Storage Management for Embedded SIMD Processors

A Dissertation
Presented to
The Academic Faculty

by
Soojung Ryu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2003

Copyright © 2003 by Soojung Ryu

Storage Management for Embedded SIMD Processors

Approved:

Dr. D. Scott Wills, Chair

Dr. Linda M. Wills, Chair

Dr. Douglas M. Blough

Dr. Sudhakar Yalamanchili

Date Approved December 15, 2003

Dedication

This work is dedicated to

my husband, Ji-Weon Jeong for his love and patience

my son, Daniel for his future

my parents, Keun-Jang Ryu and Sun-Sook Park for their love

my father-in-law, Han-Young Jeong for his love

my sister and brother, Hye-Jung and Jae-woan for their love

and my friends for their love

Acknowledgments

I would like to express my deepest appreciation to my thesis advisor, Dr. Scott Wills and my co-advisor, Dr. Linda Wills. They have contributed the most to my research. During my Ph.D. program at Georgia Institute of Technology, they provided me advice on everything – research, family, and life. Their encouragement made me continue to pursue my Ph. D. I really appreciate their love and also I would like to express my sincere love for them and their lovely children, Rosemary and Frank.

I am also grateful to Dr. Sudhakar Yalamanchili for serving as a thesis reading committee member. I would like to thank Dr. Douglas Blough for his encouragement for my research as well as careful reading of this dissertation. I would like to thank Dr. Bonnie Heck and Dr. Ellen Zegura for serving as committee of my dissertation defense.

I would like to acknowledge all of the members of the PICA group and EASL group at Georgia Institute of Technology for their friendship and useful comments – Mark, Keh, Jong-Myun, Lewis, Hong-Kyu, Jin-Sung, Cameron, Chris, Brett, Cory, Krit, Murat, Roy, Nidhi and Peter. Many thanks are given to Dr. Mondira Deb Pant for her support and friendship. I also would like to thank Dr. Kee-Shik Chung for his valuable advice on my Ph.D. research.

I'd like to also thank to my friends, Chung-Seok Seo, Jiwon Park, Joanne Kim, Seungmin Lee, Hyun-Ah Kim, Hasup Lee, So-Young You, Hyung-Su Lee, Jeongseok Ha, Jungwon Kang, and Jaehong Kim. I would like to give my special appreciation to Ji-Eun Park's family for their love and friendship.

Last, I'd like to express my deepest appreciation to my parents, family, husband, and my son who have continued to support me with love to finish this thesis.

Table of Contents

Dedication iii

Acknowledgments iv

Table of Contents v

List of Tables viii

List of Figures ix

Summary xiii

CHAPTER 1 Introduction 1
1.1. Introduction 1
1.2. Problem Statement and Contributions 1
1.3. Related Research 3
1.3.1. SIMD Instruction Broadcast 3
1.3.2. Systolic Arrays 3
1.3.3. Instruction Systolic Architecture (ISA) 5
1.3.4. Compilation Techniques for Storage Optimization 7
1.3.5. Summary of Related Research 9
1.4. Thesis Contribution Summary 10
1.5. Thesis Outline 12

CHAPTER 2 Efficient Storage Usage in Embedded SIMD Systems 13
2.1. Summary 13
2.2. Introduction 13
2.3. Related Work 14
2.3.1. Application Retargeting 14
2.3.2. Register Allocation 14
2.3.3. Memory Optimization 16
2.4. Approach: Application Retargeting for Different Memory Configurations 17
2.4.1. Register Allocation 18
2.4.2. Memory Optimization 21
2.5. Energy and Area Estimation for Storage 24
2.6. Validation and Evaluation 25
2.7. Metrics and Analysis 26
2.8. Results and Analyses 28
2.8.1. Example Application: Vector Quantization (VQ) Encoding 28
2.8.2. Experimental Results 30
2.9. Chapter Conclusion 36

