor with hazard detection unit and the proposed scheduling. This result insists that large programs tend to perform fast execution with the proposed scheduler. For example, `matrix1` and `matrix2`, largest programs in this benchmark, run faster with the proposed scheduler than the hazard detection unit. On the other hand, the hazard detection unit improves execution cycle of some small programs since small basic blocks in their programs have no affords to optimize their code for partial forwarding. This ineffectiveness is limited when I apply the proposed scheduler to practical programs.

The proposed scheduling algorithm can be also applied for partial forwarding processor with hazard detection unit. However, since the proposed scheduler removes all hazards, the execution cycle of the scheduled code on the processor with hazard detection unit equals that on the processor without hazard detection unit. In other words, the proposed algorithm can be used to reduce hazard detection unit for partial forwarding processor.

Table 4.5: Execution time for each benchmark program with the proposed scheduler

Processor Name	complex_ multiply	complex_ update	con- volution	dot_ product	fft	fir	fir2dim	iir_biquad_ N_sections
DLX_no	8.0	10.2	37.1	7.0	342.2	30.6	266.3	92.1
DLX_X	8.3	10.5	38.4	7.4	289.8	35.1	272.3	97.2
DLX_M	8.1	10.0	36.4	6.9	330.6	32.9	258.7	92.0
DLX_W	7.8	9.8	37.1	7.0	300.2	31.8	255.2	92.9
DLX_XM	8.6	10.5	38.4	7.4	288.5	34.9	280.5	99.3
DLX_XW	8.5	10.5	38.8	7.5	292.0	34.0	280.7	99.7
DLX_MW	7.9	9.8	36.2	6.9	310.2	32.5	262.7	92.9
DLX_XMW	7.8	10.1	37.0	6.9	283.6	33.6	256.2	93.1

Processor Name	iir_biquad_ one_section	lms	matrix1	matrix2	matrix1x3	n_complex_ updates	n_real_ updates	real_ update
DLX_no	10.7	56.6	1976.2	1888.7	15.4	150.1	55.6	6.0
DLX_X	11.8	57.0	2020.0	1866.5	16.1	150.9	56.6	5.5
DLX_M	11.4	53.7	1982.6	1895.3	15.0	145.0	54.6	5.7
DLX_W	11.1	52.7	1968.4	1880.4	15.1	146.6	54.2	5.6
DLX_XM	12.0	58.7	2089.0	2001.8	16.4	155.5	58.5	6.0
DLX_XW	12.0	58.3	2139.1	2053.5	16.5	157.3	54.1	6.0
DLX_MW	11.2	53.9	1944.4	1869.8	15.2	146.6	54.6	5.6
DLX_XMW	12.0	58.7	2080.1	2000.3	15.7	157.9	56.7	5.9

4.3 Summary

In this chapter, heuristic scheduling method for partial forwarding processors is proposed. In experiments, the heuristic instruction scheduler generated nearly optimal solution within feasible compilation time. Experimental results also showed the advantage of the proposed scheduling method compared with hazard detection unit.

Table 4.6: Execution cycle for each benchmark program with the proposed scheduler

Processor Name	complex_ multiply	complex_ update	con- volution	dot_ product	fft	fir	fir2dim	iir_biquad_ N_sections
DLX_no	1133	1445	5255	991	48469	4334	37719	13045
DLX_X	1032	1306	4776	920	36045	4366	33869	12090
DLX_M	1044	1289	4691	889	42604	4240	33339	11856
DLX_W	1020	1281	4850	915	39242	4157	33360	12144
DLX_XM	1035	1264	4621	891	34718	4200	33755	11950
DLX_XW	1018	1257	4647	898	34970	4072	33617	11940
DLX_MW	1006	1248	4612	879	39516	4140	33465	11835
DLX_XMW	929	1202	4405	821	33763	4000	30501	11084

Processor Name	iir_biquad_ one_section	lms	matrix1	matrix2	matrix1x3	n_complex_ updates	n_real_ updates	real_ update
DLX_no	1516	8017	279909	267515	2181	21260	7875	850
DLX_X	1468	7090	251248	232155	2003	18769	7040	684
DLX_M	1469	6920	255498	244247	1933	18686	7036	735
DLX_W	1451	6889	257309	245806	1974	19164	7085	732
DLX_XM	1444	7064	251390	240897	1974	18713	7040	722
DLX_XW	1437	6982	256179	245927	1976	18838	6479	719
DLX_MW	1427	6866	247697	238194	1936	18675	6955	713
DLX_XMW	1429	6988	247636	238136	1869	18798	6750	702

Input: A data dependency graph $DDG(V, E)$

V: A set of instruction nodes, E: A set of directed edges in DDG

Output: A scheduled instruction sequence $I = \{i_1, i_2, ..., i_m\}$

01: begin
02: // Initialize part
03: $SchCycle = 0$;
04: $ExecutableList =$ list of nodes which have no predecessor nodes in DDG,
sorted by descending order of the longest reachable path length for each node;
05: // Main loop
06: while (unscheduled node exists) {
07: do {
08: // Choose an instruction to schedule
09: $v_{target} =$ The first schedulable node at $SchCycle$ in $ExecutableList$;
10: if (the number of child nodes of $v_{target} >= 2$
|| v_{target} is the only parent node of v_c
|| the available distance with the dependency of v_p,
the other parent node of v_c, to v_c, (D_p, L_p), has the empty D_p
|| In the available distance with the dependency of v_{target} to v_c, (D_t, L_t),
the smallest value of $D_t >=$ the smallest value of D_P) {
11: Schedule v_{target} at $SchCycle$;
12: Remove v_{target} from $ExecutableList$;
13: for each ($v_{succ} =$ successors of v_{target}) {
14: if (all predecessors of v_{succ} are scheduled) Add v_{succ} into $ExecutableList$;
15: }
16: }
17: Check v_{target} to avoid to schedule at cycle $SchCycle$;
18: } while (an instruction is scheduled at cycle $SchCycle$
|| All instructions in $ExecutableList$ have been checked);
19: $SchCycle$++;
20: }
21: end

