DESIGNING COST-EFFECTIVE COARSE-GRAINED RECONFIGURABLE ARCHITECTURE

A Dissertation

by

YOONJIN KIM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2009

Major Subject: Computer Engineering

DESIGNING COST-EFFECTIVE COARSE-GRAINED RECONFIGURABLE ARCHITECTURE

A Dissertation

by

YOONJIN KIM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Rabi N. Mahapatra
Committee Members,	Eun Jung Kim
	Duncan Henry M. Walker
	Gwan Choi
Head of Department,	Valerie E. Taylor

May 2009

Major Subject: Computer Engineering

ABSTRACT

Designing Cost-Effective Coarse-Grained Reconfigurable Architecture. (May 2009)

Yoonjin Kim, B.S., SungKyunKwan University;

M.S., Seoul National University

Chair of Advisory Committee: Dr. Rabi N. Mahapatra

Application-specific optimization of embedded systems becomes inevitable to satisfy the market demand for designers to meet tighter constraints on cost, performance and power. On the other hand, the flexibility of a system is also important to accommodate the short time-to-market requirements for embedded systems. To compromise these incompatible demands, coarse-grained reconfigurable architecture (CGRA) has emerged as a suitable solution. A typical CGRA requires many processing elements (PEs) and a configuration cache for reconfiguration of its PE array. However, such a structure consumes significant area and power. Therefore, designing cost-effective CGRA has been a serious concern for reliability of CGRA-based embedded systems.

As an effort to provide such cost-effective design, the first half of this work focuses on reducing power in the configuration cache. For power saving in the configuration cache, a *low power reconfiguration technique* is presented based on *reusable context pipelining* achieved by merging the concept of context reuse into context pipelining. In addition, we propose *dynamic context compression* capable of supporting only required bits of the context words set to enable and the redundant bits set to disable. Fi-

nally, we provide *dynamic context management* capable of reducing reduce power consumption in configuration cache by controlling a read/write operation of the redundant context words

In the second part of this dissertation, we focus on designing a cost-effective PE array to reduce area and power. For area and power saving in a PE array, we devise *a cost-effective array fabric* addresses novel rearrangement of processing elements and their interconnection designs to reduce area and power consumption. In addition, *hierarchical reconfigurable computing arrays* are proposed consisting of two reconfigurable computing blocks with two types of communication structure together. The two computing blocks have shared critical resources and such a sharing structure provides efficient communication interface between them with reducing overall area.

Based on the proposed design approaches, a CGRA combining the multiple design schemes is shown to verify the synergy effect of the integrated approach. Experimental results show that the integrated approach reduces area by 23.07% and power by up to 72% when compared with the conventional CGRA.

DEDICATION

To my family and friends

ACKNOWLEDGEMENTS

It has been my life-long dream to become a professional in the engineering/science field and to infuse my passion into my research work. For this reason, this acknowledgement is very meaningful for me. The completion a Ph.D. in computer engineering at Texas A&M University has been the best way to accomplish my goals and achieve my dream.

First of all, I am sincerely grateful to my advisor Dr. Rabi N. Mahapatra for allowing me to conduct research with him and for his guidance during my Ph.D. program. His exceptional commitment to research and strong demand for excellence have guided me this far. I am truly grateful to his insightful advice, encouragement, and constant motivation throughout this work. Many thanks also go to my previous advisor, Professor Kiyoung Choi of Seoul National University, for his encouragement and helpful discussions. For two years of my master's course, he taught me to appreciate that a successful graduate student must have an arduous and passionate attitude. I would also like to thank the other members of my dissertation committees: Professors Eun Jung Kim, Duncan Henry M. Walker, and Gwan Choi. Their insightful comments and constructive criticisms helped me improve my research. Without their feedback, this dissertation would not have been made in it present form. In addition, I am deeply grateful to Professor Jundong Cho of SungKyunKwan University for his teaching and advice in my undergraduate days.

Furthermore, I would like to thank my friends and fellow students at Texas A&M University for numerous discussions about various issues related to research and life. I

sincerely thank current and former members of Embedded Systems and Co-design Group for being supportive of me during this work. I thank them all, including Woo-Seok Hong, In-choon Yeo, Baik-Song Ahn, Ja-Ryeong Koo, Sun-Young Choi, Ju-Young Jung, Young-Ah Kim, Moon-Jeong Kang and Young-Ho Koh, for being great friends and always being available whenever I need their assistance and help. Members of the Design Automation Lab in Seoul National University have helped me in various ways during the years of my Ph.D. program. I thank them all, especially Yong-Jin Ahn, Dong-Kwan Suh, Young-Chul Cho, Imyong Lee, Il-Hyun Park, Dong-Wook Lee, and Man-Hwee Jo.

Last, but not least, I am especially grateful to my parents and my elder brother for their incredible support and trust for me. Without their dedication and belief in me, I couldn‘t have completed this work in due time.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION v

ACKNOWLEDGEMENTS vi

TABLE OF CONTENTS viii

LIST OF FIGURES xiii

LIST OF TABLES xvii

CHAPTER

I INTRODUCTION 1

A. Objective and Approach 2
B. Contributions 3
C. Dissertation Organization 5

II BACKGROUND AND RELATED WORKS 6

A. Coarse-Grained Reconfigurable Architecture 6
B. Related Works 8

III BASE CGRA IMPLEMENTATION 13

A. Reconfigurable Array Architecture Coupling with Processor 13
B. Base Reconfigurable Array Architecture 15
1. Processing Element 16
2. PE Array 16
3. Frame Buffer 18
4. Configuration Cache 18
5. Execution Controller 19
C. Breakdown of Area, Delay, and Power Cost 19
1. Area and Delay 20
2. Power 23

CHAPTER Page

IV LOW POWER RECONFIGURATION TECHNIQUE 25

A. Motivation 25
1. Loop Pipelining 25
2. Spatial Mapping and Temporal Mapping 31
B. Individual Approaches to Reduce Power in Configuration Cache .. 32
1. Spatial Mapping with Context Reuse 33
2. Temporal Mapping with Context Pipelining 34
3. Limitation of Individual Approaches 35
C. Integrated Approach to Reduce Power in Configuration Cache 36
1. Reusable Context Pipelining 47
2. Limitation of Reusable Context Pipelining 41
3. Hybrid Configuration Cache Structure 43
D. Application Mapping Flow 44
1. Temporal Mapping Algorithm 45
a. Covering 45
b. Time Assignment 46
c. Place Assignment 47
2. Context Rearrangement 47
E. Experiments 49
1. Experimental Setup 49
2. Results 50
a. Necessary Context Registers for Evaluated Kernels 50
b. Configuration Cache Size 52
c. Performance Evaluation 52
d. Power Evaluation 53

V DYNAMIC CONTEXT COMPRESSION FOR LOW POWER CGRA 55

A. Preliminary 55
1. Context Architecture 55
B. Motivation 57
1. Power Consumption by Configuration Cache 57
2. Valid Bit-Width of Context Words 57
3. Dynamic Context Compression for Low Power CGRA 59
C. Design Flow of Dynamically Compressible Context Architecture . 59
1. Context Architecture Initialization 62
2. Field Grouping 62
3. Field Sequence Graph Generation 64
4. Generation of Field Control Signal 65
a. Control Signals for ALU-Dependent Fields 66

CHAPTER Page

b. Control Signals for ALU-Independent Fields 66
5. Field Positioning 67
a. Field Positioning on Uncompressed Context Word 67
b. Field Positioning on Compressed Context Word 69
6. Compressible Context Architecture 78
7. Context Evaluation 78
D. Experiments 78
1. Experimental Setup 78
2. Results 79
a. Area Cost Evaluation 79
b. Performance Evaluation 80
c. Context Compression Ratio and Power Evaluation 81

VI DYNAMIC CONTEXT MANAGEMENT FOR LOW POWER CGRA 82

A. Motivation 82
1. Power Consumption by Configuration Cache 82
2. Redundancy of Context Words 83
a. NOP Context Words 83
b. Consecutively Same Part in Context Words 85
c. Redundancy Ratio 86
B. Dynamic Context Management 86
1. Context Partitioning 87
2. Context Management at Transfer Time 90
3. Context Management at Run Time 92
C. Experiments 94
1. Experimental Setup 94
2. Results 94
a. Area Cost Evaluation 94
b. Power Evaluation 95
c. Performance Evaluation 96

VII COST-EFFECTIVE ARRAY FABRIC 97

A. Preliminary 97
1. Resource Sharing 98
2. Resource Pipelining 101
B. Cost-Effective Reconfigurable Array Fabric 103
1. Motivation 103
a. Characteristics of Computation-Intensive and Data-Parallel Application 103

CHAPTER Page

b. Redundancy in Conventional Array Fabric 104
2. New Cost Effective Data Flow-Oriented Array Structure 105
a. Derivation of Data Flow-Oriented Array Structure 105
b. Mitigation of Spatial Limitation in the Proposed Array Structure 109
3. Data Flow-Oriented Array Design Flow 110
a. Input Reconfigurable Array Fabric 112
b. New Array Fabric Specification-Phase I 112
c. New Array Fabric Specification-Phase II 118
d. Connectivity Enhancement 122
4. Cost-Effective Array Fabric with Resource Sharing and Pipelining 123
C. Experiments 125
1. Experimental Setup 125
a. Evaluated Applications 125
b. Hardware Design and Power Estimation 126
2. Results 127
a. Area Evaluation 127
b. Performance Evaluation 127
c. Power Evaluation 130

VIII HIERARCHICAL RECONFIGURABLE COMPUTING ARRAYS 131

A. Motivation 131
1. Limitation of Existing Processor-RAA Communication Structures 131
2. RAA-based Computing Hierarchy 133
B. Computing Hierarchy in CGRA 134
1. Computing Hierarchy –Size and Speed 135
2. Resource Sharing in RCC and RAA 136
3. Computing Flow Optimization 140
C. Experiments 142
1. Experimental Setup 142
a. Architecture Implementation 142
b. Evaluated Applications 144
2. Results 144
a. Area Cost Evaluation 144
b. Performance Evaluation 144
c. Power Evaluation 146

IX INTEGRATED APPROACH TO OPTIMIZE CGRA 149

CHAPTER Page

A. Combination among the Cost-Effective CGRA Design Schemes .. 149
B. Case Study for Integrated Approach 150
1. An CGRA Design Example Merging Three Design Schemes.. 150
2. Results 151
a. Area and Performance Evaluation..... 151
b. Power Evaluation 152
C. Potential Combinations and Expected Outcomes 153

X CONCLUSIONS 155

REFERENCES..... 158

VITA 168

LIST OF FIGURES

FIGURE		Page
1	Block diagram of general CGRA	7
2	Basic types of reconfigurable array coupling	14
3	Block diagram of base CGRA	14
4	Processing element structure of base RAA	16
5	Interconnection structure of PE array	17
6	Distributed configuration cache structure	18
7	Area cost breakdown for CGRA	20
8	Cost analysis for a PE	21
9	Power cost breakdown for CGRA running 2D-FDCT	22
10	4x4 reconfigurable array	26
11	C-code of Eq. (2)	28
12	Execution model for CGRA	29
13	Comparison between temporal mapping and spatial mapping	32
14	Configuration cache structure for context reuse	33
15	Cache structure for context pipelining	35
16	Proposed configuration cache structure	37
17	Reusable context pipelining for Eq. (2)	40
18	Reusable context pipelining with temporal cache	41
19	Reusable context pipelining according to the execution time for one iteration (i > 1)	42

FIGURE Page

20 Hybrid configuration cache structure 44

21 Application mapping flow for base architecture and proposed architecture 45

22 Temporal mapping steps 46

23 Context rearrangement 48

24 PE structure and context architecture of MorphoSys 56

25 Valid bit-width of context words 58

26 Entire design flow 60

27 Context architecture initialization 61

28 Field grouping 63

29 Field sequence graph 64

30 Control signals for 'MUX_B' and 'PRED' 65

31 Updated FSG from flag merging 67

32 Default field positioning 68

33 Field concurrency graph 69

34 Examples of ‘Find_Interval’ 75

35 Multiplexer port-mapping graph 76

36 Compressible context architecture 77

37 Consecutively same part in context words 84

38 Redundancy ratio of context words 85

39 An example of PE and context architecture 87

40 Context partitioning 88

FIGURE Page

41 Comparison between general CE and proposed CE 89
42 Context management when context words are transferred 89
43 Context management at run time 92
44 Snapshots of three mappings 98
45 Eight multipliers shared by sixteen PEs 99
46 The connection between a PE and shared multipliers 100
47 Critical paths 101
48 Loop pipelining with pipelined multipliers 102
49 Subtask classification 104
50 Data flow on square reconfigurable array 105
51 Data flow-oriented array structure derived from three types of data flow. 106
52 An example of data flow-oriented array 107
53 Snapshots showing the maximum utilization of PEs 109
54 Overall design flow 111
55 Basic concept of *local triangulation* method 114
56 *Local triangulation* method 115
57 Interconnection derivation in Phase I 116
58 New array fabric example by Phase I 117
59 *Global triangulation* method when $n = 2$ (L2) 120
60 New array fabric example by Phase II 122
61 New array fabric example by connectivity enhancement 123

FIGURE Page

62 New array fabric with resource sharing and pipelining 124

63 Mapping example on new array fabric..... 125

64 Analogy between Memory and RAA-computing hierarchy 134

65 Computing hierarchy of CGRA 134

66 CGRA configuration with RCC and RAA..... 136

67 Two cases of functional resource assignment..... 138

68 Critical resource sharing and pipelining in L1 and L2 PE array..... 139

69 Interconnection structure among RCC, shared critical resources and L2 PE array..... 139

70 Four cases of computing flow according to the input/output size of application..... 141

71 Performance comparison..... 145

72 Power comparison..... 147

73 Combination flow of the proposed design schemes..... 150

74 A combination example combining three design schemes 151

75 Potential combination of multiple design schemes..... 154

LIST OF TABLES

TABLE		Page
I	Architecture Specification of Base and Proposed Architecture	51
II	Necessary Context Registers for Evaluated Kernels	51
III	Size of Configuration Cache and Context Registers	52
IV	Power Reduction Ratio by Reusable Context Pipelining	53
V	Notations for Port-Mapping Algorithm	71
VI	Area Overhead by Dynamic Context Compression	80
VII	Power Reduction Ratio by Dynamic Context Compression	80
VIII	Area Overhead by Dynamic Context Management	94
IX	Power Reduction Ratio by Dynamic Context Management	95
X	Area Reduction Ratio by RSPA and NAF	127
XI	Applications Characteristics and Performance Evaluation	128
XII	Power Reduction Ratio by RSP+NAF	129
XIII	Comparison of the Basic Coupling Types	133
XIV	Comparison of the Architecture Implementations	142
XV	Applications Characteristics	143
XVI	Area Cost Comparison	144
XVII	Area Reduction Ratio by Integrated RAA	152
XVIII	Entire Power Comparison	153

CHAPTER I

INTRODUCTION

With the growing demand for high quality multimedia, especially over portable media, there has been continuous development on more sophisticated algorithms for audio, video, and graphics processing. These algorithms have the characteristics of data-intensive computation of high complexity. For such applications, we can consider two extreme approaches to implementation: software running on a general purpose processor and hardware in the form of ASIC. In the case of general purpose processor, it is flexible enough to support various applications but may not provide sufficient performance to cope with the complexity of the applications. In the case of ASIC, we can optimize best in terms of power and performance but only for a specific application. With a coarse-grained reconfigurable architecture (CGRA), we can take advantage of the above two approaches. This architecture has higher performance level than general purpose processor and wider applicability than ASIC.

As the market pressure of embedded systems compels the designer to meet tighter constraints on cost, performance, and power, the application specific optimization of a system becomes inevitable. On the other hand, the flexibility of a system is also important to accommodate rapidly changing consumer needs. To compromise these incompatible demands, domain-specific design is focused on as a suitable solution for recent

---

The journal model is *IEEE Transactions on Very Large Scale Integration Systems*.

embedded systems. Coarse-grained reconfigurable architecture is the very domain-specific design in that it can boost the performance by adopting specific hardware engines while it can be reconfigured to adapt to ever-changing characteristics of the applications.

In spite of the above advantages, the deployment of CGRA is prohibitive due to its significant area and power consumption. This is due to the fact that CGRA is composed of several memory components and the array of many processing elements including ALU, multiplier and divider, etc. Especially, processing element (PE) array occupies most of the area and consumes most of the power in the system to support flexibility and high performance. Therefore, reducing area and power consumption in the PE array has been a serious concern for the adoption of CGRA.

A. Objective and Approach

This dissertation explores the problem of reducing area and power in CGRA based on architecture optimization. To provide cost-effective CGRA design, the following questions are considered.

- *How to reduce area and power consumption in CGRA?* For power saving in CGRA, We should obtain area and power breakdown data of CGRA to identify area and power-dominant components. Then the components may be optimized for area and power by removing redundancies of CGRA wasting area and power. Such redundancies may depend on the characteristics of computation model or applications.
- *How to design cost-effective CGRA with non-sacrificing or enhancing perform-*

ance? Ultimately, the goals of designing cost-effective CGRA is that proposed approaches do not cause performance degradation with saving area and power. It means that the proposed cost-effective CGRA keeps original functionality of CGRA intact and does not increase critical path delay. In addition, the performance may be enhanced by optimizing the performance bottleneck with keeping the area and power-efficient approaches.

In this dissertation, these central questions are addressed for area/power-critical components of CGRA and we suggest new frameworks to achieve these goals. The validation of the proposed approaches is demonstrated through the use of real application benchmarks and gate level simulations.

B. Contributions

This work makes the following contributions:

- *Low power reconfiguration technique for CGRA*. It presents a novel power-conscious architectural technique called *reusable context pipelining* (RCP) for CGRA to close the power-performance gap between low power-oriented spatial mapping and high performance-oriented temporal mapping prevailing in existing CGRA architectures. A new configuration cache structure has been proposed to support reusable context pipelining with negligible overheads. The temporal mapping with RCP has been shown to be a universal approach in reducing power and enhancing performance for CGRA.

- *Dynamic context compression for low power CGRA*. A new design flow for

CGRA design has been proposed to generate architecture specifications that are required for modifying configuration cache dynamically. Design methodology for dynamically compressible context architecture and a new cache structure to support the configurability are being presented to reduce the power consumption in configuration cache without performance degradation.

• *Dynamic context management for low power CGRA.* It presents a novel control mechanism of configuration cache called dynamic context management to reduce the power consumption in configuration cache without performance degradation. A new configuration cache structure is proposed to support dynamic context management.

• *A new array fabric for CGRA*. A novel array fabric design exploration method has been proposed to generate cost-effective reconfigurable array structure. Novel rearrangement of processing elements and their interconnection designs are introduced for CGRA to reduce area and power consumption without any performance degradation.

• *Hierarchical reconfigurable computing arrays for efficient CGRA-based embedded systems.* A new reconfigurable computing hierarchy has been proposed to design cost-effective CGRA-based embedded systems. Efficient communication structure between processor and reconfigurable computing blocks is introduced to reduce performance bottleneck in the CGRA-based architecture.

C. Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter II, we describe background and related work of this dissertation. Chapter III presents base architecture implementation and its cost breakdown. In Chapter IV, we propose *low power reconfiguration technique* to reduce power in configuration cache. Chapters V and VI present *dynamic context compression* and *dynamic context management* capable of reducing power consumption in configuration cache. In Chapter VII, we device a cost-effective array fabric for CGRA to reduce area and power in PE array. Chapter VIII presents hierarchical reconfigurable computing array to reduce area and power with enhancing performance. Finally, we present integrated approach to merge the multiple design schemes and conclude this work in Chapters IX and X.

CHAPTER II

BACKGROUND AND RELATED WORKS

A. Coarse-Grained Reconfigurable Architecture

A recent trend in the architectural platforms for embedded systems is the adoption of reconfigurable computing elements for cost, performance, and flexibility issues [1]. Coarse-Grained Reconfigurable Architectures (CGRAs) [1] exploit both the flexibility and efficiency, and are shown to be a generally better solution for compute-intensive applications than fine-grained reconfigurable architectures. There are different styles of CGRAs, but many architectures are based on 2D array of ALU-like datapath blocks. These are particularly interesting due to the wide acceptance in recent reconfigurable processors as well as their expected high performance for many heavy-load applications in the domains of signal processing, multimedia, communication, security, and so on.

Typically, a CGRA consists of a main processor, a Reconfigurable Array Architecture (RAA), and their interface as Fig. 1. The RAA has identical processing elements (PEs) containing functional units and a few storage units such as ALU, multiplier, shifter and register file. The data buffer provides operand data to PE array through a high-bandwidth data bus. The configuration cache (or context memory) stores the context words used for configuring the PE array elements. The context register between a PE and a cache element (CE) in configuration cache is used to keep the cache access path from being the critical path of the CGRA.

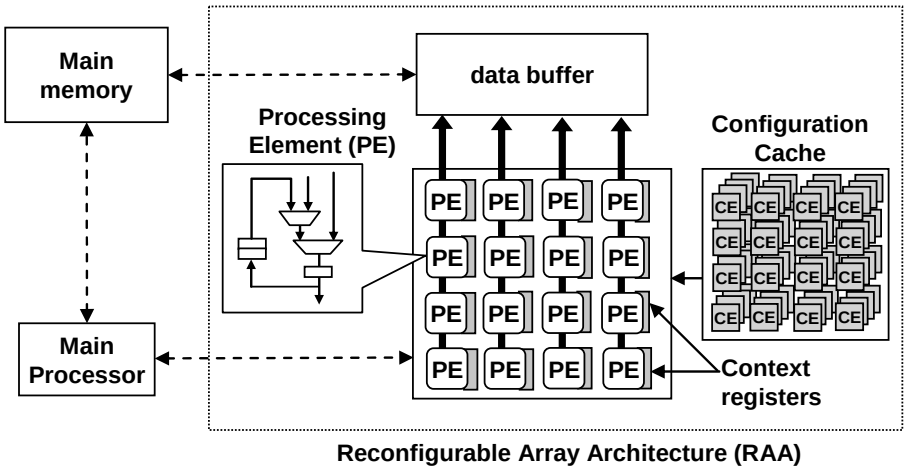


Fig. 1. Block diagram of general CGRA.

Unlike FPGA (most typical of a fine-grained reconfigurable architecture), which are built with *bit-level* configurable logic blocks (CLBs), CGRA is built with PEs, which are *word-level* configurable functional blocks. By raising the granularity of operations from a bit to a word, CGRA can improve on the speed and the performance as well as the resource utilization for compute-intensive applications. Another consequence of this raised granularity is that whereas FPGA can be used for implementing any digital circuits, CGRA is targeted only for a limited set of applications, although different CGRAs may target different application domains. Still, CGRA retains the idea of "reprogrammable hardware" in the reprogrammable interconnects as well as in the configurable functional blocks (i.e., PEs). Moreover, since the amount of the configuration bit-stream is greatly reduced through the raised granularity, the configuration can be actually changed even at the runtime very fast. Most of the CGRAs feature single-cycle configuration change, fetching the configuration data from a distributed local cache. This unique com-

bination of efficiency and flexibility, which in the main advantage of CGRA, explains an evaluation result [2] that under certain conditions CGRAs are actually more cost-effective for wireless communication applications than alternatives such as FPGA implementations as well as DSP architectures. It is worth mentioning that the improved efficiency of CGRAs in terms of the speed, performance, and area is a result of the architecture specialization for compute-intensive applications.

B. Related Works

Many kinds of coarse-grained reconfigurable architecture have been proposed with the increasing interests in reconfigurable computing until 2001 [1]. These CGRAs can be classified into two cases: mesh-based reconfigurable array and linear reconfigurable array. Mesh-based reconfigurable arrays arrange their processing elements (PEs) mainly as a rectangular 2-D array with horizontal and vertical connections, which support rich communication resources for efficient parallelism. In the case of linear reconfigurable arrays, they support pipelined execution for stream-based applications with static or dynamic reconfiguration. MorphoSys [3] and REMARC [4] are representations of mesh-based architectures. MorphoSys consists of Tiny_RISC processor, RC (Reconfigurable Cell) array, frame buffer, context memory and DMA controller. RC array is an 8×8 array of ALUs that performs 16-bit operations based on SIMD programming model. REMARC consists of a global control unit and an 8x8 array of nano processors. A nano processor consists of an ALU, a 16-entry data RAM, an 8-entry register file, data input registers and data output registers. The configuration for each nano processor is stored in the 32-entry instruction RAM to support MIMD execution model as well as SIMD

model. RaPiD [5] and PipeRench [6][7] have linear array structure. RaPiD provides different computing resources like ALUs, RAMs, multipliers and registers. These resources are irregularly distributed on one dimension and are mostly static reconfigured. However, PipeRench [6][7] relies on dynamic reconfiguration, allowing the reconfiguration of a processing element (PE) in each execution cycle. It consists of strips composed of interconnect and PEs with registers and ALUs. The reconfigurable fabric allows the configuration of a pipeline stage in every cycle, while concurrently executing all other stages.

Since then, many more new CGRAs [2][8][9][10][11][12][13][14] [15][16][17][18] [19] have been continuously proposed and evolved. Most of them comprise of a fixed set of specialized processing elements (PEs) and interconnection fabrics between them. The run-time control of the operation of each PE and the interconnection provides the reconfigurability.

However, such fixed architecture has limitations in optimizing the area cost and performance for various applications. For example, MorphoSys [3] consists of 8x8 array of Reconfigurable Cell coupled with Tiny_RISC processor through system bus. It shows good performance for regular code segments in computation intensive domains but requires large amount of area and power consumption. XPP configurable system-on-chip architecture [10] is another example. XPP has 4 x 4 or 8 x 8 reconfigurable array and LEON processor with AMBA bus architecture. A processing element of XPP is composed of an ALU and some registers. Since the processing elements do not include heavy resources, the total area cost is not high but the range of applicable domains is restricted. In addition, XPP shows significant communication overhead between the proc-

essor and RAA through the system bus. REMARC [4] is reconfigurable Multimedia Array Coprocessor that consists of a global control unit and an 8x8 array of nano processors. The nano processors do not also include heavy resources like XPP but it also restricts the range of applicable domains. However, the communication with main processor is faster than [3] or [20] because the processor can access the register-set by coprocessor data transfer instructions. However, limited size of the register-set causes heavy registers-array traffic restricting performance enhancement. ADRES [21] tightly couples a VLIW processor and a reconfigurable matrix through shared register file. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way, whereas the VLIW processor executes the non-kernel code by exploiting instruction-level parallelism. Even though it also provides the fast communication speed between VLIW and the matrix but the entire structure is very dependent on VLIW processor architecture and it require huge register file for the communication. Therefore, the performance is limited by size of the register file. Most design space exploration techniques previously suggested are limited to the configuration of the internal structure of a PE and the interconnection scheme. Such configuration techniques are in general good at obtaining high performance but require high hardware cost. This is mainly because even a primitive PE design should be equipped with basic functional resources to gain reasonable performance. Moreover, adding a small functional block to a primitive PE design increases the total cost of the aggregate architecture a lot. In ADRES template [21], an XML-based architecture description language is used to define the overall topology, supported operation set, resource allocation, timing, and even internal organization of

each processing element. KressArray [20] also defines the exploration properties such as array size, interconnections, and functionality of certain processing elements. However, both templates do not support common resources shared among processing elements, thus some critical functional resources may have low utilization while occupying large area.

The research on low power CGRA has three different aspects: architecture exploration, code compilation & mapping and physical implementation. Although the architecture exploration flows that have been suggested in [8][20][21] [22][23][24][25][26][27] [28] generate a good instance of CGRA considering area and performance, they do not deal with power consumption. Interconnect architecture explorations have been suggested for low energy [21][29]. Because CGRA has complex interconnection for performance and flexibility, power consumption due to interconnection is crucial. In [8][29] the authors have proposed energy-aware interconnection exploration to minimize energy by changing the topology between global register file and function units. However, this exploration only provides the trade-off between performance and energy. In [30] the authors have suggested hierarchical generalized mesh structure exploration that continues to exploit locality while reducing the cost of long connections but it has been only evaluated for specific reconfigurable DSPs. In the case of code compilation and mapping, loops are exploited mainly for performance [9][31][32][33][34][35][36][37][38][39][40] [43][44]. Many reconfigurable architectures have been implemented with various technologies [6][10][12][43][44][45][46]. Most of these researches have focused on efficient design with respect to small area and high performance. In [6][8], even though authors

have presented power estimation data of the implemented architectures, these are only accessorial results and they do not offer power/energy-aware implementation. In [2][14], authors have emphasized that the implemented architectures are power-efficient as compared to fine-grained architectures such as FPGA running specific applications. These architectures are not general CGRA but specific for running some applications with low power consumption. In [6], the authors have fabricated PipeRench [7] in a 0.18 micron process. Their experimental results show that the power consumption is significantly high. Authors describe that the increase in power consumption is due to the dynamic reconfiguration requiring frequent configuration and state memory accesses. Hence, that power consumption by dynamic reconfiguration is a serious overhead as compared to other types of IP cores such as ASIC or ASIP.

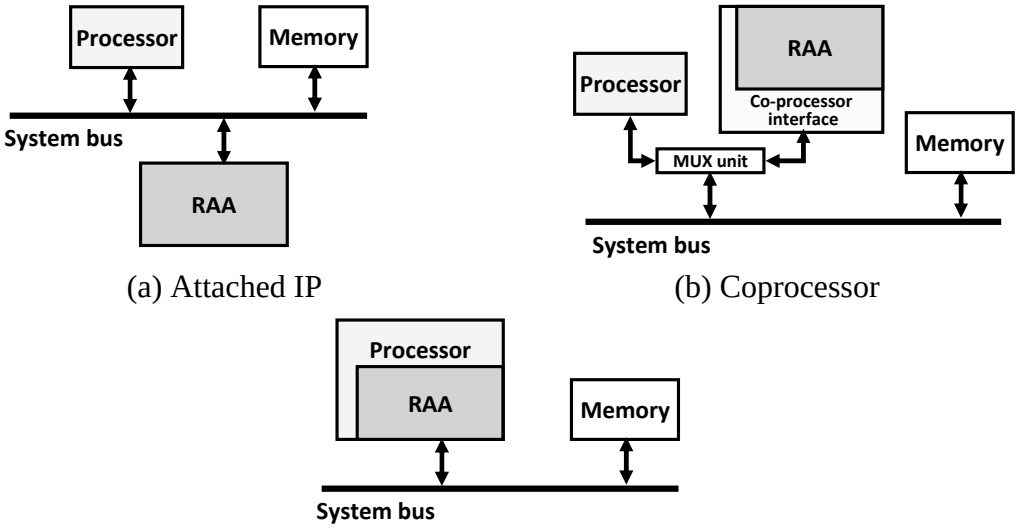
CHAPTER III

BASE CGRA IMPLEMENTATION

We have first designed a conventional CGRA as the base architecture and implemented it at the RT-level. This conventional architecture will be used throughout this dissertation as a reference for quantitative comparison with our cost-effective approaches.

A. Reconfigurable Array Architecture Coupling with Processor

A typical coarse-grained reconfigurable architecture consists of a microprocessor, a Reconfigurable Array Architecture (RAA), and their interface. We can consider three ways of connecting the RAA to the processor [47]. First, the array can be connected to a bus as an ‘Attached IP’ shown in Fig. 2(a). Secondly, the array can be placed next to the processor as a ‘Coprocessor’ as shown in Fig. 2(b). In this case, the communication is done using a protocol similar to those used for floating point coprocessors. Finally, the array can be placed inside the processor like a ‘FU (Functional Unit)’ as shown in Fig. 2(c). In this case, the instruction decoder issues special instructions to perform specific functions on the reconfigurable array as if it were one of the standard functional units of the processor.



(a) Attached IP (b) Coprocessor

(c) Functional unit

Fig. 2. Basic types of reconfigurable array coupling.

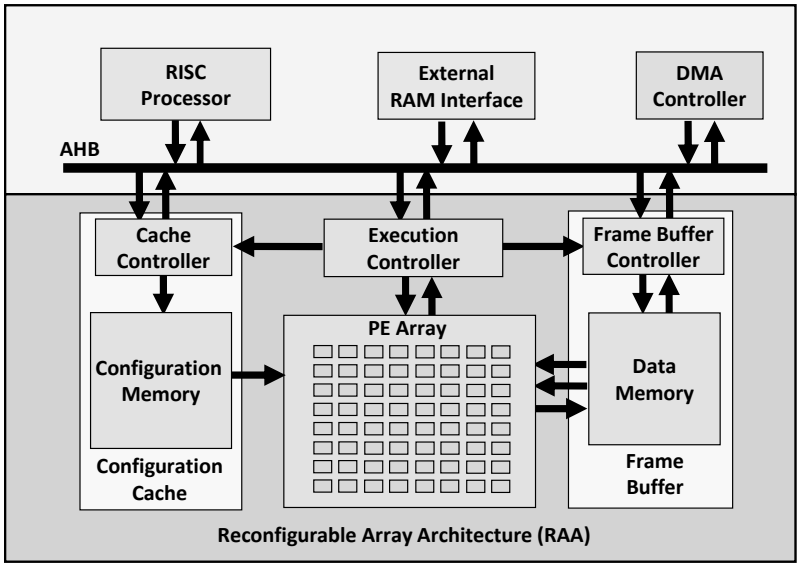


Fig. 3. Block diagram of base CGRA.

We have implemented the first type of reconfigurable architecture connecting the RAA as an Attached IP. In this case, the speed improvement using the RAA may have to

compensate for significant communication overhead. However, the main benefit of this type is the ease of constructing such a system using a standard processor and standard reconfigurable array without any modification. It consists of a RISC processor, a main memory block, a DMA controller, and an RAA. The RISC processor is a 32-bit processor which is small and simple with three pipeline stages and the communication bus is AMBA AHB [48], which couples the RISC processor and the DMA controller as master devices and the RAA as a slave device. The RISC processor executes control intensive, irregular code segments and the RAA executes data-intensive kernel code segments. The block diagram of the entire reconfigurable architecture is shown in Fig. 3.

B. Base Reconfigurable Array Architecture

Base RAA is similar to MorphoSys [3], which is a very representative CGRA showing high performance and flexibility as well as physical implementation. The difference from MorphoSys is that the proposed architecture supports both SIMD and MIMD execution model whereas the memory structure (frame buffer and configuration cache) of MorphoSys supports only the SIMD model. The SIMD model is efficient for data parallelism since it saves configurations and cache storage by sharing an instruction for multiple data. But its execution models are limited in that each individual PE cannot execute different instructions independently at the same time. Therefore, we take MIMD-style CGRA in which each PE can be configured separately to facilitate processing its own instructions. Since it allows more versatile configurations than their SIMD-style siblings, we adopt more general forms of loop pipelining [32] through simultaneous execution of multiple iterations of a loop in a pipeline.

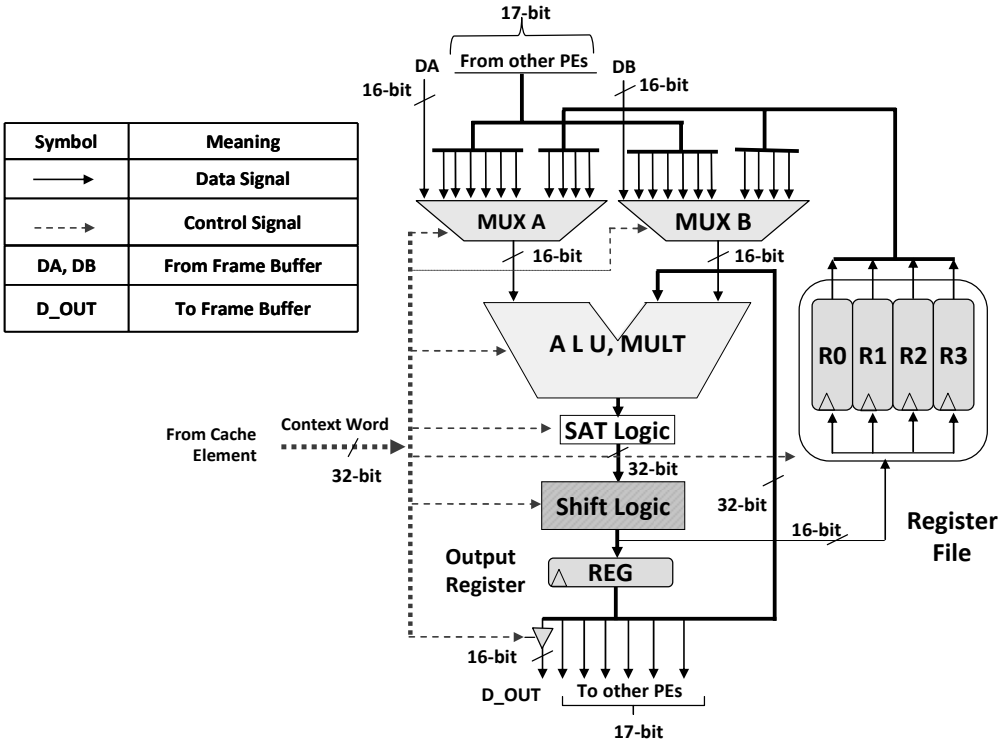


Fig. 4. Processing element structure of base RAA.

Base architecture specification is determined by our target application domain including audio/video codec as well as various benchmark kernels. Detailed features of each component of the architecture are as follows.

1. Processing Element

Each PE is a dynamically reconfigurable unit executing arithmetic and logical operations. The inner structure of a PE is shown in Fig. 4. A PE contains a 16-bit ALU, 16 x 16-bit array multiplier, shift logic, Arithmetic saturation (SAT_Logic), multiplexors and registers.

2. PE Array

The PE array is an 8x8 reconfigurable array of PEs, which we think is big enough for

most of the applications considered in our experiments. We assume that computation model of the array is loop pipelining based on temporal mapping [32] for high performance - each iteration of application kernel (critical loop) is mapped onto each column of

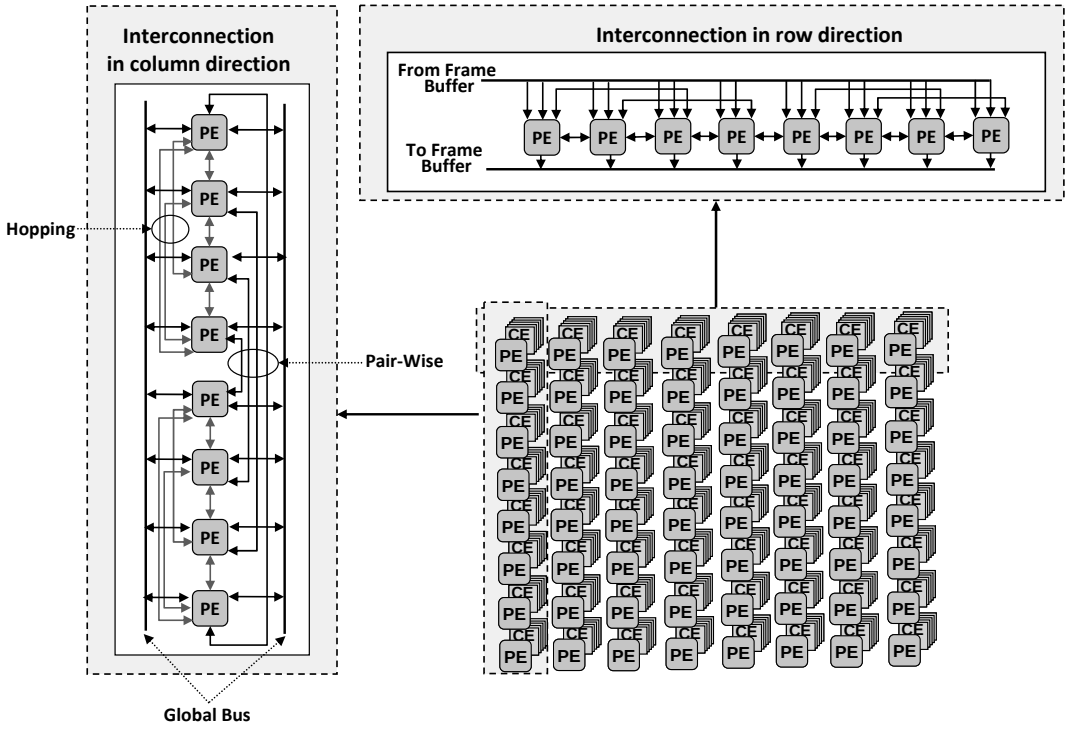


Fig. 5. Interconnection structure of PE array.

square array. Therefore, in this PE array, columns have more interconnection than rows. Fig. 5 shows interconnection structure of the PE array. The interconnection in rows is used mainly for the communication taking care of loop-carried dependencies. Columns and rows have nearest-neighbor and hopping interconnections for connectivity between two PEs in a half column and a half row. In addition, each column has pair-wise interconnections and two global buses for connectivity between two half columns. Each row shares two read-buses and one write-bus.

3. Frame Buffer

Frame buffer (FB) of MorphoSys does not support concurrency between the load of two operands and the store of result in a same column, since it is not needed in SIMD-style mapping. However, in the case of MIMD-style execution, concurrent load and store operations can happen between different loop iterations. So our FB has two sets of buffers, each having three banks: one bank connected to the write bus and the other two banks

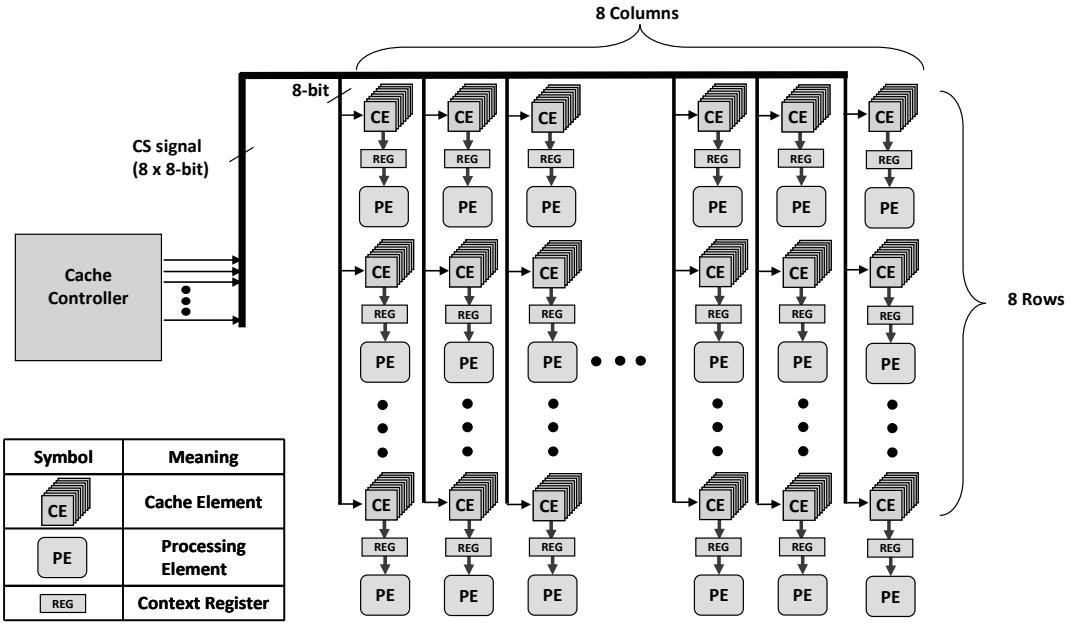


Fig. 6. Distributed configuration cache structure.

connected to the read buses. However, any combination of one-to-one mapping between the three banks and the three buses is possible.