CHAPTER 3 Systolic Instruction Broadcast for Embedded SIMD Architectures 37

3.1. Summary 37
3.2. Introduction 37
3.3. Related Work 39
3.3.1. Pipelined Instruction Broadcast 39
3.3.2. Instruction Systolic Architecture (ISA) 40
3.3.3. Systolic Instruction Broadcast 41
3.4. Approach: Systolic Instruction Broadcast Architecture 43
3.4.1. Data and Structural Hazards 44
3.4.1.1. Nearest Neighbor Communication 44
3.4.1.2. Data and Structural Hazard Analyses 45
3.4.2. Implementation of Systolic Instruction Broadcast 47
3.4.2.1. Software Approach 47
3.4.2.2. Two Write-port Register File Method 48
3.4.2.3. Instruction Scheduling Techniques for Systolic Instruction Broadcast 49
3.4.2.4. Hardware Approach 56
3.5. Results and Analysis 57
3.5.1. Metrics 57
3.5.2. SIMPil Applications 58
3.5.3. Clock Count Penalty 58
3.5.4. System Performance 60
3.5.5. Hardware Overhead 63
3.5.6. Area Efficiency 64
3.6. Chapter Conclusion 66

CHAPTER 4 Systolic Virtual Memory 67
4.1. Summary 67
4.2. Introduction 68
4.3. Related Work 69
4.3.1. Linear Mapping Technique 69
4.3.2. Data Prefetching Technique 72
4.4. Approach: SIMD-systolic System with Systolic Virtual Memory 74
4.4.1. Systematic Design Approach 74
4.4.2. Case Study of Extended Mapping Techniques: Vector Quantization 76
4.4.3. Memory Operations in SIMPil Architecture 79
4.4.4. Systolic Virtual Memory (SVM): SIMD-systolic System with Off-Chip Memory Accesses 80
4.4.4.1. Systolic Load 80
4.4.4.2. Systolic Store 80
4.4.5. Data Prefetch 83
4.4.5.1. Prefetch Instruction 83
4.4.5.2. Address Table 85
4.4.6. Instruction Scheduling 87
4.5. Results and Analysis 92
4.5.1. Channel Utilization 93
4.5.2. System Performance 94
4.5.3. Clock Count Penalty 95
4.5.4. Application: Matrix Multiplication 96
4.5.4.1. Matrix Multiplication: Execution Time 98
4.5.4.2. Memory Requirement 98
4.5.4.3. Memory Area Efficiency 102
4.6. Chapter Conclusion 103

CHAPTER 5 Concluding Remarks 104
5.1. Conclusions 104
5.2. Future Work 105

APPENDIX A SIMPil Architecture 106

References 107

Vita 113

List of Tables

Table 1: High-level comparisons between Systola and SIMD-systolic system 6

Table 2: Low-level comparisons between Systola and SIMD-systolic system 6

Table 3: Metrics for experiments. 26

Table 4: Description of selected workloads. 30

Table 5: Explored storage configurations.* 30

Table 6: Storage area efficiency (x 10^{-6}). 31

Table 7: Area efficiency (x 10^{-6}). 31

Table 8: Energy efficiency (x 10^{-6}). 31

Table 9: Metrics for experiments. 57

Table 10: Number of instructions issued by controller. 59

Table 11: Clock count penalties (delay cycles) of SIMD-systolic systems in three approaches. 59

Table 12: Definitions for linear mapping method. 69

Table 13: Transformation table for each edge. 77

Table 14: Memory addressing modes for each type of memory operation. 79

Table 15: Distances between consecutive off-chip memory operations. 92

Table 16: Metrics for experiments. 92

List of Figures

Figure 1: The principles of memory interfaces for (a) conventional processor and (b) systolic array [12]. 3

Figure 2: Relations among related research and our approaches in three contributions. 9

Figure 3: Chatin's register allocator [45] 15

Figure 4: Brigg's register allocator [46,47] 15

Figure 5: Overall framework for finding optimal storage configurations by application retargeting. 17

Figure 6: Application retargeting module with register reassignment 18

Figure 7: Example of register reassignment. 20

Figure 8: Application retargeting module with memory optimization. 21

Figure 9: Example of memory optimization based on the lifetimes. 22

Figure 10: Results of memory optimization: memory words in byte used in the original program and in the optimized program for median filtering, TAK, complement, and brightness slicing image processing applications. 23

Figure 11: GENESYS structure [63]. 25

Figure 12: Verification steps. 26

Figure 13: Overall VQ processes. 29

Figure 14: Results of code sizes and execution time in clock cycles for retargeting VQ application to different storage configurations. 29