Figure 4.3: The proposed scheduling algorithm

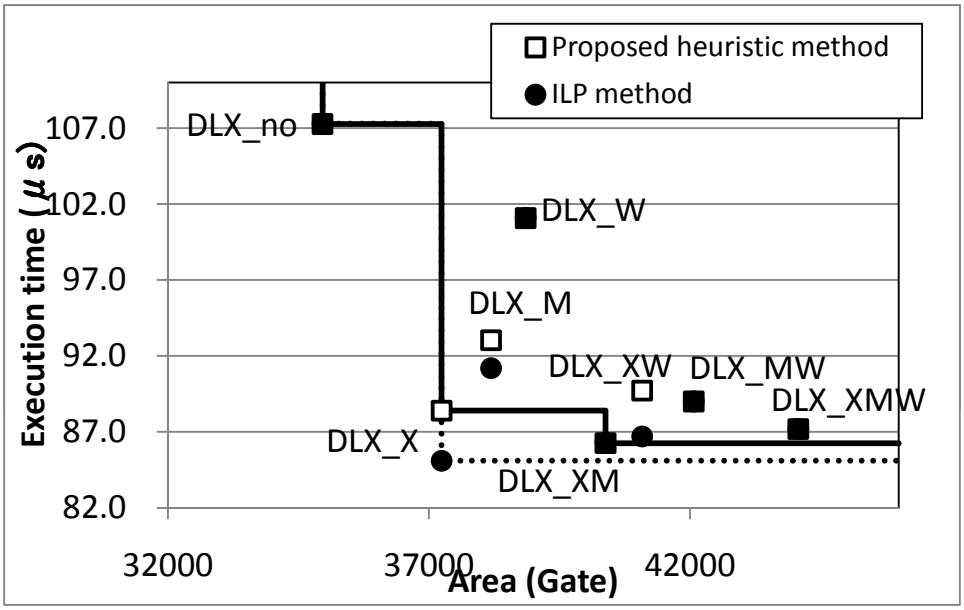


Figure 4.4: Trade-off graph of sieve

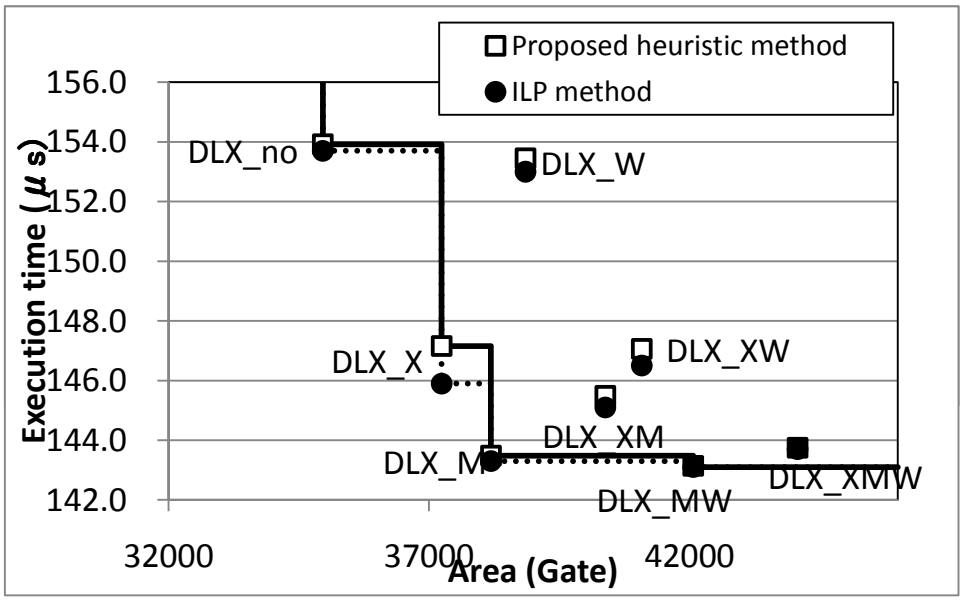


Figure 4.5: Trade-off graph of mat1x3

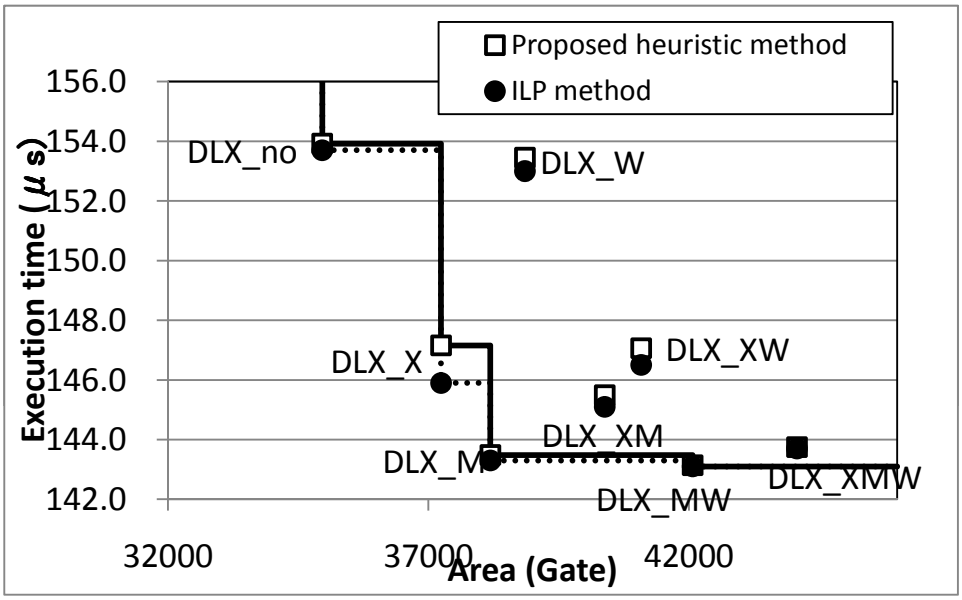


Figure 4.6: Trade-off graph of inssort

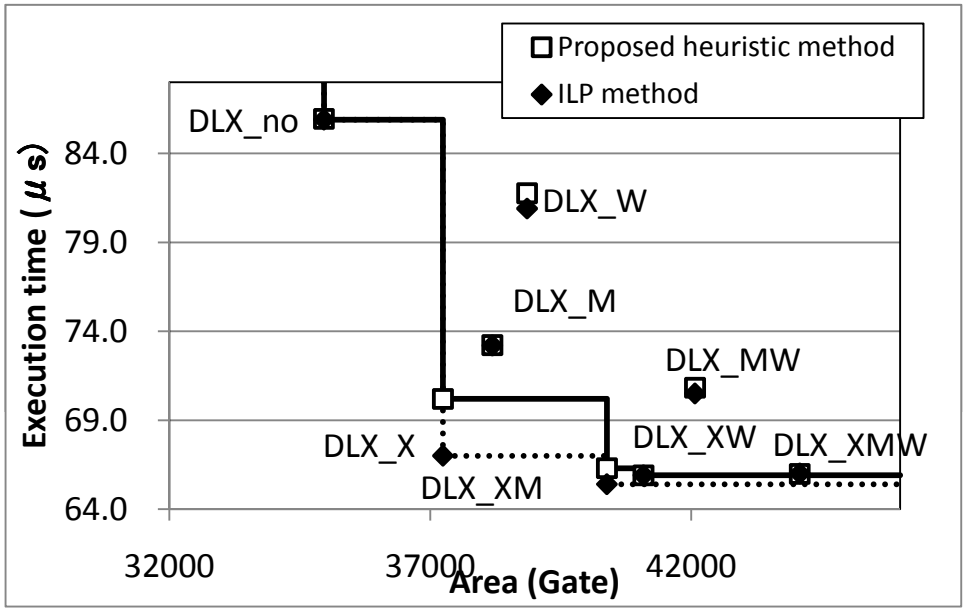


Figure 4.7: Trade-off graph of crc16

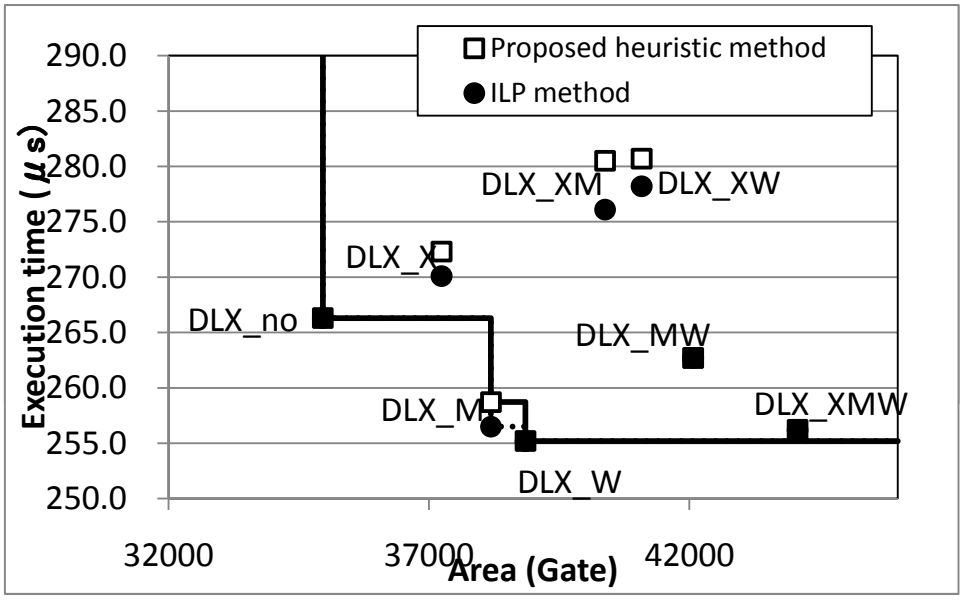


Figure 4.8: Trade-off graph of fir2dim

Table 4.7: Execution cycle for each benchmark program with hazard detection unit

Processor Name	complex_ multiply	complex_ update	con- volution	dot_ product	fft	fir	fir2dim	iir_biquad_ N_sections
DLX_no	1112	1526	5357	1000	49444	4430	37377	13084
DLX_X	1023	1396	4734	917	40491	4135	32978	12564
DLX_M	1029	1331	4796	913	44482	4063	32361	12318
DLX_W	999	1341	5012	951	41324	4271	33720	12417
DLX_XM	1020	1297	4438	864	37001	3927	31307	12772
DLX_XW	1003	1311	4569	889	35879	4084	32675	11838
DLX_MW	994	1272	4741	897	40689	4131	31557	11949

Processor Name	iir_biquad_ one_section	lms	matrix1	matrix2	matrix1x3	n_complex_ updates	n_real_ updates	real_ update
DLX_no	1486	8197	292572	279878	2136	23345	8760	808
DLX_X	1474	7006	266335	247842	1958	21421	7916	663
DLX_M	1427	7067	261855	252404	1834	20822	7924	711
DLX_W	1427	7192	271247	259444	1956	20643	7748	711
DLX_XM	1402	6740	254042	243549	1812	20021	7388	692
DLX_XW	1398	6871	257544	247892	1931	19768	6788	689
DLX_MW	1421	6992	250016	240213	1810	19251	7099	707

Table 4.8: Differences of execution cycles on processor with hazard detection unit from the result of scheduled code

Processor Name	complex_ multiply	complex_ update	con- volution	dot_ product	fft	fir	fir2dim	iir_biquad_ N_sections
DLX_no	-21	81	102	9	975	96	-342	39
DLX_X	-9	90	-42	-3	4446	-231	-891	474
DLX_M	-15	42	105	24	1878	-177	-978	462
DLX_W	-21	60	162	36	2082	114	360	273
DLX_XM	-15	33	-183	-27	2283	-273	-2448	822
DLX_XW	-15	54	-78	-9	909	12	-942	-102
DLX_MW	-12	24	129	18	1173	-9	-1908	114

Processor Name	iir_biquad_ one_section	lms	matrix1	matrix2	matrix1x3	n_complex_ updates	n_real_ updates	real_ update
DLX_no	-30	180	12663	12363	-45	2085	885	-42
DLX_X	6	-84	15087	15687	-45	2652	876	-21
DLX_M	-42	147	6357	8157	-99	2136	888	-24
DLX_W	-24	303	13938	13638	-18	1479	663	-21
DLX_XM	-42	-324	2652	2652	-162	1308	348	-30
DLX_XW	-39	-111	1365	1965	-45	930	309	-30
DLX_MW	-6	126	2319	2019	-126	576	144	-6

Chapter 5

Code Optimization for SIMD Instruction-set Processors

This chapter proposes the data permutation optimization method for SIMD (Single-Instruction Multiple-Data) instructions. This chapter is organized as follows. SIMD instruction-set architecture for embedded processor is summarized in section 5.1. Automatic code generation for SIMD instructions by compiler is described in section 5.2 and data permutation optimization problem and optimization algorithm are described in section 5.3. Experimental results are described in section 5.4. Finally, this chapter is concluded in section 5.5.

5.1 SIMD Instructions in Embedded Processor

SIMD is acronym from Single-Instruction Multiple-Data, which means a number of operations are executed by one instruction. SIMD instructions can be considered as fixed length vector operations whose vector length are short, less than ten in most cases of embedded processors. These instructions load a number of data from memory at once and store them into wide registers. In this thesis, such a wide register is called *SIMD register*. SIMD arithmetic instructions perform their operations for each data in SIMD register in parallel. Memory store operation is also executed in the same way of the load instruction. Figure 5.1 shows datapath of `ADD2`, a SIMD instruction of Texas Instruments C62xx processor [54], that performs two additions

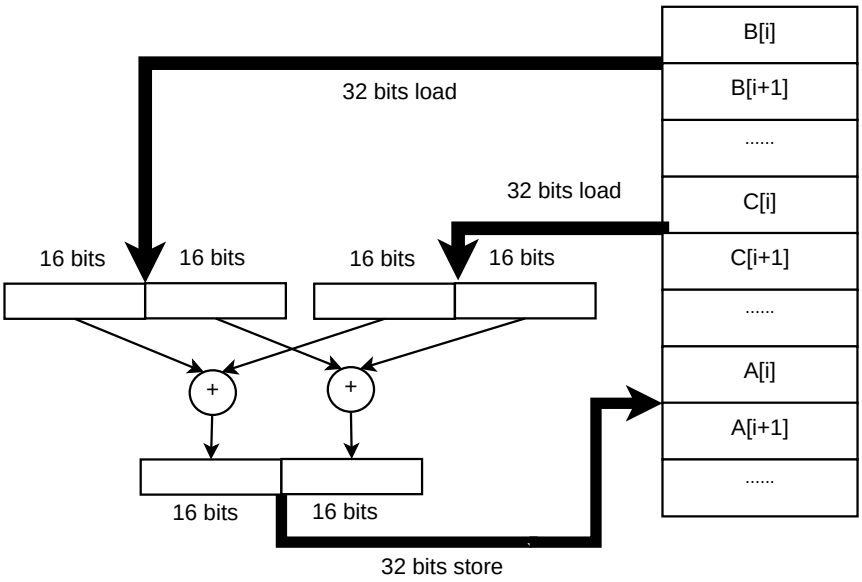


Figure 5.1: Example of datapath of ADD2, a SIMD instruction in TI C62xx.

in parallel. This SIMD instruction reads upper parts of the input SIMD registers and writes the result of addition into the same position of the output SIMD register. The same operation are processed for lower parts of the SIMD registers at the same time. Figure 5.1 also shows an example of datapath of SIMD load and store instructions. Note that memory accesses by SIMD load and store instructions are always contiguous so that they load and store 32 bits data.

Besides arithmetic operations, SIMD instruction set architecture also has instructions for data replacement within SIMD registers. Such instructions are called *permutation instructions*. Figure 5.2 shows an example of dataflow of permutation instructions. In Fig. 5.2, there are two SIMD registers, one contains B[i] and B[i+1], and the other contains C[i] and C[i+1]. If the program is written in `A[i] = B[i] + C[i+1];` and `A[i+1] = B[i+1] + C[i];`, SIMD addition cannot be executed right after loading since position of values are not located regularly. To replacing values by permutation instructions, SIMD instructions can be applied. In general, not all of the sentences in the program can be mapped into SIMD arithmetic instructions directly due to mis-order of index of the array so that permutation instructions is essential for SIMD instruction set architecture.