4. Configuration Cache

The context memory of MorphoSys is designed for broadcasting configuration. So PEs

in the same row or column share the same context word for SIMD-style operation [3]. However, in the case of MIMD-style operation, each PE can be configured by different context word. Our configuration cache is composed of 64 Cache Elements (CEs) and a cache controller for controlling the CEs (Fig. 6). Each CE has 32 layers, each of which stores a context that configures the corresponding PE. The context register between a PE and a CE is used to keep the cache access path from being the critical path of the CGRA.

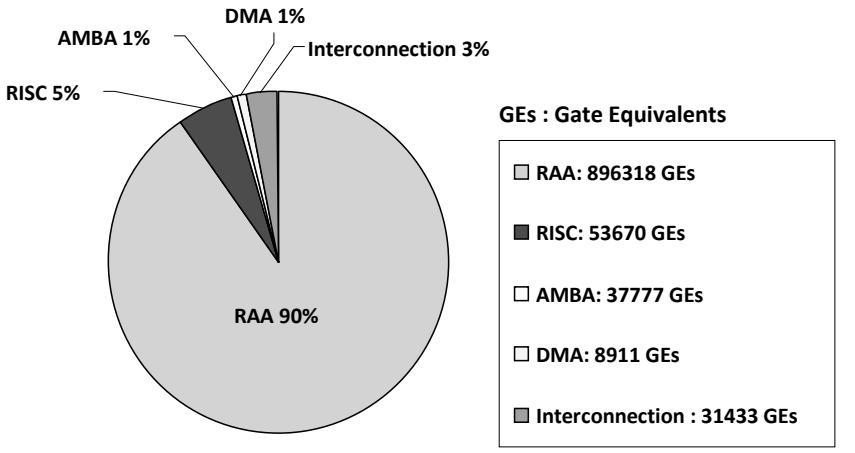
5 Execution Controller

Controlling the PE array execution directly from the main processor through AMBA AHB will cause high overhead in the main processor. In addition, the latency of the control will degrade the performance of the whole system, especially when dynamic reconfiguration is used. So a separate control unit is necessary to control the execution of the PE array every cycle. The execution controller receives the encoded control data from the main processor. The control data contains read/write mode and addresses of frame buffer and cache for guaranteeing correct operations of the PE array.

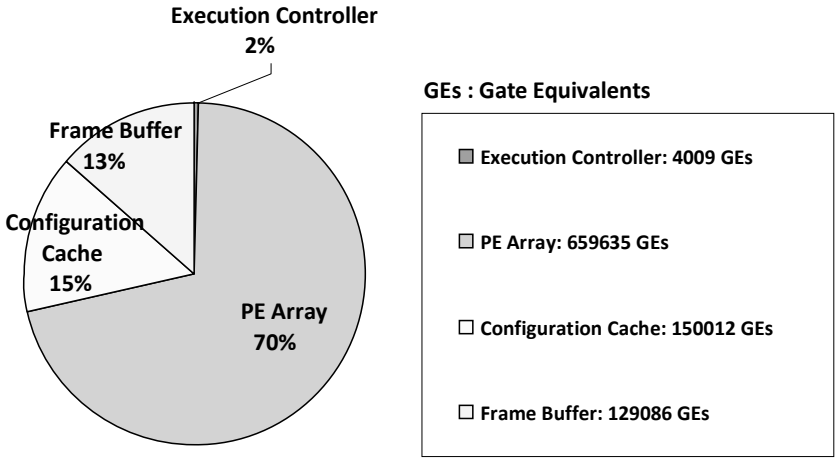
C. Breakdown of Area, Delay, and Power Cost

We have implemented the base architecture shown in Fig. 2 at the RT-level with VHDL. We have synthesized a gate-level circuit from the VHDL description and analyzed area, delay, and power cost. The synthesis has been done using Design Compiler [49] with 0.18 ㎛ technology. We have used DesignWare [49] library for the multipliers (carry-save array synthesis model) and dividers (restoring carry-look-ahead, 2-way overlapped synthesis model). SRAM Macro Cell library is used for the frame buffer and configuration cache. ModelSim [50] and PrimePower [49] have been used for gate-level simula-

tion and power estimation.



(a) Entire CGRA



(b) RAA

Fig. 7. Area cost breakdown for CGRA.

1. Area and Delay

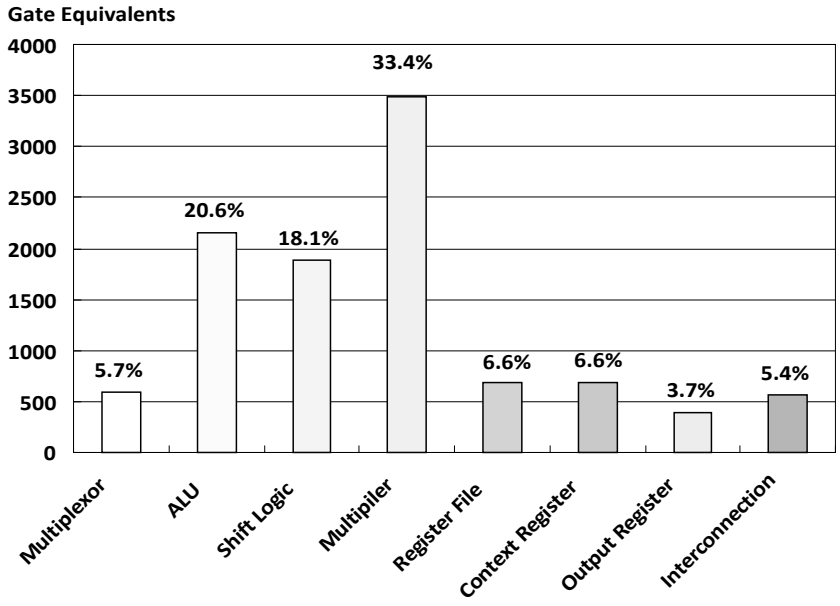
As shown in Fig. 7 (a), the RAA occupies as much as 90 % of the total area of the CGRA. Fig. 7 (b) shows more detailed area breakdown in the RAA. The PE array occupies as much as 70.5 % of the total area of the RAA, which is mainly due to heavy com-

putational resources such as ALU, multiplier, etc. in each PE. The critical path of the entire RAA is also in the PEs and its delay is given by

$$T_{Critical\ path} = T_{Multiplexor} + T_{Multiplier} + T_{Shift\_logic} + T_{others} \quad (1)$$

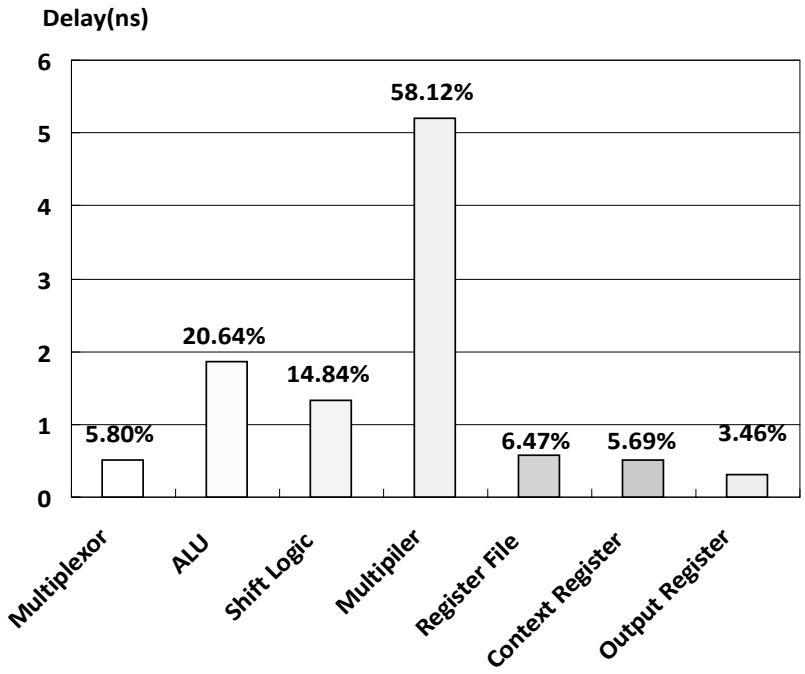
$$(8.96\text{ns} = 0.32\text{ns} + 5.21\text{ns} + 1.42\text{ns} + 1.78\text{ns})$$

From the area and delay cost breakdown of the RAA as shown in Figs. 7 and 8, we see that PE array design is crucial for cost-effective design. In the case of area, Fig. 8 (a) shows that multiplier occupies about 33.4% of the total area in a PE. In the case of delay, the multiplier again takes as much as 58.12 % (Fig. 8 (b)). Therefore, in our PE design, the multiplier is considered to be area-critical and delay-critical resource.



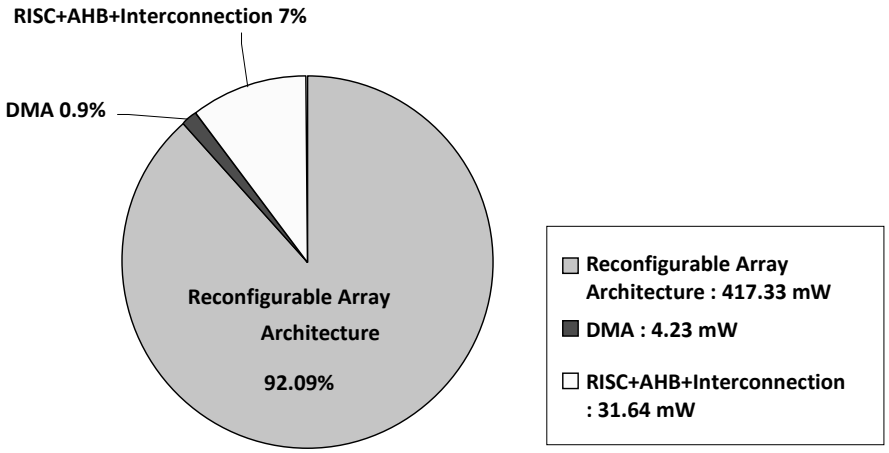
(a) Area

Fig. 8. Cost analysis for a PE.



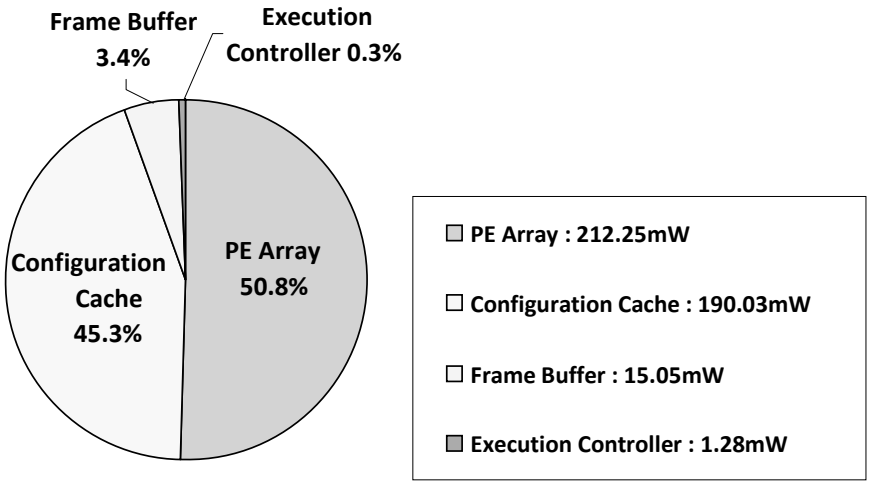
(b) Delay

Fig. 8. Continued.



(a) Entire CGRA

Fig. 9. Power cost breakdown for CGRA running 2D-FDCT.



(b) RAA

Fig. 9. Continued.

2. Power

To obtain power breakdown data, we have used 2D-FDCT as the kernel for simulation-based power measurement. The simulation has been done under the typical operating condition of 100 MHz frequency, 1.8 V Vdd, and 27℃ temperature. As can be observed from Fig. 9 (a), the RAA spends about 92.09% of the total power consumed in CGRA. Fig. 9 (b) shows more detailed power breakdown in the RAA. The RAA spends about 50.8% of its total power in the PE array, which consists of many components such as ALUs, multipliers, shifters and register files. The PE array consumes most of the power, which is natural because coarse-grained architecture aims to achieve high performance and flexibility with plenty of resources. The configuration cache spends about 45.3% of the overall power, which is the second largest. Even though the frame buffer uses the same kind of SRAM as the configuration cache, it consumes much less power (3.4%).

This is because the configuration cache performs read operations frequently to load the context words, one for each PE, whereas the frame buffer performs load/store operations less frequently to access data on row basis rather than for every PE.

CHAPTER IV

LOW POWER RECONFIGURATION TECHNIQUE

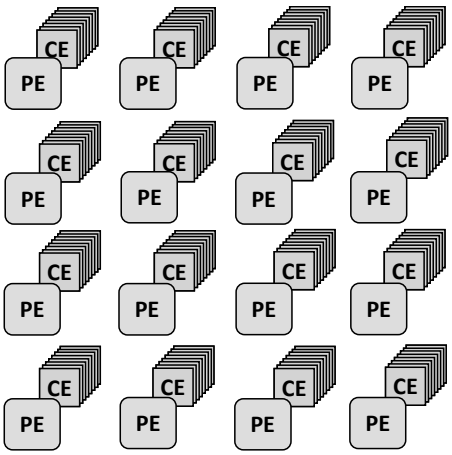
In this chapter, we suggest a novel power-conscious architectural technique called *reusable context pipelining* (RCP) to reduce power consumption in configuration cache [51]. RCP is a universal approach in reducing power and enhancing performance for CGRA because it can be achieved by closing the power-performance gap between low power-oriented spatial mapping and high performance-oriented temporal mapping. Furthermore, we propose new configuration cache structure (called hybrid configuration cache) to support reusable context pipelining with reduced memory size. Experimental results show that the proposed approach saves much power even with reduced configuration cache size. Power reduction ratio in the configuration cache and the entire architecture are up to 86.33 % and 47.60 % respectively compared to the base architecture.

A. Motivation

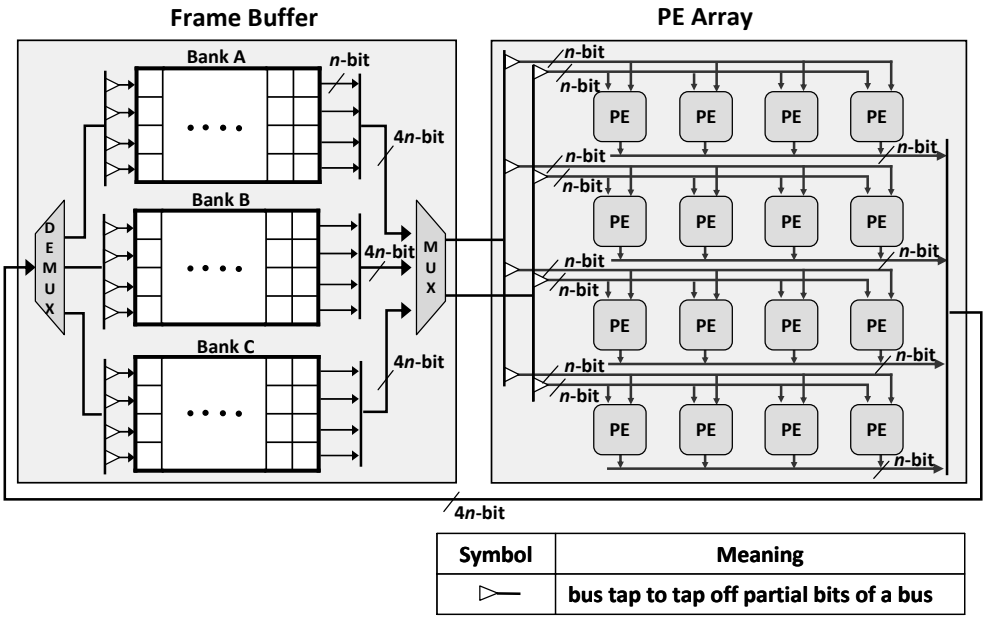
In this section, we present the motivation of our power-conscious approaches. The main motivation is due to the characteristics of loop pipelining (spatial mapping and temporal mapping) [32] based on MIMD-style execution model.

1. Loop Pipelining

To represent the characteristics of loop pipelining [32], we examine the difference between SIMD and MIMD in the RAA with a simple example. We assume a mesh-based 4x4 coarse-grained reconfigurable array of PEs, where a PE is a basic reconfigurable

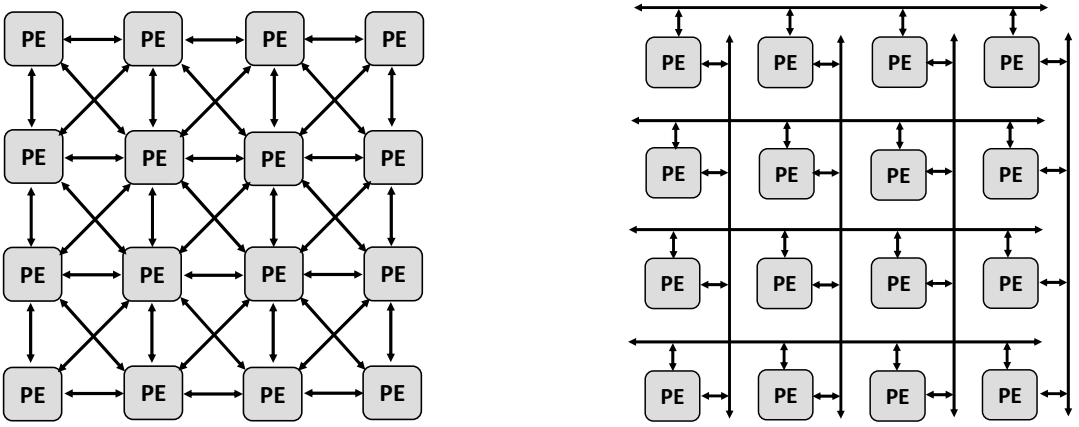


(a) Distributed cache structure



(b) Frame buffer and data bus

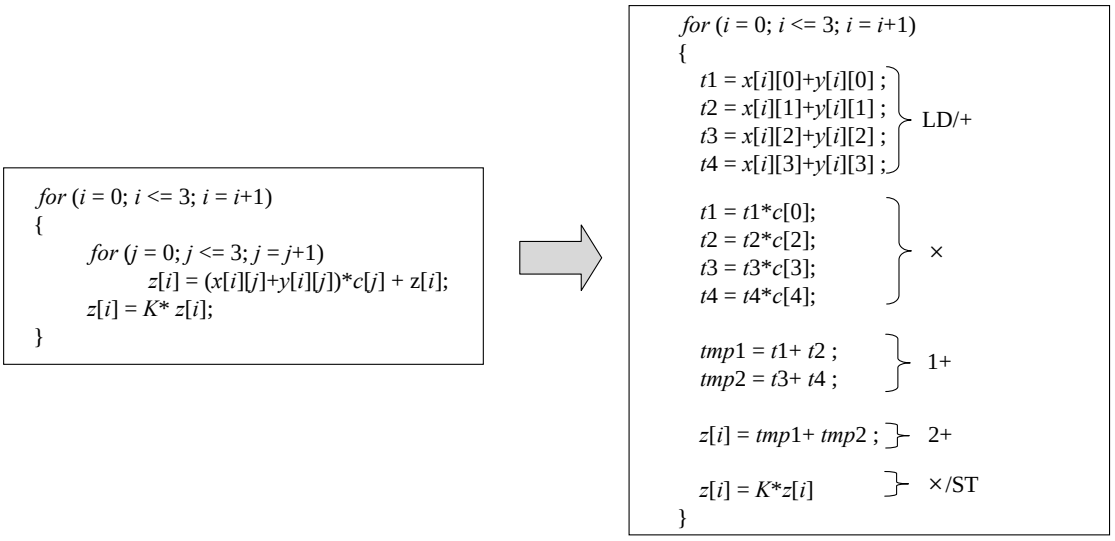
Fig. 10. 4x4 reconfigurable array.



(c) Nearest neighbor interconnection (d) Global bus interconnection

Fig. 10. Continued.

element composed of an ALU, an array multiplier, etc. and the configuration is controlled by the words stored in the CE as shown in Fig. 10 (a). In addition, we assume that Frame Buffer has simply one set having three banks and two read-ports and one write-port, supporting any combination of one-to-one mapping between the three banks and the three buses. Fig. 10 (b) shows such a Frame Buffer and data bus structure, where the PEs in each row of the array share two read buses and one write bus. The 4x4 array has nearest neighbor interconnections as shown in Fig. 10 (c) and each row or each column has a global bus as shown in Fig. 10 (d).



(a) Before parallelization (b) After parallelization

Fig. 11. C-code of Eq. (2).

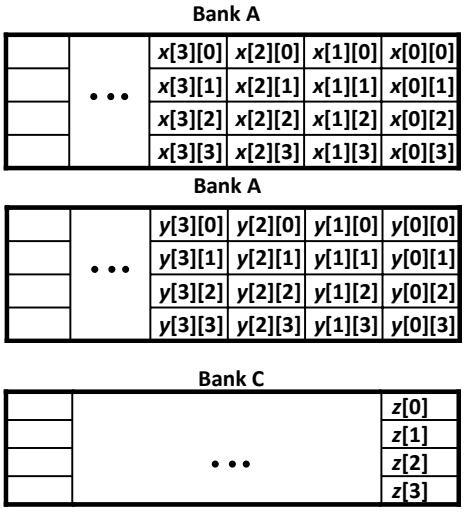
Consider a square matrix X and Y, both of order N, and the computation of Z, an N element vector, given by

$$Z(i) = K \times \sum_{j=0}^{N-1} \{(X(i,j) + Y(i,j) \times C(j)\} \tag{2}$$

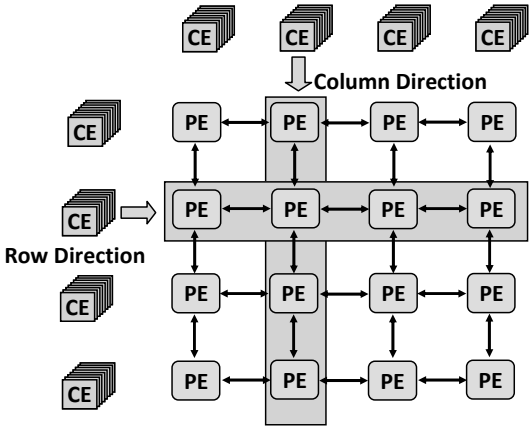
where $i, j = 0,1,\ldots,N\text{-}1$, $C(j)$ is a constant vector, and K is a constant.

Consider $N = 4$ for the mapping of the computation defined in Eq. (2) on our 4x4 PE array and let the computation be given as a C-program (Fig. 11 (a)). It is assumed that the input matrix X, Y, constant vector C and output vector Z are stored in the arrays $x[i][j]$, $y[i][j]$, $c[j]$ and $z[i]$, and $z[i]$ is initialized to zero. Fig. 11 (b) shows parallelized code for execution on the array as shown in Fig. 12, where we assume that matrix X and Y have been loaded into the Frame Buffer (FB) and all of the constants (C and K) have been already saved in a register file of each PE. Vector Z is stored in the FB after it has been

processed by the PE array as shown in Fig. 12 (a).



(a) Operand and result data in FB



(b) Configuration broadcast

Symbol	Meaning
LD/+	Data Load and Addition
NOP	No Operation
×	Multiplication
1+, 2+	Addition
×/ST	Multiplication and Store

Broadcast	Column Direction				Row Direction			Column Direction			
Cycle Time	1	2	3	4	5	6	7	8	9	10	11
Column#1	LD/+	NOP	NOP	NOP	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	NOP	NOP	×	1+	2+	NOP	×/ST	NOP	NOP
Column#3			LD/+	NOP	×	1+	2+	NOP	NOP	×/ST	NOP
Column#4				LD/+	×	1+	2+	NOP	NOP	NOP	×/ST

(c) SIMD model

Cycle Time	1	2	3	4	5	6	7	8
Column#1	LD/+	×	1+	2+	×/ST	NOP	NOP	NOP
Column#2		LD/+	×	1+	2+	×/ST	NOP	NOP
Column#3			LD/+	×	1+	2+	×/ST	NOP
Column#4				LD/+	×	1+	2+	×/ST

(d) Loop pipelining schedule

Fig. 12. Execution model for CGRA.

The SIMD-based scheduling enables parallel execution of multiple loop iterations as shown in Fig. 12 (c), whereas the MIMD-based scheduling enables loop pipelining as shown in Fig. 12 (d). The first row of Fig. 12 (c) represents the direction of configuration broadcast. The second row of Fig. 12 (c) and the first row of Fig. 12 (d) indicate the schedule time in cycles from the start of the loop. In the case of SIMD model, load and addition operations in PEs are executed on all columns till 4th cycle with broadcast in column direction. Then the PEs in a row perform the same operation with broadcast in row direction. In the case of loop pipelining, PEs in the first column perform load and addition operations in the first cycle and then perform multiplications in the second cycle. In the next two cycles, the PEs in the first column perform summations, while the PEs in the next column perform multiplication and summation operations. When the first column performs the multiplication/store operation in the 5th cycle, the fourth column performs multiplication. Comparing the latency, SIMD takes three more cycles.

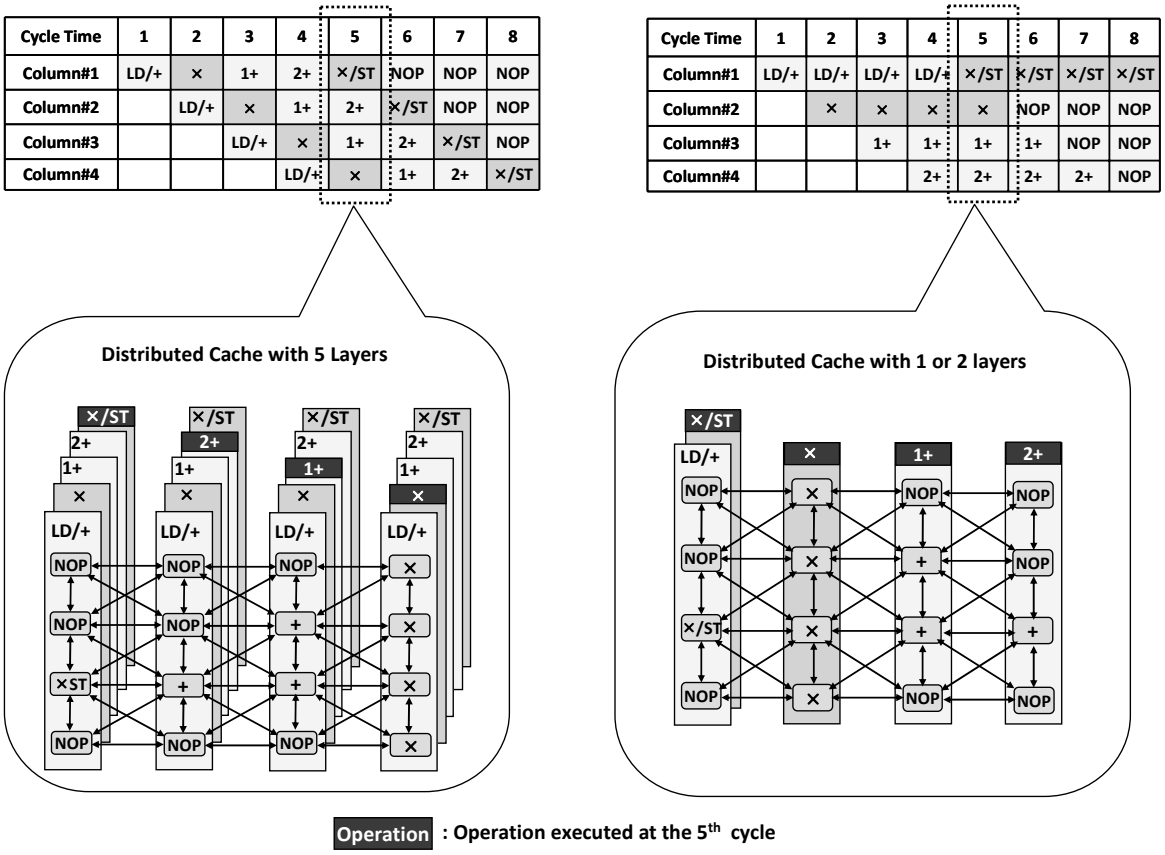
As shown in this example, SIMD model does not utilize PEs efficiently since all data should be loaded before the computations of the same type are performed synchronously. On the other hand, since MIMD allows any type of computations at any moment, it does not need to wait for a specific data to be loaded but can process other data that is readily available. Loop pipelining is an effective way of exploiting this fact, thereby utilizing PEs better. The loop pipelining in the example of Fig. 11 improves the performance by three cycles compared to the SIMD, but for loops with more frequent memory operations, it will have higher performance improvement.

2. Spatial Mapping and Temporal Mapping

When mapping kernels onto the reconfigurable architecture with loop pipelining, we can consider two mapping techniques [32]: spatial mapping and temporal mapping. Fig. 13 shows the difference between the two techniques with the previous example. In the case of temporal mapping (Fig. 13 (a)), like the previous illustration of loop pipelining in Fig. 12 (d), a PE executes multiple operations within a loop by changing the configuration dynamically. Therefore, complex loops having many operations with heavy data dependencies can be mapped better in temporal fashion, provided that the configuration cache has sufficient layers to execute the whole loop body.

In the case of spatial mapping, a loop body is spatially mapped onto the reconfigurable array implying that each PE executes a fixed operation with static configuration as shown in Fig. 13 (b). The advantage of spatial mapping is that it may not need reconfiguration during execution of a loop. As can be seen from Fig. 13, spatial mapping needs only one or two cache layers whereas temporal mapping needs 4 cache layers. One disadvantage of spatial mapping is that spreading all the operations of the loop body over the limited reconfigurable array may require too many resources. Moreover, data dependencies between the operations should be taken care of by allocating interconnect resources to provide a path and inserting registers (or using PEs) in the path to synchronize the arrival of operands. Therefore, if the loop is simple enough to map the loop body to the limited reconfigurable array and there is not much data dependency between the operations, then spatial mapping is the right choice. The effectiveness of the mapping strategies depends on the characteristics of the target architecture as well as the target

application.



(a) Temporal mapping (b) Spatial mapping

Fig. 13. Comparison between temporal mapping and spatial mapping.

B. Individual Approaches to Reduce Power in Configuration Cache

In this section, we suggest individual power-conscious approaches for two different execution models (spatial mapping and temporal mapping) and describe their limitations. These approaches achieve the goal by making use of the characteristics of spatial mapping and temporal mapping [52][53][54].

1. Spatial Mapping with Context Reuse

Because most power consumption in the configuration cache is due to memory read-operations, one of the most effective ways to achieve power reduction in the configuration cache is to reduce the frequency of read operations.

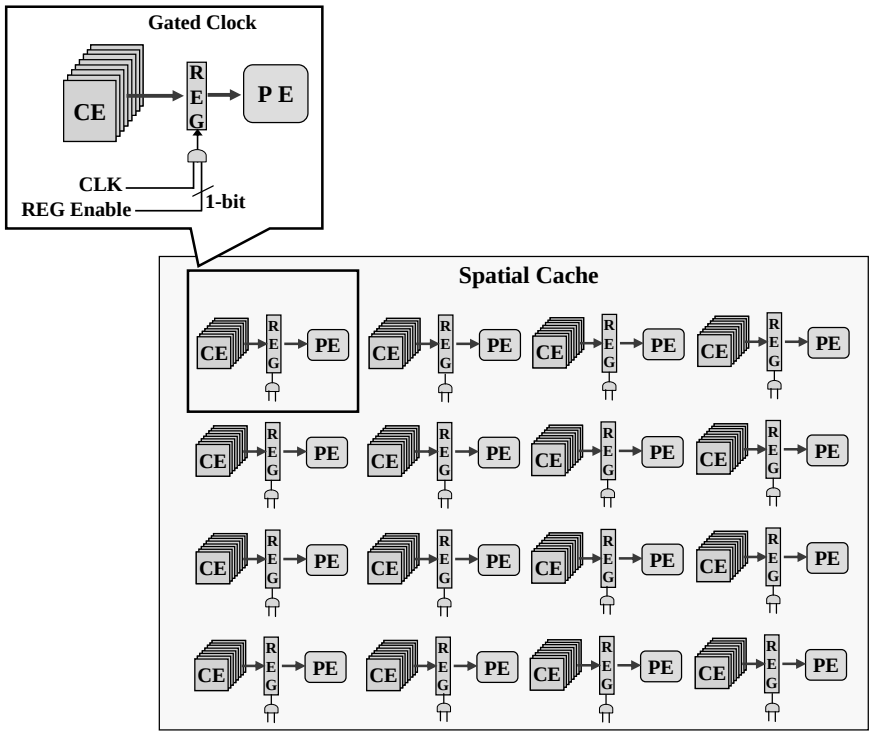


Fig. 14. Configuration cache structure for context reuse.

Even though temporal mapping is more efficient in mapping complex loops onto the reconfigurable array, it requires many configuration data layers for each PE and performs power consuming read-operations in every cycle. On the other hand, spatial mapping does not need to read a new context word from the cache every cycle because each

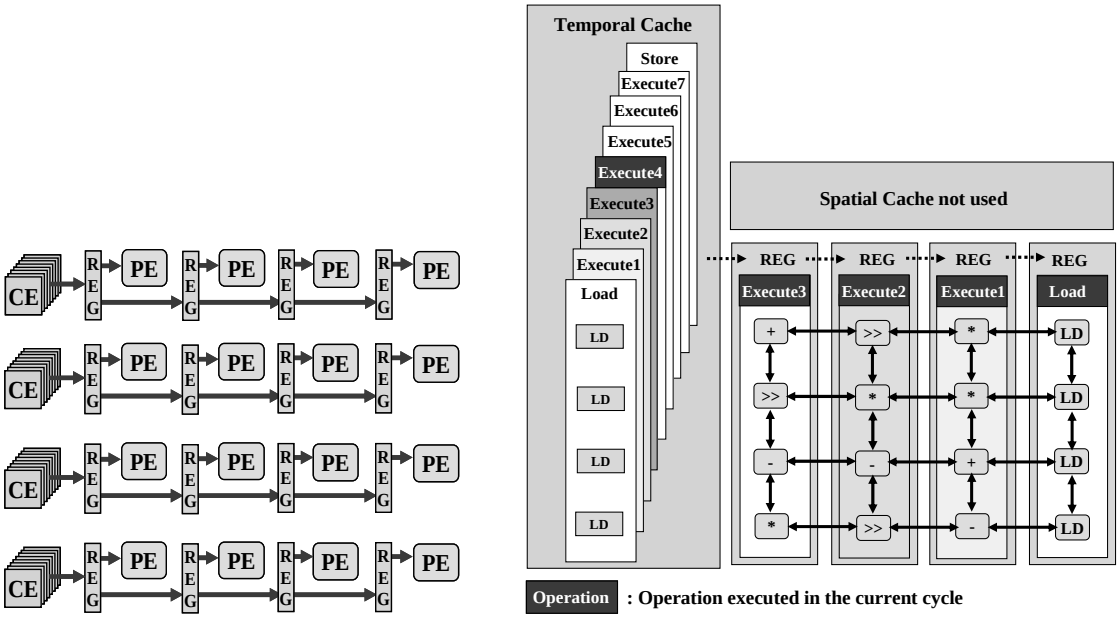
PE executes a fixed operation within a loop. As shown in Fig. 14, if a context register between a CE and a PE is implemented by a gated clock, one spatial cache[1] read-operation is enough in spatial mapping to configure PEs for static operations with fixed output of the context register caused by non-oscillated clock. In summary, spatial mapping with context reuse is more efficient than temporal mapping from the viewpoint of power consumption in configuration cache. However, all kinds of loops cannot be spatially mapped because of the limitation of the spatial mapping. Moreover, if we consider performance alone, temporal mapping is a better choice for loops having long and complex loop body. In the next subsection, we propose a new cache structure and mapping technique that reduce power consumption while retaining the merits of temporal mapping.

2. Temporal Mapping with Context Pipelining

As shown in Fig. 13 (a), in temporal mapping with loop pipelining, operations flow column by column from left to right. In Fig. 13 (a) for example, the first column executes 'LD/+' in the first cycle and then in the second cycle, the second column executes 'LD/+' while the first column executes '×'. In temporal mapping, there is no need for a PE to have a CE. Instead, only PEs in the first column have CEs and the context word can be fetched from the left neighboring column. By organizing a pipelined cache structure as shown in Fig 15, we can propagate the context words column by column through the pipeline. In this way, we can remove most of the CEs from the array keeping temporal

[1] We use the term ‘spatial cache’. Spatial cache is connected to context registers implemented by gated clock. ‘spatial’ means that such configuration cache is used for spatial mapping with context reuse. This naming is to differentiate spatial cache from general configuration cache.

cache[2], thereby saving power consumption without any performance degradation. In summary, temporal mapping with context pipelining can efficiently support long and complex loops reducing power consumption in configuration cache. However, temporal mapping with context pipelining still needs cache-read operations for providing context words to the first column of PE array whereas spatial mapping with context reuse can remove cache-read operation after initial cache-read operation.



(a) Cache structure (b) Context pipelining

Fig. 15. Cache structure for context pipelining.

3. Limitation of Individual Approaches

As mentioned in previous section, even though individual low power techniques provide

[2] We use the term ‘temporal cache’. Temporal cache is composed of the cache elements connected to the PEs in the first column. ‘temporal’ means that such CEs are used for temporal mapping with context pipelining. This naming is to differentiate temporal cache from general configuration cache and spatial cache.

solution to reduce power consumption for spatial mapping and temporal mapping, each case has both advantage and disadvantage. Spatial mapping with context reuse only need one cache-read operation for initialization but it can not support the complex loops that cannot be spatially mapped. However, temporal mapping with context pipelining support such complex loops but cache-read operations still remain in context pipelining for the running time. Therefore we should consider the trade-off between performance and power while deploying these techniques.

We can consider two ways to close the gap between spatial mapping and temporal mapping. One is to implement more complex architecture to support high performance spatial mapping by adding additional interconnections or global register files for data dependency. However, in this case the area cost and mapping complexities will increase. Another way is to implement low power temporal mapping taking advantage of spatial mapping with negligible over-head. However, the problem is how to implement this method. In the next section, we propose new technique to guarantee the advantage of spatial mapping and temporal mapping. This is achieved by merging the concept of context reuse into context pipelining.

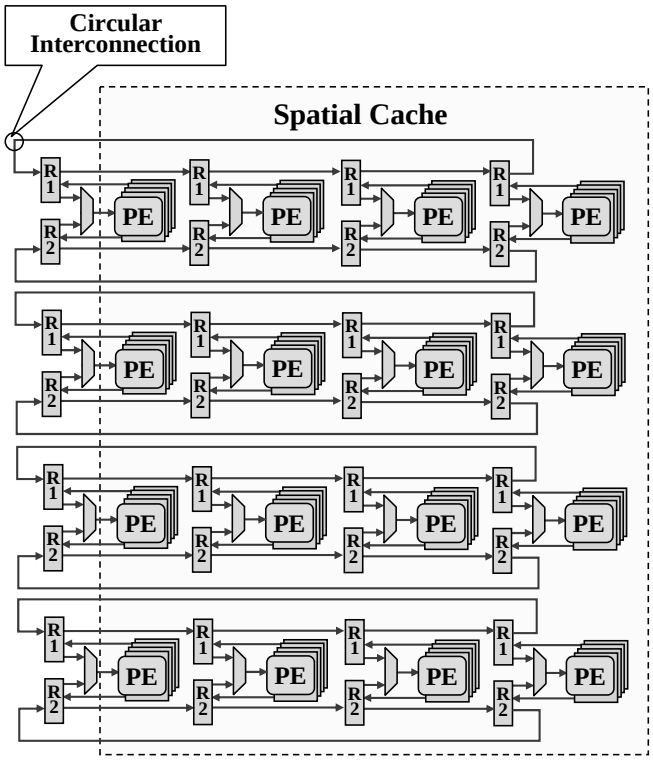
C. Integrated Approach to Reduce Power in Configuration Cache

Filling the gap between two mappings means that context pipelining is executed by reusable context words. However, it means conjunction of two mappings that are contrary to each other. This is because spatial mapping with context reuse requires spatially static position of each context whereas temporal mapping with context pipelining is performed with temporally changed context words. To solve this contradiction, we propose to add

circular interconnection between the first PE and the last PE in the same row and suggest a reusable context pipelining using this interconnection.

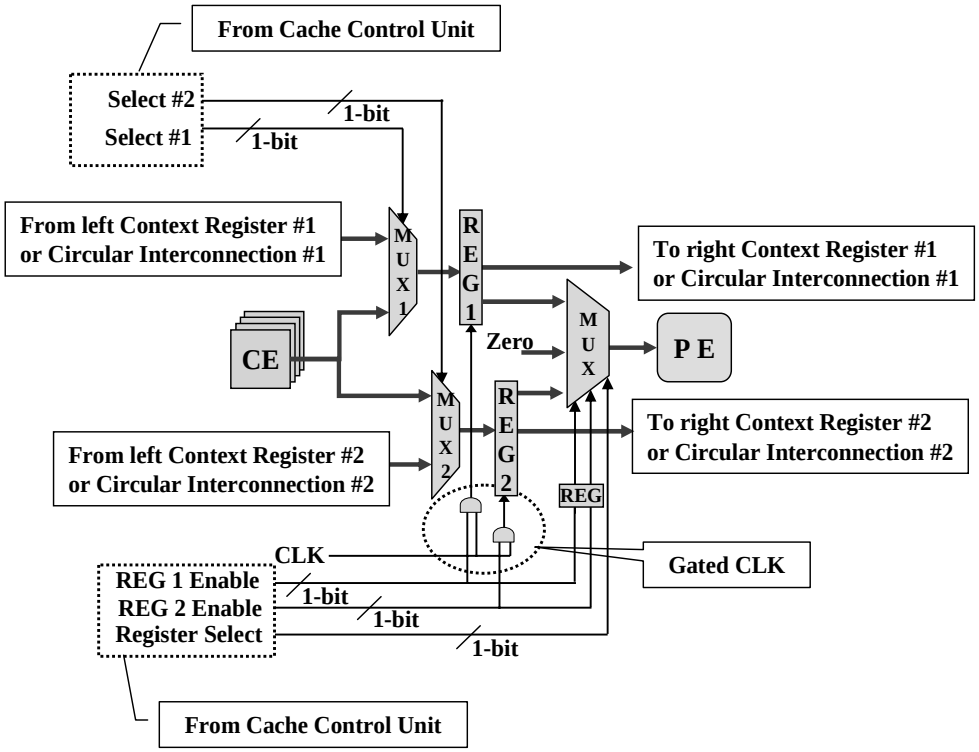
1. Reusable Context Pipelining

Reusable context pipelining (RCP) means that reusable context words in spatial cache are pipelined through context registers as context pipelining. Fig. 16 (a) depicts the proposed configuration cache structure for RCP. Even though it is similar to the structure of Fig. 14 (spatial mapping with context reuse), the new one has two context registers (‘R1’ and ‘R2’) connected to each PE, circular interconnections and less cache layers whereas the original model had one context register and more cache layers.