Figure 15: Code size increase. 32

Figure 16: Execution time increase. 33

Figure 17: Normalized storage area efficiency for selected workloads. 33

Figure 18: Normalized energy efficiency. 34

Figure 19: Average of normalized storage area efficiency. 35

Figure 20: Average of normalized energy efficiency. 35

Figure 21: Reachable fraction of a chip for future VLSI technology (from ITRS). 38

Figure 22: Two methods for delivering instructions to PEs [4]. 39

Figure 23: The architecture of Systola 1024 [30]. 40

Figure 24: Control flow in an instruction systolic array [30]. 41

Figure 25: The structure of the dedicated communication register in Systola 1024 [30]. 41

Figure 26: A 4x4 mesh of PEs showing how instructions are pumped from an ACU to PEs [1]. . 42

Figure 27: The overall framework of the SIMD-systolic architecture. 43

Figure 28: Neighboring PEs showing how data is transferred. 44

Figure 29: Conditions causing delays in systolic instruction broadcast. 45

Figure 30: Hazard detection logic. 46

Figure 31: Sample code that introduces a delay due to a write following a communication instruction. 47

Figure 32: Clock cycle splitting – RD for a data read from register file and WR for a data write to register file. 48

Figure 33: An example of an execution of a communication instruction with split clock cycle . . 48

Figure 34: Sample codes that eliminate a delay due to a following write after communication instruction by having two write register ports. 48

Figure 35: Framework of an instruction scheduler for systolic instruction broadcast. 50

Figure 36: An example of instruction scheduling technique for software approach. 53

Figure 37: An example of instruction scheduling technique for two write-port register file approach. 55

Figure 38: Bypass logic. 56

Figure 39: Projected system performance in sustained system throughput. 62

Figure 40: Normalized sustained throughput. 62

Figure 41: Register file size increase for an additional write port: R(2R,1W) – 2 read-port, 1 write-port register file; R(2R,2W) – 2 read-port, 2 write-port register file. 63

Figure 42: Area efficiencies in (GOP/mm^2). 65

Figure 43: Normalized area efficiency projected in year 2010. 66

Figure 44: Mechanism of systolic instruction broadcast and systolic data movement. 69

Figure 45: An example of data movement with systolic load instruction. 70

Figure 46: Performance gap between processor and DRAM (from ITRS). 72

Figure 47: Performance increase rates for processor and DRAM (from ITRS). 73

Figure 48: The overall framework of the systematic design method for a SIMD-systolic system. 75

Figure 49: DG for VQ application. 76

Figure 50: One-Column system designed for VQ based on transformations in Table 13. 77

Figure 51: Space-time representation. 78

Figure 52: Segment-level system designed for VQ. 78

Figure 53: High level view of space-time diagram assuming 4 x 4 processor array. 79

Figure 54: Systolic virtual memory: systolic load mechanism. 81

Figure 55: Systolic virtual memory: systolic store mechanism. 81

Figure 56: An Example of Systolic Store. 82

Figure 57: An example of software data prefetching using the 'prefetch' instruction. 83

Figure 58: Extended instruction format for data prefetching. 84

Figure 59: An example of an extended instruction format: 'sys_load R1, SR0.' 84

Figure 60: An example of data movement in one column of an 8 x 8 processor array for a 'sys_load.' 85

Figure 61: Address table. 85

Figure 62: An example of data prefetching using the address table given in Figure 61. 86

Figure 63: An example of data prefetching. 87

Figure 64: An example of consecutive systolic load instructions. 88

Figure 65: An example of consecutive systolic store instructions. 89

Figure 66: An example of three consecutive systolic load instructions. 90

Figure 67: Framework of an instruction scheduler for systolic virtual memory 91

Figure 68: Channel utilization for a given number of consecutive memory operations. 93

Figure 69: Channel utilization for a given combination of memory operations. (L:sys_load, S: sys_store) 94

Figure 70: Normalized execution time of off-chip memory operation relative to the on-chip memory operation. 95

Figure 71: Clock count penalty for systolic virtual memory. 96

Figure 72: Matrix multiplication application with shared memory address space. 97

Figure 73: Matrix multiplication application with private memory address space. 97