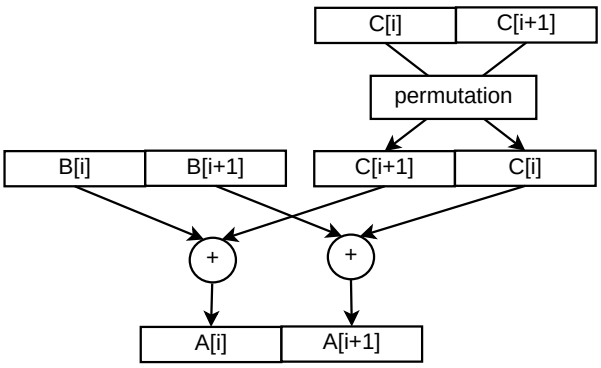


Figure 5.2: Example of dataflow of permutation instructions.

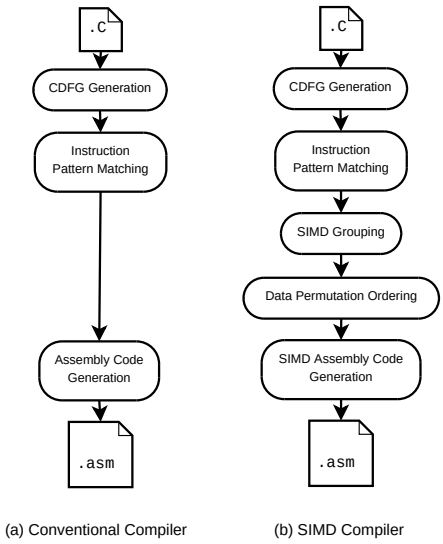


Figure 5.3: Automatic SIMD code generation flow.

5.2 Code Generation for SIMD Processor

In this section, automatic SIMD code generation method from source code is summarized.

Figure 5.3 shows processing flow of SIMD code generation. Instruction pattern matcher generates scalar instructions from CDFG (Control-Data Flow Graph) which is generated by compiler front end. Conventional compiler back end generates assembly code through instruction scheduling and register allocation after pattern matching phase as Fig. 5.3 (a). To generate SIMD assembly code by compiler, SIMD grouping and data permutation ordering phases are

inserted between pattern matching and assembly code generation phase as Fig. 5.3 (b).

5.2.1 Grouping SIMD Operations

In SIMD grouping phase, SIMD instructions are generated using tree matching algorithm [55, 35]. This step constructs a data flow graph (DFG) whose nodes correspond to elements of SIMD operations. Note that these elements are represented as scalar operations in DFG. After DFG construction, a DFG is divided into data flow trees (DFT). Pattern matching, DFG construction and DFT construction methods used by the proposed method are similar to [6]. The elements of SIMD operations are grouped into several SIMD instructions. Finally, a DFG whose nodes are SIMD instructions is constructed.

Groups of operations performed by SIMD instructions are determined as follows.

1. Leaves of DFTs which have the same operations are selected.

2. Selected nodes are grouped as a SIMD instruction if the selected nodes can be performed by one SIMD instruction, otherwise split the nodes into smaller group to fit size of SIMD instruction.

3. Grouped SIMD nodes are removed from the DFTs.

4. Repeat the above steps until all nodes are removed from DFTs.

Note that load and store operations have to be grouped into SIMD instructions only these memory address are contiguous and aligned with memory alignment. This constraints are due to memory access operations of SIMD load/store instructions. Figure 5.4 shows an example of SIMD grouping [8]. The load operations have been removed from DFTs as shown in Fig. 5.4 (a). In this example, a0, a1, a2, and a3 are on contiguous memory addresses. Similarly, b0 and b1 are on. The add operations, nodes with a plus operator, are grouped and added to the DFT in Fig. 5.4 (b). This procedure is called *packing* and generated SIMD node is also called *packed node*. The add nodes will be removed from Fig. 5.4 (a) in the next packing step. After this grouping, the DFT in Fig5.4 (b) is constructed.

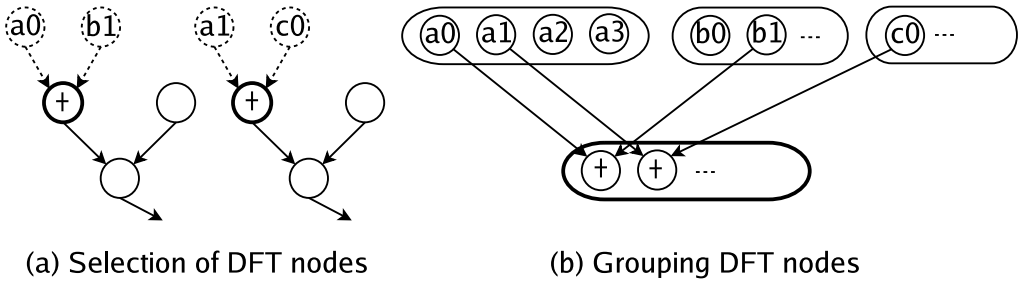


Figure 5.4: Operation grouping

5.2.2 Data Permutation Ordering

After SIMD grouping, SIMD DFG of the input program is constructed with SIMD DFTs. Each SIMD node in SIMD DFG has more than one scalar operations executed in parallel. However, data order for each SIMD node is not yet determined except for SIMD load/store nodes. In data permutation ordering phase, data order of operations for each SIMD node is determined to minimize the number of permutation instructions. Figures 5.1 and 5.2 show the importance of data permutation ordering. In the case of Fig. 5.2, permutation instructions are required to arrange the order of C[i] and C[i+1]. After this permutation, ADD2 instruction can be executed as Fig. 5.1.

Although data permutation ordering is an important optimization problem for SIMD code generation, there are few studies focused on this problem. First study of data permutation ordering is shown in [37]. This method generates SIMD code considering the order of each data by integer programming. However, this method only generates 2-way SIMD code since it focuses on TI 62xx processor which has only 2-way SIMD instructions. [6] supports more than 2-way SIMD instructions to split SIMD code generation problem into SIMD grouping and data permutation ordering. SIMD instructions are generated by grouping operations in a basic block represented by DFT. Data permutation ordering is also solved by integer programming as [37]. A heuristic method to solve data permutation ordering is shown in [7]. This method optimizes data order of SIMD nodes for each basic block to propagate data order across statements in the input program. It aims to reduce the number of permutations to merge consecutive permutations. There are no global data permutation ordering method so that I propose a global optimization method described in section 5.3.

5.2.3 Data Permutation Instruction Sequence Generation

After data permutation ordering, data permutation instruction sequence is generated for each mis-order point.

For each mis-order point where the pair of SIMD registers have different data order, instruction sequence for data permutation is required. Since permutation instructions do not contribute for program execution result, it is an important optimization problem to minimize the number of permutation instructions. Hereafter, the contents of packed data are called *permutation* because they are naturally represented by permutations. Data permutation instruction sequence is generated by the following steps:

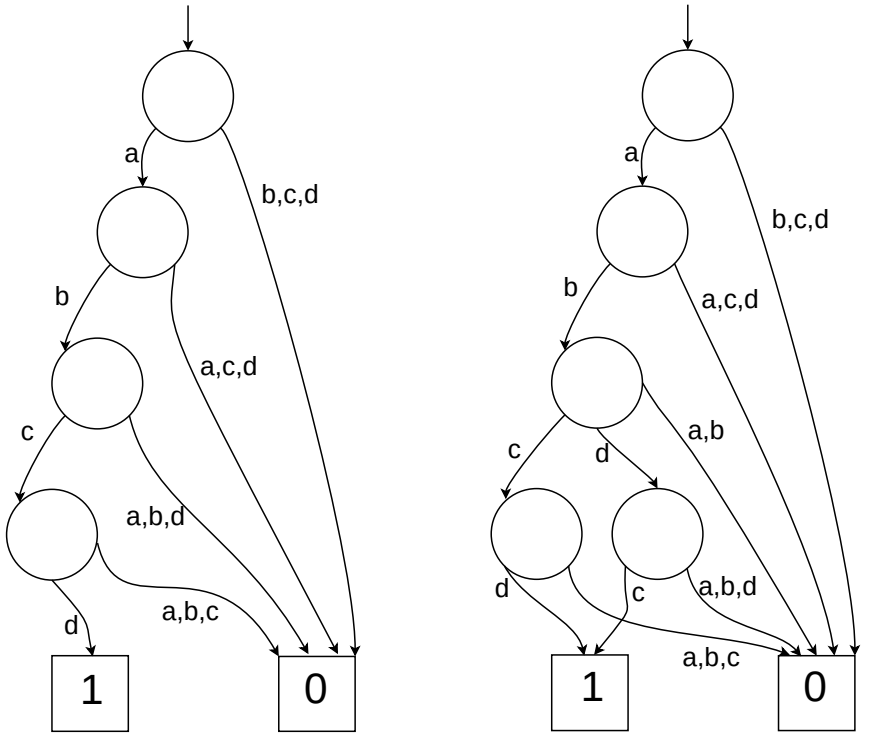
1. generate all permutation patterns from source permutations using available permutation instructions.

2. construct expression tree which represents permutation steps from source permutation to output permutation pattern.

3. generate permutation instruction sequence.

These steps are processed based on Multi-valued Decision Diagram (MDD) [8]. Multi-valued Decision Diagram represents binary-valued output function which takes multi-valued input parameters. In this problem, MDD represents a set of permutation patterns and manipulation on MDD correspond to operations on the sets of permutations. Figure 5.5 (a) shows a MDD of {`abcd`} and Fig. 5.5 (b) shows a MDD of {`abcd`,`abdc`}. Fig.5.5 (b) is constructed by the logical-or of MDDs representing {`abcd`} and {`abdc`}, which corresponds to the union of {`abcd`} and {`abdc`} [8]. Logical-and operation on MDD corresponds to the intersection operation, similarly.

The generation algorithm is summarized as follows; detail of the generation algorithm is described in [8]. The generation algorithm calculates the minimum length of permutation instruction sequence based on MDD manipulation. The algorithm calculates the set of permutations from the previous set of permutations and given set of permutation instructions. In first step, this step starts from the source permutation set. This loop continues until the required permutation is found or the set of permutations equals to the previous set of permutations. When



(a) Set { abcd } in MDD (b) Set { abcd, abdc } in MDD

Figure 5.5: MDDs for { abcd } and { abcd,abdc }

the required permutation is found, the algorithm generates permutation instruction sequence to backtrack the steps of permutation set generation to the source permutations.

5.3 Data Permutation Optimization in SIMD Registers

As summarized in section 5.2, instruction sequences for data permutation are generated where the data orders differ between input and output SIMD registers. If program is designed to consider data parallelism, permutation instructions may be not required. However, automatic SIMD code generation compilers have to compile arbitrary program codes including sequential function codes or legacy codes which are not considered about parallelism. Permutation instruction generation method in [8] focuses on minimizing the number of permutation instructions for each mis-order point. To improve efficiency of the execution cycles, minimizing the number of mis-order point in SIMD DFGs have to be considered.

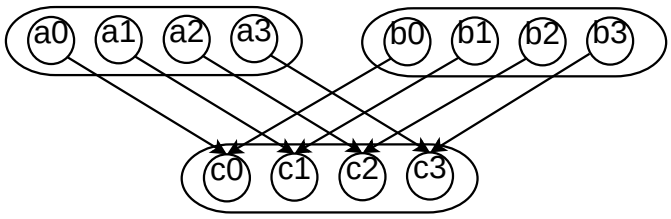


Figure 5.6: An example of SIMD DFG

5.3.1 Data Permutation Optimization Problem

In this section, data permutation optimization problem is discussed. Objective of the optimization problem is to minimize the number of mis-order points which require permutations on input SIMD DFG generated by SIMD grouping phase.