(a) Entire structure

Fig. 16. Proposed configuration cache structure.



(b) Connection between a CE and a PE

Fig. 16. Continued.

The circular interconnections and the context registers are necessary for pipelining of reusable context words from spatial cache. Fig. 16 (b) shows the detailed structure between a CE and a PE for RCP. A multiplexer (‘MUX’) is added between context registers (‘REG1’ and ‘REG2’) and PE for selecting one of the context registers or ‘Zero’. Each context register is connected to each multiplexer (‘MUX 1’ or ‘MUX 2’) having two inputs: context word from left context register and context word from spatial cache. The input from spatial cache is for loading a reusable context word to the context register and the input from left context register is for pipelining execution of the loaded con-

text word in left context register. Each select signal (‘Select #1’ or ‘Select #2’) connects one from two inputs to the single output connected with right context registers. Each context register is implemented by gated clock for holding the output as well as reducing the wasteful power consumption. All of the select-signals of the multiplexers are generated by cache control unit.

To present the detailed process of RCP, it is assumed that the matrix-vector multiplication given as Eq. (2) is mapped onto the proposed structure like the one in Fig. 17. Fig 17 (a) shows the context words stored in spatial cache for RCP and Fig. 17 (b) ~ (i) shows the RCP process from the first cycle to the eighth cycle. Before starting execution, the cont ext words of first layer in spatial cache are loaded into the first context registers (‘REG 1’). At the first cycle, the PEs in the first column performs ‘Load’ and the context word (‘Store’) in spatial cache is loaded to the ‘REG 2’ in the first column while other columns perform no operation (‘NOP’). At the second cycle, the first column performs ‘Execute1’ from circular interconnection while PEs in the next column perform ‘Load’ from the first column. Then context words in the first registers are sequentially pipelined for two cycles (the third and forth cycle) and the first column perform ‘Store’ from the second register at the fifth cycle. Such a context pipelining is continually executed and finished at the eighth cycle. Therefore, if reusable context words are loaded into context registers in the circular order, the context words from spatial cache can be rotated for temporal mapping without temporal cache. It means that spatiality of the array structure and the added context registers can be utilized for low power in temporal mapping.

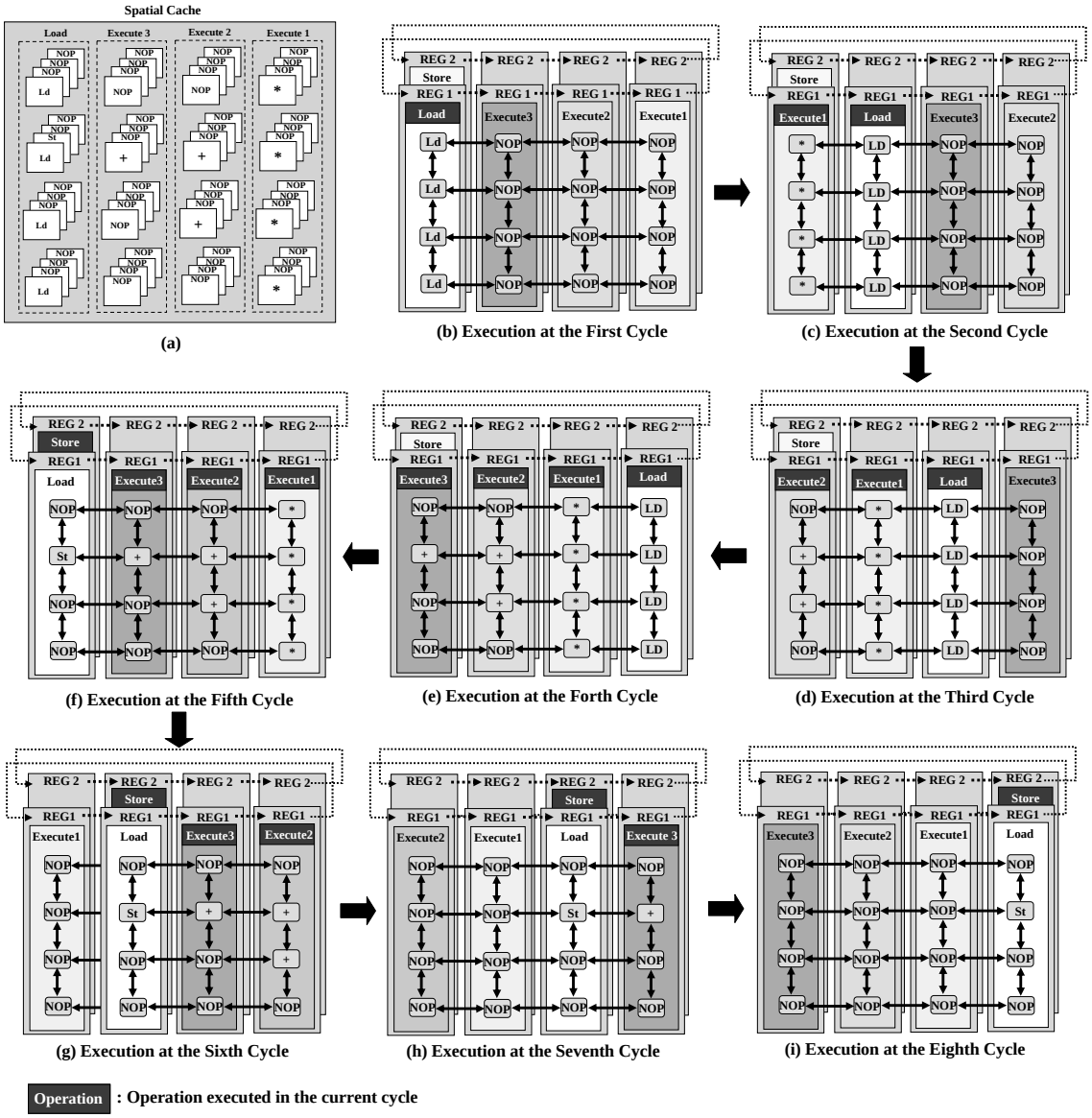


Fig. 17. Reusable context pipelining for Eq. (2).

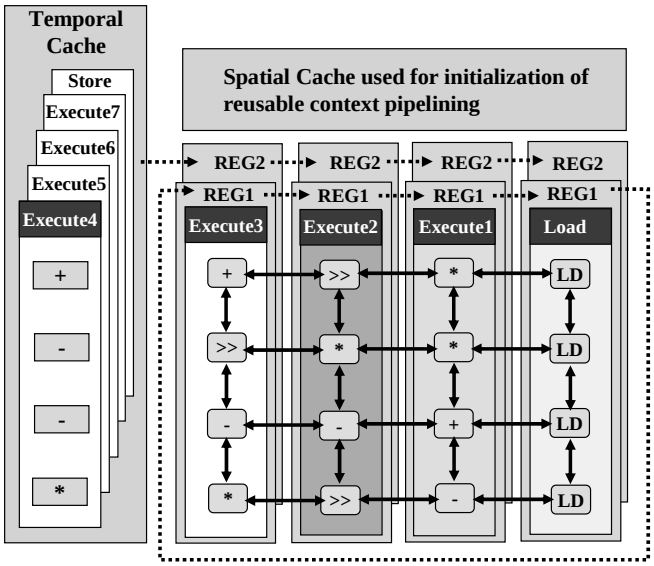


Fig. 18. Reusable context pipelining with temporal cache.

2. Limitation of Reusable Context Pipelining

If the loop given in Fig. 15 (b) is mapped onto the 4x4 PE array with added context registers like Fig. 16 (a), RCP cannot finish entire execution because the given architecture only supports a maximum number of 8 cycles (2 context registers and 4 columns) for an iteration of the loop whereas the loop has loop body taking 9 cycles. Therefore, in this case, temporal cache is necessary to support the entire execution as Fig. 18 - RCP is performed for 4 cycles by register 1 and original context pipelining is performed for 5 cycles by register 2. Hence, RCP guarantees reduction of 4 cache-read operations after execution of the first iteration. This example shows that power efficiency of reusable context pipelining can be varied according to the complexity of evaluated loops and architecture specification.

	Cycle	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9
Col#1	Cache/REG	REG1	REG1	REG1	REG1	Cache	Cache	Cache	Cache	Cache
	Operation	LD	EX1	EX2	EX3	EX4	EX5	EX6	EX7	ST

(a) i^{th} iteration in the case of loop body taking 9 cycles

	Cycle	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8
Col#1	Cache/REG	REG1	REG1	REG1	REG1	REG2	REG2	REG2	REG2
	Operation	LD	EX1	EX2	EX3	EX4	EX5	EX6	ST

(b)) i^{th} iteration in the case of loop body taking 8 cycles

Fig. 19. Reusable context pipelining according to the execution time for one iteration (i > 1).

Therefore, we can estimate how many cache-read operations occur after the first iteration under architecture constraints. This is given as follows:

$$N_{Tcache\_read} = \begin{cases} 0 & if \quad C_{iter} \leq m \times N_{ctxt} \quad (3) \\ C_{iter} - m \times (N_{ctxt} - 1) & if \quad C_{iter} > m \times N_{ctxt} \quad (4) \end{cases}$$

where

- N_{Tcache_read} : cycle count of temporal cache-read operations after the first iteration
- C_{iter} : cycle count for an iteration of loop
- m : number of columns on reconfigurable array
- N_{ctxt} : number of context registers for a PE

Based on above formula, the optimal case is when the N_{Tcache_read} is zero - context registers are sufficient to support entire loop body without temporal cache read-operations after the first iteration. Fig. 19 shows two cases of temporal mapping with RCP after the first iteration. In the case of Fig. 19 (a), it shows the scheduling for previous example in Fig. 18 and it corresponds to Eq. (4). However, Fig. 19 (b) shows other case that execution time for an iteration is 8 cycles and it corresponds to Eq. (3).

3. Hybrid Configuration Cache Structure

Based on modified interconnection structure as in Fig. 16 (b), we propose a power-conscious configuration cache structure that supports reusable context pipelining - we call it hybrid configuration cache including two cache parts – spatial cache for reusable context pipelining and temporal cache to making up for the limitation of RCP. Fig. 20 shows the modified configuration cache structure to support the example given in Fig. 18. It is composed of cache controller, spatial cache, temporal cache, multiplexer and de-multiplexer. The cache controller supports the same functions as the previous controller and in addition it controls increased context registers as well as the selection between spatial cache and temporal cache. Therefore, the new cache controller is more complex than the base one but the cache controller supports reusable context pipelining with negligible area and power overheads. As compared to the distributed cache of base architecture, both spatial cache and temporal cache have much less number of layers since spatial mapping does not require many layers and RCP can save the layer of temporal cache by up to the number of columns using context registers - the number of spatial cache layers should be more than the number of context registers connected to a PE because spatial cache should be able to include context words of several applications. Therefore, the area cost overhead caused by added context registers offsets because temporal cache size can be reduced by same size of total added registers. As mentioned earlier, the approach does not incur any performance degradation and this hybrid structure saves cache area since we keep only one column with reduced number of temporal CEs and less layers of spatial CEs compared to distributed configuration cache that has much more layers

of CEs.

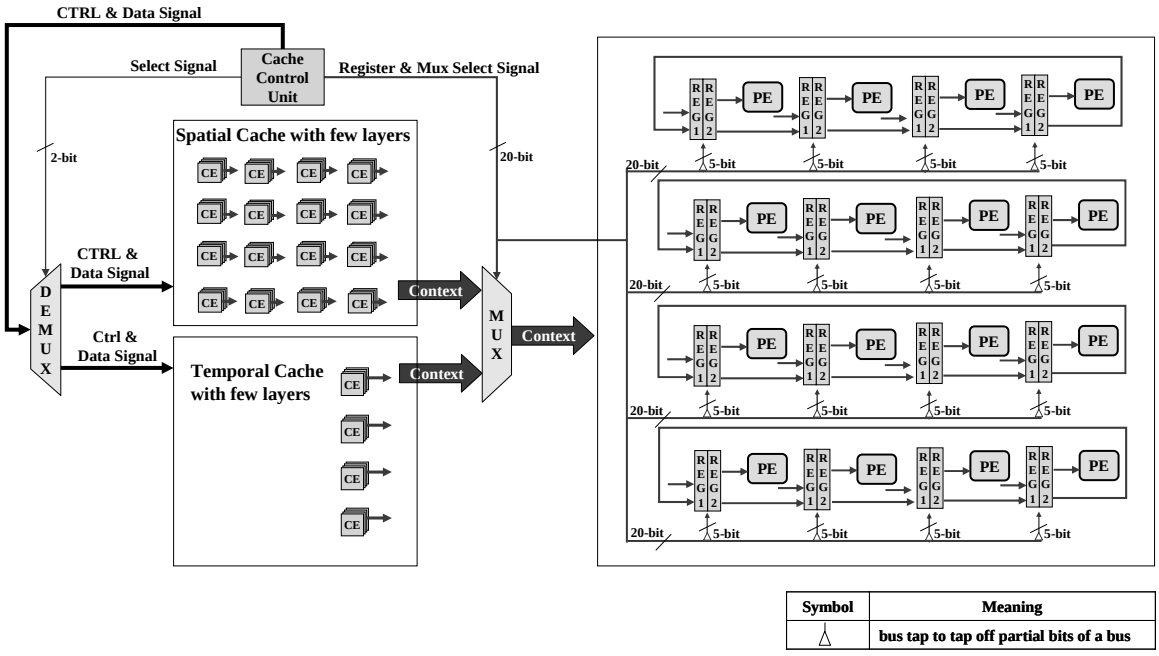


Fig. 20. Hybrid configuration cache structure.

D. Application Mapping Flow

We have implemented automatic compilation flow to map applications onto the base architecture for supporting temporal mapping [37]. The binary context words for reusable context pipelining are basically the same as the context words used for the temporal mapping but these context words should be rearranged for context pipelining with circular interconnection. Fig. 21 shows entire mapping flow for the base architecture and proposed architecture. Binary context words are automatically generated from the compiler for temporal mapping. The timing and control information that is used to operate execution controller is manually optimized and the final encoded data is loaded onto registers of the execution controller.

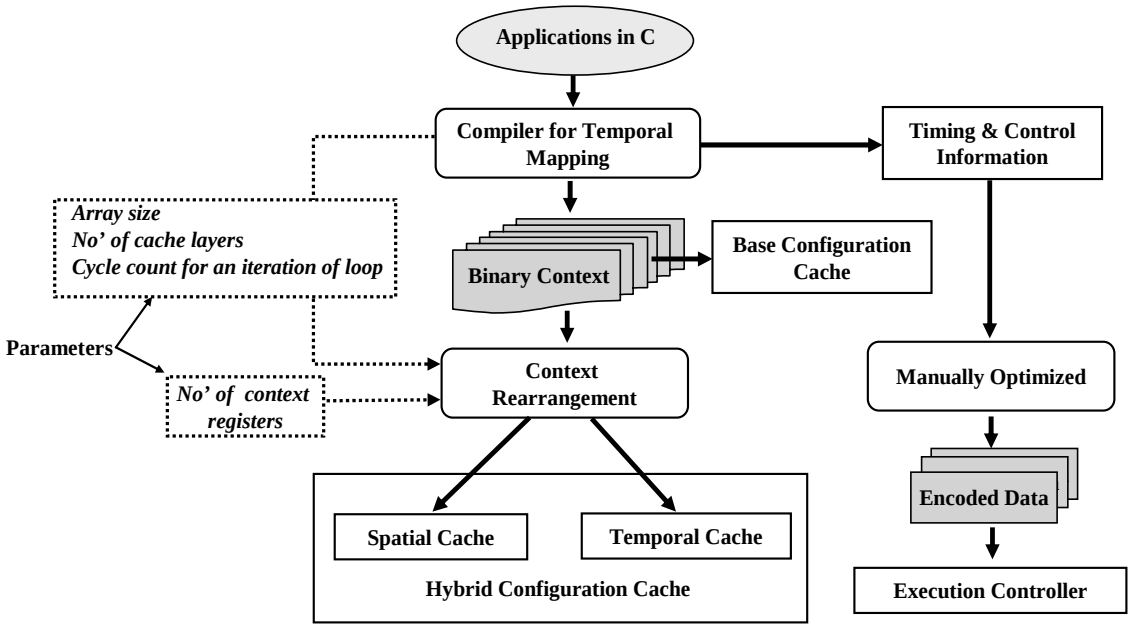


Fig. 21. Application mapping flow for base architecture and proposed architecture.

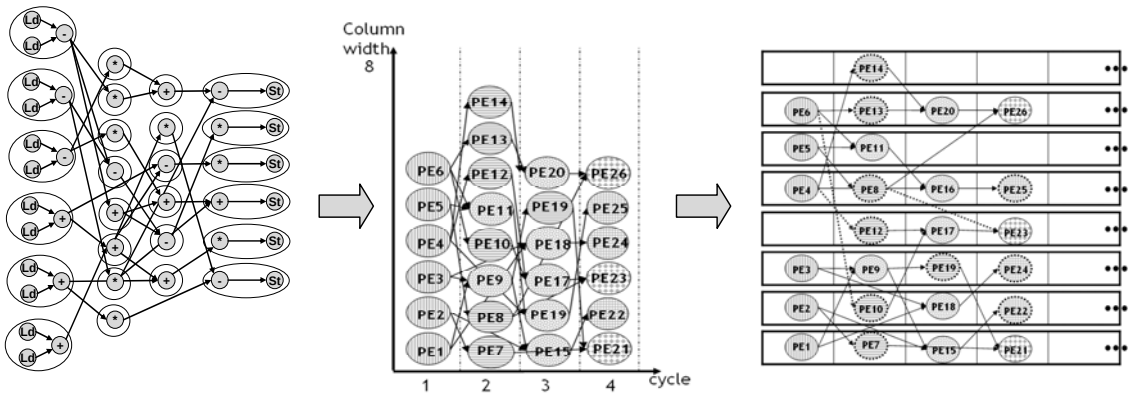
1. Temporal Mapping Algorithm

The temporal mapping algorithm minimizes the execution time of kernel codes on the PE array. This execution time is directly proportional to the number of cache layers in configuration array. The time, $T_{critical}$ is considered as a parameter to be minimized during temporal mapping. We implement the temporal mapping in three sequential steps: covering, time assignment, and place assignment.

a. Covering

For compilation, the original kernel code is initially transformed into a DAG form, called the kernel DAG using common sub expression elimination technique [55]. One or more operation nodes in a kernel DAG are scheduled in a single configuration of a PE.

For this, we generate a configuration DAG (CDAG) by clustering the nodes in kernel DAG. A CDAG is used to find the minimum number of configurations for kernel code execution. To perform this task, we formulate it into a DAG covering problem where one has to find the minimal cost set of patterns that cover all the nodes in input CDAG. To efficiently solve our DAG covering problem, we implement our algorithm based on binate covering [56]. For example, Fig. 22 (a) shows CDAG generation from an input DAG.



(a) Covering (b) Time assignment (c) Place assignment

Fig. 22. Temporal mapping steps.

b. Time Assignment

Each node in the CDAG is assigned to a cycle in which the node will be executed. In order to minimize $T_{critical}$, we must fully exploit the parallel resources provided by the $m \times n$ PE array using modulo scheduling [57]. For example, Fig. 22 (b) shows assignment schedule obtained after applying modulo scheduling to the CDAG. Note that the cycle in

which a node in the CDAG is scheduled as part of a configuration in this phase, and it represents a layer location inside a configuration cache.

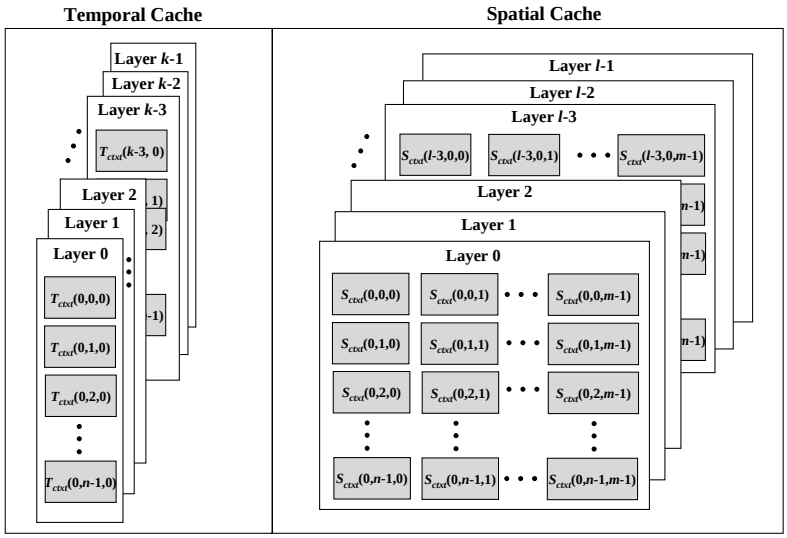
c. Place Assignment

In this phase, we assign all nodes in the CDAG to actual PEs by storing each of them as a configuration entity in the cache of a PE. We split the PEs in a column into two groups, called slots. In this phase, the CDAG nodes are first assigned to either slot with resorting to the ILP solver, and then within each slot, nodes are finally mapped onto actual PEs. Fig. 22 (c) shows the final mapping results after a place assignment is deployed.

2. Context Rearrangement

In the case of base architecture, the binary context words generated from the compiler can be loaded into configuration cache without any modification. However, in the case of proposed architecture the generated context words are rearranged and properly assigned to spatial and temporal cache. The address of each context word in hybrid configuration cache can be represented by three-dimensional position as Fig. 23 (a). Fig. 23 (b) shows pseudo code for context rearrangement algorithm that is easily implemented based on Eq. (3) and (4). Before explaining the algorithm in detail, we introduce the notations used in the algorithm - N_{Tcache_read} , C_{iter}, N_{ctxt} and m are defined in subsection C.2.

- n : number of rows in reconfigurable array
- k, l : number of temporal cache layers, the number of spatial cache layers
- T_{ctxt} , : set of the context words having positions in temporal cache
- S_{ctxt} : set of the context words having positions in spatial cache
- $T_{ctxt}(x, y, z)$: context word corresponding the position (x, y, z) in temporal cache
- $S_{ctxt}(x, y, z)$: context word corresponding the position (x, y, z) in spatial cache
  (x : layer index, y : row index, z : column index)



(a) positions of binary contexts in hybrid configuration cache

CONTEXT REARRANGEMENT (T_{ctxt}, m, n, k, C_{iter}, N_{ctxt})

```
L1   S_ctxt ← Ø
L2   p ← 0
L3   r ← 0
L4   u ← 0
L5   if   C_iter ≤ m×N_ctxt
L6        then for i ← 0 to k-1
L7                  do  for j ← 0 to n-1
L8                            do S_ctxt(p, j, r) ← T_ctxt(i, j,0)
L9                        if r = 0
L10                         then r ← m - 1
L11                             else if  r > 1
L12                                  then r ← r – 1
L13                       else r ← 0, p ← p + 1
L14       else u ← m×(N_ctxt-1)
L15             for i ← 0 to k-1
L16                  do  if i ≤ u
L17                       then for j ← 0 to n-1
L18                                 do S_ctxt(p, j, r) ← T_ctxt(i, j,0)
L19                            if r = 0
L20                              then r ← m - 1
L21                              else if  r > 1
L22                                     then r ← r – 1
L23                              else r ← 0, p ← p + 1
L24                       else for j ← 0 to n-1
L25                                 do T_ctxt(i-u-1, j,0)  ← T_ctxt(i, j,0)
L26   return T_ctxt, S_ctxt
```

(b) rearrangement algorithm.

Fig. 23. Context rearrangement.

The code between L1 and L4 initialize temporary variables (p, r, u) and S_{ctxt}. If $N_{ctxt} \times m$ is sufficient to support entire loop body without temporal cache read-operations (L5), all of the context positions in temporal cache are remapped to the positions in spatial cache with rearrangement in the circular order (L6 ~ L13). Otherwise, the limited number of temporal cache layers which can be executed by reusable context pipelining is estimated (L14), all of the context positions within the limited temporal cache layers are remapped to the positions in spatial cache (L16 ~ L23) in the same manner as (L6 ~ L13). Then the layer indices of context positions remaining in temporal cache are updated to fill up the empty layers.

E. Experiments

1. Experimental Setup

For a fair comparison between the base model and the proposed one, we have implemented two cases of reconfigurable architectures as given in Table I. Base architecture is as specified in Chapter III. Proposed architecture is same as base architecture but also includes increased context registers and hybrid configuration cache to support reusable context pipelining. Two models have been designed at RT-level with VHDL and synthesized using Design Compiler [49] with 0.18 μm technology. We have used SRAM Macro Cell library for the frame buffer and configuration cache. ModelSim [50] and PrimePower [49] have been used for gate- level simulation and power estimation respectively. To estimate the power consumption overhead in the proposed model, the context registers and multiplexers in each case (previous model and proposed architecture) have been separated from the PE array and those have been included in the configuration

cache for each model while implementation. To obtain the power consumption data, we have used various kernels (Table II) for simulation with same simulation conditions as the previous one mentioned in Chapter III (subsection C.3) - operation frequency of 100 MHz and typical case of 1.8 V Vdd and 27℃. We have implemented the context rearrangement algorithm (Fig. 23) in C++ and the application mapping flow as given in Fig. 21 by adding the algorithm to the compiler for temporal mapping.

2. Results

a. Necessary Context Registers for Evaluated Kernels

We have applied several kernels of Livermore loops benchmark [58], DSPstone [59] and representative loops in MPEG-4 AAC decoder, H.263 encoder and H.264 decoder to the base and proposed architectures. To determine necessary number of context registers to support reusable context pipelining for selected kernels, we have analyzed each case of selected kernels and Table II shows execution cycle count for an iteration and necessary number of context registers for each kernel. In the case of 2D-FDCT, it shows 11 execution cycles and the maximum number of context registers among selected kernels. It means that composing a PE having 2 context registers is necessary to support reusable context pipelining for all of the selected kernels. Therefore, each PE in the proposed architecture has 2 context registers for reusable context pipelining while base architecture has one context register as shown in Table I.

Table I. Architecture Specification of Base and Proposed Architecture

Parameter		Base architecture	Proposed architecture
PE Array	Number of context registers for a PE	1	2
	Number of rows	8	8
	Number of columns	8	8
Frame buffer	Number of sets and banks	2 sets and 3 banks	2 sets and 3 banks
	Bit width	16-bit	16-bit
	Bank size	1 KB	1 KB
Configuration Cache	Number of layers for a CE	32	16
	Number of Cache Elements (CEs)	64	72
	Bit width of a CE	32-bit	32-bit

Table II. Necessary Context Registers for Evaluated Kernels

Kernels	Execution cycle count for an iteration	Necessary number of context registers
[a]First_Diff	10	2
[a]Tri-Diagonal	4	1
[a]Hydro	7	2
[a]ICCG	5	1
[b]Dot_Product	5	1
[b]24-Taps FIR	8	2
Complex Multiplication in MPEG-4 AAC decoder	10	2
ITRANS in H.264 decoder.	9	2
2D-FDCT in H.263 encoder.	11	2
SAD in H.263 encoder	5	1
Matrix(10x8)-Vector(8x1) Multiplication(MVM)	5	1

[a] Livermore loop benchmark suite. [b] DSPstone benchmark suite.

Table III. Size of Configuration Cache and Context Registers

Size of memory elements	Architecture		Reduced(%)
	Base	Proposed	
Context registers	256-Byte	512-Byte	-
Spatial cache	8192-Byte	4096-Byte	43.75
Temporal cache		512-Byte	
Total amount	8448-Byte	5120-Byte	**39.39**

b. Configuration Cache Size

Both temporal cache and spatial cache of the proposed architecture have 16 layers, which is half the size of the base architecture. Reducing cache size does not affect performance degradation of evaluated kernels - the size is sufficient to perform the selected kernels with reusable context pipelining. Table III shows memory size evaluation between the base architecture and the proposed one. It shows that added context registers offsets by reduction of temporal cache layers. Compared to the base architecture, we have reduced the size of memory elements by up to 39.39%. This means that reconfigurable architecture with new configuration cache structure is more efficient than previous one in terms of memory size and power saving.

c. Performance Evaluation

The execution cycle counts of the evaluated kernels on proposed architecture do not vary from the base architecture because the functionality of proposed architecture is same as the base model. It also indicates the reusable context pipelining does not cause performance degradation in terms of the execution cycle count. In addition, the synthesis results show that the critical path delay of the proposed architecture is same as the base model i.e. 8.96 ns. It indicates the proposed approach does not cause performance degradation

in terms of the critical path delay.

d. Power Evaluation

To demonstrate the effectiveness of our power-conscious approach, we have evaluated the power consumption of only base architecture with temporal mapping and proposed architecture with reusable context pipelining on hybrid configuration cache.

Table IV. Power Reduction Ratio by Reusable Context Pipelining

Kernels	Power(mW)				Reduced(%)	
	Cache		Entire		Cache	Entire
	base	proposed	base	proposed		
First_Diff	171.77	28.08	376.17	232.48	83.65	38.20
Tri- Diagonal	174.18	31.58	400.19	257.59	81.87	35.63
Dot_Product	117.84	29.87	328.54	240.57	74.65	26.78
Complex_Mult	180.63	32.82	452.00	304.19	81.83	32.70
Hydro	148.23	32.40	356.47	240.64	78.14	32.49
ICCG	205.80	32.64	434.45	261.29	84.14	39.86
24-Taps FIR	227.56	31.11	471.44	274.99	**86.33**	41.67
MVM	227.57	34.45	405.70	212.58	84.86	**47.60**
ITRANS	204.85	69.96	417.95	283.06	65.85	32.27
2D-FDCT	190.03	37.59	417.33	264.89	80.22	36.53
SAD	185.30	75.08	415.27	305.05	59.48	26.54

Table IV shows comparison of power consumption between the two architectures. Selected kernels were executed with 100 iterations. Compared to the base architecture, we have saved up to 86.33% of the total power consumed in the configuration cache and 47.60 % of that in the entire architecture using reusable context pipelining. These results

show that reusable context pipelining is a good solution for power saving in CGRA. ITRANS and SAD show less reduction in power compared to other kernels because they need additional spatial cache-read operations for data arrangement. In the case of 24-Taps FIR showing the maximum reduction ratio, the total power consumption of proposed architecture is much less than the result of PipeRench [6]. PipeRench has been fabricated in a 0.18 micron process and [6] shows power measurement with varying FIR filter tap sizes. The power consumption has been measured using a 33.3 MHz fabric clock and a 16.7 MHz IO clock. The power measurement shows that the power consumption of 24-Taps FIR ranges from 600 mW to 700 mW.

CHAPTER V

DYNAMIC CONTEXT COMPRESSION FOR LOW POWER CGRA

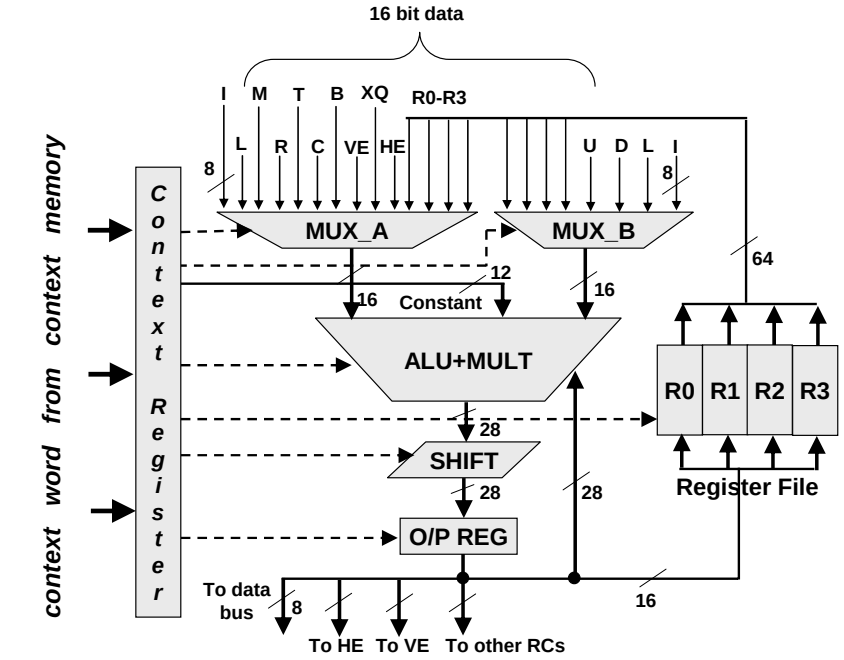
In this chapter, we address the power reduction issues in CGRA and provide a framework to achieve this. A new design flow and a new configuration cache structure are presented to reduce power consumption in configuration cache [60]. The power saving is achieved by dynamic context compression in the configuration cache – only required bits of the context words are set to enable and the redundant bits are set to disable. Therefore, the new design flow for CGRA has been proposed to generate architecture specifications that are required for supporting dynamically compressible context architecture without performance degradation. Experimental results show that the proposed approach saves up to 39.72% power in configuration cache with negligible area overhead (2.16%).

A. Preliminary

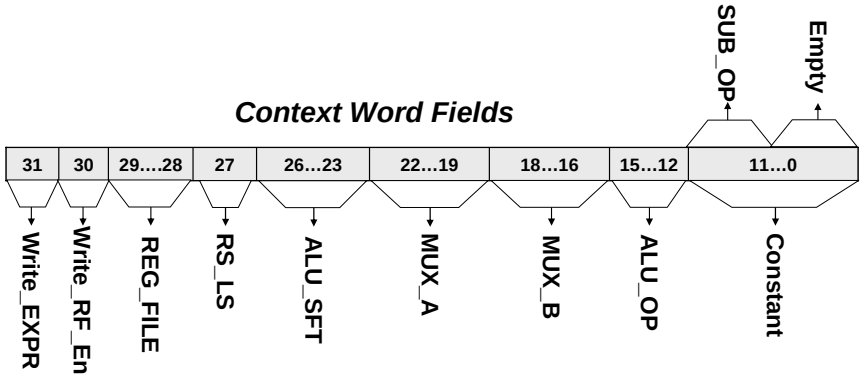
1. Context Architecture

The configuration cache provides context words to the context register of each PE on a cycle by cycle basis. From the context register, these context words configure the PEs. Fig. 24 shows an example of PE structure and context architecture for MorphoSys [3]. 32-bit context word specifies the function for the ALU-multiplier, the inputs to be selected from MUX_A and MUX_B, the amount and direction of shift of the ALU output, and the register for storing the result as Fig. 24 (a). Context architecture means organiza-

tion of context word with several fields to control resources in a PE as Fig. 24 (b). The



(a) PE structure



(b) Context architecture

Fig. 24. PE structure and context architecture of MorphoSys.

context architectures of other CGRAs such as [2][8][9][10][11][12][13][14] [15][16][17] [18] are similar to the case of MorphoSys although there is a wide variance in context-

width and kind of fields used by different functionality.

B. Motivation

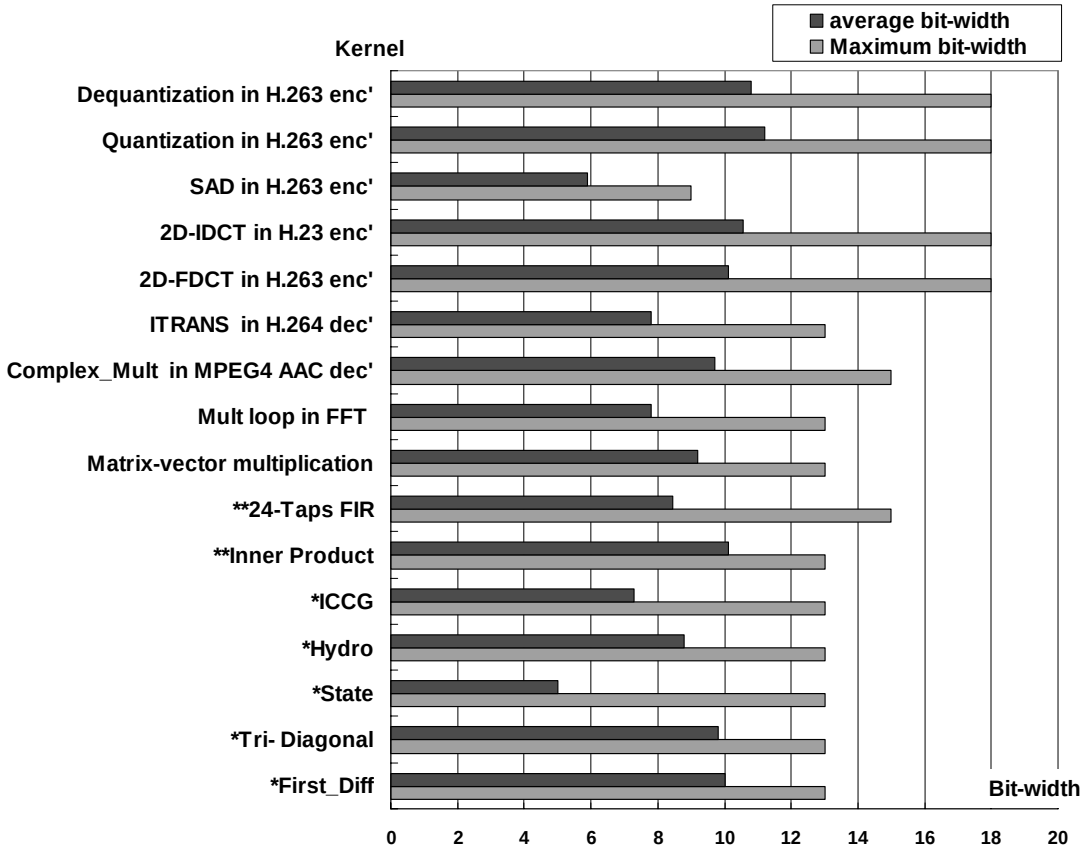
1. Power Consumption by Configuration Cache

By loading the context words from the configuration cache into the array, we can dynamically change the configuration of the entire array within just one cycle. However, such dynamic reconfiguration of CGRA causes many SRAM-read operations in configuration cache. In [6], the authors have fabricated a CGRA (PipeRench) in a 0.18 ㎛ process. Their experimental results show that the power consumption is significant high due to the dynamic reconfiguration requiring frequent configuration memory access. In Fig. 9, power break-down for the CGRA running 2D-FDCT is proposed with gate-level implementation at 0.18 ㎛ technology based on MorphoSys architecture. It is shown that the configuration cache spends about 43% of the overall power, which is the second largest after the PE arrays consuming 48% of overall power budget. This is because the configuration cache performs SRAM-read operations to load the context words in every cycle at run time. In addition, [8][30] also shows power break-down for another CGRA (ADRES) running IDCT based on 90nm technology. In this case, the configuration memory spends about 37.22% of the overall power. Therefore, it is explicit that power consumption by configuration cache (memory) is serious overhead compared to other types of IP cores such as ASIC or ASIP.

2. Valid Bit-Width of Context Words

When a kernel is mapped onto CGRA and application gets executed, the usable context

fields are limited to types of operations involved due to the kernel executed at run time.



*Livermore loops benchmark [58], **DSPstone [59]

Fig. 25. Valid bit-width of context words.

Furthermore, operation types of an executed kernel on PE array are changed in every cycle. It means the valid bit-width of executed context word is frequently less than the full bit-width of a context word even though full bit-width can be less often used.

For statistical evaluation of valid bit-width of contexts, we selected 32-bit context architecture of the base architecture (Fig. 4) and mapped several kernels onto its PE ar-

ray in order to maximize the utilization of the context fields. Fig. 25 shows the results for various benchmark kernels and critical loops in real applications. In Fig. 25, average bit-width is the average value of valid bit-widths of all the executed context words at run-time and the maximum bit-width is the maximal valid bit-width among all the context words considered at run-time. The statistical result shows that average bit-widths vary from 7 to 11 bits and the maximum bit-width is less than or equal to 18 bits whereas the full bit-width is 32-bit.

3. Dynamic Context Compression for Low Power CGRA

If the configuration cache can provide only required bits (valid bits) of the context words to PE array at run time, it is possible to reduce power consumption in configuration cache. The redundant bits of the context words can be set to disable and make those invalid at run time. That way, one can achieve low-power implementation of CGRA without performance degradation while context architecture dynamically supports both the cases at run time: one case is uncompressed context word with full bit-width and another case is compressed context word with setting unused part of configuration cache disabled. In order to support such a dynamic context compression, we propose a new context architecture and configuration cache structure in this chapter.

C. Design Flow of Dynamically Compressible Context Architecture

In order to design and evaluate dynamically compressible context architecture, we propose a new context architecture design flow. Entire design flow is shown in Fig. 26. This design starts from context architecture initialization, which is similar to the architecture

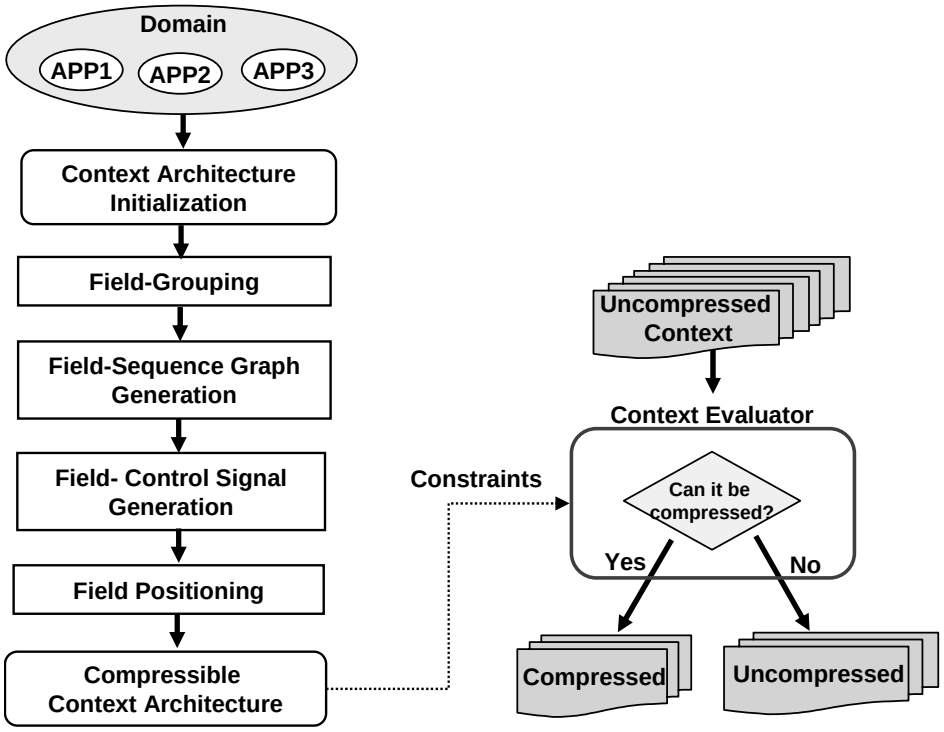
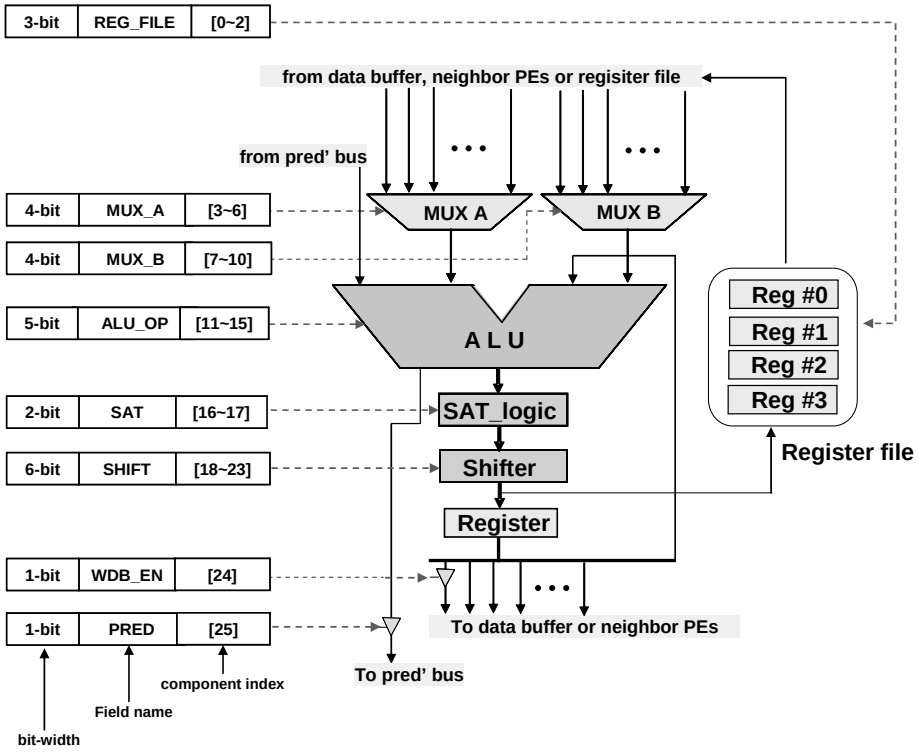


Fig. 26. Entire design flow.

specification stage of general CGRA design flow given in [21][22][27][29]. Based on such architecture specifications, PE operations are determined and initial context architecture is defined. From the context initialization, fields are grouped by essentiality of PE operation and dependency with ALU operation to provide some criterions for context compression. A field sequence graph (FSG) is generated to show possible field combinations for PE operation. Then field control signals are generated to make some field enable or disable when contexts are compressed. Based on former stages, the position of each field is defined and final context architecture is generated. Finally, one can determine whether the initially uncompressed contexts can be compressed or not by context evaluator. From subsection C.1 to subsection C.5, we describe more detailed process for

each stage in entire design flow.