Figure 74: Matrix multiplication: normalized execution time. 98

Figure 75: Estimation of the number of transistors and area for each type of memories: DRAM, SRAM, and register file. 99

Figure 76: Estimation of the number of transistors and area for each type of memories: DRAM, SRAM, and register file (log scale). 99

Figure 77: Memory requirements for each implementation in the number of memory words. ... 100

Figure 78: Memory requirements in area where both memories (on-chip & off-chip) are implemented by SRAM. 101

Figure 79: Memory requirements in area where off-chip memory is implemented by DRAM and on-chip memory is implemented by SRAM. 101

Figure 80: Normalized area efficiency 102

Figure 81: SIMPil microarchitecture. 106

Summary

SIMD parallelism offers a high performance and efficient execution approach for today's broad range of portable multimedia consumer products. However, new methods are needed to meet the complex demands of high performance, embedded systems. This research explores new storage management techniques for this focused but critical application. These techniques include memory design exploration based on the application retargeting technique, storage-based systolic instruction broadcast, and systolic virtual memory to improve both the performance and efficiency of embedded SIMD systems. A selection of image processing applications serves as the workload for the study. Code retargeting software, architectural simulation, and technology models are used to evaluate these methods.

CHAPTER 1

Introduction

1.1. Introduction

Growing consumer demand for portable multimedia products is focusing architectural research on high efficiency, high performance computing architectures. These computing applications exhibit significant data parallelism that can be effectively exploited using well-studied execution models, such as Single Instruction Multiple Data (SIMD) architectures. Limited attempts to harness data parallelism through subword parallelism have already been incorporated in multimedia instruction extensions on general-purpose microprocessors and more recently in DSP processors. Much greater parallelism is available in applications; but SIMD architectures techniques developed in the 1970's and 1980's do not fully address issues that arise in implementing a monolithic SIMD array in an embedded system. One critical area is the handling of storage (registers, local processing element (PE) memory, and off-chip memory). Technology advances in transistors, on-chip interconnects, and packaging have dramatically altered the relative performance, implementation cost, and level of integration of storage.

This thesis presents research on techniques to efficiently handle the storage hierarchy in embedded SIMD processors for multimedia applications. It includes distribution of register and local memory storage, as well as a systolic approach to support off-chip dense memory arrays with minimum latency.

The basic research problems being addressed are defined in the next section.

1.2. Problem Statement and Contributions

There are many research issues to be addressed in the development of an embedded SIMD system. This thesis addresses three basic research problems.

First, in a replicated cell (a SIMD processing element), efficient silicon area usage is critical. Data storage (registers and local memory) is the single largest allocation of silicon area in a PE design. Registers have multiple access ports and the fastest access time whereas local memory can store more bits in a given area. An optimal allocation between registers and local memory requires a methodology for evaluating performance and cost across a set of target applications.

Second, SIMD execution is defined as simultaneous execution of broadcast instructions at every PE. As VLSI technology advances, PEs get smaller and SIMD arrays get larger. To avoid inevitable clock frequency limitations, a segmented, temporally and/or spatially shifted broadcast technique must be employed. This scheme must employ a combination of software reordering and hardware mechanisms to avoid the resulting data hazards.

Third, while local PE memory provides the fastest and most accessible (greatest access bandwidth) operand storage, off-chip dense memories can provide significantly greater density due to their specialized fabrication process and amortized interface circuitry. But accessing off-chip memory is complicated by limited per PE memory bandwidth, especially for large PE arrays. The situation is further exacerbated by SIMD's synchronous instruction execution that creates magnified peak off-chip bandwidth demands.

This thesis research attacks these problems with three related architectural approaches:

Approach 1: Improve methods to evaluate storage usage within a PE

Develop automatic techniques to examine application characteristics and explore the space of feasible register and local memory configurations. This includes automatic application retargeting and compilation techniques and technology modeling so cost and efficiency can be accessed.

Approach 2: Explore systolic instruction broadcast and resulting data hazard avoidance

Systolic instruction broadcast [1] eliminates long broadcast wires at the expense of execution simplicity. Develop and evaluate techniques for eliminating data hazards with this staggered execution of instructions with minimal performance penalty. Use implementation strategies and technology models to area costs and overall area efficiency.