5.3.1.1 SIMD DFG

SIMD DFG represents dependences among SIMD nodes and each operation in SIMD nodes. Figure 5.6 shows an example of SIMD DFG. Small circles correspond to operations and rounded rectangles which encloses some operations are SIMD operations. Source and sink nodes represent load and store instructions, respectively. Each directed edge shows dependence between operations. Each node in SIMD DFG has a number of operations. In the grouping example presented in Fig. 5.6, all SIMD nodes contain four operations. In this thesis, the number of operations in the SIMD instructions or width of SIMD operations is called *pack count*. Maximum pack count is limited by register size and data size for each operation. If each operation treats 8 bit data and these operations are packed in 64 bit register, maximum pack count of the SIMD node equals 8.

There are three types of dependences between SIMD nodes;

- Regular dependence,
- Partial dependence, and
- Spreading dependence.

Regular dependence means that all operations in the precedence SIMD node connects operations in the succeeding SIMD node. Both pack counts of the SIMD nodes are the same and there

are one-to-one correspondence hence bijection. Figure 5.6 shows an example of regular dependence. There are two source SIMD nodes and each source SIMD node has regular dependence onto the sink SIMD node. Most of dependences in SIMD DFG are regular dependences and permutation may be not required to adjust data order of the operations between precedence and succeeding SIMD nodes. On the other hand, partial and spreading dependences must require permutations between them. In this thesis, partial and spreading dependences are classified as irregular dependences. Partial dependence although connects in an injective manner, however, not in an bijective manner. Partial dependence occurs in two cases; pack counts differ between the SIMD nodes, or not all the operations connects onto the succeeding nodes. Figure 5.7 (a) shows the first case. Two source SIMD nodes pack two operations and connects onto one sink SIMD node which pack count is 4. Operations `a0`, `a1`, `b0`, and `b1` are the precedence operations onto `c0`, `c1`, `c2`, and `c3`, respectively. In the second case, Fig. 5.7 (b), only 2 of 4 packed operations from `aX` and `bX` are connected onto `cX` operations. Spreading dependence do not has one-to-one connection though partial dependence has. The operation `a0` connects all of operations `cX` in Fig. 5.8 so that there is spreading dependence while the other source SIMD node which packs operations bX have regular dependence onto the sink SIMD node. Partial and spread dependences can be combined like Fig. 5.9. In this case there are two operations which have dependences onto succeeding operations and each of them have two succeeding operations.

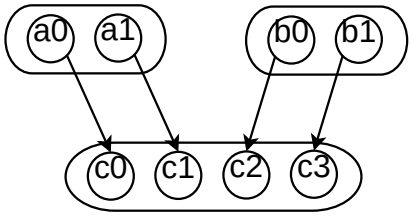
For all SIMD nodes, data orders of the operations are not determined at first, except for load nodes and store nodes due to memory address constraints.

5.3.1.2 Computational Complexity of the Problem

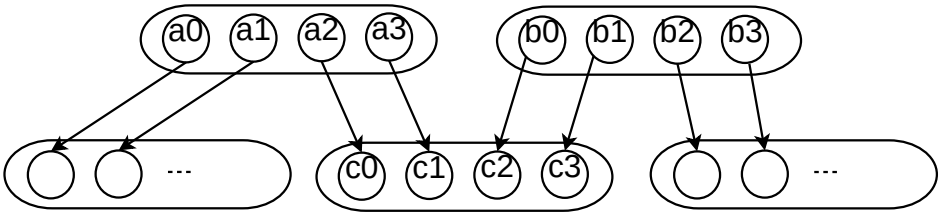
As Ren, et al. described, data permutation optimization problem is a NP-hard problem [7].

5.3.2 Data Permutation Optimization Method

In the proposed method, optimizer with heuristic methods described in this section solves data permutation optimization problem based on SIMD DFG generated by the whole input program.



(a) Differences of the number of PACK nodes



(b) Propagation of partial PACK nodes

Figure 5.7: Examples of partial propagation

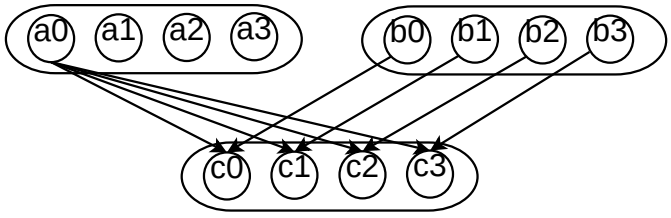


Figure 5.8: Example of spreading propagation

The proposed data permutation optimization algorithm are consists of two strategies based on propagation; the most frequently used position where sources and destinations are arranged is selected for each operation [8]. Figure 5.10 shows an example of propagation with a part of SIMD DFG [8]. The most left add operation has two sources `a1` and `b0`, one destination `d1`. The order in the grouped node of `a1` is the second element, `b0` is the first and `d1` is the second. Therefore, the most left add operation in Fig. 5.10 (a) is reordered to the second in the grouped node as shown in Fig. 5.10 (b). Similarly the second add operation is reordered to the most left in the grouped node.

In the proposed optimization method applies two strategies; one is data order propagation

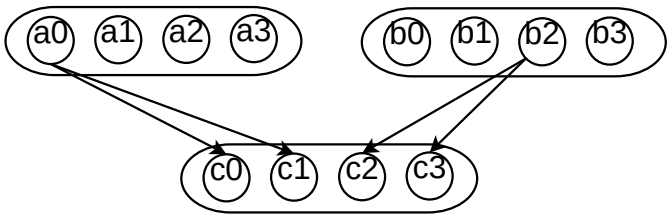


Figure 5.9: Composition of partial and spreading propagations

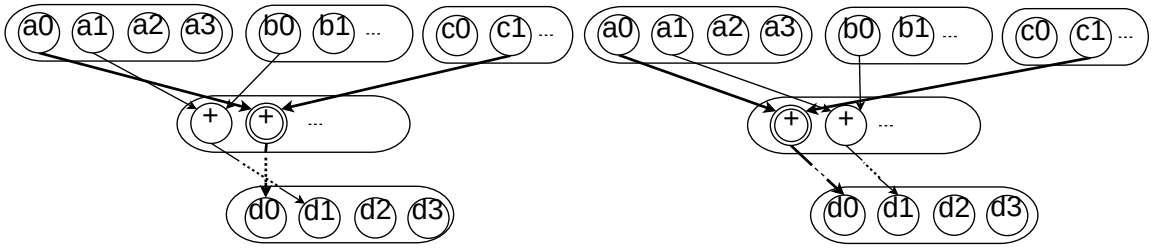


Figure 5.10: Operation ordering

from SIMD load/store instructions, the other is data order propagation from neighbor nodes. The algorithm applied the better solution as a result of the optimization.

5.3.3 Data Order Propagation from SIMD Load Nodes and Store Nodes

The order of SIMD nodes are determined by load and store instructions since SIMD load and store instructions are performed on contiguous memory locations. This strategy is implemented as [8]. The basis of this strategy is to adjust data order of SIMD nodes according to the dominated order of SIMD load nodes and store nodes. For each SIMD load node and store node, the order of the node is propagated onto all SIMD nodes except the other SIMD load/store nodes. Order of SIMD load nodes and store node is represented by position of each operation in SIMD load nodes and store nodes. The most left operation is numbered by 0, and the most right one is $p - 1$, where p means the pack count of the SIMD load/store nodes. This position number of operation are propagated according to the dependence edges. While visiting the other SIMD nodes from a SIMD load/store node, The order of a SIMD load node is propagated to following edges in the forward direction. On the other hand, the order of a SIMD store node is propagated

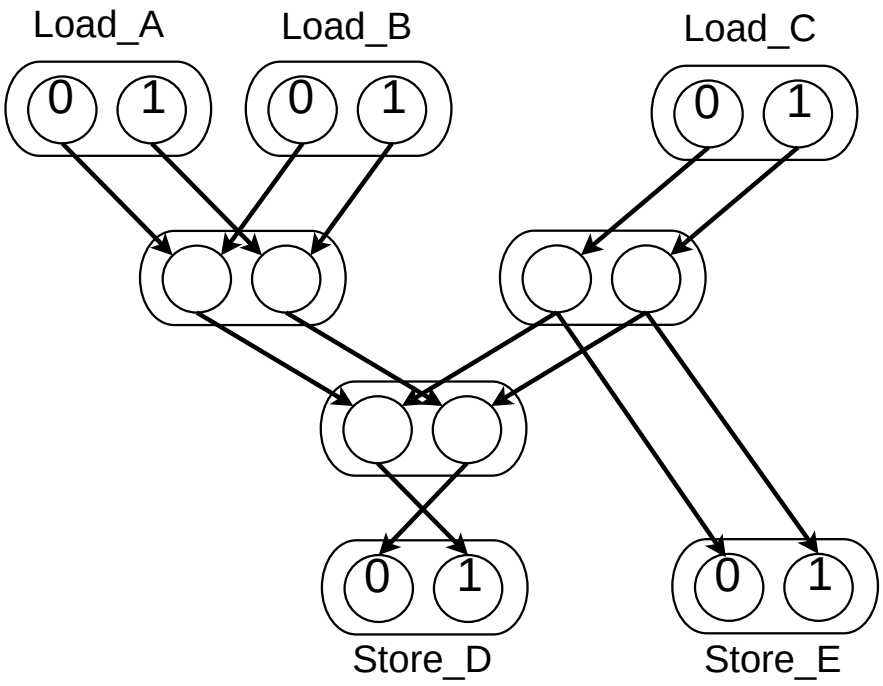


Figure 5.11: Example of propagation from SIMD Load/Store Nodes

following edges in the backward direction. For example, SIMD DFG shown in Fig. 5.11 has three SIMD load nodes and two SIMD store nodes. The remaining three SIMD nodes which pack arithmetic operations are propagation target SIMD nodes. All dependences are regular propagation in this example. Figure 5.12 shows the propagation result from `Load_A`. From `Load_A`, only two SIMD nodes are visited and propagated by the same order of `Load_A`, $(0, 1)$. The order of `Load_A` is not propagated onto the rest of one SIMD node. In contrast, all three propagation target SIMD nodes are visited from `Store_D` to traverse edges backward as shown in Fig. 5.13. Contrary to the case of `Load_A`, `Store_D` propagates $(1, 0)$ caused by crossed edges from the precedence of `Store_D`. Similarly, the orders of `Load_B`, `Load_C`, and `Store_E` are propagated. Finally, the order of each target SIMD nodes are determined by majority rule. This result is shown in Fig. 5.14 with voting result of the order for each the target SIMD node. In this case, all three target SIMD nodes have the same order, $(0, 1)$.