(a) PE structure

Field name	Bit-width	Component index	Control
SHIFT	6-bit	[18,19,20,21,22,23]	Processing Element
ALU_OP	5-bit	[11,12,13,14,15]	
MUX_A	4-bit	[3,4,5,6]	
MUX_B	4-bit	[7,8,9,10]	
REG_FILE	3-bit	[0,1,2]	
SAT	2-bit	[16,17]	
WDB_EN	1-bit	[24]	
PRED	1-bit	[25]	
CTXT_CTRL	6-bit	-	Context register

(b) Context architecture initialization

Fig. 27. Context architecture initialization.

1. Context Architecture Initialization

Context rchitecture in CGRA design depends on architecture specification. In the process of architecture specification, CGRA structure is evolved with PE array size, PE functionalities and their interconnect scheme. The proposed approach starts from the conventional context architecture selection and makes it dynamically compressible context architecture through the proposed design flow. We have defined generic 32-bit context architecture as an example to illustrate the design flow to support the kernels in Fig. 25. It is similar to the representative CGRAs such as MorphoSys [3], REMARC [4], ADRES [8] [22][30][43], PACT_XPP [9][10][31]. The PE structure and bit-width of each field are shown in Fig. 27. It supports various arithmetic and logical operations with two operands (MUX_A and MUX_B), predicated execution (PRED), Arithmetic saturation (SAT_logic), shift operation (SHIFT) and saving temporal data with register file (REG_FILE). In Fig. 27 (a), all of the fields are classified by 'Control' of 2 cases - 'Processing element' and 'context register'. It means that each case is configured by the fields included in that case. Furthermore, Fig. 27 (b) shows the bit-width of each field and the component index to identify each component configured by each field.

Even though each field can be positioned on context word under conventional design flow, this initialization stage does not define any field position. It means field position for uncompressed case should be assigned by considering context compression.

2. Field Grouping

All of the context fields are grouped into three sets - necessary set, optional set and unnecessary set. Necessary set includes indispensable fields for all of the PE operations

and optional set includes optional fields for PE operations. Unnecessary set is composed of fields unrelated to PE operations. It means necessary fields should be included in context words even if context words are compressed whereas optional and unnecessary fields can be excluded out of context words. In addition, we classify optional set into two subsets. One is a subset composed of fields dependent on the field of 'ALU_OP' and another is a subset composed of fields independent of 'ALU_OP'. This classification is necessary for generating field control signals in subsection C.4. Fig. 28 shows field grouping based on the context initialization presented in subsection C.1.

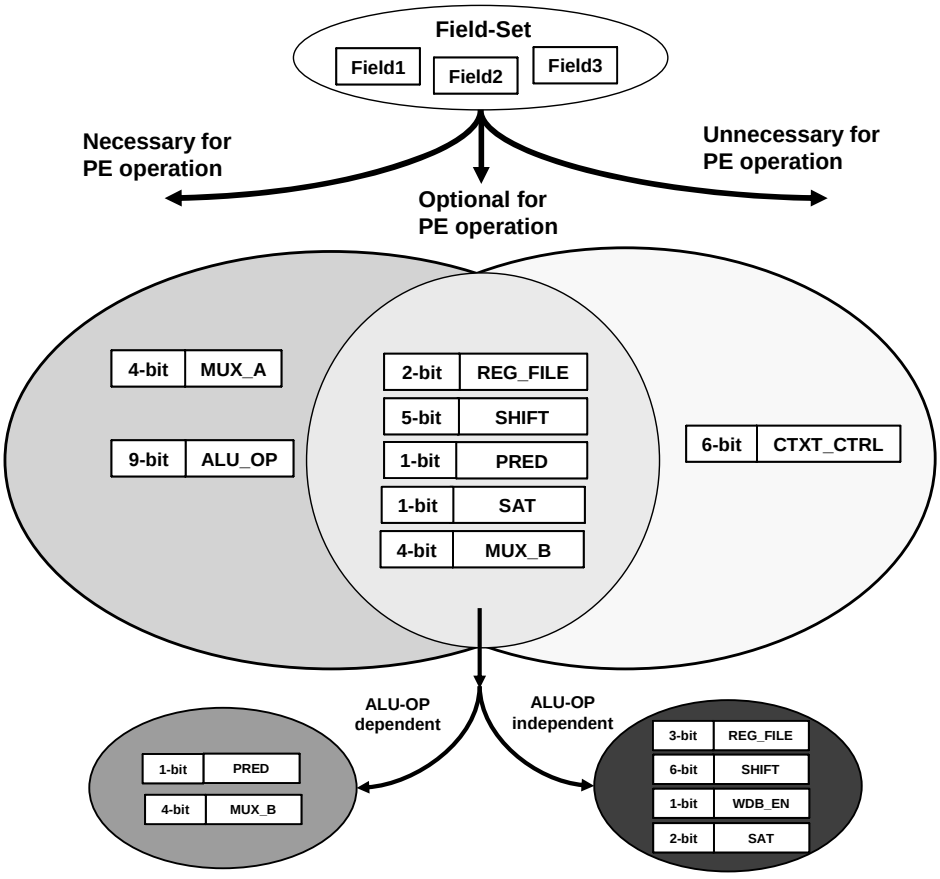


Fig. 28. Field grouping.

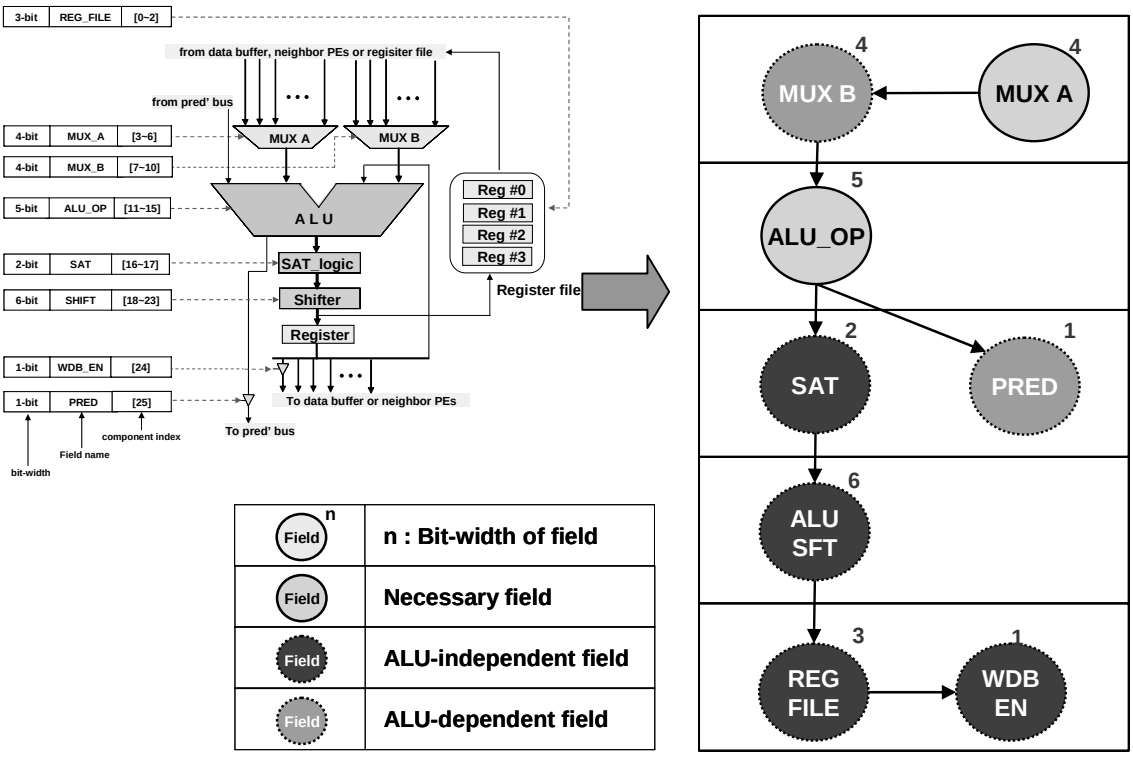


Fig. 29. Field sequence graph.

3. Field Sequence Graph Generation

Field sequence graph (FSG) is generated from context architecture initialization and field grouping. FSG is a directed graph composed of necessary and optional fields and it shows possible field combinations for PE operations based on PE structure. Each vertex of FSG corresponds to a necessary or optional field in field grouping and each edge of FSG shows a possible field combination between two fields. The possible field combinations can be found by vertex tracing in the edge directions and the combinations should include all of the necessary fields. Furthermore, optional fields can be skipped out of vertex tracing to search possible field combinations. Fig. 29 shows an example of FSG

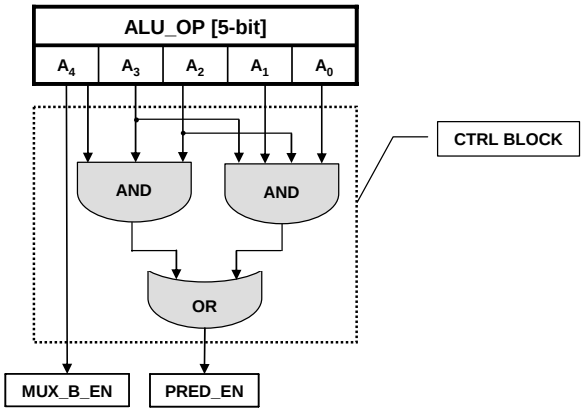
from Fig. 27 and Fig. 28. While searching possible field combinations, some times it is possible (for example, MUX_A, ALU_OP, SAT is possible) whereas (MUX_A, ALU_OP, SAT, PRED) is not possible. FSG is a useful data structure for field positioning as described in subsection C.5.

4. Generation of Field Control Signal

When contexts are compressed, optional fields are relocated on compressed space and the positions of these fields may be overlapped with each other. Therefore, each optional field should be disabled when it is not being compressed in the context word. It means that compressed context should have control information for all of the optional fields in order to make unused fields disable. In this subsection, control signals generation for optional fields has been described.

Logical Operation	ALU_OP [5-bit]				
	A_4	A_3	A_2	A_1	A_0
A && B	1	1	1	0	0
A || B	1	1	1	0	1
A < B	1	1	1	1	0
A ≤ B	1	1	1	1	1
A!	0	1	1	1	1

(a) logical operations



(b) control signals

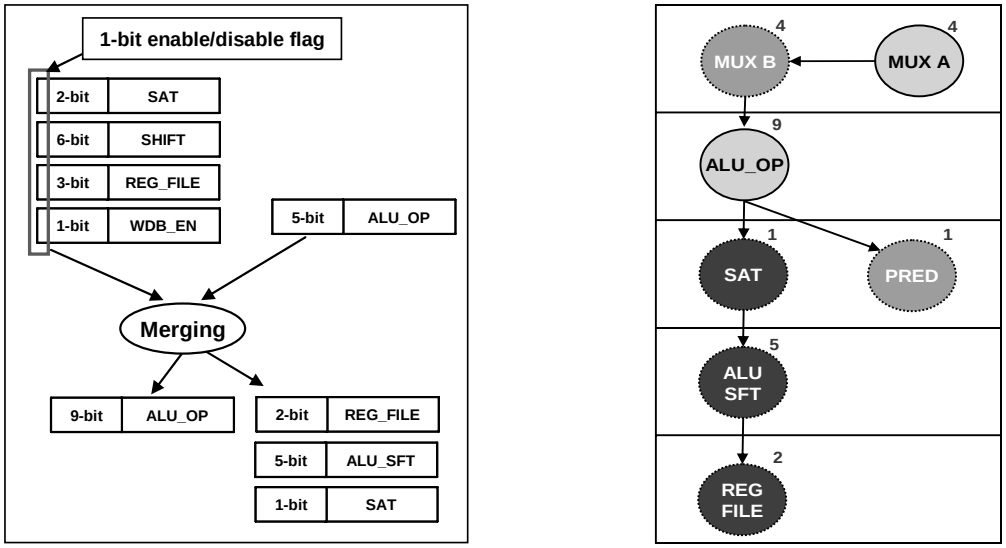
Fig. 30. Control signals for 'MUX_B' and 'PRED'.

a. Control Signals for ALU-Dependent Fields

If the truth table of 'ALU_OP' is classified by the operation type, enable/disable signals for ALU-dependent fields can be generated from 'ALU_OP' with some combinational logic. Fig. 30 (a) shows the truth table manipulated by classifying operations for the example given in subsection C.1. MSB (A4) of 'ALU_OP' is used for classifying operations according to the number of operands. For example, MSB =1 is used for the operations with two operands and MSB =0 is used for the operations with one operand. In addition, A_3~A_0 are used for classifying logical operations. Based on the truth table, we can generate control signals for two fields with some combinational logic as Fig. 30 (b). We define such a combinational logic as 'CTRL BLOCK'.

b. Control Signals for ALU-Independent Fields

In order to control ALU-independent fields when context words are compressed, the enable/disable flag bit on each of the ALU-independent field should be merged with a necessary field. Fig. 31 (a) shows the process that 1-bit flags of ALU-independent fields are merged with 'ALU_OP'. After flag merging, the FSG should be updated because the bit-widths of some of the fields are changed and 1-bit field such as 'WDB_EN' is no longer valid in FSG. Fig. 31 (b) shows an updated FSG with modified bit-widths of some of the fields.



(a) Flag merging (b) Updated FSG

Fig. 31. Updated FSG from flag merging.

5. Field Positioning

The final stage of proposed design flow is positioning each field on the context word. Field positioning should be considered for two cases (uncompressed and compressed) modes to support dynamic compression.

a. Field Positioning on Uncompressed Context Word

All the fields should have default positions for the case when contexts cannot be compressed. First of all, the necessary fields are positioned to the part near to MSB and the unnecessary fields are positioned near the LSB as shown in Fig. 32. Then the optional fields are positioned on the available space between the already occupied context word. For optional field positioning, the bit-width of compressed context word should be determined. Compressed bit width can be different according to the definition of the capacity of compressed context word. The large capacity of compressed context word can

show high compression ratio but the amount of power reduction is limited by long bit-width. However, the little capacity of compressed context word may cause low compression ratio but the power reduction ratio can be high in short bit-width. To prevent the extreme cases (much short or much long bit-width of compressed context word), we determine compressed bit-width based on following criterions.

i) Compressed context words should be able to support all of the ALU-dependent fields.

ii) Compressed context words should be able to include at least an ALU-independent field.

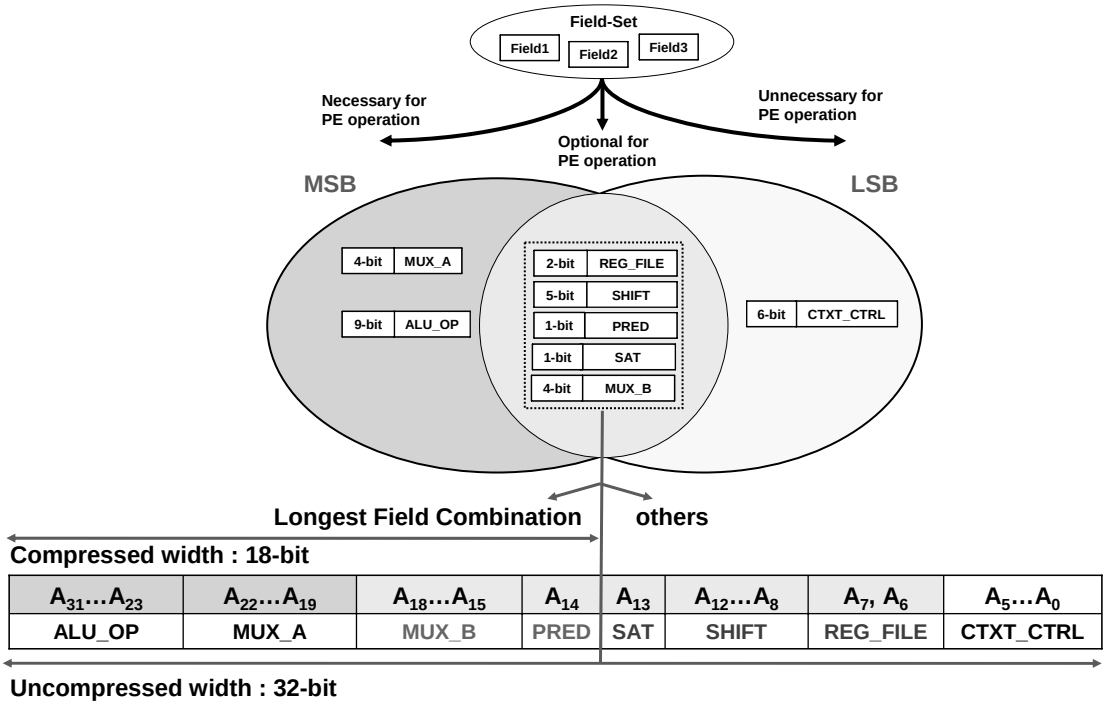
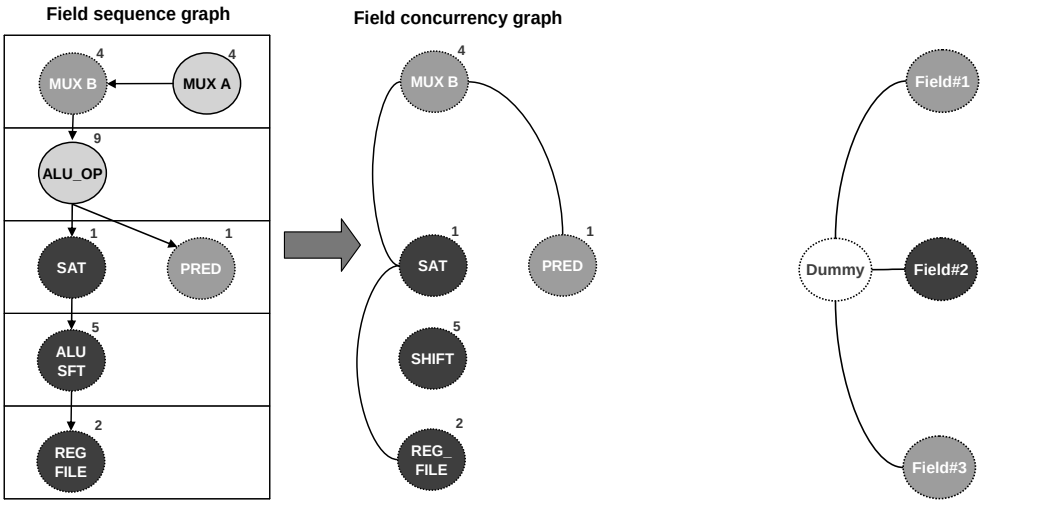


Fig. 32. Default field positioning.

To satisfy criterions, we determine the longest field combination showing the maxi-

mum bit-width among *i)* and *ii)*. The maximum width for satisfying i) and ii) is found to be 18-bit that consists of 'ALU_OP', 'MUX_A', 'MUX_B' and 'PRED'. Therefore, 18-bit is the compressed bit-width. Optional fields that are included in the longest field combination are preferentially positioned on the compressed zone near the MSB and other fields are positioned on uncompressed zone near the LSB as Fig. 32.

After this, the positions of the necessary fields on FSG are firmly determined and the positions of the field control signals are also determined because they are included in 'ALU_OP' as necessary field.



(a) FCG from FSG (b) FCG with dummy vertex

Fig. 33. Field concurrency graph.

b. Field Positioning on Compressed Context Word

This stage is for positioning fields on compressed context word to guarantee that all the possible field combinations are not exceeding the compressed bit-width. Therefore, first

of all, all the possible field combinations should be found. This process can be achieved by searching them from FSG and then generating field concurrency graph (FCG) such as Fig. 33 (a). The FCG shows the concurrency between the optional fields. Therefore the FCG is used for preventing position that is overlapping between the concurrent optional fields. An edge between two fields means that the two fields are included in one of the possible field combinations. Even though this example does not show concurrency among more than 2 optional fields, such a case can be represented by adding a dummy field connected with the fields as Fig. 33 (b).

Based on a given FCG, the next step is to position the optional fields on compressed context word. The positioning means that some optional fields have additional positions as well as default positions on uncompressed context words. To select a position among default and additional positions, multiplexers can be used that are composed of multiple position inputs and one feasible position output. Therefore, in this step, the field positioning is a mapping among inputs, outputs and control signals for multiplexers connected with the optional fields. Thus, we propose a port-mapping algorithm for the multiplexers. Before we explain the procedure in detail, we introduce notations we use in the explanation as Table V.

Table V. Notations for Port-Mapping Algorithm

Notation	Meaning
G_{FCG}	field concurrency graph, $G_{FCG} = (V, E)$: V is a set composed of the optional field set and E is a set composed of edges showing the concurrency between two fields.
G_{MUX}	multiplexer port mapping graph $G_{MUX} = (V_{MUX}, E_{MUX})$: V_{MUX} is a set composed of input signals and control signals for multiplexers and E_{MUX} is a set composed of weighted edges connecting input data with control signal.
defV	subset of *V*, *defV* is composed of the fields having their default positions on compressed context word
ndefV	subset of *V*, *ndefV* is composed of the fields not having their default positions on compressed context word
ctxt[A_i, A_j]	bit interval from index A_i to index A_j on the uncompressed context word, it is used for showing bit position of a field.
width[*v*]	bit-width of field *v*
cmp_lsb	LSB of compressed context word
def_pos[*field*]	default position of field such as interval type of *ctxt*[A_i, A_j]
ctrl_pos[*field*]	one bit position of control signal for *field* such as *ctxt*[A_i], *ctrl_blk*[A_i]
field[*i*, *j*]	component index corresponding to the interval that is from the i_{th} bit position to the j_{th} bit position on *field*
cmp_ctrl	one-bit signal from cache control unit. ‘1’ means executed context word compressed and ‘0’ means executed context word not compressed.
pdone[*field*]	‘1’ means positioning firmly done and ‘0’ means positioning not finished.
mux[*field*]	*mux* (multiplexer) connected with *field*.
data_in[*mux*]	set composed of *mux* input data signals
ctrl_in[*mux*]	set composed of field control signals for *mux*
data_out[*mux*]	set composed of *mux* ouput data signals
Adj[*field*]	adjacency list of *field* on graph G_{FCG}, if an adjacent field is dummy, it return adjacency list of the dummy field

Algorithm 1 Mux_**P**ort **M**apping (G_{FCG}) - fields having default position

L1 $V_{MUX} \leftarrow \emptyset$, $E_{MUX} \leftarrow \emptyset$, $G_{MUX} \leftarrow (V_{MUX}, E_{MUX})$

L2 Add *cmp_ctrl* on V_{MUX}

L3 **for** each $v \in defV$ **do**

L4 *data_in***[***mux*[*v*]] ← *data_in***[***mux*[*v*]] ∪ {*def_pos*[*v*]} ∪ {[*null*]}

L5 *ctrl_in*[*mux*[*v*]] ← *ctrl_in*[*mux*[*v*]] ∪ {*ctrl_pos*[*v*]}

L6 Add *data_in***[***mux*[*v*]] and *ctrl_in*[*mux*[*v*]] to V_{MUX}

L7 Add an edge between *def_pos*[*v*] and *ctrl_pos*[*v*] with weight '1' to E_{MUX}

L8 Add an edge between [*null*] and *ctrl_pos*[*v*] with weight '0' to E_{MUX}

L9 *data_out*[*mux*[*v*]] ← *v*[*width*[*v*]-1, 0]

L10 *pdone*[*v*] = 1

L11 **end do**

The input to the port-mapping algorithm is FCG and the output is multiplexer port-mapping graph (PMG) showing the relationship among field control signals and input data signals (field position). The algorithm is composed of two parts – The first part is for the optional fields having default position on compressed context word and the second part is for the optional fields not having default position on compressed context word. The procedure of the first part is described in Algorithm 1. The algorithm starts with initialization step (L1 and L2). In this part, input data signals of multiplexers are only two cases - default field position and 'zero' selected when the field is not used. This is because the fields already have default positions on compressed context space. Therefore the default field position, 'zero' and the field control signal of each field are mapped to the input of the multiplexer (L4~L6). Next process is to define the relationship between field control signal and a field position by adding a weighted edge between them (L7 and L8). Weight '1' (or '0') means the input signal is selected when the control sig-

nal is ‘1’ (or ‘0’). Finally, the outputs of multiplexers are connected with the component index defined in subsection C.1 (L9) and positioning of the field is firmly done (L10).

Algorithm 2 Mux_**P**ort **M**apping (G_{FCG}) - fields not having default position

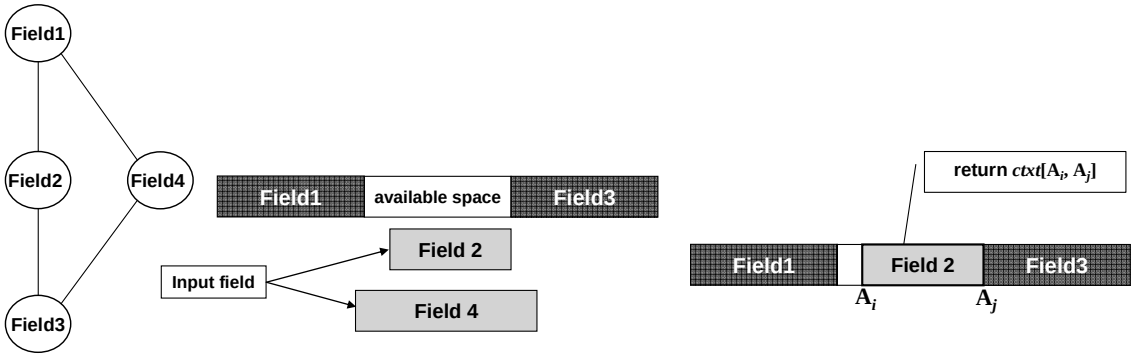
```
L1    for each v ∈ ndefV do
L2        data_in[mux[v]] ← data_in[mux[v]] ∪ {def_pos[v]}
L3        ctrl_in[mux[v]] ← ctrl_in[mux[v]] ∪{cmp_ctrl}
L4        Add def_pos[v] to V_MUX
L5        Add an edge between def_pos[v] and cmp_ctrl with weight '0' on E_MUX
L6        if Adj[v] = Ø on G_FCG then
L7            tmp_interval ← ctxt[(width[v]+cmp_lsb), cmp_lsb]
L8            data_in[mux[v]] ← data_in[mux[v]] ∪{tmp_interval}
L9            ctrl_in[mux[v]] ← ctrl_in [mux[v]] ∪{ctrl_pos[v]}
L10           Add data_in[mux[v]] and ctrl_in[mux[v]] to V_MUX
L11           Add an edge between tmp_interval and ctrl_pos[v] with weight '1' to E_MUX
L12           Add an edge between tmp_interval and cmp_ctrl with weight '1' to E_MUX
L13       else Check_Adjacency(v)
L14       end if
L15   data_out[mux[v]] ← v[width[v]-1, 0]
L16   pdone[u] ← 1
L17   end do
```

The procedure of the second part is described in Algorithm 2. The algorithm starts with mapping default field position and signal ‘*cmp_ctrl*’ to the input of the multiplexer for each field (L2 and L3). Signal ‘*cmp_ctrl*’ is one-bit signal from cache control unit and it gives information whether the context word is compressed (‘1’) or not (‘0’). Then the algorithm defines the relationship between signal ‘*cmp_ctrl*’ and a default position by adding a edge showing weight ‘0’ between them (L5). Next process is split into two cases – one is for the fields having no adjacent fields on FCG and another is for the fields having adjacent fields on FCG. The first case means the fields can be positioned to

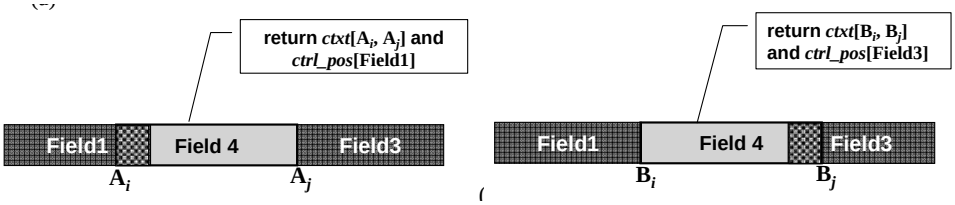
any part of compressed zone except the positions of necessary fields whereas the second case means the fields should be positioned to the part not overlapped with the positions of their adjacent fields. In the first case (L6), the field is positioned to the part near to LSB of compressed context word (L7). Then new field position and field control signal are mapped to the input of the multiplexer (L8 and L9). Next process is to define the relationship between field control signal (or '*cmp_ctrl*') and new field position by adding a edge showing weight '1' between them (L11 and L12).

Algorithm 3 Check_**A**djacency (*field*)

```
L1    for each u ∈ Adj[v] on G_FCG do
L2        if pdone[u] = 1 then
L3            tmpV ← tmpV ∪ {u}
L4        end if
L5    end do
L6    position_set and ctrl_set ← Find_Interval (v, tmpV)
L7    for each ctxt[A_i, A_j] ∈ position_set do
L8        if {ctxt[A_i, A_j]} ∩ data_in[mux[v]] = Ø then
L9            data_in[mux[v]] ← data_in[mux[v]] ∪ {ctxt[A_i, A_j] }
L10           ctrl_in[mux[v]] ← ctrl_in[mux[v]] ∪ {ctrl_pos[v]} ∪ ctrl_set
L10           Add data_in[mux[v]] and ctrl_in[mux[v]] to V_MUX
L11           Add an edge between ctxt[A_i, A_j] and ctrl_pos[v] with weight '1' to E_MUX
L12           Add an edge between ctxt[A_i, A_j] and cmp_ctrl with weight '1'to E_MUX
L13           for each ctrl ∈ ctrl_set do
L14               if ctrl overlapped with ctxt[A_i, A_j] then
L15                   Add an edge between ctxt[A_i, A_j] and ctrl with weight '0' to E_MUX
L16               end if
L17           end do
L18       end if
L18   end do
```



(a) FCG (b) Available space on compressed zone (c) When target field ('Field2') not overlapped with adjacent fields



(d) When target field ('Field4') overlapped with adjacent fields ('Field1'or 'Field3').

Fig. 34. Examples of 'Find_Interval'.

In the second case (L13), 'Check_Adjancency' function is used and it is described as algorithm 3. The algorithm start with gathering the adjacent fields firmly positioned. Then new position on compressed zone is assigned by 'Find_Interval' function (L6). Fig. 34 shows examples for this function with two cases - (c) when new position of input field is not overlapped with the adjacent field positions and (d) when new position of input field is overlapped with the adjacent field positions. 'Find_Interval' only returns a new position (*ctxt*[A_i, A_j]) in Fig. 34 (c) because of no confliction with the adjacent fields. However, it returns two positions (*ctxt*[A_i, A_j] and *ctxt*[B_i, B_j]) and field control signals from overlapped fields in Fig. 34 (d). This is because the adjacent field control

signals are necessary to select proper a field position when multiple field positions exist on compressed zone. Such returned new position set and control signal set are mapped to the input of multiplexer for the input field (L9 and L10) and the relationship among field control signals and a new position is made by adding weighted edges among them (L11~L17). Finally, the outputs of multiplexers are connected with the component index (L15) and positioning of the field is firmly done (L16) in Algorithm 2.

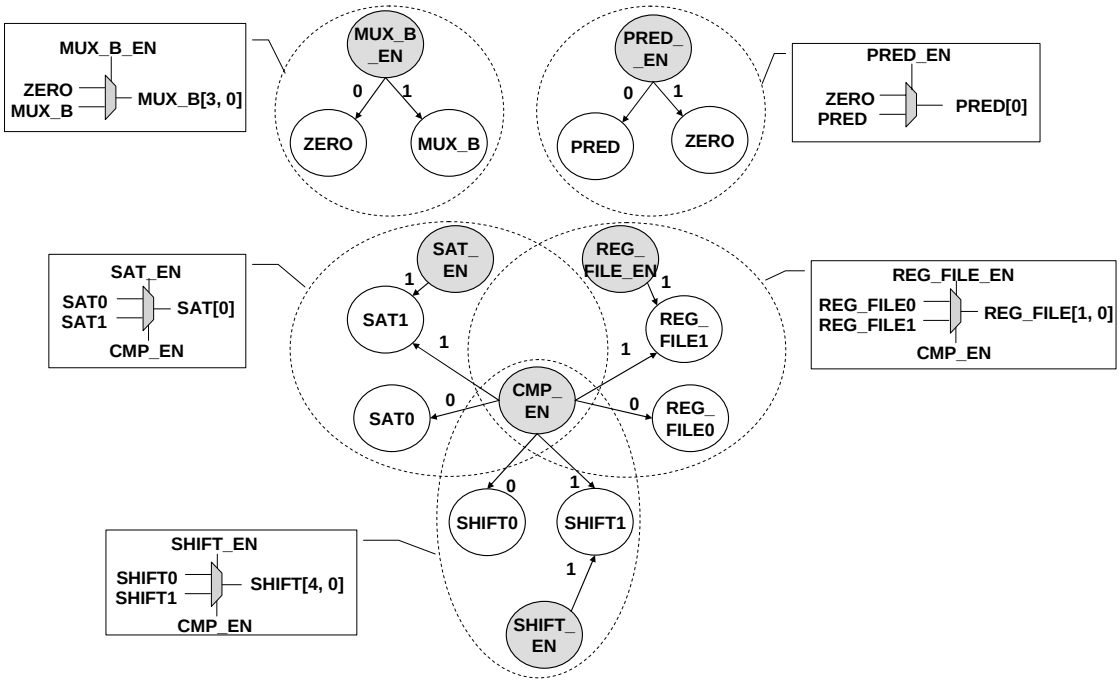
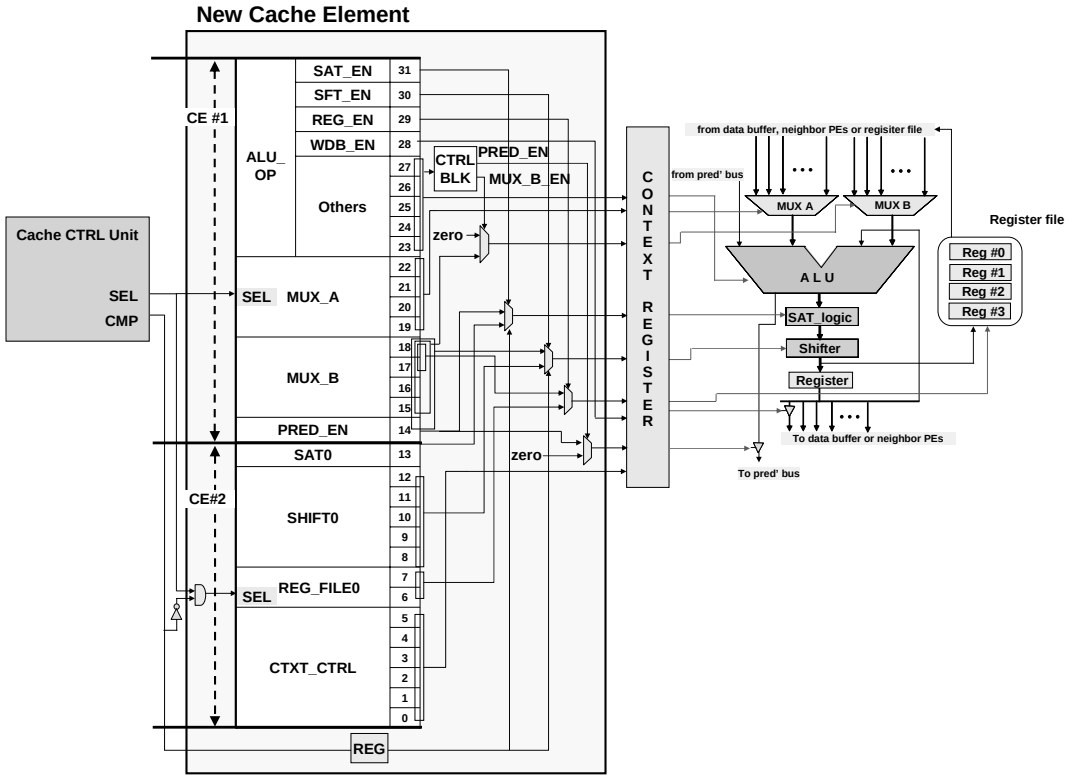


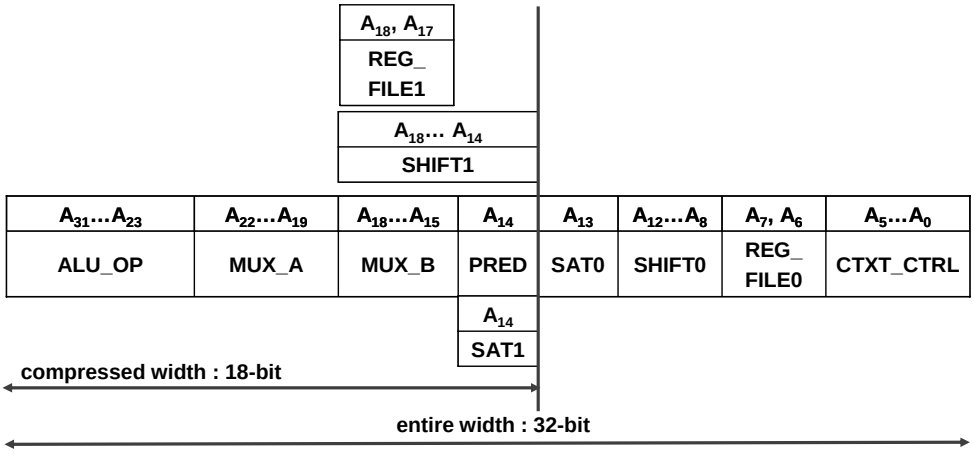
Fig. 35. Multiplexer port-mapping graph.

PMG example from the port-mapping algorithm is shown in the Fig. 35. Each vertex of PMG corresponds to an input or control signal of multiplexer and each edge shows the relationship between control signal and a position that is selected by the weight of the edge from control signals such as 'SAT_EN', 'MUX_B_EN', etc. Then the

outputs of



(a) Field layout of compressible context architecture



(b) Modified structure between a PE and a CE

Fig. 36. Compressible context architecture.

multiplexers are connected with the component index defined in Fig. 27 (b). Therefore we can implement the multiplexers for the optional fields by the PMG.

6. Compressible Context Architecture

After the field positioning, we have generated a specification of dynamically compressible context architecture like one in the Fig. 36. Fig. 36 (a) shows the final field layout of compressible context architecture. 'REG_FILE', 'SHIFT' and 'SAT' have double positions for compressed and uncompressed cases. Fig. 36 (b) shows a modified structure between a PE and a cache element (CE). New cache element is composed of CE1 and CE2 and cache control unit provides compression information from port 'CMP' whether executed contexts are compressed or not. CE1 is always selected but CE2 is not selected under compression ('CMP'=1) to remove power consumption in CE2.

7. Context Evaluation

The context evaluator in Fig. 26 determines whether initially uncompressed contexts can be compressed or not. This evaluation process can be implemented by checking the fact that a given context word is compared with one of the possible field combinations not exceeding compressed bit-width. Using FCG, we can easily check this and generate compressed context words with using position information from PMG.

D. Experiments

1. Experimental Setup

We have implemented entire design flow in Fig. 26 with C++. We have initialized con-

text architecture as the example described in Section C. The implemented design flow generated the specification of dynamically compressible context architecture. For quantitative evaluation, we have designed two CGRAs based on the 8x8 reconfigurable array at RT-level with VHDL - one is conventional base CGRA and the other is the proposed CGRA supporting compressible features in context architecture. The architectures have been synthesized using Design Compiler [49] with 0.18 ㎛ technology. We have used SRAM Macro Cell library for the frame buffer and configuration cache. ModelSim [50] and PrimePower [49] tools have been used for gate- level simulation and power estimation. To obtain the power consumption data, we have used the kernels (Fig. 25) for simulation with operation frequency of 100 MHz and typical case of 1.8 V Vdd and 27℃. These kernels have been executed with 100 iterations while varying test vectors.

2. Results

a. Area Cost Evaluation

Table VI shows the synthesis results from Design Compiler [49] of proposed architecture and base architecture. It shows that area cost of new configuration cache including cache control unit, added interconnects and multiplexers has increased by 10.35% but the overall area-overhead is only 1.62 %. Thus, the new configuration cache structure can support dynamic context compression with negligible overheads.