Approach 3: Define and evaluate a systolic virtual memory system for off-chip storage

When local PE memory is insufficient or inefficient to meet application needs, develop an off-chip memory access technique utilizing the staggered instruction execution that results from systolic instruction broadcast. Employ data prefetching and scheduling techniques to minimize

penalties resulting from increased access latency and limited access bandwidth. Technology models are again developed to evaluate cost and efficiency as well as performance.

The following section summarizes related research on which this thesis builds.

1.3. Related Research

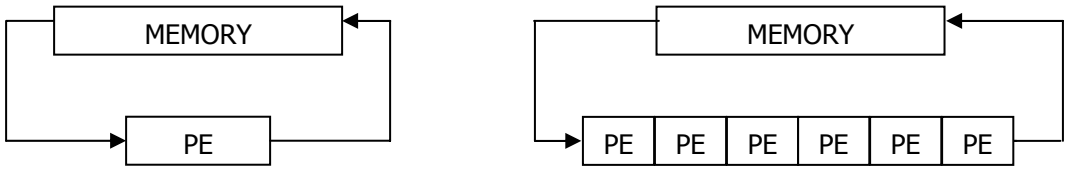
1.3.1. SIMD Instruction Broadcast

As mentioned earlier, long wires for global instruction broadcast in SIMD directly limit clock rates [1,2,4]. Recent designs attempt to address this problem with limited success; a board level instruction issue rate of 100MHz was achieved in 1995 by Bolotski's Abacus [5] and a chip level issue rate of 200MHz is achieved in products from PixelFusion which are currently developed with the clock rate of 400MHz for 256 PEs [6]. This instruction bottleneck limits the scalability of SIMD architectures [1,4,7].

Two alternatives to overcome instruction bottlenecks are pipelined instruction broadcast and the use of instruction caches [4]. The first approach hides the average broadcast latency by pipelined instruction broadcast. The Blitzen project [8] and MasPar's MP-2 [9] employed this method. The disadvantage of pipelined instruction broadcast is the required additional instruction latches. The second approach is proposed by Rockoff which uses a SIMD instruction cache [10]. However, this method consumes significant chip area and depends on temporal instruction locality for its effectiveness.

1.3.2. Systolic Arrays

H. T. Kung and C. E. Leiserson introduced systolic arrays in 1978 [11]. Figure 1 shows the conventional processor with one PE and systolic array with an array of PEs named cells due to their regularities [12,13,14].



(a) Conventional processor with one PE (b) Systolic array with an array of PEs

Figure 1: The principles of memory interfaces for (a) conventional processor and (b) systolic array [12].

As shown in Figure 1-(b), systolic array can achieve high throughput with the same I/O bandwidth by having array of many PEs. In a systolic system, data is pumped from the memory module rhythmically, passing through the array of PEs until it returns to the memory module. Though, the systolic array shown in Figure 1 is a linear array, it can be multi-dimensional to achieve a high degree of parallelism.

Systolic architectures have been proposed to design special-purpose systems whose efficiency derives from modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting, fan-in, fast response and balancing computation with I/O [2,12,15]. However, since the I/O problem is also related to the available internal memory, an appropriate memory structure should be designed to achieve a balance between computation time and I/O time [2,12,15]. Later surveys in [2,15] categorize systolic system into specific-purpose systolic system and general-purpose systolic system. Early design efforts in systolic systems were mainly related to solving a specific problem, e.g., matrix multiplication or convolution. These systems can achieve high performance due to application specific hardware. However, application specific architectures are expensive since their cost cannot be amortized across multiple applications. Thus many research efforts on general-purpose systolic architectures are gaining importance. General-purpose architectures can be divided into two categories – programmable models and reconfigurable models. The former can be implemented in SIMD or Multiple Instruction Streams and Multiple Data Streams (MIMD) architectures and the latter can be implemented by Field Programmable Gate Arrays (FPGAs). FPGAs have great advantage in flexibility by allowing system to be reconfigured. However, FPGAs are still more expensive, lower performance, and consume more power than those of Application Specific Integrated Circuits (ASICs). Thus programmable model is more appropriate for the current embedded systems running some set of applications while having some degree of flexibility thanks to the programmability.