Figure 5.15 shows the algorithm of this strategy. Computational complexity of this algorithm

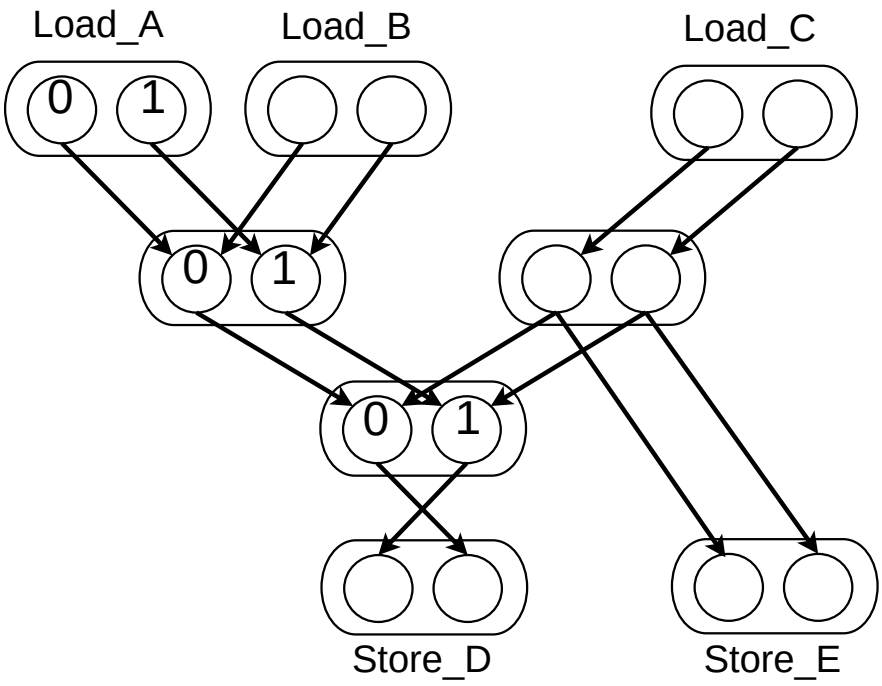


Figure 5.12: Propagation from Load_A

is $O(N^2)$ where N means the set of nodes of the input SIMD DFG since SIMD nodes in the DFG are visited for each SIMD load/store nodes. In practical, the number of SIMD load nodes and store nodes in SIMD DFG is less than $O(N)$, such as $(\log N)$ or constant, so that the complexity of the strategy is smaller than $O(N^2)$.

5.3.4 Data Order Propagation from Neighbor Nodes

Although the previous strategy propagates order of SIMD nodes through irregular dependences, inappropriate result may be obtained so that another propagation strategy is required. To solve this problem, the another data order propagation strategy to propagate data orders from neighbor SIMD nodes is proposed. This strategy is similar to data order propagation from load nodes and store nodes. The order of SIMD load/store nodes are also propagated by this strategy, however, the order of neighbor SIMD nodes are applied as source order for each propagation and majority rule is applied only on junction nodes which have more than two precedence nodes.

Figure 5.16 shows an example of order decision on junction nodes. Similar to the previous strategy, the order of junction nodes are adjusted by dominated order of the precedence nodes.

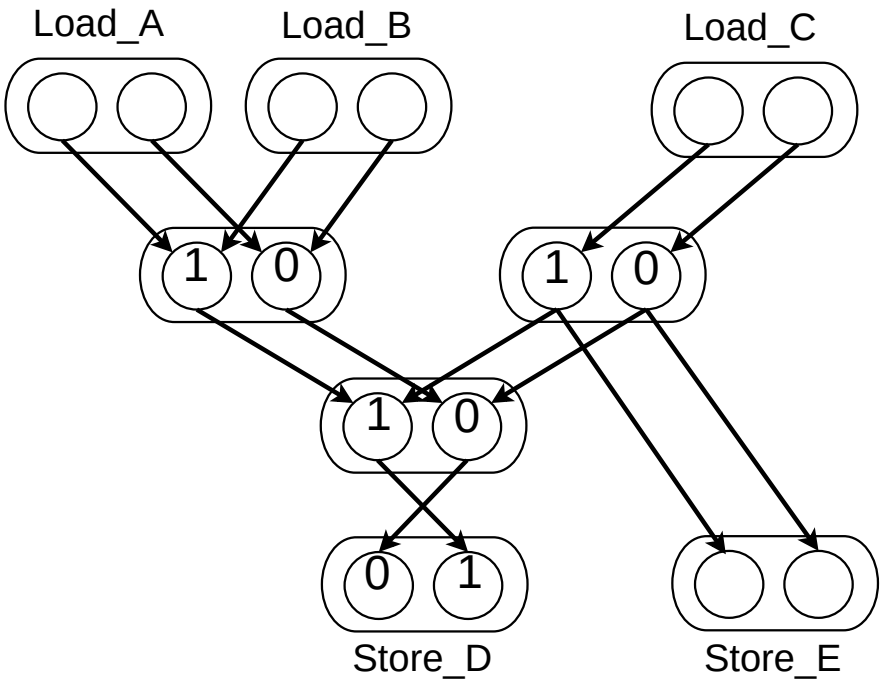


Figure 5.13: Propagation from Store_D

In this example, the order of `Node_D` is the same as `Node_A`, not as `Node_B` and `Node_C` in Fig. 5.16 (a). If the order of `Node_D` is fixed as Fig. 5.16 (a), two permutations are required. Both `Node_B` and `Node_C` have the same order so that one permutation can be reduced to adjust the order of `Node_D` with `Node_B` and `Node_C`.

The proposed method targets on SIMD DFG of the whole program so that branch path and loop path may exist in SIMD DFG. Figure 5.17 shows an example of SIMD DFG with the pair of branch paths. A number sequence written in each node means permutation order. This DFG assumes that the pack count of all SIMD nodes equals to four. `Node_X` is connected with two source SIMD nodes and one sink SIMD node. Inside of the branch paths, the orders of the source and sink SIMD nodes are corresponded. However, the source SIMD node outside of the branch paths has different order from the others. To adjust the order of `Node_X` to the outside SIMD node, two permutations are required as shown in Fig. 5.17 (a). The number of permutations can be reduced to adjust `Node_X` according to the inside SIMD nodes inside of the branch paths as shown in Fig. 5.17 (b). It costs only one permutation between the SIMD

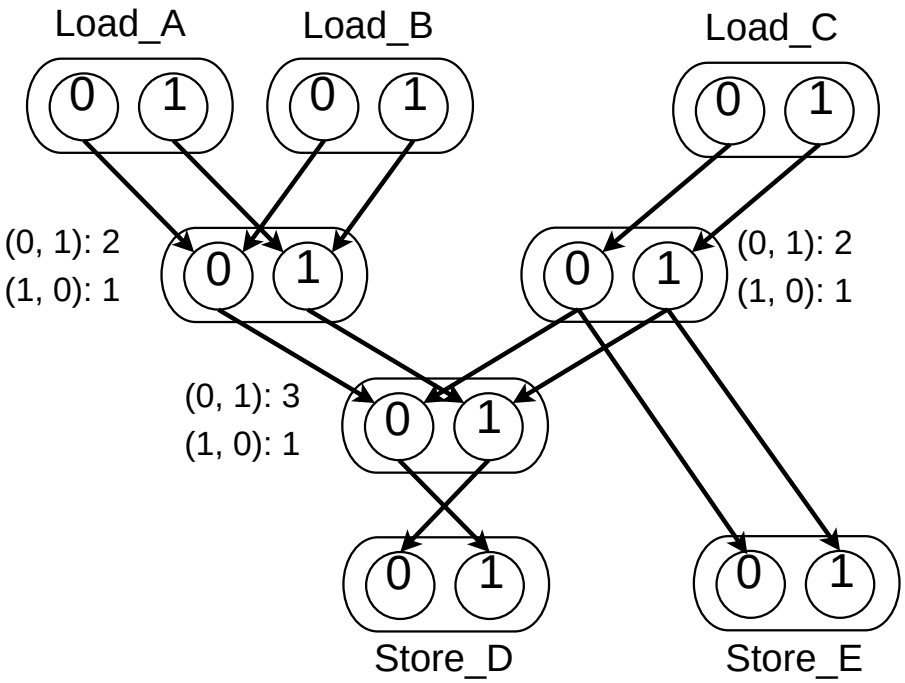


Figure 5.14: Result of the propagation from Load/Store nodes

source node outside of the branch paths.

Data order propagation from neighbor nodes strategy supports propagation for these paths. Both branch and loop paths are the same circulation form to ignore direction of the edge in SIMD DFG. Figure 5.18 shows the concept of propagation on branch or loop paths. For each propagation, to find the branch or loop paths, neighbor nodes are visited from the current target SIMD node of which the algorithm prepares to determine the order. The algorithm searches to find a pair of SIMD nodes which are start and end node of two paths, one path includes the target SIMD node and the another path does not. If found such a pair, the order of the target SIMD node is adjusted according the pair. If the orders of the pair differs, one of the order is selected according to the number of nodes which are directly connected with the pair. Figure 5.19 shows the algorithm of this strategy. Line 03 can be calculated using Floyd-Warshall algorithm by $\theta(N^3)$ [53]. There are two loops in the main body of the algorithm and one traversing step in line 08. Traversing cost is $O(E)$ so that the complexity of the algorithm is $O(N^2E)$. In practical SIMD DFG, $O(E)$ can be regarded as $O(N)$ hence the complexity of

Input: SIMD DFG $G(V, E)$
V: A set of SIMD nodes E: A set of directed edges in DFG
Output: SIMD DFG with reordered SIMD nodes

```
01: begin
02:     for each ( root nodes v_r in G ) {
03:         Traverse G in forward and propagate the order of v_r;
04:     }
05:     for each ( leaf nodes v_s in G ) {
06:         Traverse G in backward and propagate the order of v_s;
07:     }
08:     for each ( nodes except roots and leaves v in G ) {
09:         Determine the order of v according to majority rule;
10:     }
11: end
```

Figure 5.15: Algorithm of data order propagation from SIMD Load/Store nodes

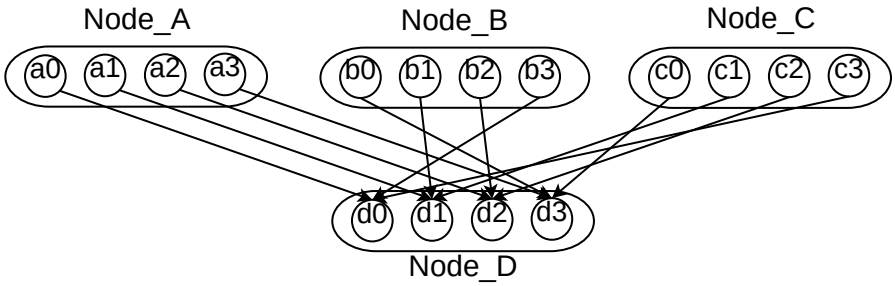
the algorithm is $\theta(N^3)$.

5.4 Experiments

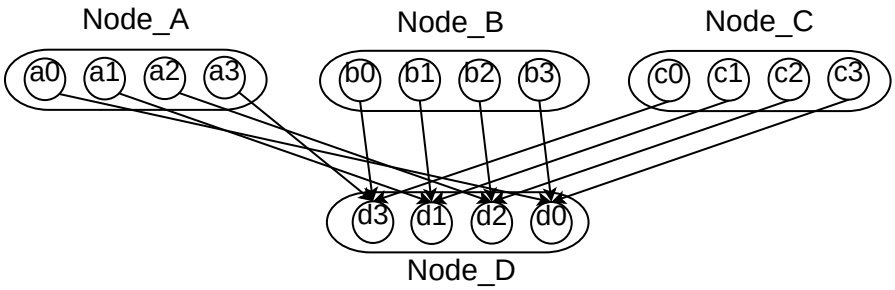
In this section, the experimental results are presented. The proposed data permutation optimization method is implemented in automatic SIMD code generator which is proposed in [8]. This SIMD code generator targets on Media embedded Processor, MeP [56, 57].