Table VI. Area Overhead by Dynamic Context Compression

Component	Area Cost (gate equivalent)		Overhead (%)
	Base Architecture	Proposed Architecture	
Configuration Cache	150012	165538	10.35
Entire RAA	942742	958268	1.62

Overhead (%): **{**(Proposed/Base) – 1}×100

Table VII. Power Reduction Ratio by Dynamic Context Compression

Kernels	Compression Ratio (%)	Configuration Cache Power(mW)		Reduced (%)
		Base Architecture	Proposed Architecture	
First_Diff	100	171.77	104.97	38.89
Tri- Diagonal	100	174.18	105.00	39.72
State	100	161.23	99.38	38.36
Hydro	100	148.23	91.50	38.27
ICCG	100	205.80	125.68	38.93
Inner Product	100	117.84	72.60	38.39
24-Taps FIR	100	227.56	139.56	38.67
MVM	100	227.57	140.43	38.29
Mult in FFT	100	175.48	107.08	38.98
Comlex Mult	100	180.63	110.18	39.00
ITRANS	100	204.85	125.27	38.85
2D-FDCT	95.53	190.03	119.87	36.92
2D-IDCT	95.49	188.47	118.98	36.87
SAD	100	185.30	113.07	38.98
Quant	95.12	185.23	117.51	36.56
Dequant	95.23	187.78	118.77	36.75

Compression Ratio (%): number of compressed context words/ number of entire context words)×100, Reduced (%): {1-(Proposed/Base)}×100, Execution Cycle Count : cycle count for an iteration.

b. Performance Evaluation

In addition, the synthesis results show that the critical path delay of the proposed architecture is same as the base model i.e. 8.96 ns. It indicates the dynamic context compression does not cause performance degradation in terms of the critical path delay. In addition, we have applied several kernels in Fig. 25 to the new and base architectures. The execution cycle count of each kernel on proposed architecture does not vary from the

base architecture because the functionality of proposed architecture is same as the base model. It also indicates the dynamic context compression does not cause performance degradation in terms of the execution cycle count.

c. Context Compression Ratio and Power Evaluation

Table VII shows context compression ratio for the evaluated kernels. Compression ratio means how many context words can be compressed among entire context words. The execution cycle count of each kernel on proposed architecture does not vary from the base architecture because the functionality of proposed architecture is same as the base model. It also indicates the dynamic context compression does not cause performance degradation in terms of the execution cycle count. All of the kernels show high compression ratio to be more than 95 %. Furthermore, the comparison of power consumption is shown in Table VII. Compared to the base architecture, it has shown to save up to 39.72% of the power. 4 kernels (2D-FDCT, 2D-IDCT, Quant and Dequant) show less reduction in power compared to other kernels. This is because all of the context words for 4 kernels are not fully compressed - the compression ratios are in the range of 95.12 ~ 95.53.

CHAPTER VI

DYNAMIC CONTEXT MANAGEMENT FOR LOW POWER CGRA

In this chapter, we present a novel control mechanism of configuration cache called dynamic context management to reduce the power consumption in configuration cache without performance degradation [61]. In addition, a new configuration cache structure is proposed to support such a dynamic context management. Experimental results show that the proposed approach saves 38.24%/38.15% of the power in write/read-operation of configuration cache with negligible area overhead compared to the base design.

A. Motivation

1. Power Consumption by Configuration Cache

By loading the context words from the configuration cache into the array, we can dynamically change the configuration of the entire array within just one cycle. However, such dynamic reconfiguration of CGRA causes many SRAM-read operations in configuration cache. In [6], the authors have fabricated a CGRA (PipeRench) in a 0.18 ㎛ process. Their experimental results show that the power consumption is significant high due to the dynamic reconfiguration requiring frequent configuration memory access. In Fig. 9, power break-down for the CGRA running 2D-FDCT is proposed with gate-level implementation at 0.18 ㎛ technology based on MorphoSys architecture. It is shown that the configuration cache spends about 43% of the overall power, which is the second largest after the PE arrays consuming 48% of overall power budget. This is because the

configuration cache performs SRAM-read operations to load the context words in every cycle at run time. In addition, [8][30] also shows power break-down for another CGRA (ADRES) running IDCT based on 90nm technology. In this case, the configuration memory spends about 37.22% of the overall power. Therefore, it is explicit that power consumption by configuration cache (memory) is serious overhead compared to other types of IP cores such as ASIC or ASIP.

2. Redundancy of Context Words

Context words are saved in configuration cache and they show redundancies at runtime. We describe two cases for redundancy of context words in following subsections.

a. NOP Context Words

Most coarse-grained reconfigurable arrays arrange their processing elements (PEs) as a square or rectangular 2-D array with horizontal and vertical connections, which support rich communication resources for efficient parallelism. However, such PE arrays have many redundant or unutilized PEs during the executions of applications onto the array.

Most of subtasks in DSP applications shows lots of redundant PEs that are not used. The redundant PEs should be configured by NOP (no operation) context words to avoid malfunction and unnecessary waste of power by the PEs. It means that configuration cache performs some redundant read-operations for NOP.

Cycle time	Context Word		
	ALU Operation	Operands	Other Operations
1	ALU_OUT <= A	A<= Data bus	R0 <= ALU_OUT
2			R1 <= ALU_OUT
3			R2 <= ALU_OUT
4			R3 <= ALU_OUT

R0~R3: registers of register file

(a) Consecutive load operations

Cycle time	Context Word		
	ALU Operation	Operands	Other Operations
1	ALU_OUT <= AXB	A <= T, B<= L	SHIFT(ALU_OUT)
2	ALU_OUT <= A+B	A <=BT, B <= R	
3	ALU_OUT <= A-B	A <= R1, B<= R2	
4	ALU_OUT <= A+B	A <= T	

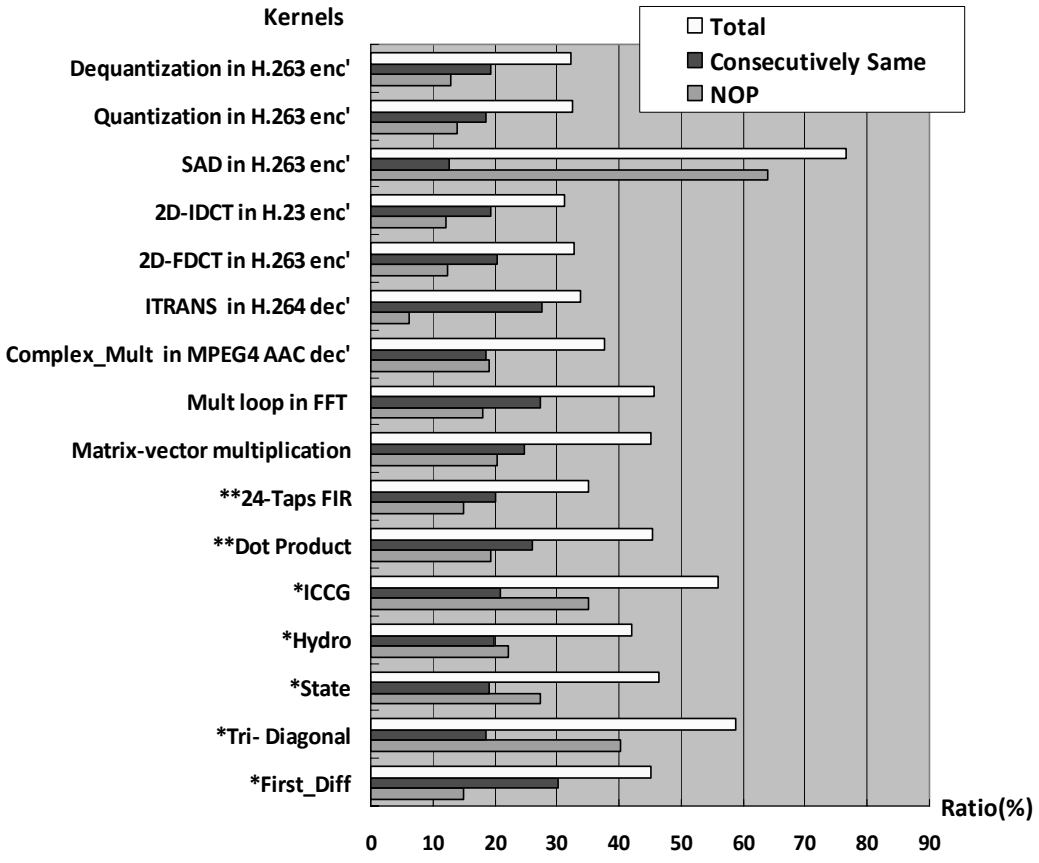
T, L, R and BT: output from Top PE, Left PE, Right PE and Bottom PE

(b) Consecutive shift operations

Cycle time	Context Word		
	ALU Operation	Operands	Other Operations
1	ALU_OUT <= A	A <= R0	Data bus <= ALU_OUT
2		A <= R1	
3		A <= R2	
4		A <= R3	

(c) Consecutive store operations

Fig. 37. Consecutively same part in context words.



*Livermore loops benchmark [58], **DSPstone [59]

Consecutively Same (%) = 100 × (consecutively same part [bits]/total context words [bits]), NOP (%) = 100 × (NOP context words [bits] / total context words [bits]), Total (%) = NOP + Consecutively Same

Fig. 38. Redundancy ratio of context words.

b. Consecutively Same Part in Context Words

When a kernel is mapped onto CGRA and application gets executed, the consecutively changed context fields are limited to types of operations involved due to the kernel executed at run time. Fig. 37 shows 3 cases for consecutively-same part in context words at run time. In the case of Fig. 37 (a), PEs perform continuous ‘Load’ operations with fixed

‘ALU Operation’ and ‘Operands’ whereas operand data are saved in different register in every cycle. The Fig. 37 (b) and (c) shows consecutive shift operations and store operations with different ‘Operand’ while keeping same ‘Other Operations’ in every cycle. It means that the context words shows consecutively same part and they are repetitively read from configuration cache without changing values.

c. Redundancy Ratio

For statistical evaluation of redundant context words, we selected 32-bit context architecture of the base architecture (Fig. 4) and mapped several kernels onto its PE array in order to maximize the utilization of the context fields. Fig. 38 shows the results for various benchmark kernels and critical loops in real applications. Each kernel shows three cases of redundancy ratios – ‘NOP’, ‘Consecutively Same’ and Total. Total redundancy ratio varies from 31% to 75%.

B. Dynamic Context Management

If the configuration cache does not perform read/write operation for redundant part of context words, it is possible to reduce power consumption in configuration cache. That way, one can achieve low-power implementation of CGRA without performance degradation while managing context words in both cases at transfer time and runtime: one case is no read/write operation for NOP and another case is one read/write-operation for consecutively same part in context words. In order to support such a dynamic context management, we propose a new configuration cache structure and efficient control mechanism in this chapter.

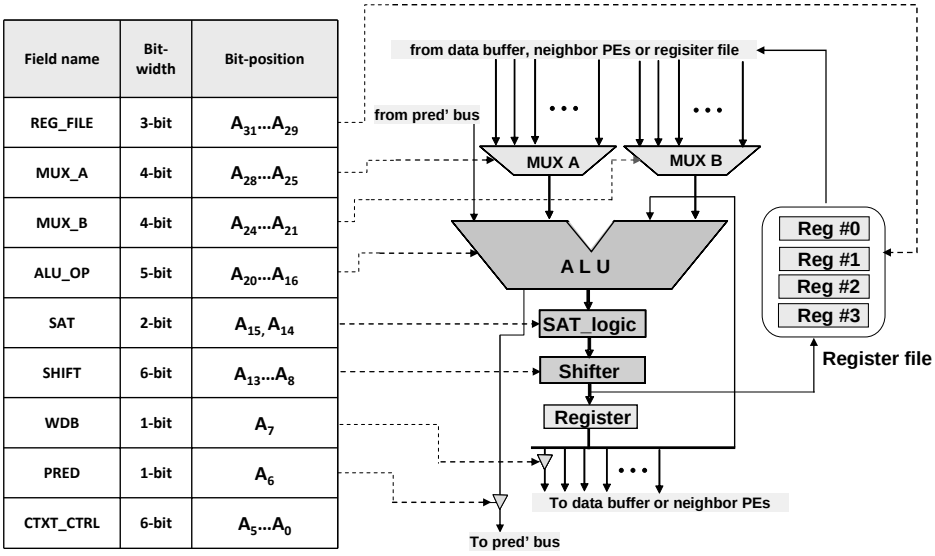


Fig. 39. An example of PE and context architecture.

1. Context Partitioning

Context partitioning is to split context architecture into two parts feasible to dynamic context management. As mentioned in subsection A.2.b, the context words shows consecutively same part and they are repetitively read from configuration cache without changing values. Therefore, if a CE is divided into two parts (CE#1 and CE#2) by context partitioning, one part of CE including continuously same part can be disabled for power saving while keeping consecutive read/write-operation of another part of CE. The partitioning starts from grouping context field for ALU operation and some context fields dependent to ALU operation. This is because ALU have the most dependency with other component and they are highly probable to be consecutively changed or unchanged together. Therefore context partitioning positions such fields on one part of context architecture and other fields on another part of context architecture. We have de-

fined generic PE structure and 32-bit context architecture like Fig. 39 as an example to illustrate context partitioning. It can support the kernels in Fig. 38. It is similar to the representative CGRAs such as MorphoSys [3], REMARC [4], ADRES [8][30] or-PACT_XPP [10]. Bit-width and initial bit-position of each field are shown in Fig. 39. It supports various arithmetic and logical operations (ALU_OP) with two operands (MUX_A and MUX_B), predicated execution (PRED), Arithmetic saturation (SAT_logic), shift operation (SHIFT) and saving temporal data with register file (REG_FILE). Fig. 40 shows context partitioning of Fig. 39. Field ‘ALU_OP’ and the fields dependent to ‘ALU_OP’ are positioned to the part near to MSB and other fields are positioned near to LSB.

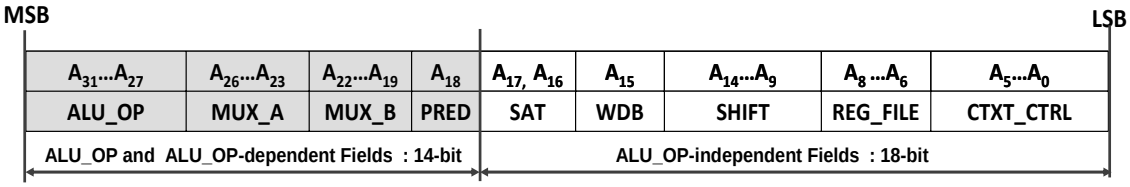
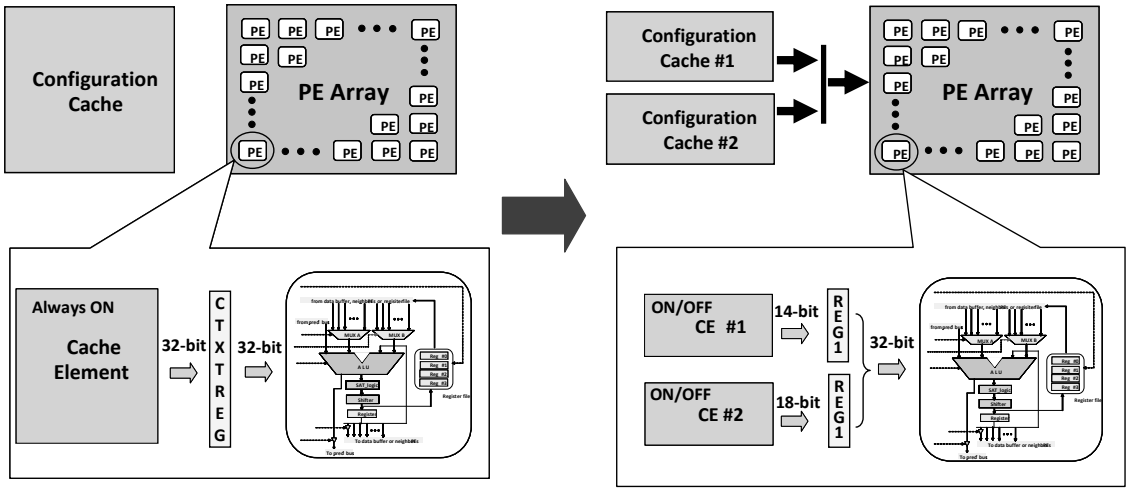


Fig. 40. Context partitioning.

After context partitioning, we can know the bit-widths of CE #1 and CE#2 and context register is also can be split into two parts with same bit-widths. Fig. 41 shows comparison between general CE and proposed CE. The proposed CE is composed of CE#1 (14-bit) and CE#2 (18-bit) whereas the general CE is a unified one (32-bit). In subsection B.2 and B.3, we describe more detailed control mechanism for dynamic context management based on the proposed CE structure.



(a) General CE (b) Proposed CE

Fig. 41. Comparison between general CE and proposed CE.

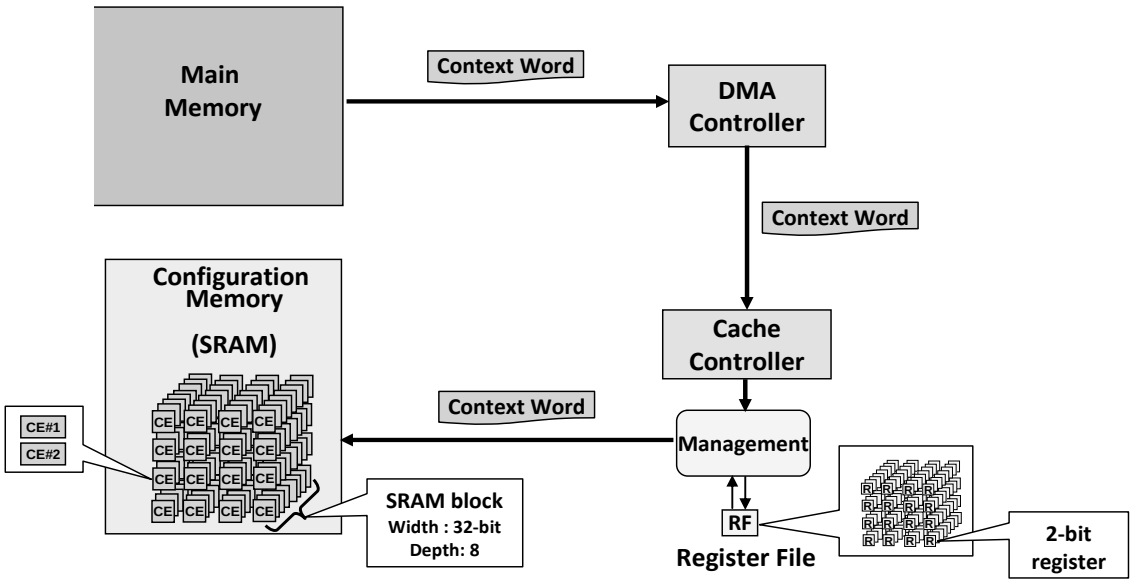


Fig. 42. Context management when context words are transferred.

2. Context Management at Transfer Time

Context management at transfer time is to remove redundant cache-write operations by using additional hardware detecting redundancy of context words. Fig. 42 shows transfer flow of context words from main memory to configuration cache in the case of 4x4 CEs. For checking the redundancy, hardware block of 'Management' is added to general cache controller. 'Management' block checks transferred context words whether it has redundancy or not. Then it controls cache-write operation as Algorithm 4. In addition, Fig. 42 shows register file connected with 'Management' block – it has same address-ability as CE but bit-width is 2. The register file store 2-bit redundancy information – the saved information in register file are used for context management at run time.

Algorithm 4 **C**ontext **M**anagement at **T**ransfer **T**ime

```
L1   begin
L2     if cur_ctxt = NOP then
L3       reg_file[ctxt_addr] ← "01"
L4       cs1 ← '0', cs2 ← '0'
L5     else if cur_ctxt[cw-1, cw-w+1] = prev_ctxt[cw-1, cw-w+1] then
L6       reg_file[ctxt_addr] ← "10"
L7       cs1 ← '0', cs2 ← '1'
L8       CE#2[ctxt_addr] ← cur_ctxt[cw-w, 0]
L9     else if cur_ctxt[cw-w, 0] = prev_ctxt[cw-w, 0] then
L10      reg_file[ctxt_addr] ← "11"
L11      cs1 ← '1', cs2 ← '0'
L12      CE#1[ctxt_addr] ← cur_ctxt[cw-1, cw-w+1]
L13    else
L14      cs1 ← '1', cs2 ← '1'
L15      CE#1[ctxt_addr] ← cur_ctxt[cw-1, cw-w+1]
L16      CE#2[ctxt_addr] ← cur_ctxt[cw-w, 0]
L17    end if
L18    prev_ctxt ← cur_ctxt
L19  end
```

Algorithm 4 shows this management process for a CE. Before we explain this management in detail, we introduce notations we use in Algorithm 4.

- *cw*: bit-width of context word
- *w*: bit-width of field group (ALU_OP and ALU_OP-dependent fields)
- *cur_ctxt*: context word currently transferred to configuration cache
- *prev_ctxt*: context word previously transferred to configuration cache
- *ctxt_addr*: address of current context word in configuration cache
- *reg_file*: register file, *CE#*1 and *CE#*2: Cache Element
- *out_ctxt*: context word currently provided to context register
- *cs*1 *and cs*2: chip select signal of *CE*1 and *CE*2

The algorithm starts with checking whether current context word is NOP or not (L2). If the context word is NOP, 2-bit information (“01”) is stored in register file and both CE#1 and CE#2 are disabled (L4). If it’s not NOP, next process is to check whether the upper part (near to MSB) of context word is the consecutively identical to one of previous context word. If it is the same part as the previous one, information (“10”) is stored in the register file (L6) and only CE#2 is enabled (L7) for cache write-operation (L8). Checking the lower part (near to LSB) of current context word (L9~L12) shows the same manner as previous process but CE#1 is enabled instead of CE#2. Finally, if current context word does not correspond to any case of previous checking processes, both CE#1 and CE#2 are enabled (L14) and full context word is stored in configuration cache (L15, L16). Finally, previous context word is updated by current context word (L18).

3. Context Management at Run Time

Context management at run time is to remove redundant cache-read operations by checking redundancy information stored in the register file. Fig. 43 shows structure between configuration cache and PE array for the context management. The hardware block of ‘Management’ controls all of CEs and a context register between a CE and a PE is implemented by a gated clock using chip select signals (CS1 and CS2). Gated clock implementation is to configure PE with fixed output of the context register caused by non-oscillated clock. Therefore, PEs can be configured without cache-read operation in the case of consecutively same context words.

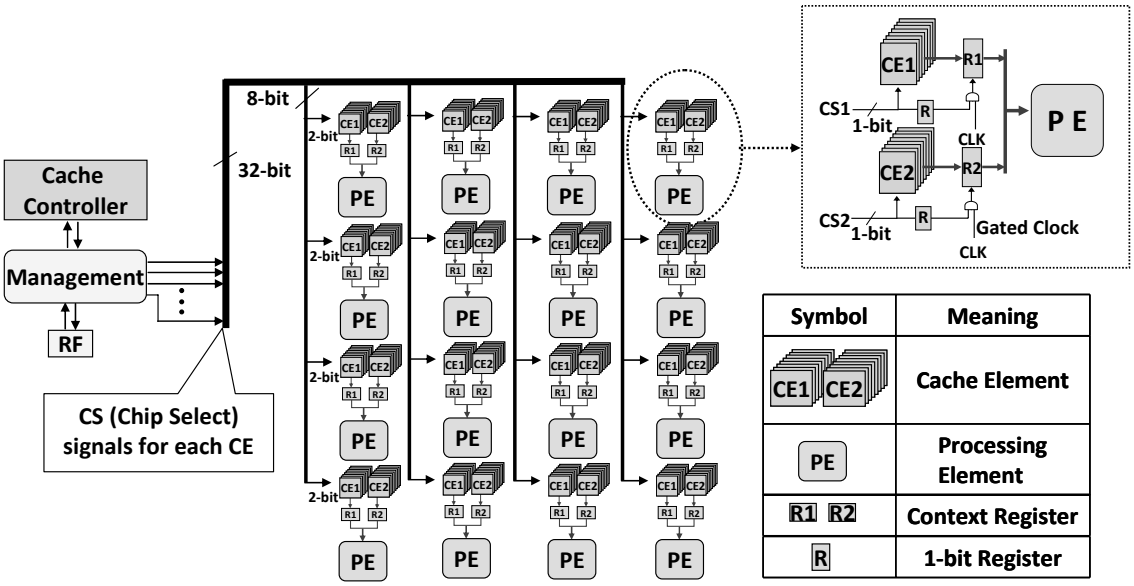


Fig. 43. Context management at run time.

Algorithm 5 **C**ontext **M**anagement at **R**un **T**ime

```
L1   begin
L2     if reg_file[ctxt_addr] = "01" then
L3       cs1 ← '0', cs2 ← '0'
L4     else if reg_file[ctxt_addr] = "10" then
L5       cs1 ← '0', cs2 ← '1'
L6       out_ctxt[cw-w, 0] ← CE#2[ctxt_addr]
L7     else if reg_file[ctxt_addr] = "11" then
L8       cs1 ← '1', cs2 ← '0'
L9       out_ctxt[cw-1, cw-w+1] ← CE#1[ctxt_addr]
L10    else
L11      cs1 ← '1', cs2 ← '1'
L12      out_ctxt[cw-1, cw-w+1] ← CE#1[ctxt_addr]
L13      out_ctxt[cw-w, 0] ← CE#2[ctxt_addr]
L14    end if
L15  end
```

Algorithm 5 shows this management process for a CE. The defined notations in Algorithm 4 are used in Algorithm 5. The algorithm starts with checking whether the information (stored in the register file) identified by current address is NOP or not (L2). If the information is NOP ("01"), both CE#1 and CE#2 are disabled (L3). If it's not NOP, next process is to check whether the information corresponds to the case ("10") of consecutively same part (near to MSB) or not (L4). If it is "10", only CE#2 is enabled (L5) for cache read-operation (L6). Next process is to check whether the information corresponds to the case ("10") of consecutively same part (near to MSB) or not (L7). It shows the same manner as previous process but CE#1 is enabled for read-operation instead of CE#2. Finally, if the information does not correspond to any case of previous checking processes, both CE#1 and CE#2 are enabled (L11) and a full context word is read from configuration cache (L12, L13).

C. Experiments

1. Experimental Setup

For quantitative evaluation, we have designed two CGRAs based on the 8x5 reconfigurable array at RT-level with VHDL – one is conventional base CGRA and the other is the proposed CGRA supporting dynamic context management. The architectures have been synthesized using Design Compiler [49] with 0.18 ㎛ technology. We have used SRAM Macro Cell library for the frame buffer and configuration cache. ModelSim [50] and PrimePower [49] tools have been used for gate- level simulation and power estimation. To obtain the power consumption data, we have used the kernels (Fig. 38) for simulation with operation frequency of 100 MHz and typical case of 1.8 V Vdd and 27℃.

Table VIII. Area Overhead by Dynamic Context Management

Component	Area cost (gate equivalent)		Overhead (%)
	Base	Proposed	
Config'cache	150012	162538	8.35
RAA	942742	955268	1.33

Base: base architecture, Proposed: proposed architecture,
Overhead(%) : {(Proposed/Base) – 1}×100

2. Results

a. Area Cost Evaluation

Table VIII shows the synthesis results from Design Compiler [49] of proposed architecture and base architecture. It shows that area cost of new configuration cache including cache control unit, hardware block of "Management" and register file increased by 8.35% but the overall area-overhead is only 1.33 %. Thus, the new configuration cache structure can support dynamic context management with negligible overheads.

Table IX. Power Reduction Ratio by Dynamic Context Management

Kernels	Configuration cache Power (mW)				Reduction Ratio (%)	
	Write-operation		Read-operation			
	Base	Proposed	Base	Proposed	Write	Read
Tri- Diagonal	14.98	6.89	171.77	79.03	54.00	53.99
First_Diff	13.34	8.25	174.18	104.51	38.12	40.00
State	15.23	9.37	161.23	93.87	38.45	41.78
Hydro	11.22	7.17	148.23	96.14	36.14	35.14
ICCG	15.39	7.56	205.80	103.35	50.87	49.78
Dot Product	12.11	7.28	117.84	72.51	39.88	38.47
24-Taps FIR	19.20	11.63	227.56	138.90	39.41	38.96
MVM	14.23	8.68	227.57	138.54	38.99	39.12
Mult in FFT	12.12	7.62	175.48	105.88	37.14	39.66
Comlex Mult	11.57	7.86	180.63	123.59	32.12	31.58
ITRANS	14.22	10.17	204.85	148.64	28.47	27.44
2D-FDCT	16.23	11.69	190.03	140.30	27.96	26.17
2D-IDCT	17.34	13.16	188.47	139.88	24.13	25.78
SAD	14.30	4.45	185.30	55.87	68.89	69.85
Quant	12.12	8.73	185.23	134.94	27.99	27.15
Dequant	15.33	11.05	187.78	137.10	27.89	26.99
Average					38.24	38.15

Base: base architecture, Proposed: proposed architecture, Reduced: {1-(Proposed/Base)}×100
Write/Read: reduction ratio in the case of write/read operation

b. Power Evaluation

To demonstrate the effectiveness of the proposed approach, we have applied several kernels in Fig. 38 to the proposed and base architectures. These kernels were executed with 100 iterations. Table IX shows power evaluation of configuration cache for two cases – read operation and write-operation. The power consumptions of write-operations are less than the cases of read-operations. This is because a CE performs write-operation at transfer time whereas all of CEs perform read-operation at run time. Compared to the base architecture, it has shown to save up to 68.89%/69.85% of the power in write/read-operation. 5 kernels (ITRANS, 2D-FDCT, 2D-IDCT, Quant and Dequant) show less reduction in power compared to other kernels. This is because they show less redundancy

ratios of context words compared with other kernels– Fig. 38 shows that the redundancy ratios of these kernels are in the range of 31.22% ~ 33.79%. Average power reduction ratios in write-operation and read-operation are 38.24% and 38.15%.

c. Performance Evaluation

The synthesis results show that the critical path delay of the proposed architecture is same as the base model i.e. 8.96 ns. It indicates the dynamic context management does not cause performance degradation in terms of the critical path delay. In addition, the execution cycle count of each kernel on proposed architecture does not vary from the base architecture because the functionality of proposed architecture is same as the base model. It also indicates the dynamic context management does not cause performance degradation in terms of the execution cycle count.

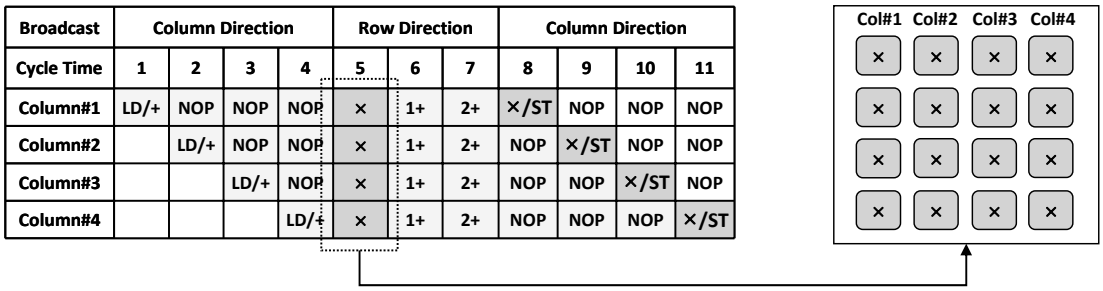
CHAPTER VII

COST-EFFECTIVE ARRAY FABRIC

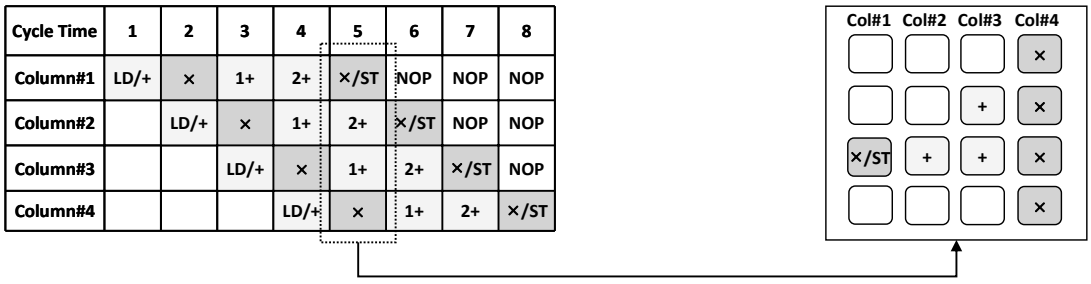
In this chapter, we propose a new domain-specific array fabric design space exploration method to generate a cost-effective reconfigurable array structure [62]. The exploration flow efficiently rearranges PEs with reducing array size and change interconnection scheme to achieve much reduction in power and area while maintaining the same performance as the original architecture. In addition, the proposed array fabric splits the computational resources into two groups (primitive resources and critical resources). Critical resources can be area-critical and/or delay-critical. Primitive resources are replicated for each processing element of the reconfigurable array, whereas area-critical resources are shared among multiple basic PEs in order to reduce more area of CGRA. Delay-critical resources can be pipelined to curtail the overall critical path so as to increase the system clock frequency. Experimental results show that for multimedia applications, the proposed approach reduces area by up to 36.75%, execution time by up to 42.86 and power by up to 35.45.% when compared with the base CGRA architecture.

A. Preliminary

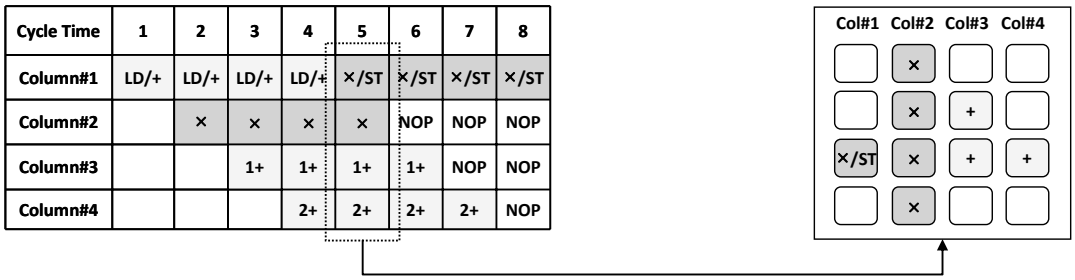
In this section, we present preliminary concepts of our cost-effective design [44]. They come from the characteristics of loop pipelining based on MIMD-style execution model. Then we propose two techniques to make an RAA cost-effective in terms of area and delay. One is resource sharing and the other is resource pipelining.



(a) SIMD



(b) Temporal mapping



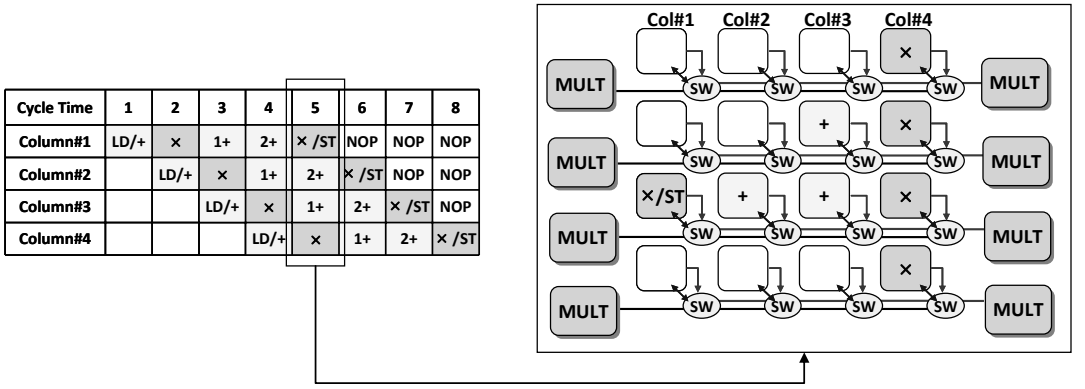
(c) Spatial mapping

Fig. 44. Snapshots of three mappings.

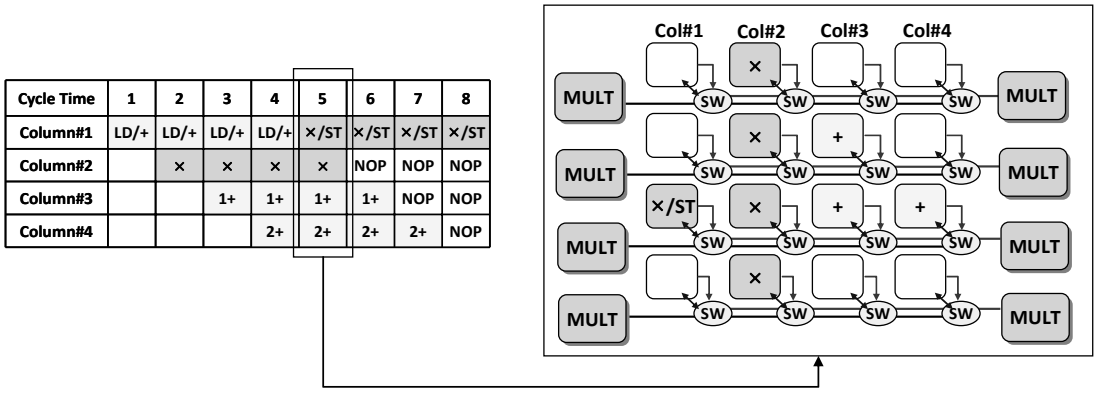
1. Resource Sharing

Fig. 44 shows the snapshot taken at the 5th cycle of execution of the previous example shown in Fig. 12 for three cases: (a) SIMD and two cases of loop pipelining - (b) temporal mapping and (c) spatial mapping. The operations in the 5th cycle for (a), (b) and (c) include multiplication and therefore the multipliers in the PE array are to be used. In the

case of SIMD, all PEs perform multiplication requiring all of them to have multipliers, thereby increasing the area cost of the PE array. However, in the case of temporal mapping, only PEs in the 1st column and the 4th column perform multiplication while PEs in the 2nd and 3rd columns perform addition. In the spatial mapping, only PEs in the 1st and



(a) Temporal mapping



(b) Spatial mapping

Fig. 45. Eight multipliers shared by sixteen PEs.

2nd columns perform multiplication. As can be observed, in the temporal mapping and spatial mapping, there is no need for all PEs to have the same functional resources at the

same time. This allows the PEs in the same column or in the same row to share area-critical resources. Fig. 45 shows four PEs in a row sharing two multipliers[3] at the 5th cycle in temporal mapping and spatial mapping. We depict only the connections related to resource sharing.

Fig. 46 depicts the detailed connections for multiplier sharing. The two n-bit operands of a PE are connected to the bus switch. The dynamic mapping of a multiplier to a PE is determined at compile time and the information is encoded into the configuration word. At run-time, the mapping control signal from the configuration word is fed to the bus switch and the bus switch decides where to route the operands. After the multiplication, the 2n-bit output is transferred from the multiplier to the original issuing PE via the bus switch.

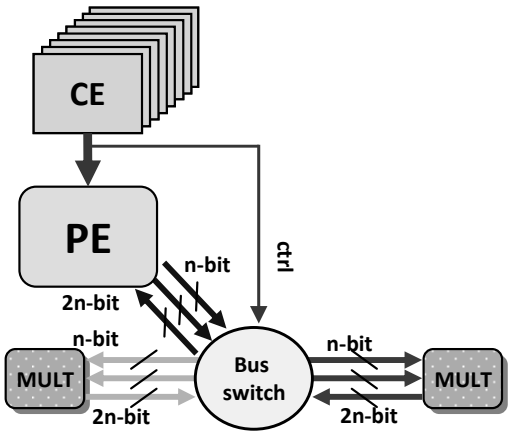
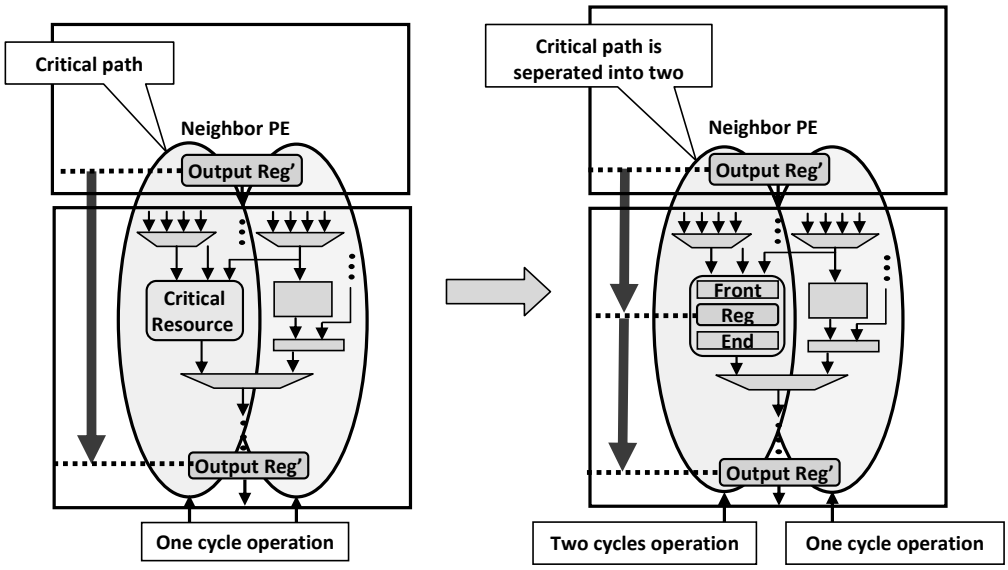


Fig. 46. The connection between a PE and shared multipliers.

[3] Since multipliers take much more area than other resources, we classify them as critical resources.

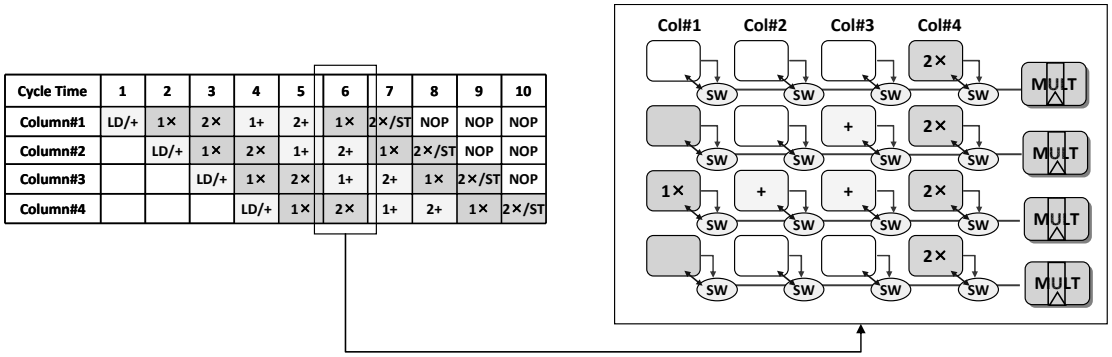
2. Resource Pipelining

If there is a critical functional resource with long latency in a PE, the functional resource can be pipelined to curtail the critical path. Resource pipelining has clear advantage in loop pipelining execution because heterogeneous functional units with different delays can run at the same time. In the traditional design (Fig. 47 (a)), the latency of a PE is fixed but in our pipelined PE design (Fig. 47 (b)), we allow multi-cycle operations and so the latency can vary depending on the operation. This helps increase the system clock frequency.