Very Large-Scale Integration (VLSI) has revived systolic systems from the early eighties because systolic architectures can be easily implemented with the growing levels of integration. However, systolic architectures remain difficult to design and implement due to the requirements of detailed information such as data and control flows, computation sequences, and spatial and temporal information of all data used by applications [16,17,18]. In addition, formal design methods involving correctness-preserving transformations are required as the complexity of VLSI system design increases [19]. There are several systematic design approaches to design specified systolic arrays algorithmically [14,16,17,18,19,20]. Such techniques, transformation methods based on data flow [21], mathematical transformations based on data dependencies [22], and

mappings for multistage algorithms [23], are well observed for several algorithms in [14,17,18,19]. These design methods start with the representation of systolic algorithms in several formats such as data-flow graphs [19], signal-flow graphs [19], or Regular Iterative Algorithms (RIAs) [24]. These representations basically capture information about data dependencies and functional requirements. A popular graph representation is the Reduced Dependence Graph (RDG) which can be used to determine processor mappings for each task and scheduling of an algorithm based on the computed delays from data dependencies. Resulting processor maps and schedules correspond to the representation of the given algorithm. Usually, space representations depend only on algorithms not on architectures. Thus processor mappings and scheduling methods should be reconsidered to design SIMD-systolic architectures with fixed architecture features to obtain proper mapping of processors while satisfying time-constraints corresponding to systolic instruction broadcast.

This research extends the SIMD architecture by employing systolic instruction broadcast and systolic data movement methodologies to overcome the bottleneck of the memory bandwidth in SIMD architectures and to eliminate long wires which may cause low clock frequency, increased interconnection area and high power consumption. Concerns of efficient sequencing of instructions and scheduling of data movement are addressed for a correctness-preserving system design. A SIMD-systolic architecture simulator is built to run a set of workloads and the current technologies are plugged in the simulator to study the implementation limits.

1.3.3. Instruction Systolic Architecture (ISA)

In this section, the comparisons between SIMD-systolic system and Systola 1024 (ISATEC Co.) [25,26,27,28,29,30] are presented. In Section 3.3.2, more details of Systola architecture will be explained. The main difference between these two systems lies in the existence of instruction scheduler and data prefetching logic. The differences in high-level design issues between Systola 1024 and SIMD-systolic system are categorized in Table 1.

Table 2 shows several details in comparisons of Systola 1024 with SIMD-systolic system which is defined as target architecture of our research.

Table 1: High-level comparisons between Systola and SIMD-systolic system

	Systola 1024	SIMD-systolic
Wires	Short	Short
Instruction Distribution	Systolic way	Systolic way
Systematic design method	N/A	Yes (Extended mapping techniques for the data dependency graph)
Scheduling Responsibility	Programmer	Scheduler
Methodology to control the data movement based on the corresponding instruction	N/A	Instruction scheduling and data sequencing
Bandwidth from one memory module	Serial bit-wise	Word wide (16bits/word)

Table 2: Low-level comparisons between Systola and SIMD-systolic system

	Systola 1024	SIMD-systolic
Flexibility	Yes (Programmability + Selectors)	Yes (Programmability + more general instruction sets)
Simplicity & Scalability	Yes (Simple regular processor array)	Yes (Simple regular processor array)
Generality	Medium (Use of selectors makes possible to run the different set of instructions on different PE. + Small set of instructions restrict the functionality of processors)	Medium-High (Instruction sets are more general to cover a large set of applications)
Suitability for VLSI	Yes	Yes
Aggregate Function	Easy and Fast	Easy and Fast
Diameter	High (2N – 2 for NxN processor array)	High (2N – 2 for NxN processor array)
Conditional jumps	N/A	Sleep, Jump, Branch instructions
Global operation	N/A	s_vectorize, s_raisehand
Value Broadcasting	Row-wise or Column-wise	Immediate value can be broadcasted from controller using s_vectorize instruction
Data Word Length	8 bits	16 bits
On-chip Memory	32 registers + 2 communication registers (C-registers)	16 registers + 256 memory words
Special constant	Registers 0, -1	N/A
State flags	zero flag, negative flag	
Number of instructions	24 instructions	70 instructions (38 PE instructions + 32 controller instructions)

1.3.4. Compilation Techniques for Storage Optimization

Compilation techniques have been heavily developed to improve performance. In this sense, memory optimization techniques are developed to minimize memory bottlenecks due to the gap between processor and memory performance. One of them is the design of storage hierarchy which tries to reduce the memory access time by maximizing data locality.