5.4.1 Target Processor Architecture

To evaluate the proposed optimization method, configurable processor with SIMD instruction set is required. MeP was selected as the target processor in this experiments. In this experiment, MeP consists of two cores; one is MeP core, the base processor of MeP, the other is SIMD co-processor. MeP core processor is a 32 bit RISC processor which has single-scalar arithmetic



(a) Before re-order: 2 misorders occur



(b) After re-order: only 1 misorder occurs

Figure 5.16: permutation adjustment on a junction node

instructions, logical instructions, branch control instructions and so on, including coprocessor move instructions [58]. Fig. 5.4.1 shows the architecture of MeP [56, 57]. There are several configuration options for MeP, such as embedding user designed logics, adding coprocessors, and so on. SIMD coprocessor for MeP is one of the customize option of MeP configurable processor. The MeP core and the coprocessor share a local memory to access directly through 64 bit data bus. Fig. 5.21 shows the architecture of SIMD coprocessor [56, 57]. SIMD coprocessor has 2-way issue instruction pipeline and 64 bit SIMD register file and 2 accumulator for multiplication. SIMD coprocessor works with MeP core in parallel so that the MeP in this experiment can be regarded as a 3-way VLIW processor. SIMD coprocessor supports 8-parallel byte, 4-parallel halfword, and 2-parallel word SIMD instructions, including addition, subtraction, shifting, logical, and permutation operations. Multiplication operations uses accumulator register which have 256 bit width. An accumulator consists of 8 registers, each of them has 32 bit width. In this experiments, multiplication nodes were mapped on pseudo SIMD

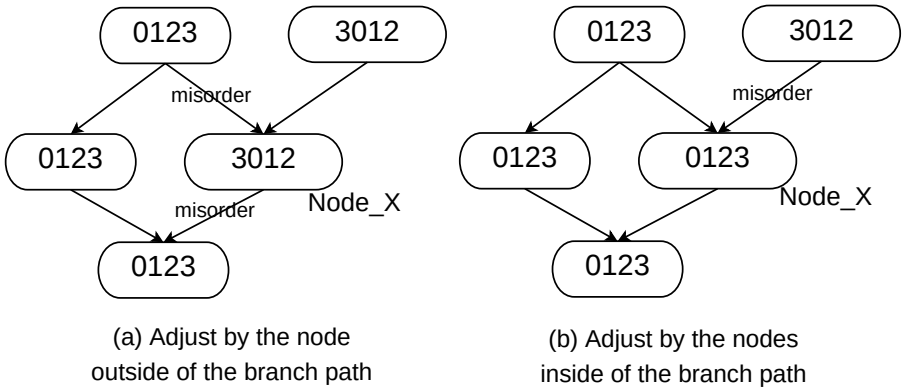


Figure 5.17: An example of permutation adjustment on branch paths

multiplication instructions to hide accumulator from the architecture model.

There are two types of permutation instructions on SIMD coprocessor; fixed pattern permutation instruction type and arbitrary permutation instruction type. Both instruction types take two input SIMD register and write the result into a SIMD register. Fixed pattern permutation instructions are classified by following properties:

- data size of each element (byte, halfword, or word),
- where of the data are read (upper or lower half of the input registers), and
- how the data are replaced (block or alternative).

Figures 5.22 and 5.23 show fixed pattern and arbitrary permutation instructions of SIMD coprocessor for 4-parallel halfword operations.

In the experiments, 10 test programs for permutation optimization were used to evaluate. Each test program has several control flows and permutation patterns. The proposed method was implemented in the automatic SIMD optimizer [8]. This optimizer is a part of optimizing C compiler for MeP with SIMD coprocessor, which generates SIMD instructions after loop unrolling. The compiler systems used in the experiments were provided by Toshiba Corp.

In these experiments, permutation optimization methods were evaluated to compare the results of two strategies. Execution cycles and the number of permutation instructions were compared. Execution cycles were obtained by the simulator which were also provided by Toshiba

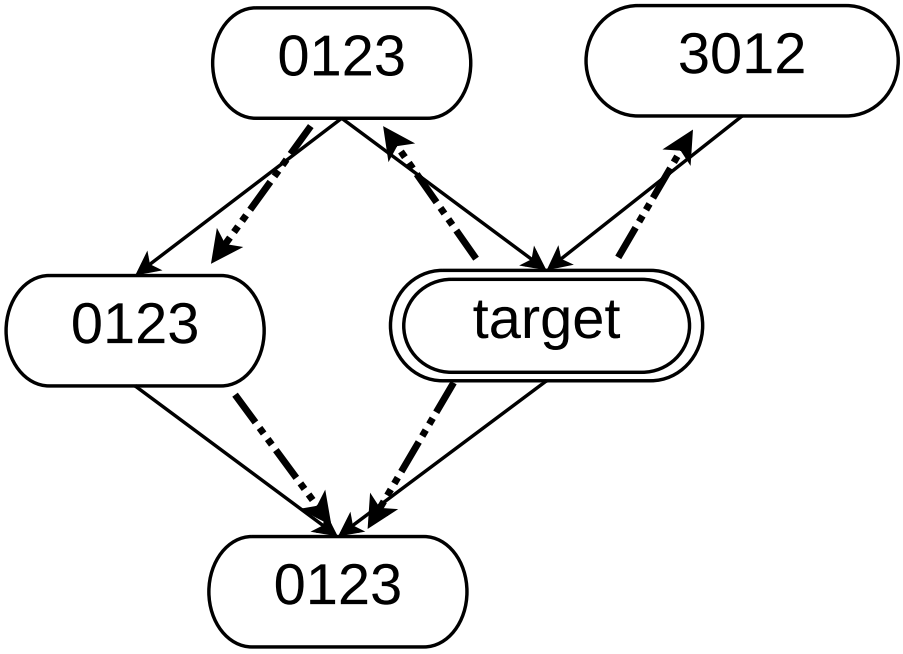


Figure 5.18: Data permutation adjustment method inside of the loop

Corp. The number of permutation instructions for each test program was counted from assembly code generated by the compiler with SIMD optimizer and the proposed method. Compilation and simulation were performed on Intel Xeon 2.8 GHz processor with 2GB of memory and RedHat Enterprise 3 operating system.

5.4.2 Experimental Results

Figure 5.24 shows the number of execution cycles of the optimized test programs. The white bars represent the execution cycles with data order propagation from load/store nodes and the gray bars represent that with data order propagation from neighbor nodes. This result shows that neighbors propagation method reduced more execution cycles in average. However, load/store propagation is superior than neighbors propagation with `test7` and `test8`.

Table 5.1 shows the number of permutation instructions of the optimized test programs. The second and third columns represent the numbers of fixed and arbitrary permutation instructions of load/store propagation while the fourth and fifth columns represent that of neighbors propagation. These results also shows that neighbors propagation method reduced more permutation

Table 5.1: Comparison of the number of permutation instructions with two strategies

	Load/Store Propagation		Neighbors Propagation	
	# of fixed perm.	# of arbitrary perm.	# of fixed perm.	# of arbitrary perm.
test0	12	3	10	3
test1	12	3	6	4
test2	10	3	4	3
test3	2	2	2	2
test4	14	4	12	4
test5	8	4	6	3
test6	4	4	2	4
test7	8	1	8	3
test8	8	4	8	4
test9	2	3	0	4

instructions in average. `test7` was the only program for which neighbors propagation method obtained inferior result. The reason of the result with `test7` was that the order of one SIMD node in SIMD DFG of `test7` was not optimized correctly. This SIMD node was a precedence node of a junction node, however, it was not on the branch nor loop paths. In this case, the current neighbors propagation decided inappropriate order on this node though the same case in other test programs generated better results. The results of `test8` shows that there were no difference between the two strategies. The reason why the result happened is due to control flow and trace of `test8`. The permutation points can move beyond the border of basic block by global optimization method. In the case of `test8`, a permutation point was moved into the basic block which was executed more times than the previous basic block. Solving these problems is in future works.

5.5 Summary

In this chapter, data permutation optimization method for SIMD instructions is proposed. In experiments, . Experimental results also showed the advantage of the proposed scheduling method compared with hazard detection unit.

```
Input: SIMD DFG G(V, E)
V: A set of SIMD nodes E: A set of directed edges in DFG
Output: SIMD DFG with reordered SIMD nodes

01: begin
02:    // Initialization: θ(N^3)
03:    Calculate distance between all pairs of nodes;
04:    for each ( root and leaf nodes v_t in G ) {
05:       v_n = one of the neighbor node of v_t;
06:       while ( the order of v_n is not determined ) {
07:          if (v_t has other neighbor nodes) {
08:             (v_s, v_e) = the pair of start and end nodes of the nearest branch or loop paths
of v_n;
09:             if ( (v_s, v_e) are not found ) {
10:                Propagate the order of v_t into v_n according to the majority rule;
11:             }
12:             else {
13:                Propagate the order of v_t into v_n according to the order of (v_s, v_e);
14:             }
15:             v_t = v_n;
16:          }
17:          else {
18:             v_n = one of the neighbor node of v_t;
19:             Propagate the order of v_t into v_n;
20:             v_t = v_n;
21:          }
22:       }
23:    }
24: end
```