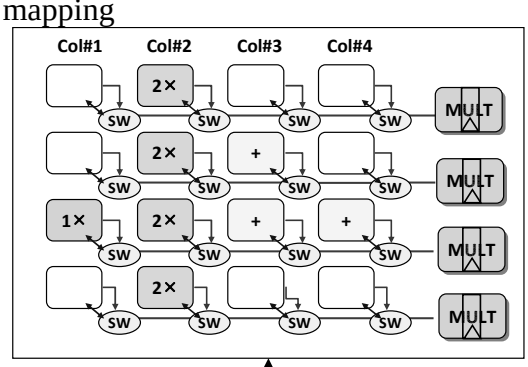
(a) General PE (b) Pipelined PE

Fig. 47. Critical paths.



Cycle Time	1	2	3	4	5	6	7	8	9	10
Column#1	LD/+	1×	2×	1+	2+	1×	2×/ST	NOP	NOP	NOP
Column#2		LD/+	1×	2×	1+	2+	1×	2×/ST	NOP	NOP
Column#3			LD/+	1×	2×	1+	2+	1×	2×/ST	NOP
Column#4				LD/+	1×	2×	1+	2+	1×	2×/ST

(a) Temporal mapping



Cycle Time	1	2	3	4	5	6	7	8	9	10
Column#1	LD/+	LD/+	LD/+	LD/+	NOP	1×	2×/ST /1×	2×/ST /1×	2×/ST /1×	2×/ST
Column#2		1×	2×/1×	2×/1×	2×/1×	2×	NOP	NOP	NOP	NOP
Column#3				1+	1+	1+	1+	NOP	NOP	NOP
Column#4					2+	2+	2+	2+	NOP	NOP

(b) Spatial mapping

1×: First pipeline stage on multiplication, 2×: Second pipeline stage on multiplication

Fig. 48. Loop pipelining with pipelined multipliers.

If a critical functional resource such as a multiplier has both large area and long latency, the resource sharing and resource pipelining can be applied at the same time in such a way that the shared resource executes multiple operations at the same time in different pipeline stages. With this technique, the conditions for resource sharing are relaxed and so the critical resources are utilized more efficiently. Fig. 48 shows this situation. Through the pipelining, we can reduce the number of multipliers from 8 to 4 to perform the execution without any stall. This is because two PEs sharing one pipelined multiplier can perform two multiplications at the same time using different pipeline stages.

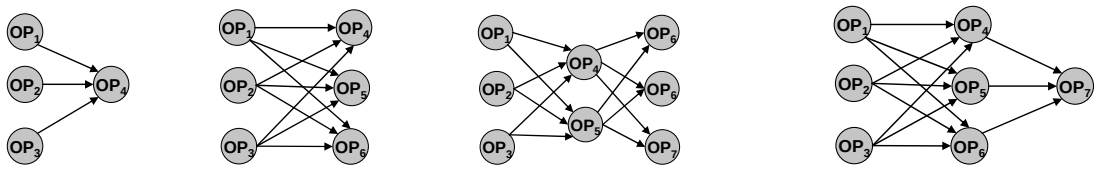
B. Cost-Effective Reconfigurable Array Fabric

In this section, we propose an array fabric design space exploration method to generate a cost-effective reconfigurable array structure in terms of area and power. It is mainly motivated by the characteristics of typical computation-intensive and data-parallel applications.

1. Motivation

a. Characteristics of Computation-Intensive and Data-Parallel Applications

Most of the CGRAs have been designed to satisfy the performance requirement of a range of applications in a particular domain. Especially, they have been designed for applications that exhibit computation-intensive and data-parallel characteristics. Common examples for such applications are digital signal processing (DSP) applications like audio signal processing, image processing, video signal processing, speech signal processing, speech recognition, and digital communications. Such applications have many subtasks such as trigonometric functions, filters and matrix/vector operations that can be mapped onto coarse-grained reconfigurable array. We have classified such subtasks into four types as shown by the data flow graphs in Fig. 49. Type (a) shows merge operation in which outputs from multiple operations in the previous stage are used as inputs to an operation in the next stage. Type (b) shows butterfly operation where output data from multiple operations in the previous stage are fed as input data to the same number of next stage operations. Finally, type (c) and (d) show the combinations of (a) and (b).



(a) Merge (b) Butterfly (c) Merge-butterfly (d) Butterfly-merge

Fig. 49. Subtask classification.

b. Redundancy in Conventional Array Fabric

Most coarse-grained reconfigurable arrays arrange their processing elements (PEs) in a square or rectangular 2-D array with rich set of horizontal and vertical connections for effective exploitation of parallelism. However, such square/rectangular array structures have many redundant or unutilized PEs during the executions of applications on them. Fig. 50 shows an example of three types of data flow (Fig. 49 (a), (c), and (d)) mapped onto 8x8 square reconfigurable arrays in the two cases of loop pipelining – temporal mapping and spatial mapping. The upper part of Fig. 50 shows the scheduling for a column of PEs based on temporal mapping and also shows how the utilization of the PEs changes for the 8 cycles of schedule. As can be seen from the figure, Some PEs have very low utilization. The lower part of Fig. 50 shows the spatial mapping of the 8x8 array, where some PEs are not used at all. All the three types of implementations show lots of redundant PEs that are not used.

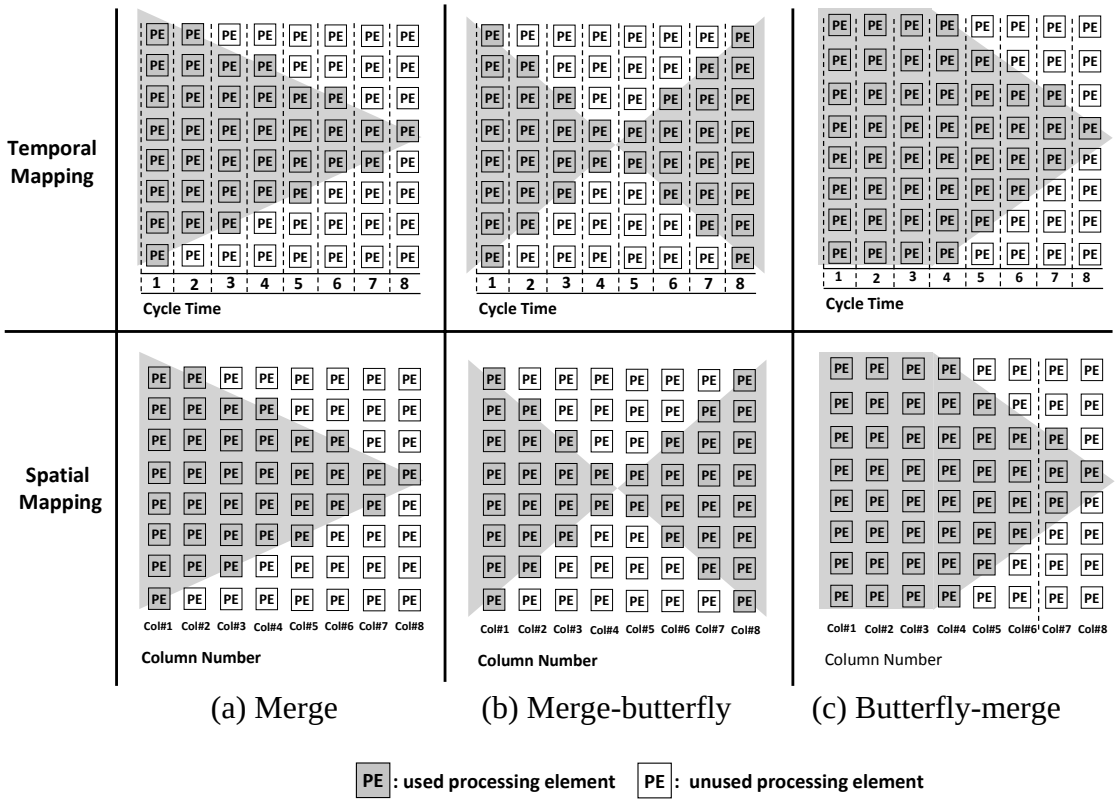


Fig. 50. Data flow on square reconfigurable array.

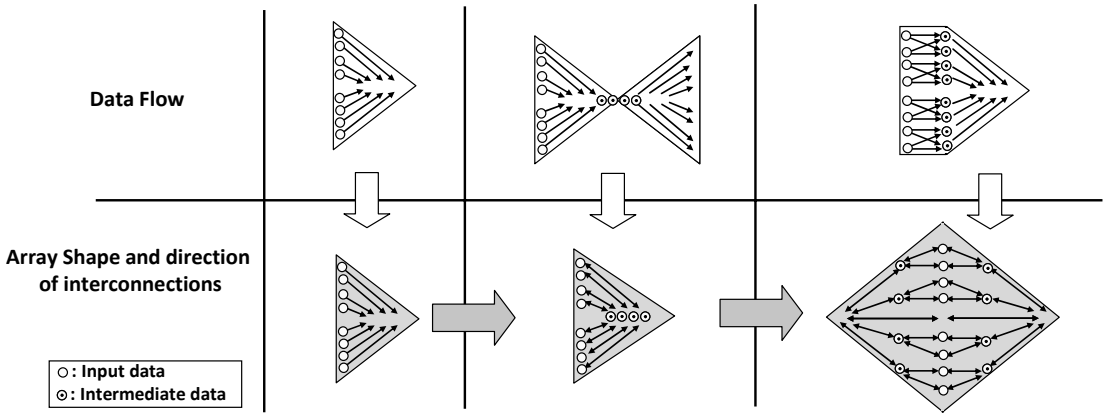
From these observations, we see that the existing square/rectangular array fabric cannot efficiently utilize the PEs in the array and therefore waste large area and power. In order to overcome such wastages in square/rectangular array fabric, we propose a new cost effective array fabric in the next subsection.

2. New Cost Effective Data Flow-Oriented Array Structure

a. Derivation of Data Flow-Oriented Array Structure

To reduce the redundancy in the conventional square/rectangular array, first of all, we can consider a specific array shape that fits well with the applications' common data flows. Fig. 51 shows such a data flow-oriented array structure derived from three types

of data flow. In Fig. 51 (a), a triangular-shaped array and uni-directional interconnections among PEs can be derived from the first data flow (merge). Then the interconnections can be made bi-directional to support the merge—butterfly data flow as shown in Fig. 51 (b). Finally, in Fig. 51 (c), the entire array becomes a diamond-shaped structure to reflect the butterfly-merge data flow. In this case, the butterfly operations are spatially spread on both sides of the array. Then intermediate data merge takes place at the end of both sides or they can merge at the center of the array.

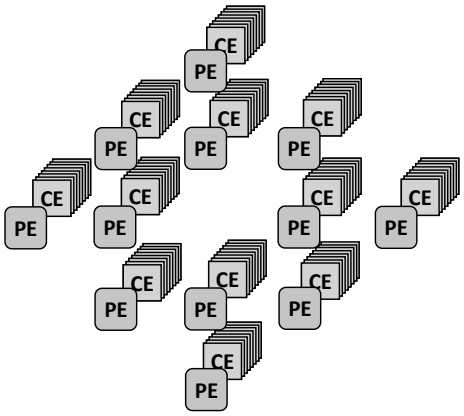


(a) Merge (b) Merge-butterfly (c) Butterfly-merge

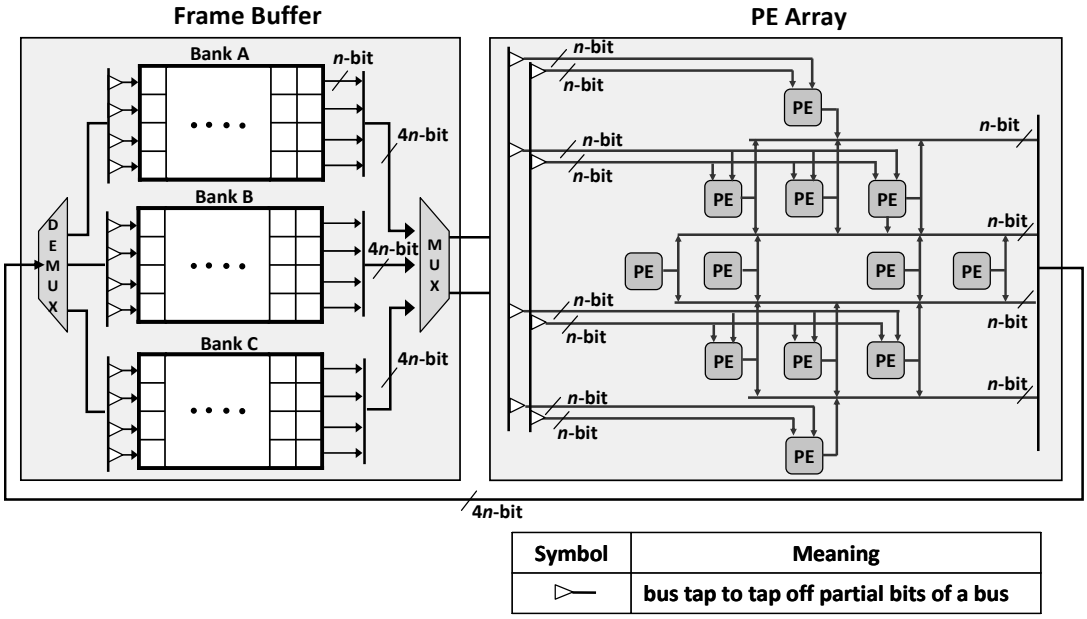
Fig. 51. Data flow-oriented array structure derived from three types of data flow.

To represent how the data-flow oriented array structure can efficiently utilize PEs, we examine the difference between the conventional square-shaped array and the proposed data flow-oriented array with a simple example. We assume a diamond-shaped reconfigurable array composed of 12 PEs as shown in Fig. 52 (a) – this is a counterpart of the 4x4 PE array shown in Fig. 10. In addition, we assume a Frame Buffer similar to the one in Fig. 10 (b) is connected to the array, where the PEs in each row of the array

share two read buses and the PEs in two neighboring rows share one write bus as shown in Fig. 52 (b). The array has nearest neighbor and global bus interconnections in diagonal and horizontal directions as shown in Fig. 52 (c) and (d).

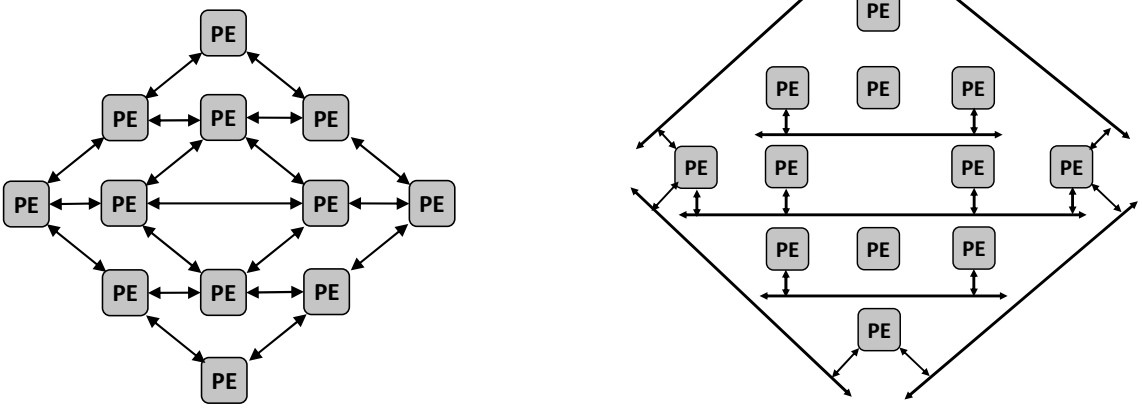


(a) Distributed cache structure



(b) Frame buffer and data bus

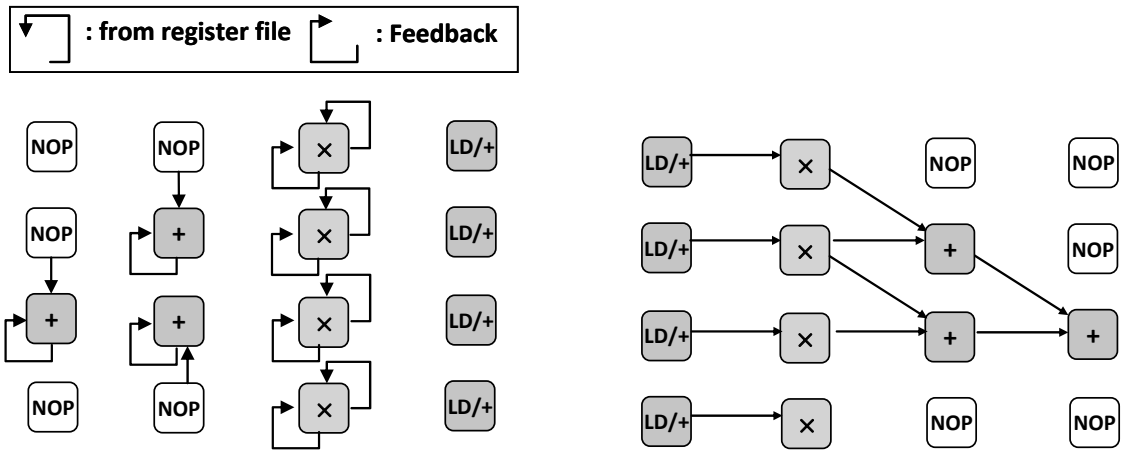
Fig. 52. An example of data flow-oriented array.



(c) Nearest neighbor interconnection (d) Global bus interconnection

Fig. 52. Continued.

Consider mapping of Eq. (2) in Chapter IV with $N = 4$ on the proposed array in the same way as we did for the 4x4 square-shaped PE array in Chapter IV. Fig. 53 shows the snapshots taken at the time of maximum utilization of PEs for three cases: (a) temporal mapping on the square-shaped array (4x4 PEs), (b) spatial mapping on the square-shaped array (4x4 PEs) and (c) spatial mapping on the data flow-oriented array (12 PEs). In the case of (a) and (b), five PEs are not used because the merging addition does not fit well with the square-shape. However, in the case of (c), the array efficiently utilizes all of the PEs without delayed operation. As can be observed, this example shows that the propose array structure can avoid the area and power wastages of the square-shaped array without performance degradation.



(a) Temporal mapping on 4x4 PE array (b) Spatial mapping on the 4x4 PE array

(c) Spatial mapping on the data-flow oriented PE array

Fig. 53. Snapshots showing the maximum utilization of PEs.

b. Mitigation of Spatial Limitation in the Proposed Array Structure

As shown in Fig. 53 (c), we spread the operations in the data flows (mostly loop bodies) over the array space, instead of spreading the operations over time for each column to implement temporal loop pipelining as shown in Fig. 53 (a). This implies that spatial loop pipelining is most suitable to the new array fabric. However, as mentioned in

Chapter IV (see subsection A.2), spatial mapping is not feasible for complex loops because of two reasons. One is that a large loop body may not fit in the limited reconfigurable array and the other is that data dependencies between the operations typically require allocating lots of interconnect resources. In order to mitigate such a limitation, the new array fabric should have rich interconnections to provide more flexible and multi-directional data communication and the PEs should be arranged in such a way to utilize such an interconnection structure efficiently. As a solution to this problem, we propose a design flow that generates a data-flow oriented array structure by determining the topology of PEs and their interconnections.

3. Data Flow-Oriented Array Design Flow

The generation of a data-flow oriented array starts from a square-shaped array fabric, considering that the original square-shaped array fabric is very well designed. We generate the new data-flow oriented array such that it can efficiently implement any application that can be implemented on the square-shaped array fabric. In the example of Fig. 52 (a), since the data-flow has a diamond-shape, we can generate a diamond-shaped array with

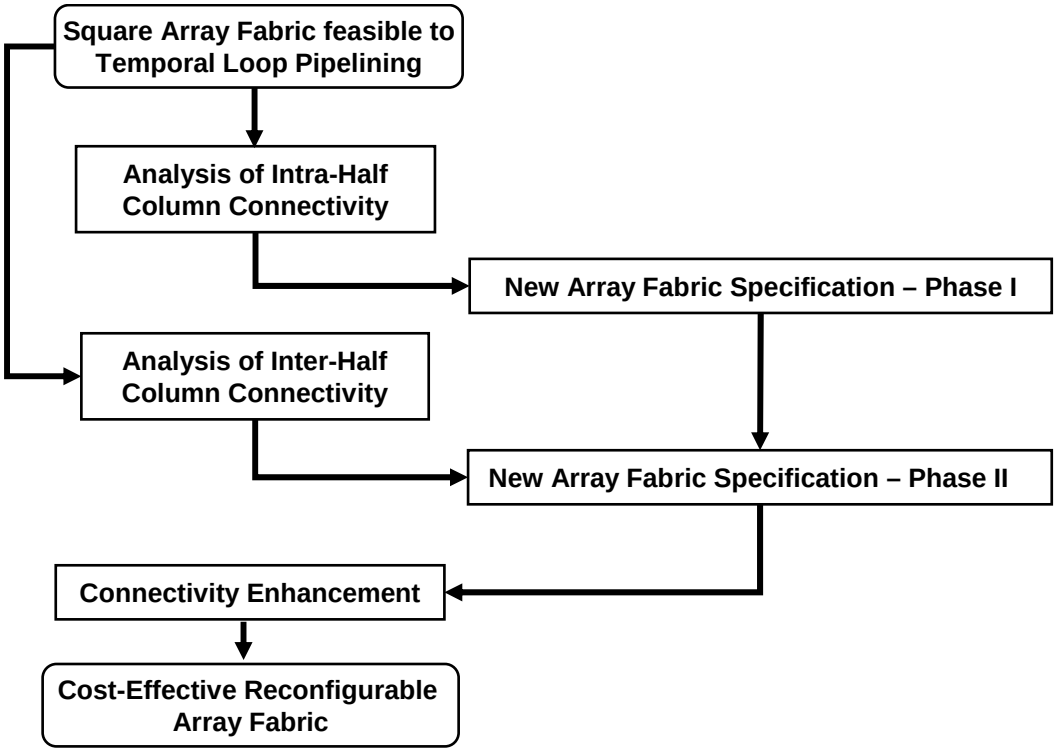


Fig. 54. Overall design flow.

less number of PEs but without any performance degradation, which is just like garment cutting. Since we want to cover the applications that can be implemented through temporal mapping on the square-shaped array fabric as well, we do not just cut the fabric but we compose the new array by transforming the temporal interconnection structure to a spatial interconnection structure.

In the temporal mapping, each loop iteration of an application kernel (critical loop) is mapped onto a column of the square-shaped array. Therefore, it is good enough to analyze the interconnection fabric only within a column to derive the new array structure.

Fig. 54 shows the entire design flow. This flow starts from analysis of *intra-half column* and *inter-half column connectivity* of general square array fabric. *Intra-half col-*

umn connectivity means nearest neighbor or hopping interconnection between PEs in a half column and *inter-half column connectivity* means pair-wise interconnection or global bus between PEs – one PE in a half column and another PE in the other half column. New array fabric is partially derived by analyzing intra-half column connectivity of square fabric in Phase I. Then Phase II elaborates new array fabric by analyzing inter-half column connectivity of square fabric. Finally, the connectivity of new array fabric is enhanced by adding vertical and horizontal global bus. In the remainder of this subsection -- from a through d below -- we describe more detailed process for each stage of the entire exploration flow.

a. Input Reconfigurable Array Fabric

The 8x8 array given in Fig. 5 is used for the input array fabric to illustrate the proposed design flow.

b. New Array Fabric Specification – Phase I

In this phase, an initial version of the new array fabric is constructed by analyzing intra-half column connectivity of the input square array. Algorithm 6 shows this procedure. Before we explain the procedure in detail, we describe the notations used in it.

- (L1) *base column* denotes a half column in the *n* x *n* reconfigurable array.
- (L3) *new_array_space* denotes 2-dimentional space of the constructed reconfigurable array.
- (L5) *source_column_group* denotes a group of PEs composed of one or two columns in the *new_array_space.* It is used as a source for deriving the new array fabric.
- (L12) |*source_column_group*| denotes the number of PEs in *source_column_group*.

• (L6) CHECK_INTERCONNECT is a function to identify nearest neighbor or hopping interconnections of PEs in *source_column_group* by analyzing the base column. If there is such an interconnection that has not been processed yet, then it returns true.

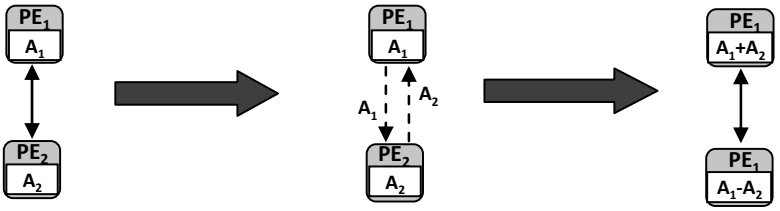
• (L8) LOC_TRI is a function that implements the *local triangulation* method, which adds PEs, assigns them new positions in *new_array_space*, and connects them with the PEs already existing in the *source_column_group*.

Algorithm 6 New **A**rray **F**abric **S**pecification – Phase I

```
L1    base ← a half column of n x n reconfigurable array
L2    m ← number of memory-read buses of n x n reconfigurable array
L3    new_array_space ← Ø
L4    begin
L5      source_column_group ← Add a column composed of n/2 PEs
                              in new_array_space
L6      while CHECK_INTERCONNECT(source_column_group, base)
L7      do
L8            LOC_TRI(source_column_group)
L9      end do
L10     source _column_group ← Ø
L11     source _column_group ← next two columns on the both sides
                               in new_array_space
L12     if |source-column_group| > 2 then
L13       goto L6
L14     end if
L15     Add nearest-neighbor interconnections
L16     Add m memory-read buses
L17     Connect the read buses with the added PEs in the same row
L18     Copy the constructed fabric on vertically symmetric position
L19   end
```

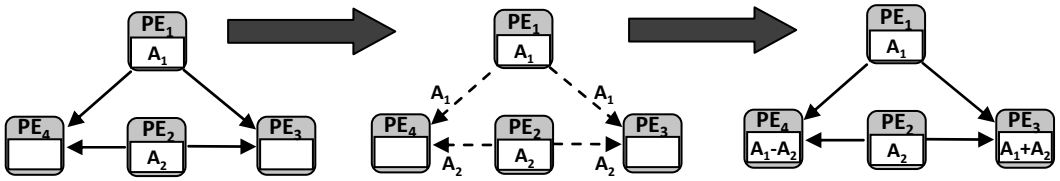
The algorithm starts with the initialization step (L1~L3). Then a half column is added into *new_array_space*, which is the initial *source_column_group* (L5). The next process is to check the nearest neighbor or hopping connectivity between two PEs (L6) in the same column included in *source_column_group.* This checking process (L6) con-

tinues until no more interconnection is found. The first checking process is performed by simply identifying interconnections of the base column. If an interconnection is found, two PEs are added into *new_array_space* and their interconnections and positions are assigned by *local triangulation* method (L8). This method is to reflect intra-half column connectivity with making the data flow oriented array structure as shown in Fig. 52.



(a) An operation fully utilizing interconnections between two PEs

(b) Butterfly operation example



(c) Butterfly operation executed on a triangle structure including four PEs

Fig. 55. Basic concept of *local triangulation* method.

We illustrate the basic concept of *local triangulation* method in Fig. 55. Consider a base column including 2 PEs and let the operation fully utilizing their interconnections as shown in Fig. 55 (a) – data (A_1 and A_2) saved in two PEs (PE_1 and PE_2) are ex-

changed with each other through the bidirectional interconnection, and then addition and subtraction are performed in PE_1 and PE_2. This is a kind of butterfly operation and Fig. 55 (b) shows an equivalent data flow graph for the butterfly operation. If we consider a triangular structure composed of four PEs reflecting the shape of the data flow graph, the example can be mapped on the PEs as shown in Fig. 55 (c) – Two PEs (PE_3 and PE_4) on both sides receive the data (A_1 and A_2) from the PEs (PE_1 and PE_2), and then addition and subtraction are performed in PE_3 and PE_4. In such a manner, *local triangulation* method is to make a data flow-oriented array structure reflecting the intra-half column connectivity.

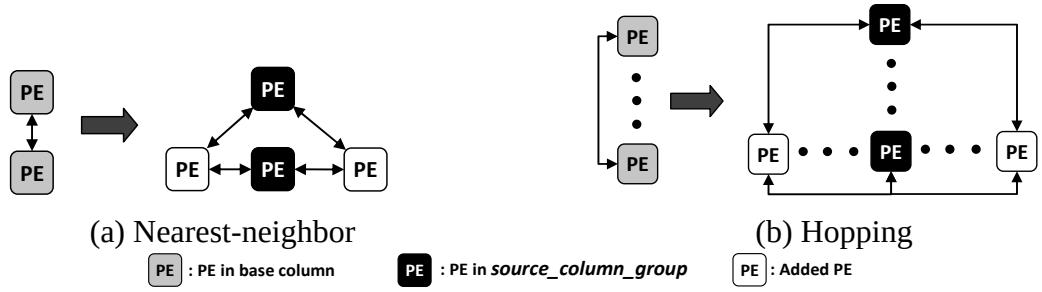


Fig. 56. *Local triangulation* method.

Fig. 56 shows two cases of the method. In the first case (a), two PEs in the base column have nearest-neighbor interconnection, which means maximum two PEs can be used for butterfly operation. Therefore, *local triangulation* method adds two PEs into *new_array_space* and assigns each PE the nearest-neighbor position on each side of the source column and the positions are vertices of a triangle. Then the method assigns nearest-neighbor interconnection between added PEs and the PEs in the

source_column_group. The second case (b) shows that two PEs in the base column have a bidirectional hopping interconnection. *Local triangulation* method is also applied to this case with the hopping interconnections instead of the nearest neighbor interconnections for the first case. Even though one-way interconnections are sufficient to perform butterfly operation in two cases of Fig. 56, the added interconnections are bidirectional. This is because it aims to keep the basic characteristics of the data flow-oriented array structure derived in Fig. 52.

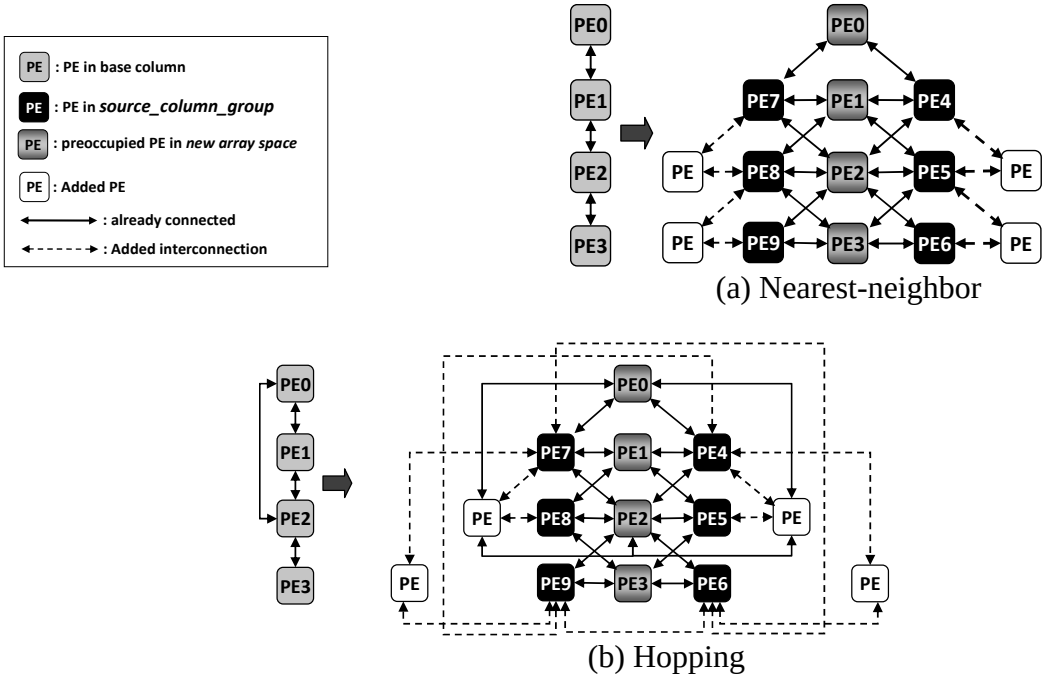
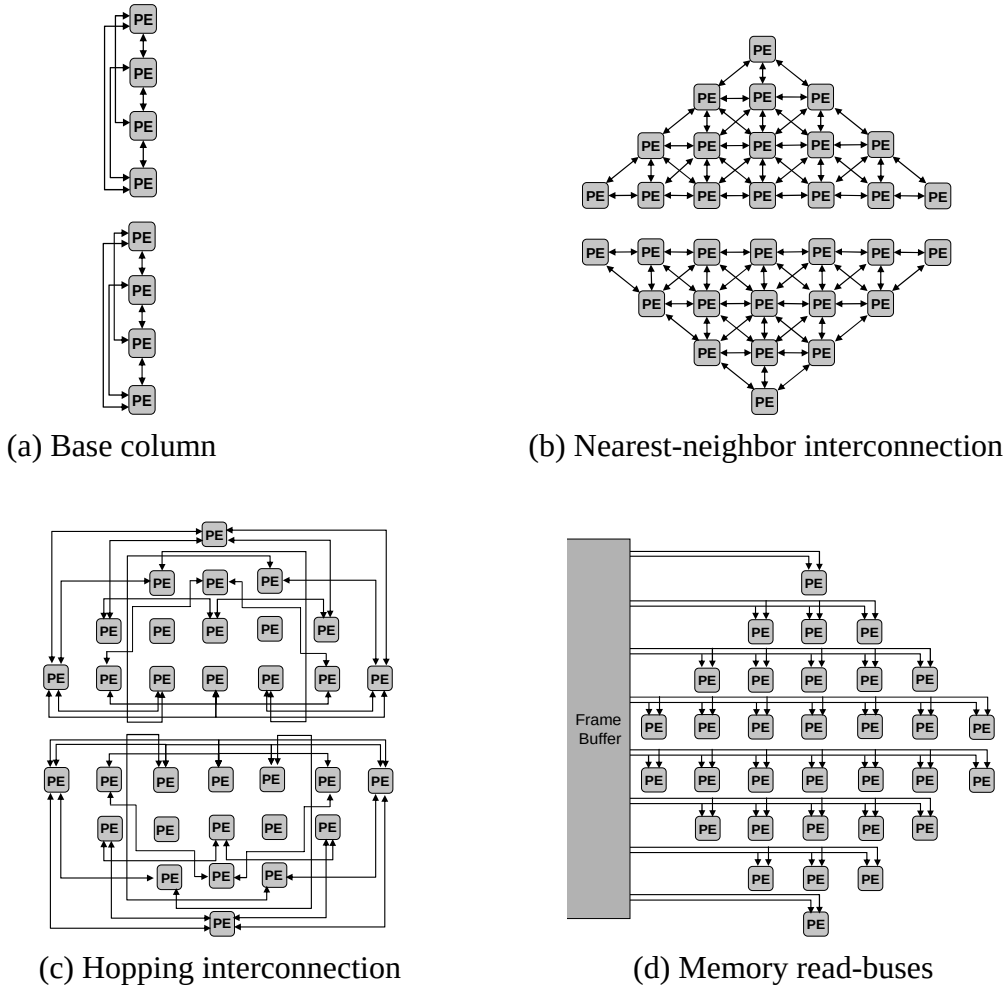


Fig. 57. Interconnection derivation in Phase I.

From the second checking process, preoccupied columns are included in *source_column_group*. Fig. 57 shows two examples on how to find connectivity on the source columns. In the case of (a), no interconnection between ‘PE4’ and ‘PE6’ (or

‘PE7’ and ‘PE9’) is added because there is no hopping connectivity between ‘PE0’ and ‘PE2’ (or ‘PE0 (or PE1)’ and ‘PE3’) in the base column. However, in the case (b), the base column has interconnection between ‘PE0’ and ‘PE2’. Therefore PEs and intercon-



(a) Base column (b) Nearest-neighbor interconnection

(c) Hopping interconnection (d) Memory read-buses

Fig. 58. New array fabric example by Phase I.

nections between ‘PE4’ and ‘PE6’ (or ‘PE7’ and ‘PE9’) are added by *local triangulation* method. After the iteration of adding PEs and interconnections (L5~L14) is finished, nearest-neighbor interconnections are added between two nearest-neighbor PEs that are

not connected with each other (L15). It is to guarantee the minimum data-mobility for data rearrangement. Finally, memory-read buses are added[4] (L16, L17) and the derived array is copied to the vertically symmetric position (L18).

Fig. 58 shows the result of the phase I procedure for the example of 8x8 reconfigurable array as shown in Fig. 5.

c. New Array Fabric Specification – Phase II

In this phase, new PEs and interconnections are added for reflecting intra-half column connectivity of the input square fabric. Phase II analyzes two kinds of interconnections – pair-wise and global bus. We propose another procedure as Algorithm 7. Before we explain the procedure in detail, we introduce notations we use in the explanation.

- (L5) CHECK_INTERCONNECT is a function to identify bus-connectivity between two PEs in the source column by analyzing the base column.
- (L6) GB_RHT_TRI means *global triangulation* method that is a function used to add global buses and PEs.

The algorithm starts with initialization step (L1, L2). Then central column in *new_array_space* is initial *source_column_group*. Next process is to check the pair-wise or global bus connectivity between two PEs (L5) - two PEs in the same column included in *source column group*. If an interconnection is found, global buses and PEs are added

[4] Memory write-buses are added in the step of connectivity enhancement in subsection VI.B.5). This is because some PEs can be added in phase II and they should be connected to memory-write buses.

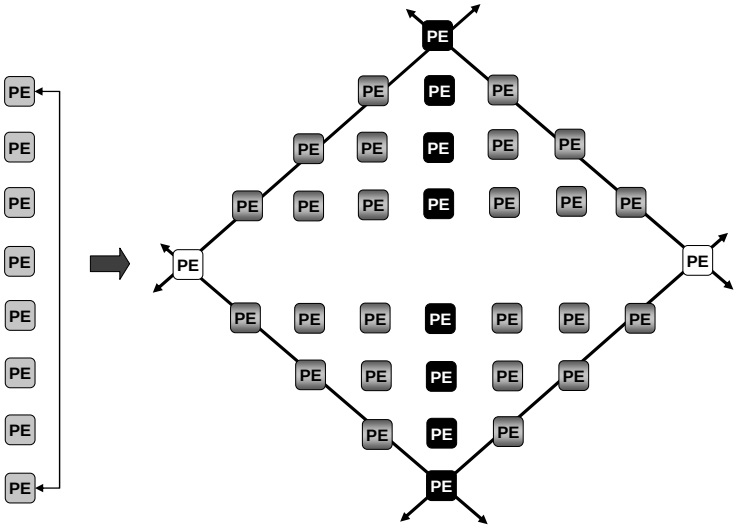
Algorithm 7 **N**ew **A**rray **F**abric **S**pecification - Phase II

```
L1    base ← a column of n x n reconfigurable array
L2    n ← number of global buses
L3    begin
L4      source_column_group ← central column in new_array_space
L5      while CHECK_INTERCONNECT(source_column_group, base) do
L6            GB__TRI (source_column)
L7      end do
L8      source _column_group ← Ø
L9      source _column_group ← next two columns on the both sides
                               in new_array_space
L10     if |source-column_group| > 2 then
L11       goto L5
L12     end if
L13     Add nearest-neighbor interconnections
L14   end
```

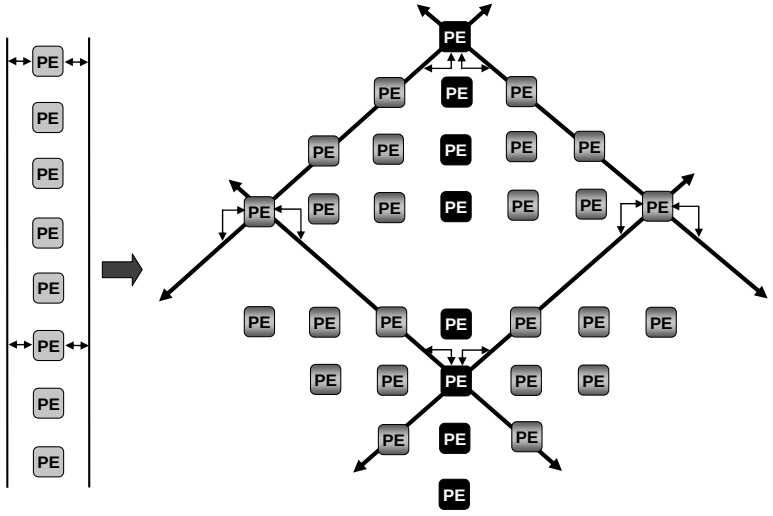
in *new_array_space* and their interconnections and positions are assigned by *global triangulation* method (L6). *Global triangulation* method has the same basic concept of *local triangulation* method in that the method is also to make a triangular-shaped array fabric suitable spatial mapping with guaranteeing the maximum inter-half column connectivity of the base column.

Fig. 59 shows three cases of *global triangulation* method when the base column has a bidirectional pair-wise interconnection and two global buses. In the first case (a), the bidirectional pair-wise interconnection means maximum two PEs can be used for butterfly operation. Therefore, *global triangulation* method adds two PEs in *new array space* and assigns each PE the intersection point of two diagonal lines from two PEs in source column. The positions are vertices of a triangle. Then the method assigns four global buses between added PEs and the PEs in the *source_column_group*. Fig. 59 (b) and (c) show the method when the base column has two global buses. In the case of (b), two di-

agonal lines from two PEs in source column intersect on already existing PE called ‘destination PE’. Therefore, four global buses are added and they connect destination PEs with PEs in the source column. However, in the case of (c), no destination PE exists on intersection point of four diagonal lines. Therefore, new PEs called global PE (GPE) as well as global buses are added on *new_array_space*.

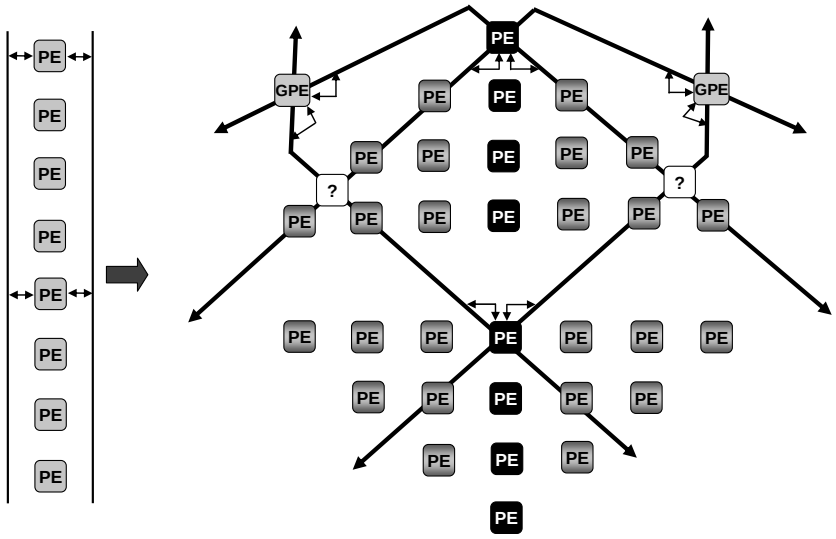


(a) When pair-wise interconnection exists in base column



(b) When destination PE exists

Fig. 59. *Global triangulation* method when n = 2 (L2).