Figure 5.19: Algorithm of data order propagation from neighbor nodes

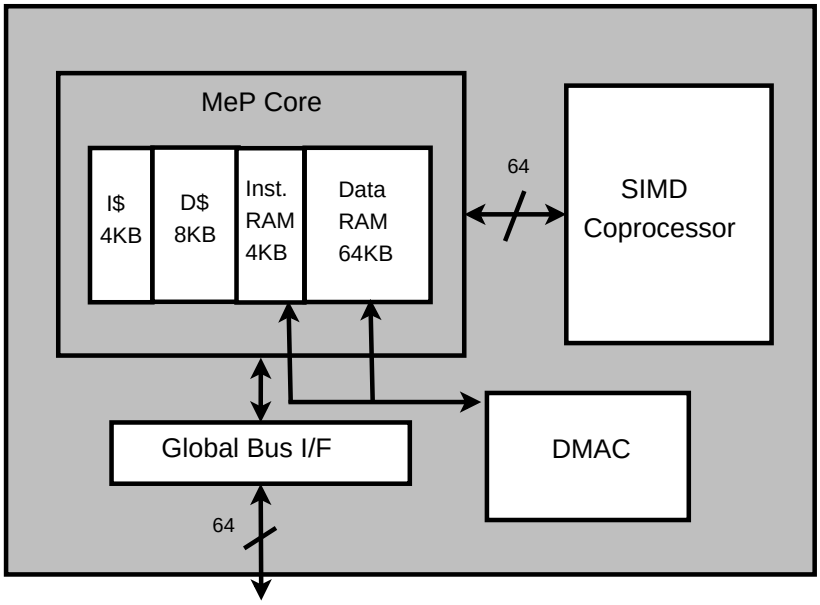


Figure 5.20: MeP processor architecture

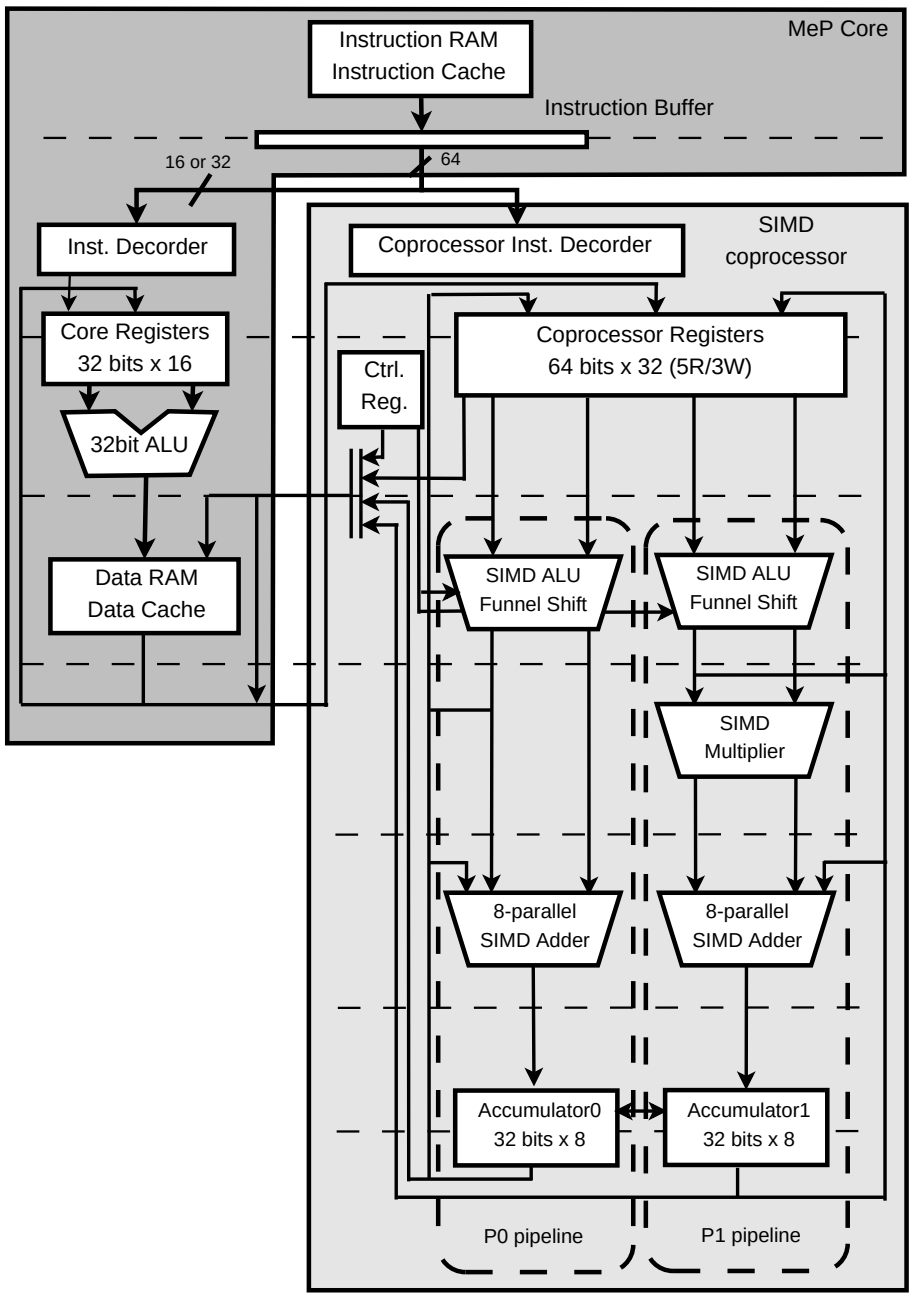


Figure 5.21: SIMD coprocessor architecture

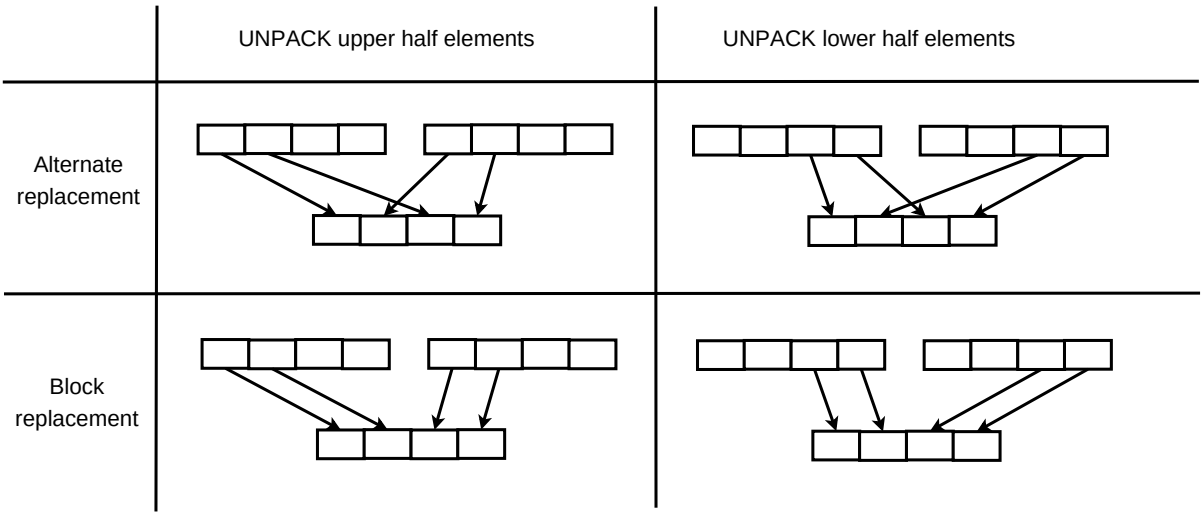


Figure 5.22: Patterns of fixed permutation for 4-parallel halfword operations

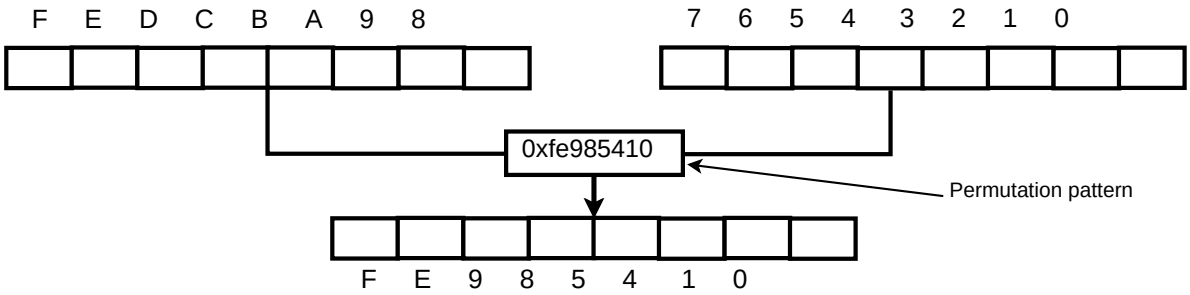


Figure 5.23: Arbitrary permutation instruction

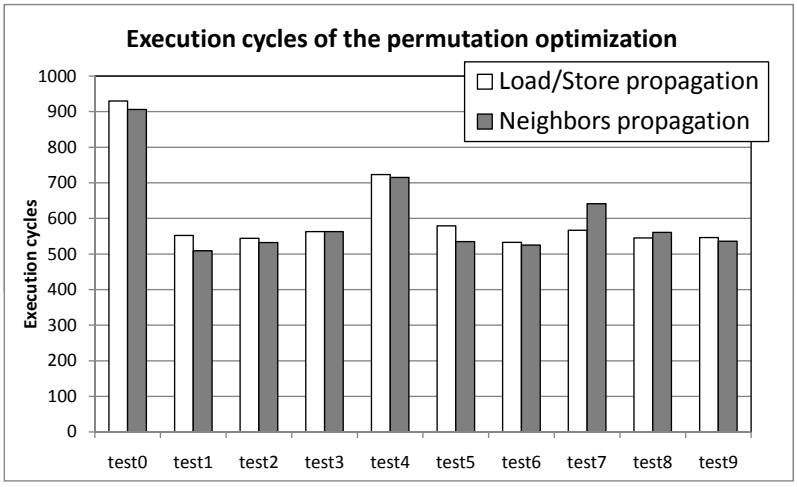


Figure 5.24: Comparison of the number of execution cycles with two strategies

Chapter 6

Conclusion and Future Work

This chapter concludes this thesis. Future direction of compiler optimization for application specific instruction-set processors is also discussed.

6.1 Conclusion

This thesis describes two types of code optimization method for configurable processors. First contribution of the thesis is instruction scheduling for partial forwarding processor. Although partial forwarding technique is supposed to be utilized with compiler support, in the aspect of design space exploration, there are rarely previous works about compiler optimization method for partial forwarding. In this thesis, optimal instruction scheduling method is proposed at first. The proposed instruction scheduler by using integer linear programming method generated more efficient code than the simple list scheduler which supports partial forwarding. Heuristic scheduling method for partial forwarding processors is proposed in the next step. Experimental results show that the proposed heuristic scheduler generates efficient code for partial forwarding processor and improves execution time. In particular, some benchmarks runs faster on partial forwarding processor than the full forwarding processor.

The second contribution of the thesis is code optimization for media processors. Exploiting data level parallelism is important for media processor, especially used in embedded systems. In this thesis, data permutation optimization problem is discussed. Obstacles when optimizing data permutations are discovered and heuristic optimization method is proposed. Experimen-

tal results show that the proposed method achieved reducing the number of data permutation instructions.

6.2 Future Work

The future work includes following items.

6.2.1 Algorithm Expansion

The code optimization methods proposed in this thesis have some constraints. Multi-issued processors such as VLIW processor are widely used in recent embedded systems such as MeP [56, 57]. However, the proposed instruction scheduling methods for partial forwarding do not support multi-issue processors and parallelism. Besides, a number of dedicated forwarding architectures which differs from the partial forwarding architecture model in this thesis are proposed [29, 30, 31]. Instruction schedulers for these architectures have to consider different characteristics from the proposed instruction scheduling methods discussed in this thesis. In out-of-order issue processor, order of the fetched instructions are shuffled so that reordering for partial forwarding in instruction buffer is required. Expanding scheduling algorithms for such dedicated forwarding architecutures is a challenging study. Design space exploration of partial forwarding for such modern architectures is also not considered so that improved instruction scheduling algorithm is required to enable to customize forwarding paths on multi-issued configurable processors.

the proposed SIMD data permutation optimization method also has some constraints without the future works described in chapter 5. SIMD multiplication instructions on MeP coprocessor use accumulator for its calculation. However, the current proposed method regards these instructions as the same format of other arithmetic instructions which use SIMD registers. There are some performance losses so that improved optimization algorithm for SIMD instructions with accumulator is required.

6.2.2 Theorical analyzation of the heuristic algorithms

Both instruction scheduling for partial forwarding and SIMD data permutation optimization are NP-hard problems so that heuristic optimization methods are proposed in this thesis. However, approximation ratio of the heuristic algorithms are only considered from the experimental results and there are no theorical analyze.

6.2.3 Compilation Techniques for Low Power

Low power consumption is desired property for embedded systems, especially battery driven PDAs or embedded chips with the small battery in the buildings to co-operate other chips for sensor area network. Although power consumption were reduced in the experiments as a result of code optimization, the code optimization methods proposed in this thesis do not consider power consumption directly in the optimization methods. Some compiler optimization methods focused on power consumption are proposed so that it is important for design space exploration to consider power consumption in the optimization methods.

Bibliography

[1] ITRS Team, “International technology roadmap for semiconductors.” http://public.itrs.net, 2010.

[2] R.E. Gonzalez, “Xtensa: a configurable and extensible processor,” IEEE Micro, vol.20, no.2, pp.60–70, 2000.

[3] M. Ganapathi, C.N. Fischer, and J.L. Hennessy, “Retargetable compiler code generation,” ACM Computing Surveys, vol.14, pp.573–592, December 1982.

[4] C. Liem, Retargetable Compilers for Embedded Core Processors, Kluwer Academic Publishers, 1997.

[5] P.S. Ahuja, D.W. Clark, and A. Rogers, “The performance impact of incomplete bypassing in processor pipelines,” Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28), pp.36–45, 1995.

[6] A. Kudriavtsev and P. Kogge, “Generation of Permutations for SIMD Processors,” Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp.147 – 156, June 2005.

[7] G. Ren, P. Wu, and D. Padua, “Optimizing data permutations for SIMD devices,” Proceeding of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI ’06), pp.118–131, 2006.

[8] H. Tanaka, Y. Takeuchi, K. Sakanushi, M. Imai, H. Tagawa, Y. Ota, and N. Matsumoto, “Generation of pack instruction sequence for media processors using multi-valued deci-

sion diagram," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol.E90-A, pp.2800–2809, December 2007.

[9] M. Imai, "ASIP Meister: A Configurable Processor Core Development System," Proceedings of ITI 3rd International Conference on Information & Communications Technology (ICICT), 2005.

[10] P.M. Sailer and D.R. Kaeli, The DLX Instruction Set Architecture Handbook, Morgan Kaufmann Publishers, Inc., 1996.

[11] S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai, "A Compiler Generation Method for HW/SW Codesign Based on Configurable Processors," IEICE Transactions on Fundamentals of Electronics, Communication and Computer Sciences, vol.E85-A, no.12, pp.2586–2595, 2002.

[12] I. Hirofumi, H. Takuji, T. Hiroaki, S. Jun, S. Keishi, T. Yoshinori, and I. Masaharu, "A highly extensible base processor for short-term ASIP design," IEICE technical report. Dependable computing, vol.107, no.339, pp.19–24, 2007. (in Japanese).

[13] A. Hoffmann, H. Meyr, and R. Leupers, Architecture Exploration for Embedded Processors with LISA, Kluwer Academic Publishers, 2003.

[14] CoWare, "Processor Designer." http://www.coware.com/products/processordesigner.php, 2007.

[15] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in Proc. European Design and Test Conference, pp.503–507, March 1995.

[16] J.G. Andrew, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A VLIW integer microprocessor," IEEE Journal of Solid State Circuits, vol.28, pp.1377–1383, 1993.

[17] A. Abnous and N. Bagherzadeh, "Pipelining and bypassing in a VLIW processor," IEEE Transactions on Parallel and Distributed Systems, vol.5, pp.658–664, June 1994.

[18] M. Buss, R. Azevedo, P. Centoducatte, and G. Araujo, "Tailoring pipeline bypassing and functional unit mapping to application in clustered VLIW architectures," Proceedings of

the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '01), pp.141–148, 2001.

[19] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W.m.W. Hwu, "IMPACT: an architectural framework for multiple-instruction-issue processors," Proceedings of the 18th annual international symposium on Computer architecture (ISCA '91), pp.266–275, 1991.

[20] M.D. Brown and Y.N. Patt, "Using internal redundant representations and limited bypass to support pipelined adders and register files," Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02), Washington, DC, USA, pp.289–298, IEEE Computer Society, 2002.

[21] P.G. Sassone and D.S. Wills, "Multicycle broadcast bypass: Too readily overlooked," the Proceedings of the Workshop on Complexity-Effective Design (WCED), 2004.

[22] M. Kudlur, K. Fan, M. Chu, R. Ravindran, N. Clark, and S. Mahlke, "FLASH: Foresighted latency-aware scheduling heuristic for processors with customized datapaths," Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04), pp.201–212, 2004.

[23] K. Fan, N. Clark, M. Chu, K.V. Manjunath, R. Ravindran, M. Smelyanskiy, and S. Mahlke, "Systematic register bypass customization for application-specific processors," In Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASSAP), pp.64–74, 2003.

[24] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau, "Operation tables for scheduling in the presence of incomplete bypassing," Proceedings of the international conference on Hardware/Software Codesign and System Synthesis: 2004 (CODES+ISSS '04), pp.194–199, 2004.

[25] A. Shrivastava, N. Dutt, A. Nicolau, and E. Earlie, "PBExplore: A framework for compiler-in-the-loop exploration of partial bypassing in embedded processors," Proceedings of the conference on Design, Automation and Test in Europe - Volume 2 (DATE '05), pp.1264–1269, 2005.

[26] S. Park, E. Earlie, A. Shrivastava, A. Nicolau, N. Dutt, and Y. Paek, “Automatic generation of operation tables for fast exploration of bypasses in embedded processors,” Proceedings of the conference on Design, automation and test in Europe: Proceedings (DATE ’06), pp.1197–1202, 2006.

[27] A. Shrivastava, P. Sanghyun, E. Earlie, N. Dutt, A. Nicolau, and P. Yunheung, “Automatic design space exploration of register bypasses in embedded processors,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.26, no.12, pp.2102 –2115, December 2007.

[28] R. Jayaseelan, H. Liu, and T. Mitra, “Exploiting forwarding to improve data bandwidth of instruction-set extensions,” Proceedings of the 43rd annual Design Automation Conference (DAC ’06), pp.43–48, 2006.

[29] K. Yasunori, A. Akira, O. Toshihiro, and N. Hiroshi, “A VLIW geometry processor with software bypass mechanism(special issue on multimedia, network, and dram lsis),” IEICE transactions on electronics, vol.81, no.5, pp.669–679, 1998.

[30] S. Toshihiro, T. Jin, M. Takefumi, and S. Nobuhiko, “Code optimization method for bypass network architecture by evaluation of dfg,” IEICE Technical Report. Communication systems, vol.107, no.531, pp.75–78, 2008-02-29. (in Japanese).

[31] T. Jin, S. Toshihiro, M. Takefumi, and S. Nobuhiko, “Power reduction in multi-processor with bypass structures connected by bus interconnection,” Proceedings of the 2008 International Conference in Embedded Systems and Intelligent Technology (ICESIT 2008), p.46, 2008.

[32] M.S. Lam, R. Sethi, J.D. Ullman, and A.V. Aho, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2006.

[33] K. Kennedy and J.R. Allen, Optimizing Compilers for Modern Architectures: A dependence-based approach, Morgan Kaufmann Publishers Inc., 2002.

[34] P. Marwedel and G. Goossens, Code Generation for Embedded Processors, Kluwer Academic Publishers, 1995.

[35] R. Leupers, Code Optimization Techniques for Embedded Processors, Kluwer Academic Publishers, 2000.

[36] Y.N. Srikant and P. Shankar, The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition, 2nd ed., CRC Press, Inc., Boca Raton, FL, USA, 2007.

[37] R. Leupers, "Code selection for media processors with SIMD instructions," Proceedings of the conference on Design, automation and test in Europe (DATE '00), pp.4–8, 2000.

[38] F. Franchetti, S. Kral, J. Lorenz, and C. Ueberhuber, "Efficient utilization of SIMD extensions," Proceedings of the IEEE, vol.93, no.2, pp.409–425, 2005. special issue on Program Generation, Optimization, and Adaptation.

[39] L. Fireman, E. Petrank, and A. Zaks, "New algorithms for SIMD alignment," Proceedings of the 16th international conference on Compiler construction (CC'07), pp.1–15, 2007.

[40] M. Suzuki, N. Fujinami, T. Fukuoka, T. Watanabe, and I. Nakata, "SIMD optimization in COINS compiler infrastructure," Proceedings of the Innovative Architecture on Future Generation High-Performance Processors and Systems, pp.131–140, 2005.

[41] M. Sassa, T. Nakaya, M. Kohama, T. Fukuoka, and M. Takahashi, "Static single assignment form in the coins compiler infrastructure," Proceedings of the 2003 International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet (SSGRR2003), 2003.

[42] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, and M. Yannakakis, "The complexity of multiterminal cuts," SIAM Journal on Computing, vol.23, pp.864–894, August 1994.

[43] J.L. Hennessy and D.A. Patterson, Computer architecture (2nd ed.): a quantitative approach, 2 ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[44] J.L. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," ACM Trans. Program. Lang. Syst., vol.5, pp.422–448, July 1983.

[45] A. Leung, K.V. Palem, and A. Pnueli, “Scheduling time-constrained instructions on pipelined processors,” ACM Transactions on Programming Languages and Systems, vol.23, pp.73–103, January 2001.

[46] A. Capitanio, N. Dutt, and A. Nicolau, “Partitioned register files for VLIWs: a preliminary analysis of tradeoffs,” Proceedings of the 25th annual international symposium on Microarchitecture (MICRO 25), pp.292–300, 1992.

[47] K. Wilken, J. Liu, and M. Heffernan, “Optimal instruction scheduling using integer programming,” Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI ’00), pp.121–133, 2000.

[48] P.v. Beek and K.D. Wilken, “Fast optimal instruction scheduling for single-issue processors with arbitrary latencies,” Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP ’01), pp.625–639, 2001.

[49] L.A. Wolsey, Integer programming, Wiley-Interscience, New York, NY, USA, 1998.

[50] ACE, “CoSy compiler development system.” http://www.ace.nl/compiler/cosy.html, 2007.

[51] P. Barth, “A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization,” Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[52] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, “DSPSTONE: A DSP-oriented benchmarking methodology,” Proceedings of the International Conference on Signal Processing and Technology (ICSPAT’94), pp.715–720, 1994.

[53] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson, Introduction to Algorithms, 2nd ed., McGraw-Hill Higher Education, 2001.

[54] Texas Instruments, TMS320C62x DSP CPU and Instruction Set Reference Guide, 2010.

[55] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang, “Code Generation Using Tree Matching and Dynamic Programming,” ACM Trans. on Programming Languages and Systems, vol.11, no.4, pp.491 – 516, Oct. 1989.

[56] J. Tanabe, Y. Taniguchi, T. Miyamori, Y. Miyamoto, H. Takeda, M. Tarui, H. Nakayama, N. Takeda, K. Maeda, and M. Matsui, “Visconti: multi-VLIW image recognition processor based on configurable processor,” Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003, pp.185 – 188, September 2003.

[57] T. Miyamori, J. Tanabe, Y. Taniguchi, K. Furukawa, T. Kozakaya, H. Nakai, Y. Miyamoto, K. Maeda, and M. Matsui, “Development of Image Recognition Processor Based on Configurable Processor,” Journal of Robotics and Mechatronics, vol.17, no.4, pp.437–446, 2005.

[58] Toshiba, MeP Core (MeP-c4) User’s Manual (Instruction Set), 2006.