(c) When GPE is added

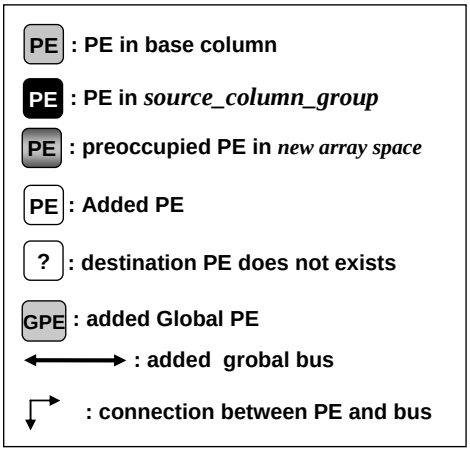


Fig. 59. Continued.

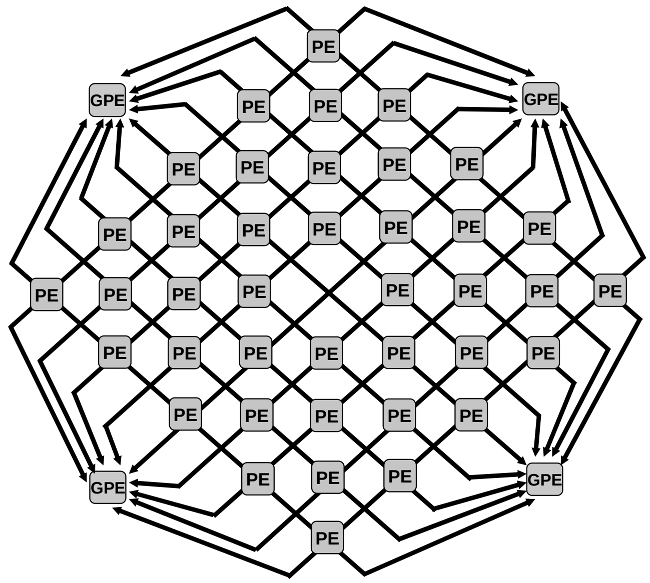


Fig. 60. New array fabric example by Phase II.

This checking process (L5) continues until no more connectivity is found. Then nearest-neighbor interconnections are added between two PEs not connected with each other. This is to guarantee the minimum data-mobility for data rearrangement.

Fig. 60 shows the result of the phase II procedure for the example of 8x8 reconfigurable arrays.

d. Connectivity Enhancement

Finally, vertical and horizontal bus can be added to enhance connectivity of new reconfigurable array. This is because new array fabric from phase I and II only has nearest neighbor or hopping interconnection in vertical and horizontal direction whereas it supports sufficient diagonal connectivity. Added horizontal bus is used as memory-write bus connected with frame buffer as well as used for data-transfer between PEs.

Fig. 61 shows the result of the connectivity enhancement for the example of 8x8 reconfigurable array. Each bus is shared by two PEs in both the sides.

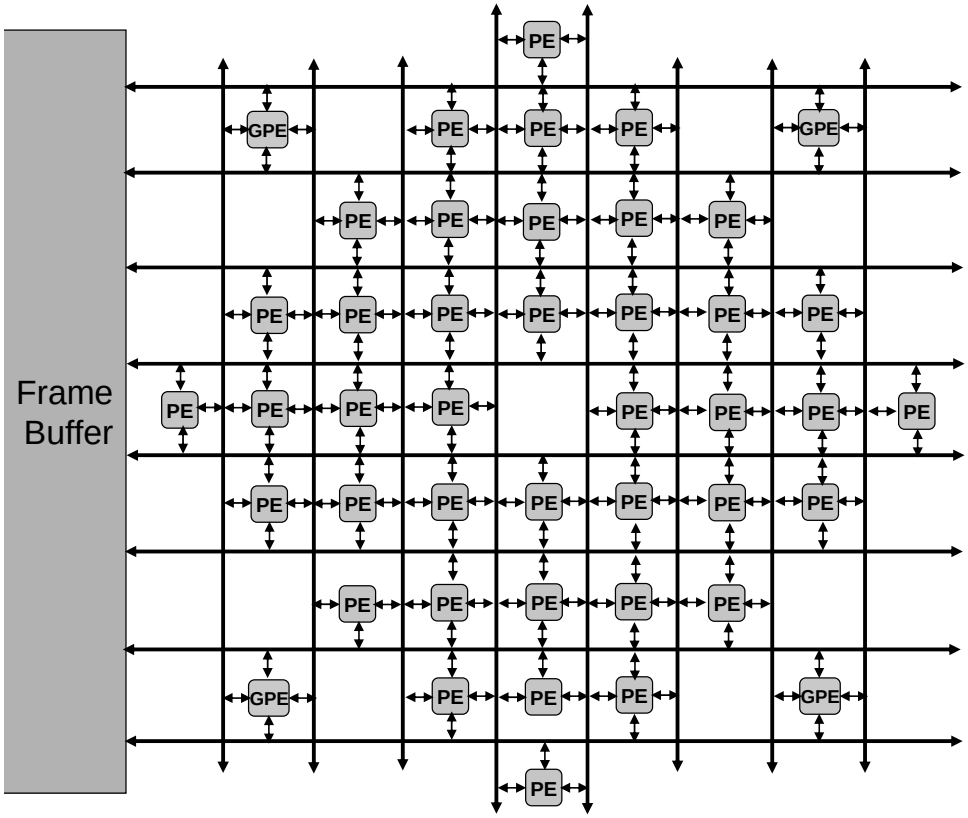


Fig. 61. New array fabric example by connectivity enhancement.

4. Cost-Effective Array Fabric with Resource Sharing and Pipelining

The resource sharing and pipelining mentioned in Section A can be applied to the proposed new array fabric because the computation model for the proposed fabric is spatial loop pipelining – spatial mapping spreads the entire loop body on the PE array, there is no need for all PEs to have the same functional resources at the same time. Fig. 62 shows the PEs in the same row share two pipelined multipliers.

Fig. 63 shows an application mapping on new array fabric example generated from the exploration flow - consider $N = 8$ for the mapping of the computation defined in Eq. (2) in Chapter IV on the new array fabric as shown in Fig. 53. Load and addition operations in PEs are executed on the central column in the first cycle. Then the next multiplications and summations are spatially spread on both sides of the array till 6th cycle. Finally, in next two cycles, a PE in the central row performs multiplication/store operations. The architecture including 16 multipliers supports the mapping example without stall caused by

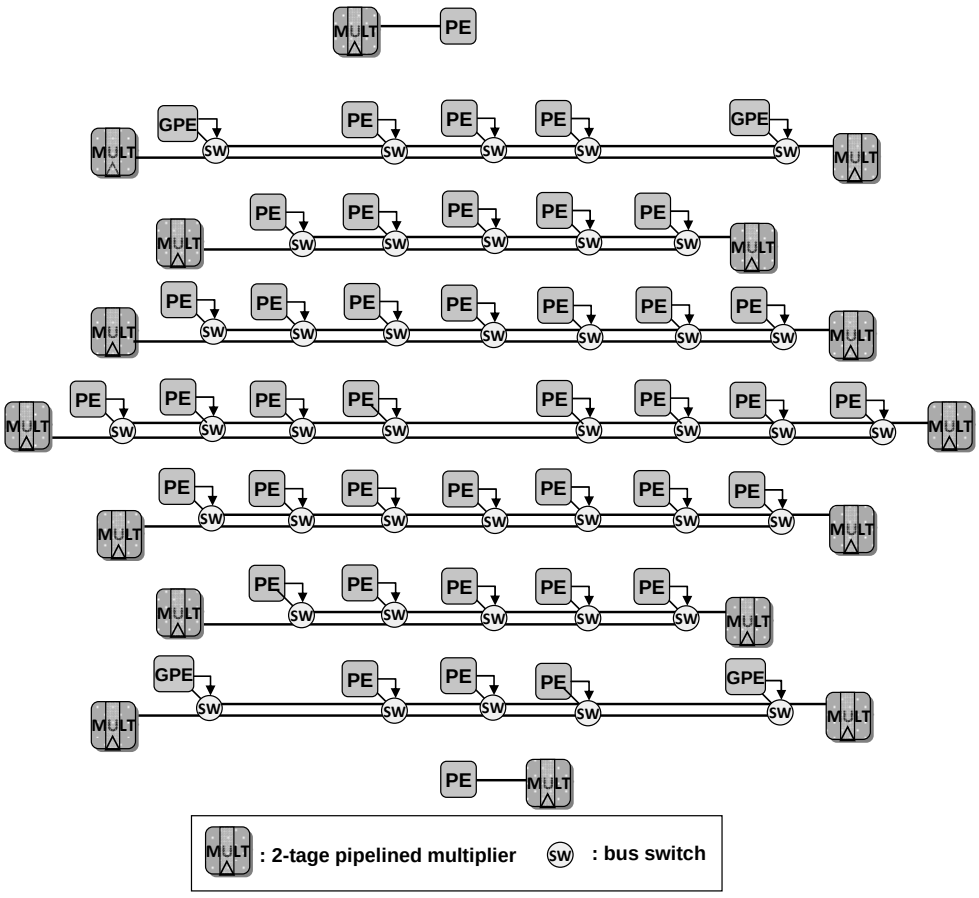


Fig. 62. New array fabric with resource sharing and pipelining.

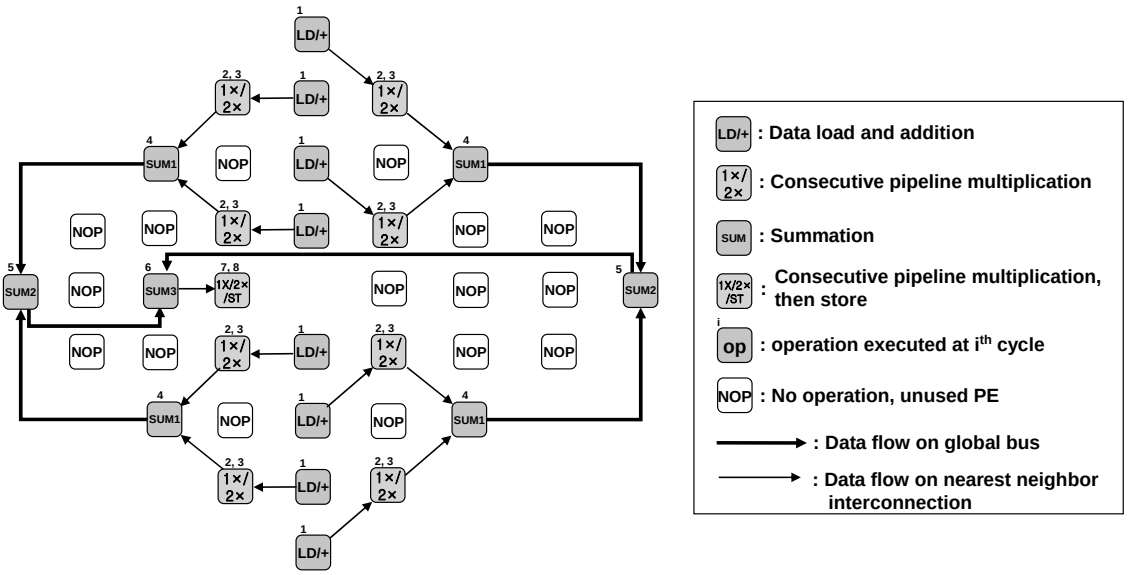


Fig. 63. Mapping example on new array fabric.

multiplier lack. In this example, nearest-neighbor interconnections and global buses are efficiently used for multi-directional data-transfer and the new array has the same performance (number of execution cycles) compared with the square array.

C. Experiments

1. Experimental Setup

a. Evaluated Applications

The target application-domain is composed of representative kernels in MPEG-4 AAC decoder, H.263 encoder, H.264 decoder, and 3D-graphics. In addition, to demonstrate the effectiveness of our approaches for benchmark domains, we have applied several kernels of Livermore loops benchmark [58] and DSPstone [59].

b. Hardware Design and Power Estimation

To demonstrate the effectiveness of Resource Sharing and Pipelining (RSP), we have applied RSP techniques to the base RAA (BASE) defined in Chapter III (see Section B) and implemented the RSP architecture (RSPA) at the RT-level with VHDL. In Chapter III (see Section C), we have confirmed that multiplier is both area-critical and delay-critical resources. Therefore we have taken the multiplier out of the PE design and arranged them to be shared and pipelined resources. From the analysis of our target applications, we have determined the sharing architecture – two pipelined multipliers shared by 8 PEs in each row. Therefore, the RSP architecture including 16 multipliers supports all of the target applications without stall caused by multiplier lack. In addition, we have implemented entire exploration flow in Fig. 54 with C++. The implemented exploration flow has generated the specification of new reconfigurable array fabric. The base RAA (Chapter III) has been used for input of the exploration flow. For quantitative evaluation, we have designed two cases of PE array based on the generated specification at the RT-level with VHDL – only new array fabric (NAF) and NAF with RSP (RSP+NAF) - 18 multipliers are shared by PEs in both row and column directions and this architecture also supports all of the target applications without stall caused by multiplier lack. The architectures have been synthesized using Design Compiler [49] with 0.18 ㎛ technology. ModelSim [50] and PrimePower [49] are used for gate-level simulation and power estimation. Simulation has been done for the typical case under the condition of 100 MHz operation frequency, 1.8 V Vdd, and 27℃ temperature.

2. Results

a. Area Evaluation

Table X shows area cost evaluation for the four cases. In RSPA, the area cost of PE array is reduced by 22.11% because it has less multipliers than BASE. In the case of NAF, the area reduction ratio (25.58%) has relatively increased compared to RSPA. This is because the number of PEs is reduced than RSPA. Finally, the area reduction ratio (36.75%) in RSP+NAF has also relatively increased compared to NAF because of reduced multipliers. However, the interconnect area of the RSPA (or RSP+NAF) has increased compared to the BASE(or NAF). This is because several buses are added to connect the shared multipliers with PEs.

Table X. Area Reduction Ratio by RSPA and NAF

PE Array Structure	Number of PEs	Number of Multipliers	Gate Equivalent			Reduction Ratio (%) compared with BASE
			Interconnect[a]	Logic[b]	Total[c]	
BASE	64	64	164908	494726	659635	-
RSPA	64	16	170008	343781	513789	22.11
NAF	44	44	156163	334737	490900	25.58
RSP+NAF	44	18	164414	252805	417219	**36.75**

Interconnect[a]: net interconnect area, Logic[b]: total cell area, Total[c] : Interconnect[a] + Logic[b]

b. Performance Evaluation

The synthesis results show that RSPA has reduced critical path delay (5.12 ns) compared to BASE (8.96 ns). This is because RSP technique excludes the combinational logic path of the multiplier from the original set of critical paths. The critical path of RSPA and its delay is given by

$$T_{Critical\ path} = T_{Multiplexor} + T_{ALU} + T_{Shift\_logic} + T_{others} \quad (3)$$

(5.12 ns = 0.32 ns + 2.22 ns + 1.42 ns + 1.16 ns)

Table XI shows that BASE and NAF (or RSPA and RSP+NAF) have same critical path delay. It indicates NAF does not cause performance degradation in terms of the critical path delay. In addition, the execution cycle count of each kernel on NAF (or RSP+NAF)

Table XI. Applications Characteristics and Performance Evaluation

Kenels	Operations[c]	PE Array Structure				
		BASE and NAF (8.96 ns)[d]		RSPA and RSP+NAF (5.12 ns)[d]		
		Cycle count	[e]ET(ns)	Cycle count	[e]ET(ns)	[f]Reduced (%)
[a]First_Diff	sub	15	134.40	15	76.80	42.86
[a]Tri- Diagonal	sub, mult	17	152.32	18	92.16	39.50
[a]State	add, mult	20	179.20	23	117.76	34.29
[a]Hydro	add, mult	15	134.40	19	97.28	27.62
[a]ICCG	sub, mult	18	161.28	19	97.28	39.68
[b]Inner Product	add, mult	21	188.16	22	112.64	40.14
[b]24-Taps FIR	add, mult	20	179.2	21	107.52	40.00
Matrix-vector multi-plication	add, mult	19	170.24	20	102.4	39.85
Mult in FFT	add, sub, mult	23	206.08	27	138.24	32.92
Comlex Mult in AAC decoder	add, sub, mult	16	143.36	17	87.04	39.29
ITRANS in H.264 Decoder	add, sub, shift	18	161.28	18	92.16	42.86
DCT in H.263 encoder	add, sub, shift, mult	32	286.72	40	204.80	28.57
IDCT in H.263 encoder	add, sub, shift, mult	34	304.64	42	215.04	29.41
SAD in H.263 encoder	add, abs	39	349.44	39	199.68	42.86
Quant in H.263 encoder	add, sub, shift, mult	39	349.44	45	230.40	34.07
Dequant in H.263 encoder	add, sub, shift, mult	41	367.36	57	240.64	34.49

[a]Livermore loop benchmark suite. [b]DSPstone benchmark suite. [c]Acronym of operations, add:addition, sub: subtraction, shift: bit-shift, mult: multiplication, [d]Critical path delay, [e]Execution time = cycle × critical path delay(ns), [f]Reduction ratio of execution time compared with BASE.

does not vary from BASE (or RSPA) because the functionality of NAF is same as the base model. It also indicates NAF does not come by performance degradation in terms of the execution cycle count.

We have applied application kernels to the implemented architectures to obtain the results in Table XI. The amount of performance improvement depends on the application. For example, compared to DCT and hydro having multiplication, we achieve much more performance improvement with RSPA and RSP+NAF for First_Diff, SAD, and ITRANS which have no multiplication. This is because the clock frequency has been increased by pipelining the multipliers whereas the execution cycle count does not vary from BASE and NAF.

Table XII. Power Reduction Ratio by RSP+NAF

Kenels	PE Array Structure		
	BASE	RSP+NAF	
	Power (mW)	Power (mW)	Reduction Ratio (%) compared with BASE
First_Diff	201.07	129.79	35.45
Tri- Diagonal	190.75	130.89	31.38
State	198.37	138.62	30.12
Hydro	190.86	129.35	32.23
ICCG	164.42	112.92	31.32
Inner Product	200.09	139.30	30.38
24-Taps FIR	174.38	116.40	33.25
Matrix-vector multiplication	163.25	113.48	30.49
Mult in FFT	187.68	125.30	33.24
Comlex Mult in AAC de-coder	222.14	148.55	33.13
ITRANS in H.264 decoder	198.32	137.89	30.47
DCT in H.263 encoder	212.25	147.90	30.32
IDCT in H.263 encoder	208.99	143.58	31.30
SAD in H.263 encoder	181.22	123.23	32.00
Quant in H.263 encoder	199.38	137.33	31.12
Dequant in H.263 encoder	196.97	131.28	33.35

c. Power Evaluation

Table XII shows the comparison of power consumptions between the two reconfigurable arrays: BASE and RSP+NAF. The two arrays have been implemented without any low power technique to evaluate their power savings. It is shown that compared to BASE, RSP+NAF could save up to 35.45% of the power. It has been possible to reduce power consumption in RSP+NAF by using less number of PEs and multipliers to do the same job compared to the base reconfigurable array. For larger array sizes, the power saving will further increase due to significant reduction in unutilized PEs.

CHAPTER VIII

HIERARCHICAL RECONFIGURABLE COMPUTING ARRAYS

In this chapter, we propose a new computing hierarchy consisting of two reconfigurable computing blocks with two types of communication structure together [63]. In addition, the two computing blocks have shared critical resources. Such a sharing structure provides efficient communication interface between them with reducing overall area. Based on the proposed architecture, optimized computing flows have been implemented according to the varying applications for low power and high performance. Experimental results show that the proposed approach reduces on-chip area by 22%, execution time by up to 72% and reduces power consumption by up to 55% when compared with the conventional CGRA-based architectures.

A. Motivation

1. Limitation of Existing Processor-RAA Communication Structures

A typical coarse-grained reconfigurable architecture consists of a microprocessor, a Reconfigurable Array Architecture (RAA), and their interface. We can consider three types of organizations in connecting RAA to the processor. First, the array can be connected to the processor through a system bus as an ‘Attached IP’ [3] [10][12][15][19][64] shown in Fig. 2 (a). In this case, the main benefit of this organization is the ease of constructing such a system using a standard processor without modifying the processor and its compiler. In addition, large data buffer of RAA can be used to support applications having

large inputs/outputs. However, the speed improvement using the RAA may have to compensate for significant communication overhead between the processor and RAA through system bus as well as SRAM-based large data buffer in RAA consumes much power. Second type of organization involves the array connected with the processor as a 'Coprocessor'[4][65][66] shown in Fig. 2 (b). In this case, the standard processor does not change and the communication is faster than 'Attached IP' type interconnects because the coprocessor register-set is used as data buffer of the RAA and the processor can access the register-set by coprocessor data transfer instructions. In addition, the register-set consumes less power than the data buffer of 'Attached IP'. Since the size of the register-set is fixed by the processor ISA, it creates performance bottleneck for registers-PE array traffic due to applications having large inputs/outputs run on the RAA. In the third type of organization, the array is placed inside the processor like a 'FU (Functional Unit)' [2][16][22][67][68] as shown in Fig. 2 (c). In this case, the instruction decoder issues special instructions to perform specific functions on the RAA as if it were one of the standard functional units of the processor. In this case, the communication speed is faster than 'Coprocessor' and power consumption of the data storage is less than 'Attached IP' because the processor register-set is used as data buffer of the RAA and the processor can directly access the register-set by the processor instructions. However, standard processor needs to be modified for due to integration with RAA and its compiler should be also changed. The performance bottleneck is caused by limited size of the processor registers as in the case of 'Coprocessor' type organization. Table XIII shows a summary about advantage and disadvantage of three coupling types.

Table XIII. Comparison of the Basic Coupling Types

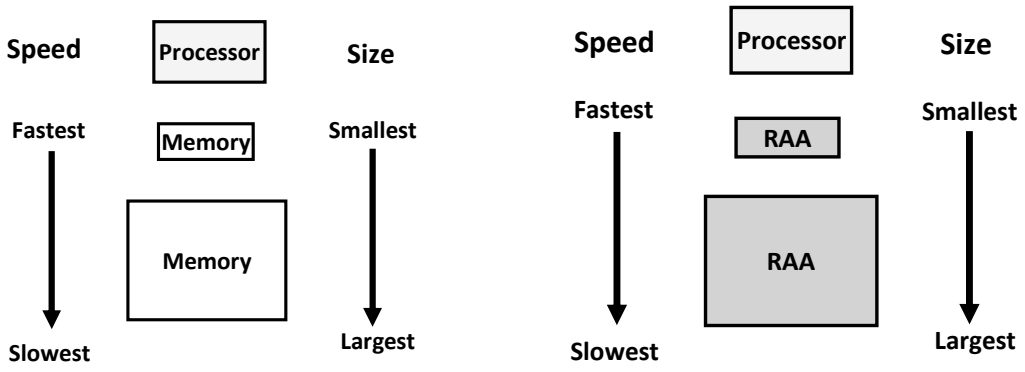
Coupling type	*Comm' power	**Comm' speed	Performance Bottleneck	Application feasibility
Attached IP	high	slow	communication through system bus	large size of input/output
Coprocessor	low	fast	limited size of coprocessor register-set	small size of input/output
Functional unit	low	very fast	limited size of processor registers	small size of input/output

*Comm' power: power consumption by data-storage (data buffer or registers)
**Comm' speed: Communication speed between processor and RAA

2. RAA-based Computing Hierarchy

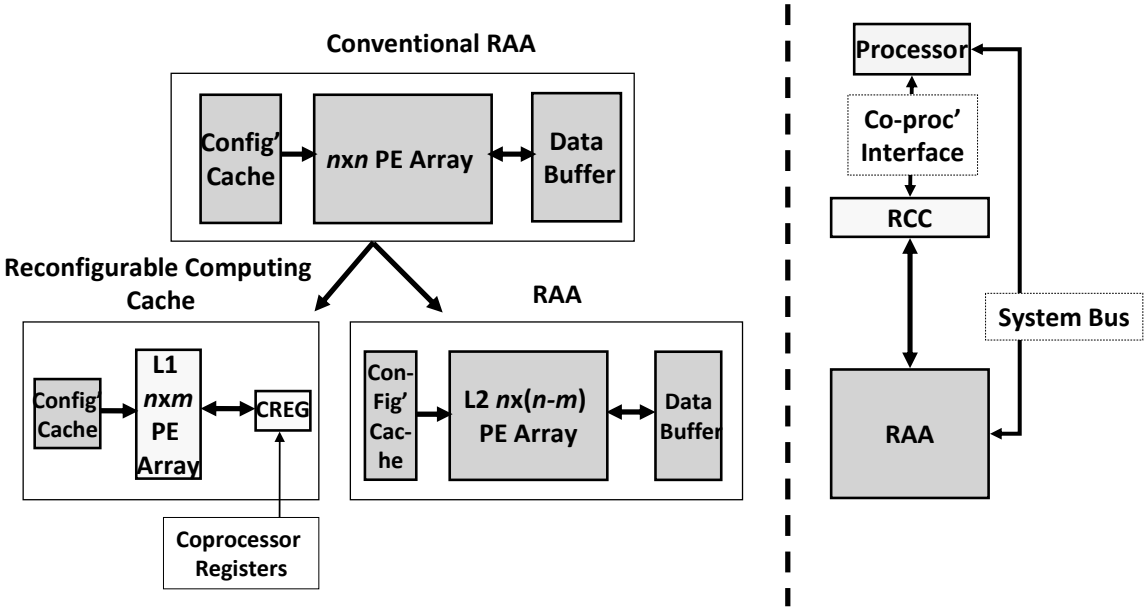
As mentioned in the previous subsection, basic three types of RAA organizations show advantage and disadvantage according the input/output size of the applications. It shows the existing coupling structure with a conventional RAA cannot be flexible to support various applications with sacrificing performance. In addition, such an RAA structure cannot efficiently utilize PE arrays and data buffers leading to high power consumption.

We hypothesize that if CGRA can maintain a computing hierarchy of its RAAs having difference size and communication speed as shown in Fig. 64 (b), the CGRA-based embedded system can be optimized for its performance and power. It is because such a hierarchical arrangement of the RAA can optimize the communication latency and efficiently utilize functional resources of PE array in various applications. In this chapter, we propose a new CGRA-based architecture that supports such a RAA-based computing hierarchy.



(a) Memory (b) RAA

Fig. 64. Analogy between Memory and RAA-computing hierarchy.



(a) Size (b) Speed

Fig. 65. Computing hierarchy of CGRA.

B. Computing Hierarchy in CGRA

In order to implement efficient CGRA-based embedded systems, we propose a new computing hierarchy consisting of two computing blocks using two types of coupling

structures together – ‘Attached IP’ and ‘Coprocessor’. In this organization, a general RAA having large size PE array is connected to a system bus and another is a small RAA composed of small PE array coupled with a processor through coprocessor interface. We call the small RAA *reconfigurable computing cache* (RCC) because it plays important role in enhancing performance and power of the entire CGRA like data cache. The RCC and the RAA share critical resources and such a sharing structure provides efficient communication interface between two computing blocks. The propose approach ensures that the RCC and the RAA are efficiently utilized to support variable size of inputs and outputs for variety of applications. In subsection B.1 and B.2, we describe computing hierarchy and resource sharing in RCC and RAA in detail. Then we show how to optimize computing flow based on reconfigurable computing cache according to the applications in subsection B.3.

1. Computing Hierarchy – Size and Speed

A CGRA-based computing hierarchy is formed by splitting a conventional computing RAA block into two computing blocks – RCC with small PE array and RAA having large PE array as shown in Fig. 65 (a). The RCC is coupled with coprocessor interface and the RAA is attached to a system bus as shown in Fig. 65 (b). The RCC provides fast communication with the processor and offers low power consumption by using coprocessor register-set and small array size. Therefore the RCC can enhance performance and reduce power consumption when small applications run on CGRA. If RCC is not sufficient to support computing requirements of in applications, intermediate data from the

RCC can be moved to the RAA through the interconnections as shown in Fig. 66. Such interconnections between the two blocks offer

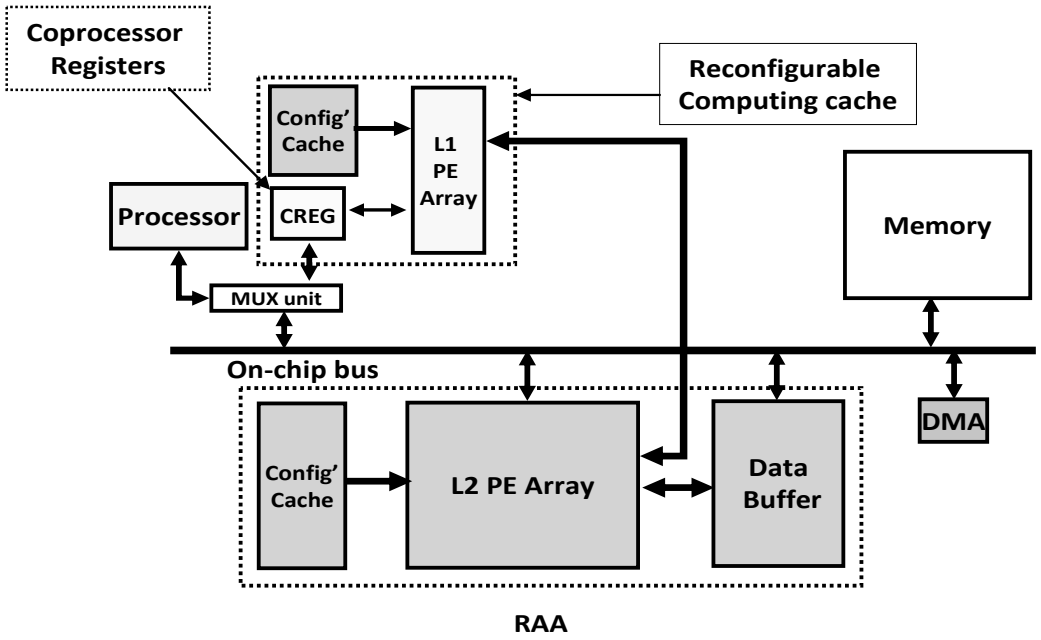


Fig. 66. CGRA configuration with RCC and RAA.

flexibility in migrating computing demands from one to another. Such computing flow may help to optimize performance and power for the applications having various sizes of inputs /outputs whereas the existing models show performance bottlenecks caused by the communication overheads or their limited sized data-storage as shown in Table XIII. We have described the computing flow optimization in detail in subsection B.3.

2. Resource Sharing in RCC and RAA

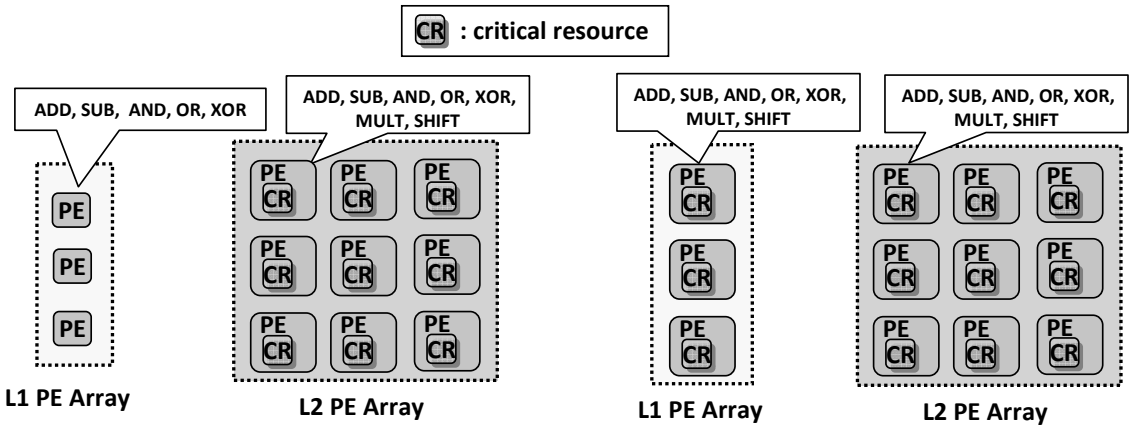
We have so far presented two factors (speed and size) in building computing hierarchy for CGRAs similar to memory hierarchy. It seems a small portion of RAA has been detached from large CGRA block and placed as the fast RCC block adjacent to the proces-

sor coupled with coprocessor interface. However, only considering two factors is not sufficient to design compact RCC for power and area benefits. This is because computing blocks can have diverse functionality which affects the system capabilities. The functionality of computing blocks is specified by functional resources of its PE such as adder, multiplier, shifter, logic operations etc. Therefore, it is necessary to examine how to select the functionalities of RCC and RAA. This leads to further studies on resource assignment/sharing between RCC and RAA.

First of all, we can classify the functional resources into two groups: primitive resources and critical resources. Primitive resources are basic functional units such as adder/subtractor and logical operators. Critical resources are area/delay-critical ones such as multiplier and divider. Based on the classification, let us consider two cases of the functional resource configurations as shown in Fig. 67. Fig. 67 (a) shows hierarchical functionality that indicates L1 PE array has primitive resources and L2 PE array includes critical resources as well as primitive resources. The Fig. 67 (b) shows identical functionalities both in the L1 and L2 PE arrays. In the case of (a), the RCC with L1 PE array is relatively lightweight computing block compared to the RAA with L2 PE array. Therefore, the RCC can perform small applications having only primitive operations with low power consumption. However, it causes ‘lack of resource’ problem when applications demand critical operations. In (b) L1 and L2 PE arrays have identical functionality with area and power overheads.

To prevent such extreme cases, we propose resource sharing for the RCC and the RAA based on [44]. L1 and L2 PE array have the same primitive resources and shared

the pipelined critical resources as shown in Fig. 68. Here the RCC and the RAA basically perform the primitive operations and their functionality will include the critical operations using the shared resources. Fig. 69 shows interconnection structure with shared critical resources along with RCC and RAA. PEs in the same row of the L1 and L2 array share the pipelined critical resources in the same manner as [44]. Such a structure avoids the 'lack of resource' problem in Fig. 67 (a) and this structure is more area and power-efficient than Fig. 67 (b) because the number of critical resources is reduced and the critical resources taken out of L1 and L2 PE array are not affected by unnecessary switching activity caused by other resources. In addition, interconnections for resource sharing can be also utilized for communication interface between the RCC and the RAA by adding multiplexer and de-multiplexer between front and end of the critical resources as shown in Fig. 69 (b).



(a) Hierarchical functionality (b) Identical functionality

Fig. 67. Two cases of functional resource assignment.

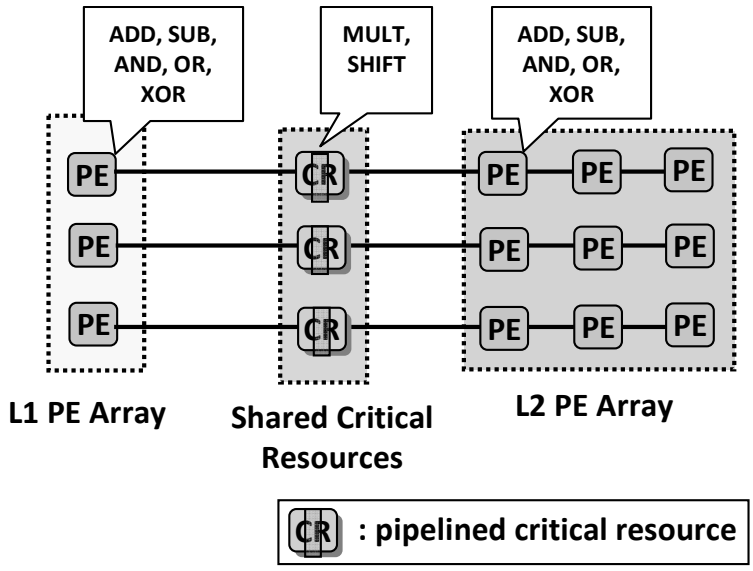
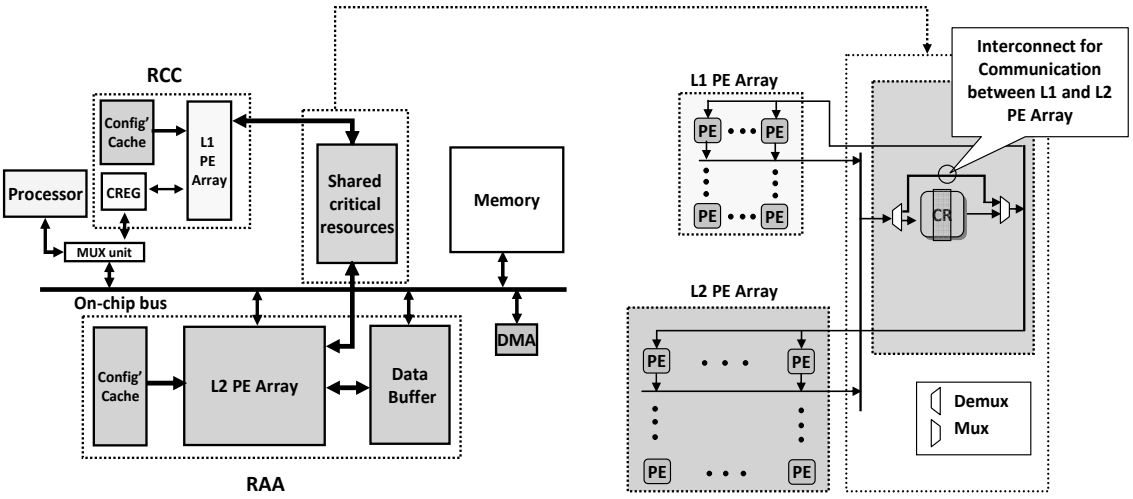


Fig. 68. Critical resource sharing and pipelining in L1 and L2 PE array.



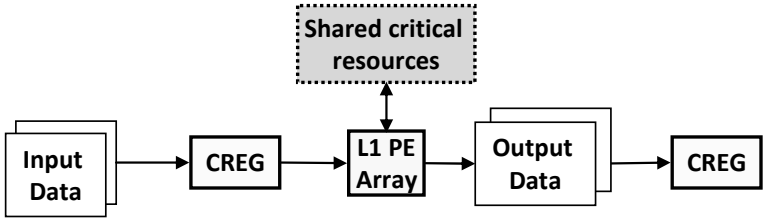
(a) Entire structure (b) Interconnection structure

Fig. 69. Interconnection structure among RCC, shared critical resources and L2 PE array.

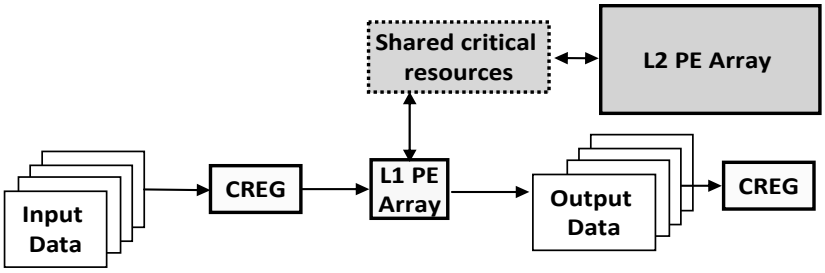
3. Computing Flow Optimization

Based on the proposed CGRA structure, we can classify four cases of optimized computing flow to achieve low power and high performance. Fig. 70 shows such four computing flows on the proposed CGRA according to variance of input and output size of applications – Subsection C.1.a shows that we can select the optimal case among the proposed computing flows for several applications with variance in their input/output size. All of the cases show that shared critical resources are used as needed because they are only utilized when applications have the operations requiring the critical resources.

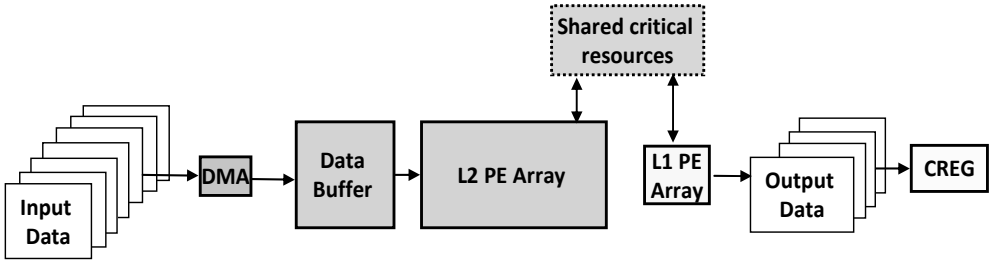
Fig. 70 (a) shows computing flow when application has the smallest inputs and outputs. In this case, only RCC functional units are used to execute the application while the RAA is disabled to reduce power consumption. However, if the application has larger inputs and outputs than (a), the computing flow can be extended to L2 PE array as shown in Fig. 70 (b). Even though L2 PE array is used for this case, data buffer of the RAA is not used because the coprocessor register-set (CREG) is sufficient to save the all of the inputs or outputs. The next case is that when RAA is used with RCC because of large inputs and small outputs as shown in Fig. 70 (c). In this case, data buffer of the RAA receives inputs using DMA which is more efficient for overall performance than CREG. This is because insufficient CREG resource for large inputs causes performance bottleneck with heavy registers-PE array traffic. Therefore, the L2 PE array may be used first for running such application and the L1 PE array can be utilized for enhancing parallelized execution as needed. However, the outputs are stored on CREG because their



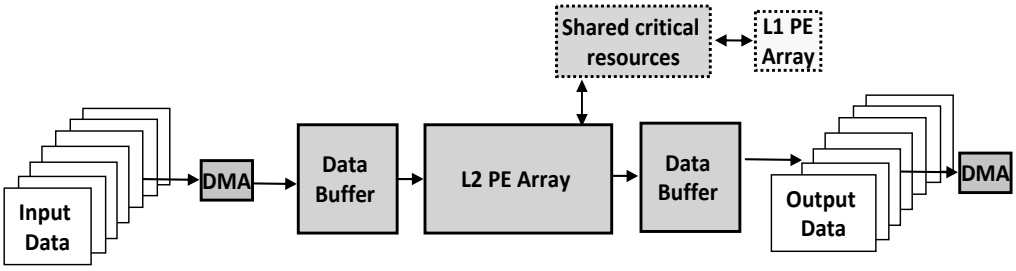
(a) Smallest inputs and outputs (STIO)



(b) Small inputs and outputs (SIO)



(c) Large inputs and small outputs (LISO)



(d) Large inputs and outputs (LIO)

Fig. 70. Four cases of computing flow according to the input/output size of application.

size is small. Finally, Fig. 70 (d) shows a case of RAA used with L1 PE array with large inputs and outputs. To avoid heavy registers-PE array traffic by the large input/output size, the data buffer with DMA is used and L1 PE array can be optionally utilized for enhancing parallelized execution.

In summary, the computing flow on the proposed CGRA can be adapted according to the input/output size of applications. It is more power-efficient than using a conventional CGRA by separated computing blocks with sharing critical resources. This way is only necessary computing blocks are utilized. In addition, computing flow with supporting two communication interfaces reduces power and enhances performance.

C. Experiments

1. Experimental Setup

a. Architecture Implementation

To demonstrate the effectiveness of the proposed RCC-based CGRA, we have designed three different organizations of CGRA with RT-level implementation using VHDL as shown in Table XIV.

Table XIV. Comparison of the Architecture Implementations

CGRA	PE array	Data storage
Attached IP	8x8 PE array	6KB data buffer
Coprocessor	8x8 PE array	512-byte coprocessor register-set
Proposed RCC-based	8x2 L1 PE array and 8x6 L2 PE array	4KB data butter and 512-byte coprocessor register set

(ARM7-compatible 32-bit RISC processor is used as main processor)

In addition, for resource sharing of RCC-based CGRA, two pipelined multipliers and two shifters are shared by PEs in the same row of L1 and L2 PE array whereas conventional two types of CGRA do not support such a resource sharing and pipelining.

The architectures have been synthesized using Design Compiler [49] with 0.18 ㎛ technology. PrimePower [49] has been used for gate-level simulation and power estimation. To obtain the power consumption data, we have used the applications in Table XV for simulation with operation frequency of 100 MHz and typical case of 1.8 V Vdd and 27℃.

Table XV. Applications Characteristics

Real Applications	SHR	Computing Flow	Benchmarks	SHR	Computing Flow
(H.263) 8x8 DCT	✓	SIO	*256-point FFT	✓	LISO
(H.263) 8x8 IDCT	✓	SIO	*256-tap FIR	✓	LISO
(H.263)8x8 QUANT	✓	SIO	*Complex Mult	✓	LISO
(H.263) 8x8 DEQUANT	✓	SIO	**State	✓	STIO
(H.263) SAD	-	LISO	**Hydro	✓	STIO
(H.264) 4x4 ITRANS	✓	STIO	**Tri-Diagonal	✓	LIO
(H.264) MSE	✓	LISO	**First-Diff	-	STIO
(H.264) MAE	-	LISO	**ICCG	✓	STIO
(H.264) 16x16 DCT	✓	LISO	**Inner Product	✓	LIO
8x8*8x1 Matrix-Vector Multiplication	✓	SIO	*: DSPstone benchmarks [71] **: Livermore loop benchmarks [70] SHR:'✓'means critical resources are used for the application. STIO: smallest inputs and outputs SIO: small inputs and outputs LISO: large inputs and small outputs LIO: large inputs and outputs		
16x16*16x1Matrix-Vector Multiplication	✓	LISO			
8x8 Matrix Multiplication	✓	SIO			
16x16 Matrix Multiplication	✓	LISO			

b. Evaluated Applications

Evaluated applications are composed of real multimedia applications and benchmarks. We have analyzed the input/output size and operation-types in the applications to identify specific computing flow in Fig. 70. Table XV shows the selected applications and the optimal computing flows for them.

Table XVI. Area Cost Comparison

PE Array	No' of PEs	No' of MULTs	No' of SHTs	Gate Equivalent			Reduction (%)
				Interconnect	Logic	Total	
Base 8x8	64	64	64	164908	494726	659635	-
Proposed	64	16	16	175434	334595	510029	22.68

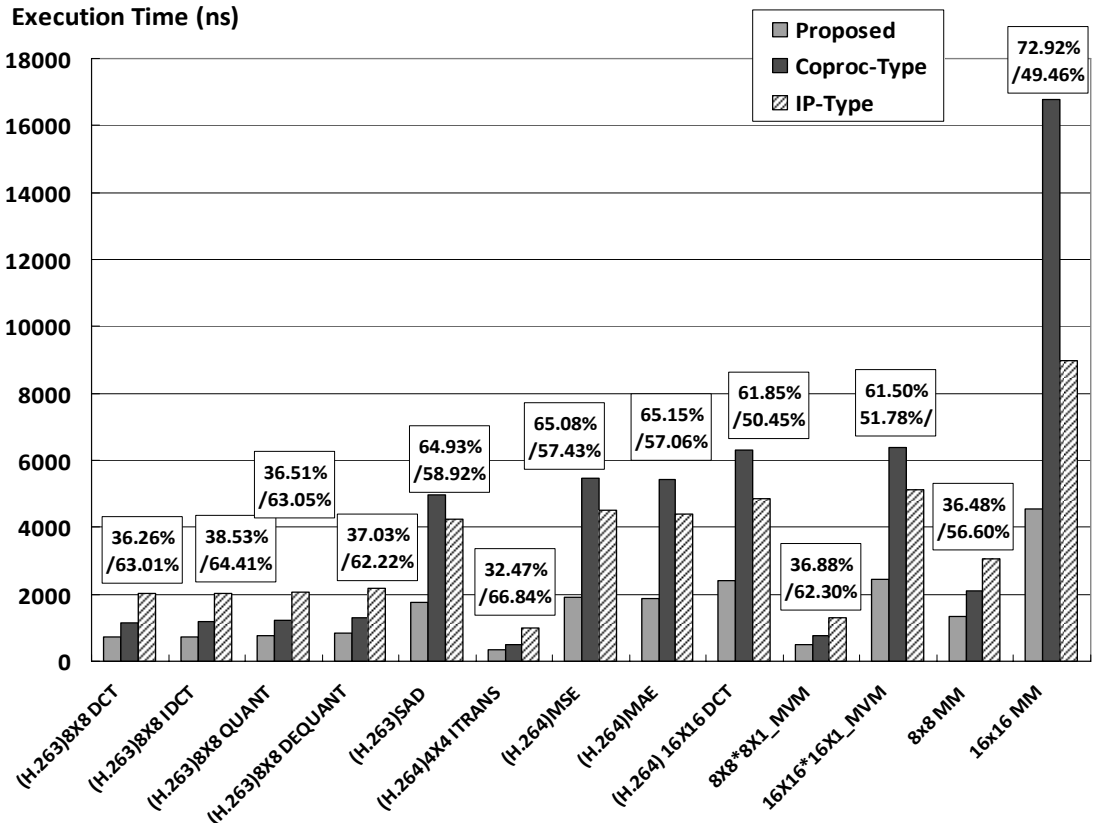
2. Results

a. Area Cost Evaluation

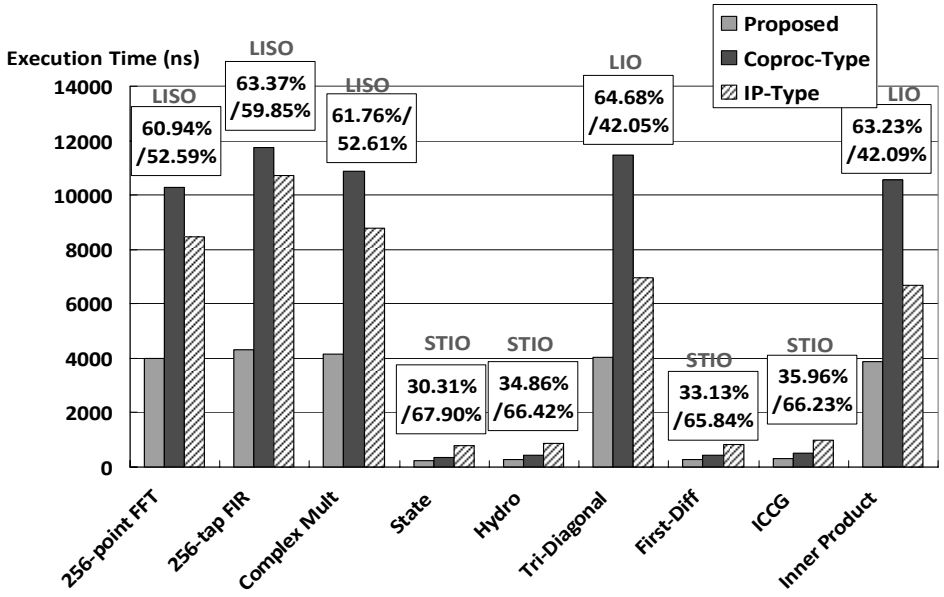
Table XVI shows area cost evaluation for the two cases. 'Base 8x8' means 8x8 PE array included in 'Attached IP' and 'Coprocessor' type CGRA. 'Proposed' means L1 and L2 PE array included in the proposed RCC-based CGRA. Even though interconnection area of the proposed model increases because of resource sharing structure, entire area of the proposed one is reduced by 22.68% because it has less critical resources than base 8x8 PE array.

b. Performance Evaluation

The synthesis results show that the proposed PE array has reduced critical path delay (5.12 ns) compared to the base PE array (8.96 ns). This is because pipelined multipliers are excluded from the original set of critical paths. Based on the synthesis results, we



(a) Real applications



(b) Benchmarks

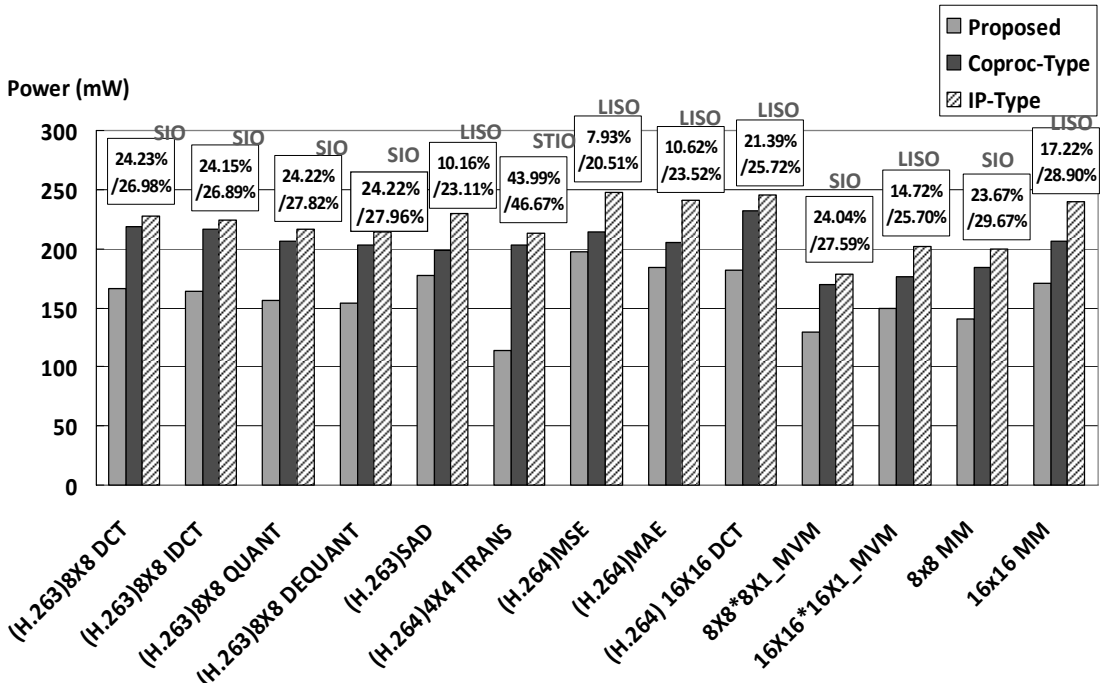
A%/B%: A% means reduced execution time ratio compared with Coproc-Type and B% means reduced execution time ratio compared with IP-Type.

Fig. 71. Performance comparison.

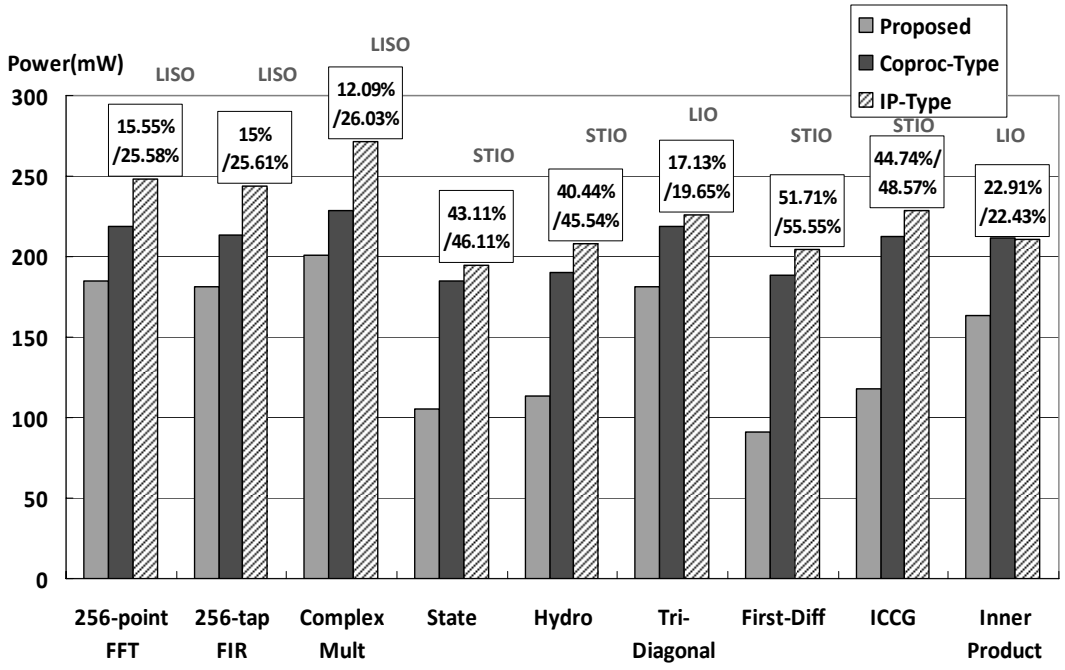
evaluate execution times of the selected applications on three cases of CGRA as shown in Fig. 71. The execution times include communication time between memory/processor and the RAA or RCC. Each application is executed on the RCC-based CGRA in the manner of selected computing flow as shown in Table XV – all of the applications are classified under 4 cases of computing flow (STIO, SIO, LISO and LIO). In the case of STIO and SIO, performance improvement compared with 'Coprocesssor' type is relatively less (30.31%~37.03%) than LIO and LISO (60.94%~72.92%). This is because the improvements of STIO and SIO are achieved by only reduced critical path delay whereas the improvements of LIO or LISO are achieved by avoiding heavy coprocessor registers-PE array traffic as well as reduced critical path delay. However, compared with 'Attached-IP' type, STIO and SIO achieve much more performance improvement (56.60%~67.90%) whereas LISO and LIO show the improvement of (42.05%~59.85%). This is because STIO and SIO do not use data buffer of the RAA causing communication overhead on system bus.

c. Power Evaluation

Fig. 72 shows the comparison of power consumptions in three different organizations of CGRA. First of all, the proposed L1 and L2 PE array is more power-efficient than the base PE array because of the reduced critical resources. With such a power-efficient PE array, the amount of power saving depends on the selected computing flow for the application. The most power-efficient computing flow is STIO that shows relatively much power saving (40.44%~55.55%) compared to other cases (7.93%~29.67%) because the STIO does not use the RAA - specially, 'First_Diff' shows the highest power saving



(a) Real applications



(b) Benchmarks

A%/B%: A% means power saving ratio compared with Coproc-Type, and B% means power saving ratio compared with IP-Type.

Fig. 72. Power comparison.

ratio of 51.71%/55.55% because of not using the shared critical resources. The next power-efficient model is SIO showing power saving (23.67%~29.67%). This is because the SIO computing flow does not use data buffer of the RAA whereas LISO (7.93%~26.03%) and LIO (17.13%~22.91%) utilizes the data buffer for input data or output data. Finally, power saving of LISO and LIO is mostly achieved by reduced critical resources and by not activating L1 PE array.

CHAPTER IX

INTEGRATED APPROACH TO OPTIMIZE CGRA

In this chapter, we present integrated approach to merge the multiple design schemes presented in the previous chapters. A case study is shown to verify the synergy effect of combining the multiple design schemes. Experimental results show that the integrated approach reduces area by 23.07% of entire RAA and power by up to 72% when compared with the conventional RAA. In addition, we discuss potential combinations among the proposed design schemes and their expected outcomes.

A. Combination among the Cost-Effective CGRA Design Schemes

From Chapter VI to Chapter VIII, we have proposed the cost-effective CGRA design schemes and such schemes can be combined with each other to optimize CGRA in terms of area, power and performance. Fig. 73 shows combination flow of the proposed design schemes. The flow shows possible scheme combinations for CGRA design. Each arrow of the flow shows a possible integration between two design schemes. The possible scheme combinations can be found by tracing in the arrow directions. The combination flow can be classified into two cases according to the computation model of CGRA. In the case of temporal mapping, *low power reconfiguration technique by reusable context pipelining* (Chapter IV) can be selected whereas *cost-effective array fabric* (Chapter VII) is applicable to the spatial mapping. This is because two design schemes have been devised while keeping the characteristics of spatial mapping and temporal mapping - we

spatially spread the operations in the data flows over the array space in the design scheme of the *cost-effective array fabric* whereas *reusable context pipelining* spread the operations over time for each column to implement temporal loop pipelining. Therefore, even though two design schemes cannot be merged, any combination of a design scheme in Chapter IV or VII with the remaining three schemes is possible.

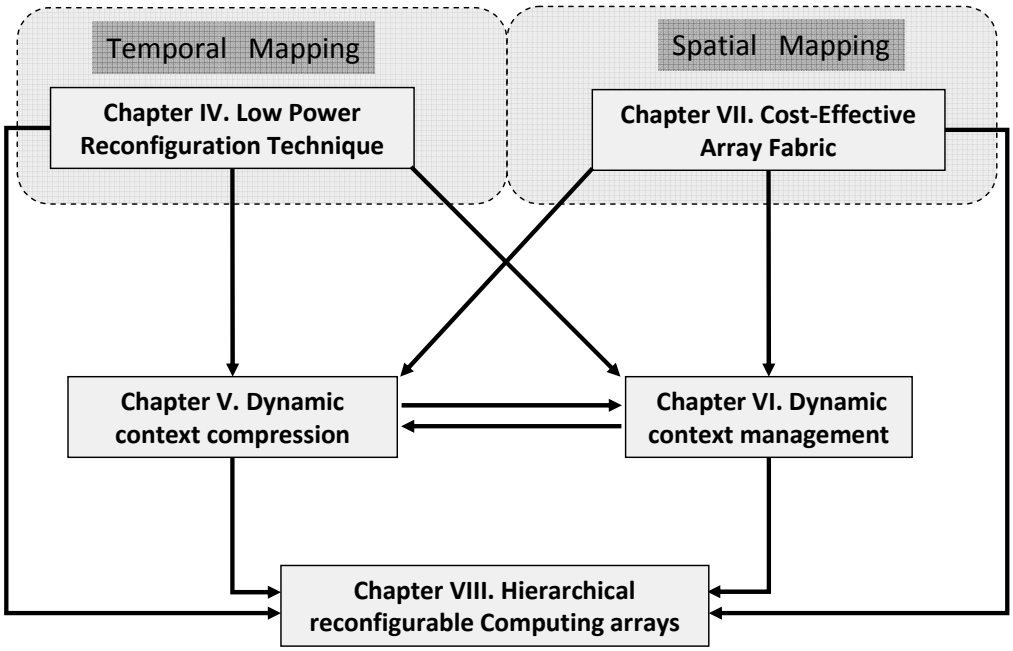


Fig. 73. Combination flow of the proposed design schemes.

B. Case Study for Integrated Approach

1. An CGRA Design Example Merging Three Design Schemes

To demonstrate the effectiveness of the integrated approach, we have designed a RAA combining three design schemes as shown in Fig. 74 with RT-level implementation using VHDL. The architectures have been synthesized using Design Compiler [49] with

0.18 ㎛ technology. PrimePower [49] has been used for gate-level simulation and power estimation. To obtain the power consumption data, we have used the same the applications shown in the previous Chapters for simulation with operation frequency of 100 MHz and typical case of 1.8 V Vdd and 27℃.

Fig. 74. A combination example combining three design schemes.

2. Results

a. Area and Performance Evaluation

Table XVII shows area cost evaluation of each component for the base RAA as specified in Chapter III and the integrated RAA combining three design schemes. In the case of configuration cache, area is reduced by 16.79% - even though *dynamic context compression* increases area as shown in Chapter V, *low power reconfiguration technique* offsets the increased area with reduced size of the configuration cache. Area of the PE array and frame buffer are also reduced by 17.27%/30% because *hierarchical reconfigurable computing arrays* supports critical resource sharing with the reduced size of the frame buffer. Therefore, the area reduction ratio of the entire RAA is 23.07% compared to the base RAA.

The synthesis results show that the integrated RAA has reduced critical path delay (5.12 ns) compared to the base RAA (8.96 ns). This is because *dynamic context management* and *low power reconfiguration technique* don't affect the original critical path

delay and pipelined multipliers are excluded from the original set of critical paths by *hierarchical reconfigurable computing arrays*. In addition, execution time evaluation of the applications shows the same results in Chapter VIII – performance enhancement of 42.05%~67.90% compared with the IP-type base RAA.

Table XVII. Area Reduction Ratio by Integrated RAA

Component	Gate Equivalent		Reduction (%)
	Base	Integrated	
Configuration Cache	150012	124824	16.79
PE Array	659635	510029	22.68
Frame Buffer	129086	90329	30.00
Entire RAA	942742	760869	**23.07**

b. Power Evaluation

To verify the synergy effect of the integrated approach, we have evaluated power consumption for the five cases:

a. Base RAA

b. RAA with *low power reconfiguration technique*

c. RAA with *dynamic context compression*

d. RAA with *hierarchical reconfigurable computing array*

e. integrated RAA.

Table XVIII shows entire power comparison among the five cases. Each design scheme (*b*, *c* and *d*) does not reduce much power of entire RAA – 26.54%~47.6% in *b*, 13.77%~21.48% in *c* and 11.09%~30.19% in *d*. However, the integrated RAA save much power (44.65% ~ 71.29%) because each component of the RAA is optimized by

the individual design scheme.

Table XVIII. Entire Power Comparison

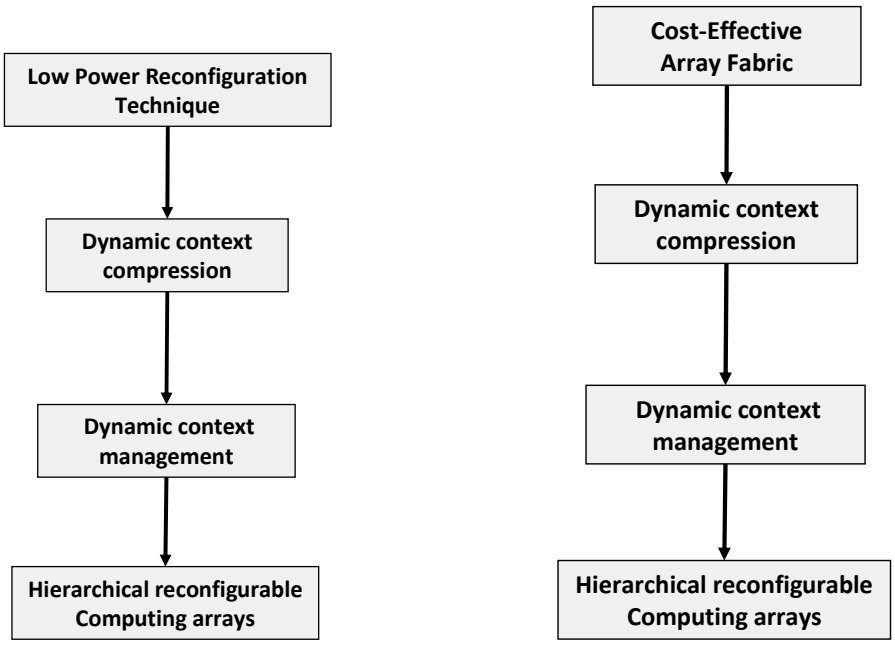
kernels	[a]Base	[b]Low Power Reconfig'		[c]Dynamic context compression		[d]Hierarchical Reconfig' Array		[e]Integrated	
	[f]P(mW)	[f]P(mW)	[g]R(%)	[f]P(mW)	[g]R(%)	[f]P(mW)	[g]R(%)	[f]P(mW)	[g]R(%)
First_Diff	376.17	232.48	38.2	309.37	17.76	262.62	**30.19**	108.01	**71.29**
Tri- Diagonal	400.19	257.59	35.63	331.01	17.29	355.79	11.09	200.65	49.86
State	356.08	228.45	35.84	294.23	17.37	266.23	25.23	125.71	64.7
Hydro	356.47	240.64	32.49	299.74	15.91	261.64	26.6	133.41	62.57
ICCG	434.45	261.29	39.86	354.33	18.44	323.39	25.56	137.52	68.35
Inner Product	328.54	240.57	26.78	283.3	13.77	281.27	14.39	181.83	44.65
24-Taps FIR	471.44	274.99	41.67	383.44	18.67	408.98	13.25	200.5	57.47
Matrix-vector multiplication	405.7	212.58	**47.6**	318.56	**21.48**	356.56	12.11	150.25	62.97
Mult in FFT	423.59	287.67	32.09	355.19	16.15	360.12	14.98	208.78	50.71
Comlex Mult in AAC decoder	452	304.19	32.7	381.55	15.59	381.38	15.62	220.77	51.16
ITRANS in H.264 decoder	417.95	283.06	32.27	338.37	19.04	318.49	23.8	156.42	62.57
DCT in H.263 encoder	417.33	264.89	36.53	347.17	16.81	356	14.7	189.68	54.55
IDCT in H.263 encoder	412.91	263.45	36.2	343.42	16.83	352.55	14.62	188.71	54.3
SAD in H.263 encoder	415.27	305.05	26.54	343.04	17.39	362.12	12.8	222.63	46.39
Quant in H.263 encoder	401.35	255.77	36.27	333.63	16.87	341.22	14.98	181.14	54.87
Dequant in H.263 encoder	401.64	252.3	37.18	332.63	17.18	341.85	14.89	178.38	55.59

[a]Base RAA (configuration cache + frame buffer + PE array), [b]RAA with low power reconfiguration technique, [c]RAA with dynamic context compression , [d]RAA with hierarchical reconfigurable computing array, [e]RAA combining three scheme, [f] Power Consumption of RAA, [g] Power reduction ratio of entire RAA compared with BASE.

C. Potential Combinations and Expected Outcomes

As mentioned in Section A, any combination of a design scheme limited by the computation model with the remaining four schemes is possible and we can consider two cases of the maximum combinations – one is the maximum power optimization for the con-

figuration cache and another is area/power optimization of the PE array. Fig. 75 shows such two cases of combinations. In the case of Fig. 75 (a), all of the design schemes reducing power in configuration cache are merged with *hierarchical reconfigurable computing arrays*. Therefore, power saving of the configuration cache can be optimized based on the computation model of the temporal mapping. The second case is area/power optimization of the PE array as shown in Fig. 75 (b). Compared with (a), instead of *low power reconfiguration technique*, the design scheme of *cost-effective array fabric* is combined with other design schemes. In this case, the area/power of the PE array can be optimized by reducing the number of PEs (*cost-effective array fabric*) and sharing critical-resource (*hierarchical reconfigurable computing arrays*).



(a) Power optimization for the configuration cache

(b) Area/power optimization of the PE array

Fig. 75. Potential combination of multiple design schemes.

CHAPTER X

CONCLUSIONS

In this chapter, we summarize the major results of this dissertation.

In Chapter IV, we propose reusable context pipelining for low power reconfiguration and hybrid configuration cache structure supporting this technique. Our architecture can be used to achieve power-savings in a reconfigurable architecture while maintaining performance same as general CGRA. In addition, new configuration cache structure is more efficient than previous one in terms of memory size. In the experiments, we show that the proposed approach saves power even with reduced configuration cache size. Power reduction ratios in the configuration cache and the entire architecture are up to 86.33% and 47.60% respectively compared to the base architecture.

In Chapter V, we introduce new context architecture (*dynamically compressible context architecture*) with its design flow and configuration cache structure to support it. The proposed dynamically compressible context architecture can save power in configuration cache without performance degradation. Experimental results show that our approach saves much power compared to conventional base model with negligible area overhead. We have reduced the power by up to 39.72% in configuration cache.

In Chapter VI, we propose novel *dynamic context management* for low power CGRA and new configuration cache structure supporting this technique. Te proposed management method can be used to achieve power-savings in configuration ache while maintaining performance same as general CGRA. In the experiments, we show that our

approach saves much power compared to conventional base model with negligible area overhead. We have reduced the power by 38.24%/38/15% in write/read operation of configuration cache.

In Chapter VII, we propose a novel reconfigurable array fabric optimized for computation-intensive and data-parallel applications. It has been shown the new array fabric is derived from a standard square-array using the proposed exploration flow. The exploration flow efficiently rearranges PEs with reducing array size and change interconnection scheme to save area and power. In addition, we suggest the new array fabric which splits the computational resources into two groups (primitive resources and critical resources). Critical resources can be area-critical and/or delay-critical. Primitive resources are replicated for each processing element of the reconfigurable array, whereas area-critical resources are shared among multiple basic PEs. Delay-critical resources can be pipelined to curtail the overall critical path so as to increase the system clock frequency. Experimental results show that the proposed approaches saves significant area and power compared to conventional base model with enhancing performance. Implementation of sixteen kernels on the new array structure demonstrates consistent results. The area reduction up to 36.75%, the performance enhancement up to 42.86% and the power savings up to 35.45% are evident when compared with the conventional array architecture.

In Chapter VIII, we propose *hierarchical reconfigurable computing array architecture* to reduce power/area and enhance performance in configurable embedded system. The CGRA-based embedded systems that consist of hierarchical configurable computing

arrays with varying size and communication speed were examined for multimedia and other applications. Experimental results show that the proposed approach reduces on-chip area by 22%, execution time by up to 72% and reduces power consumption by up to 55% when compared with the conventional CGRA-based architectures.

In Chapter IX, we present integrated approach to merge the multiple design schemes. A case study is shown to verify the synergy effect of combining the multiple design schemes. Experimental results show that the integrated approach reduces area by 23.07% of entire RAA and power by up to 72% when compared with the conventional RAA.

REFERENCES

[1] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in *Proc. of Design Automation and Test in Europe Conf.*, pp. 642-649, March 2001.

[2] F. Barat, M. Jayapala, T. Vander A. Corporaal, G. Deconinck, and R. Lauwereins, “Low power coarse-grained reconfigurable instruction set processor,” in *Proc. of Int. Conf. on Field Programmable Logic and Applications*, pp. 230-239, September 2003.

[3] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho, “MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications,” *IEEE Trans. on Computers*, vol. 49, no. 5, pp. 465-481, May 2000.

[4] T. Miyamori and K. Olukotun, “A quantitative analysis of reconfigurable coprocessors for multimedia applications,” in *Proc. of IEEE Symp. on FPGAs for Custom Computing Machines,* pp 15-17, April 1998.

[5] C. Ebeling, D. Cronquist, and P. Franklin, “Configurable computing: The catalyst for high-performance architectures,” in *Proc. of IEEE Int. Conf. Appl.-Specific Syst., Arch., Process*., pp. 364–372, July 1997.

[6] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology," in *Proc. of IEEE Custom Integrated Circuits Conf.*, pp 63 –66, May 2002.

[7] Y. Chou, P. Pillai, H. Schmit, and J. Shen, "PipeRench implementation of the instruction path coprocessor," in *Proc. of Annual IEEE/ACM Int. Symp. on Microarchitecture*, pp 147-158, December 2000.

[8] F. Bouwens, M. Berekovic, A. Kanstein and G. Gaydadjiev, "Architectural exploration of the ADRES coarse-grained reconfigurable array," in *Proc. of Int.Workshop on Applied Reconfigurable Computing*, pp. 1-13, March 2007.

[9] F. Hanning, H. Dutta, and J. Teich, “Regular mapping for coarse-grained reconfigurable architectures,” *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing,* pp. 57-60, May 2004.

[10] J. Becker and M. Vorbach, “Architecture, memory and interface technology integration of an indutrial/academic configurable system-on-chip (CSoC),” in *Proc. of IEEE Computer Society Annual Symp. on VLSI*, pp. 107-112, February 2003.

[11] Y. Kim, J. Lee, J. Junng, S. Kang, and K. Choi, "Design of coarse-grained reconfigurable hardware," in *Proc. of IEEK SoC Design Conf.*, pp. 312-317, April 2004.

[12] Y. Kim, C. Park, S. Kang, H. Song, J. Jung, and K. Choi, “Design and evaluation of coarse-grained reconfigurable architecture,” in *Proc. of Int. SoC Design Conf.*, pp. 227-230, October 2004.

[13] N. Suzuki, S. Kurotaki, M. Suzuki, N. Kaneko, Y. Yamada, K. Deguchi, Y. Hasegawa, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima, “Implementing and evaluating stream applications on the dynamically reconfigurable processor,” in *Proc. of Field-Programmable Custom Computing Machines,* pp. 328-329, April 2004.

[14] S. Khawam, T. Arslan, and F. Westall, “Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications,” in *Proc. of IEEE Int. Parallel & Distributed Processing Symp.*, pp. 150-157, April 2004.

[15] A. Deledda, C. Mucci, A.Vitkovski, M. Kuehnle, F. Ries, M. Huebner, J. Becker, P. Bonnot, A. Grasset, P. Millet, M. Coppola, L. Pieralisi, R. Locatelli, and G. Maruccia,, "Design of a HW/SW communication infrastructure for a heterogeneous reconfigurable processor," in *Proc. of Design, Automation, and Test in Europe Conf.*, pp.1352-1357, March 2008.

[16] M. Galanis and C. Goutis, "Speedups from extending embedded processors with a high-performance coarse-grained reconfigurable data-path," *Journal of Systems Architecture - Embedded Systems Design,* vol. 50, no. 2, pp. 479-490, February, 2008

[17] G. Rauwerda, P. Heysters, and G. Smit, "Towards software defined radios using coarse-grained reconfigurable hardware," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 16, no. 1, pp. 3-13, January 2008.

[18] M. Myjak, and J. Delgado-Frias, "A medium-grain reconfigurable architecture for DSP: VLSI design, benchmark mapping, and performance," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 16, no. 1, pp. 14-23, January 2008.

[19] A. Poon, "An energy-efficient reconfigurable baseband processor for wireless communications," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 15, no. 3, pp. 319-327, March 2007.

[20] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures," in *Proc. of Asia and South Pacific Design Automation Conf.*, pp. 163-168, January 2000.

[21] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, “Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study,” in *Proc. of Design Automation and Test in Europe Conf.*, pp. 1224-1229, March 2004.

[22] N. Bansal, S. Gupta, N. Dutt, and A. Nicolau, “Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations,” presented at the Workshop on Application Specific Processors, San Diego, CA, December 2003.

[23] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, “Interconnect-aware mapping of applications to coarse-grain reconfigurable architectures,” in *Proc. of Int. Conf. on Field Programmable Logic and Applications*, pp. 891-899, August 2004.

[24] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, “Network topology exploration of mesh-based coarse-grain reconfigurable architectures,” in *Proc. of Design Automation and Test in Europe Conf.*, pp. 474-479, February 2004.

[25] J. Lee, K. Choi, and N. Dutt, "Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures," in *Proc, of IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors*, pp.166-176, June 2003.

[26] J. Lee, K. Choi, and N. Dutt, "Design space exploration of reconfigurable ALU array (RAA) architectures," in *Proc. of IEEE SOC Design Conf.* pp. 302-307, November 2003.

[27] J. Lee, K. Choi, and N. Dutt, "Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures," *Int. Journal of Embedded Systems,* vol. 3 no. 3, pp.119-127, October 2008.

[28] Y. Kim, M. Kiemb, and K. Choi, "Efficient design space exploration for domain-specific optimization of coarse-grained reconfigurable architecture," in *Proc. of IEEK SoC Design Conf.*, pp. 19-24, May 2005.

[29] A. Lambrechts, P. Raghavan, and M. Jayapala, “Energy-aware interconnect-exploration of coarse-grained reconfigurable processors,” presented at the Workshop on Application Specific Processors, New York, September 2005.

[30] H. Zhang, M. Wan, V. George, and J. Rabaey, “Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs,” in *Proc. of VLSI’ 99*, April 1999.

[31] F. Hannig, H. Dutta, and J. Teich, “Mapping of regular nested loop programs to coarse-grained reconfigurable arrays – Constraints and methodology,” in *Proc. of EEE Int. Parallel & Distributed Processing Symp.*, pp. 148-155, April 2004.

[32] J. Lee, K. Choi, and N. Dutt, “Mapping loops on coarse-grained reconfigurable architectures using memory operation sharing,” Center for Embedded Computer Systems (CECS), University of California, Irvine, Tech. Rep. 02-34, 2002.

[33] J. Lee, K. Choi, N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Design & Test of Computers*, vol. 20 no. 1, pp.26-33, January 2003.

[34] J. Lee, K. Choi, and N. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures", in *Proc. of ACM Workshop on Languages, Compilers, Tools for Embedded Systems*, pp.183-188, June 2003

[35] J. Lee, K. Choi, and N. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," *ACM Sigplan Notices*, vol. 38 no. 7 pp.183-188, July. 2003

[36] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proc. of Design Automation and Test in Europe Conf.*, pp. 262-268, March 2006.

[37] J. Yoon, Y. Kim, M. Ahn, Y. Paek, and K. Choi, "Temporal mapping for loop pipelining on a MIMD style coarse-grained reconfigurable architecture," presented at the IEEE Int. SoC Design Conf., Seoul, Korea, October 2006.

[38] G. Lee, S. Lee, and K. Choi, "Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques," in *Proc. of IEEE Int. SoC Design Conf.*, pp.395-398, September 2008.

[39] J. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proc. of Asia and South Pacific Design Automation Conf.*, pp. 776-782, March 2008.

[40] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proc. of Int.*

Conf. on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 136-146, October 2006.

[41] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. Kim, “Edge-centric modulo scheduling for coarse-grained reconfigurable architectures,” in *Proc. of 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 166-176, October 2008.

[42] G. Dimitroulakos, N. Kostaras, M. Galanis, and C. Goutis, “Compiler assisted architectural exploration for coarse grained reconfigurable arrays,” in *Proc. of Great Lakes Symp. on VLSI*, pp.164-167, March 2007.

[43] F. Vererdas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes,” in *Proc. of Int. Conf. on Field Programmable Logic and Applications,* pp. 106-111, August 2005.

[44] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, “Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization,” in *Proc. of Design Automation and Test in Europe Conf.*, pp. 12-17, March 2005.

[45] C. Park, Y. Kim, and K. Choi, "Domain-specific optimization of reconfigurable array architecture," presented at the US-Korea Conference on Science, Technology, & Entrepreneurship, Irvine, CA, August 2005.

[46] M. Lanuzza, M. Margala, and P. Corsonello, “Cost-effective low-power processor-in-memory-based reconfigurable datapath for multimedia applications,” in *Proc. of Int. Symp. on Low Power Electronics and Design,* pp. 161-166, August 2005.

[47] F. Barat and R.Lauwereins, “Reconfigurable instruction set processors: A survey,” in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 168-173, April 2000.

[48] ARM Corp., Cambridge, U.K., “ARM Corp. home page,” 2002. [Online]. Available: http://www.arm.com/arm/AMBA

[49] Synopsys Corp., Mountain View, CA, “Synopsys Corp. home page,” 2005. [Online]. Available: http://www.synopsys.com

[50] Model Technology Corp., Wilsonville, OR, “Model Technology Corp. home page,” 2005. [Online]. Available: http://www.model.com

[51] Y. Kim and R. Mahapatra, “Reusable context pipelining for low power coarse-grained reconfigurable architecture,” in *Proc. of Int. Parallel & Distributed Processing Symp.,* pp. 1-8, April, 2008.

[52] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Proc. of Int. Symp. on Low Power Electronics and Design*, pp. 310-315, October 2006.

[53] I. Park, Y. Kim, C. Park, J. Son, M. Jo, and K. Choi, "Chip implementation of a coarse-grained reconfigurable architecture," in *Proc. of IEEE Int. SoC Design Conf.,* pp. 628-629, October 2006.

[54] I. Park, Y. Kim, M. Jo, and K. Choi, "Chip implementation of power conscious configuration cache for coarse-grained reconfigurable architecture," in *Proc. of the 15th Korean Conf. on Semiconductors*, pp.527-528, February 2008.

[55] J. Cocke, “Global common sub expression elimination,” in *Proc. of Symposium on Compiler Construction, ACM SIGPLAN Notices 5*, pp 850-856, July 1970.

[56] S. Keutzer, S. Tjiang, and S. Devadas, “A new viewpoint on code generation for directed acyclic graphs,” *ACM Transactions on Design Automation of Electronic Systems* vol. 3, no. 1, pp 51-75, January 1998.

[57] R. Rau, “Iterative modulo scheduling,” *Technical Report, Hewlett-Packard Lab*: HPL-94-115, 1995.

[58] Netlib Repository at the Oak Ridge National Laboratory, Oak Ridge, TN. [Online]. Available: http://www.netlib.org/benchmark/livermorec

[59] Institute for Integrated Signal Processing Systems, Aachen, Germany. [Online]. Available: http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE

[60] Y. Kim and R. Mahapatra, “Dynamically compressible context architecture for low power coarse-grained reconfigurable array,” in *Proc. of Int. Conf. on Computer Design*, pp. 295-400, October 2007.

[61] Y. Kim and R. Mahapatra, “Dynamic context management for low power coarse-grained reconfigurable architecture,” presented at the ACM Great Lake Symp. on VLSI, Boston, MA, May 2009.

[62] Y. Kim and R. Mahapatra, “A new array fabric for coarse-grained reconfigurable architecture,” in *Proc. of EuroMicro Conf. on Digital System Design*, pp. 584-591, September 2008.

[63] Y. Kim, and R. Mahapatra, “Hierarchical reconfigurable computing arrays for efficient CGRA-based embedded systems,” presented at the Design Automation Conf., San Francisco, CA, July 2009.

[64] M. Jo, V. Arava, H. Yang, and K. Choi, "Implementation of floating-point operations for 3D graphics on a coarse-grained reconfigurable architecture, " in *Proc. of IEEE Int. SoC Conf.*, pp.127-130, September 2007.

[65] M. Galanis, G. Dimitroulakos, S. Tragoudas, and C. Goutis, “Speedups in embedded systems with a high-performance coprocessor datapath,” *ACM Transactions on Design Automation of Electronic Systems*, vol.12, no 35, pp. 1-22, August 2007.

[66] T. Callahan, J. Hauser, and J. Wawrzynek, “The Garp architecture and C compiler”, *IEEE Computer*, vol. 33, no. 4, pp. 62-69, April 2000.

[67] C. Arbelo1, A. Kanstein, S. López1, J.F. López1, M. Berekovic, R. Sarmiento1 and J.-Y. Mignolet, “Mapping control-intensive video kernels onto a coarse-grain reconfigurable architecture: the H.264/AVC deblocking filter,” in *Design Automation and Test in Europe Conf.*, pp. 642-649, March 2007.

[68] M. Galanis, G. Dimitroulakos, and C. Goutis, “Speedups and energy savings of microprocessor platforms with a coarse-grained reconfigurable data-path,” in *Proc. of Int. Parallel & Distributed Processing Symp.,* pp. 1-8, March, 2007.

VITA

Yoonjin Kim received the B.S. degree in information and communication engineering from SungKyunKwan University, Suwon, Korea, in 2003, and the M.S. degree in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 2005. He graduated with the Ph.D. in computer engineering at Texas A&M University May 2009. His research interests are system-on-chip design, embedded systems, and reconfigurable computing. He may be contacted at:

Yoonjin Kim

Department of Computer Science and Engineering

Texas A&M University

TAMU 3112

College Station, TX 77843-3112

U.S.A.