However, storage (data and program) reduction techniques must be considered for resource-constraint systems, such as embedded systems [31,32,33]. This class of architectures is often used as portable devices in which power consumption and area cost are very important factors to achieve the long battery life and small size, in addition to high performance.

Until the early nineties, memory optimization techniques for embedded systems have been mainly developed to decrease code size, such as addressing optimization [34,35,36,37], mode optimization [38,39,40] and code compression [40]. These techniques utilize special architectural features such as addressing modes for a particular embedded system to minimize the code size. Program memory optimization is not the main concern in this proposed research since PEs will receive an instruction from a central controller without storing it.

Recently, many data memory optimization techniques are being developed as embedded systems flourish. The main effort is to reduce an allocated memory size especially for array data which usually takes a large fraction of data [41,42]. Data memory optimization techniques can be categorized into two approaches – architecture dependent optimizations and architecture independent optimizations [32]. A recent survey on data memory optimizations is available in [32].

Architecture independent optimization techniques are based on source-level transformations such as loop transformation and code rewriting to increase the data reusability. Loop transformation techniques are developed for long periods to increase the performance of the program [43]. However, these techniques also can be used to decrease the number of data accessed by merging two loops which access the same data [32].

Architecture dependent optimization techniques are designed for particular memory architectures. One of effective ways of reducing memory requirements is through better utilization of given registers. This is not only for maximizing memory utilization, and also for reducing power consumption and maximizing performance. The most frequently used variable should get the highest priority to be assigned a register since access to a register is less power consumed, and results in shorter access time. Register allocation is a standard part of the compilation process [44], usually using a common graph-coloring algorithm [45,46,47].

Typically, register allocation is performed on an internal representation (IR) right before code generation.

However, in this research, a cost-based register allocation is performed on assembly code. The result of our technique is assembly code that is retargeted to a given number of registers. If an input assembly code can run with a given number of registers, the input application is rewritten automatically to minimize the number of registers used in the application. Otherwise, the data in registers will be spilled into the data memory [45]. The register allocation and memory spilling are based on a cost model which is used to select the spilling-candidate registers.

Application retargeting is often performed by retargetable compilers [48]. A significant amount of research in this area has been conducted for embedded processors including CHESS [38], SPAM [34], AVIV [49], RECORD [50], and CodeSyn [48]. Since a single retargetable compiler is sufficient for different configurations of architecture, retargetable compilers are gaining popularity as reconfigurable architectures are emerging that can be tailored to a given workload. However, these techniques take high-level programs or some descriptions as an input. Thus hand-coded assembly code cannot be retargeted with these techniques directly. We develop retargeting techniques which can be used for general patterns of assembly applications – hand-written codes and compiler generated codes. Consequently retargeting techniques can play a great role in developing register size dependent embedded applications which still are written in assembly languages to achieve real-time performance requirements.

In this research, memory optimization is performed based on variable lifetimes under the assumption that effective addresses can be determined in compile time. The lifetime based optimizations [52,53] are reassigning the same memory location for different data items of which lifetimes are not overlapped. The resulting memory optimized code is also written in assembly code running on the same architecture as an input platform. Our retargeting techniques for different register sizes may increase memory uses due to the spilling. Thus memory optimization technique, called memory retargeting, is placed after register retargeting to increase data reusability by variable lifetime analysis. In addition, lifetime analysis techniques to place the frequently used data in higher level of memory in a memory hierarchy are useful when off-chip memory is used, to deal with its long access time.

Our retargeting techniques – register retargeting and memory retargeting – can be used to decide the optimal storage configurations for a selected set of workloads by analyzing the resulting retargeted applications in terms of the area efficiency and energy efficiency. This technique is applicable to either SIMD architectures or general architectures.

1.3.5. Summary of Related Research

Related research is described based on categories of techniques from Section 1.3.1 through Section 0. The proposed research is built on this related research. The summary of related research is depicted in Figure 2. In addition, Figure 2 shows the relations among these research efforts to express how this research is built on the previous research works.

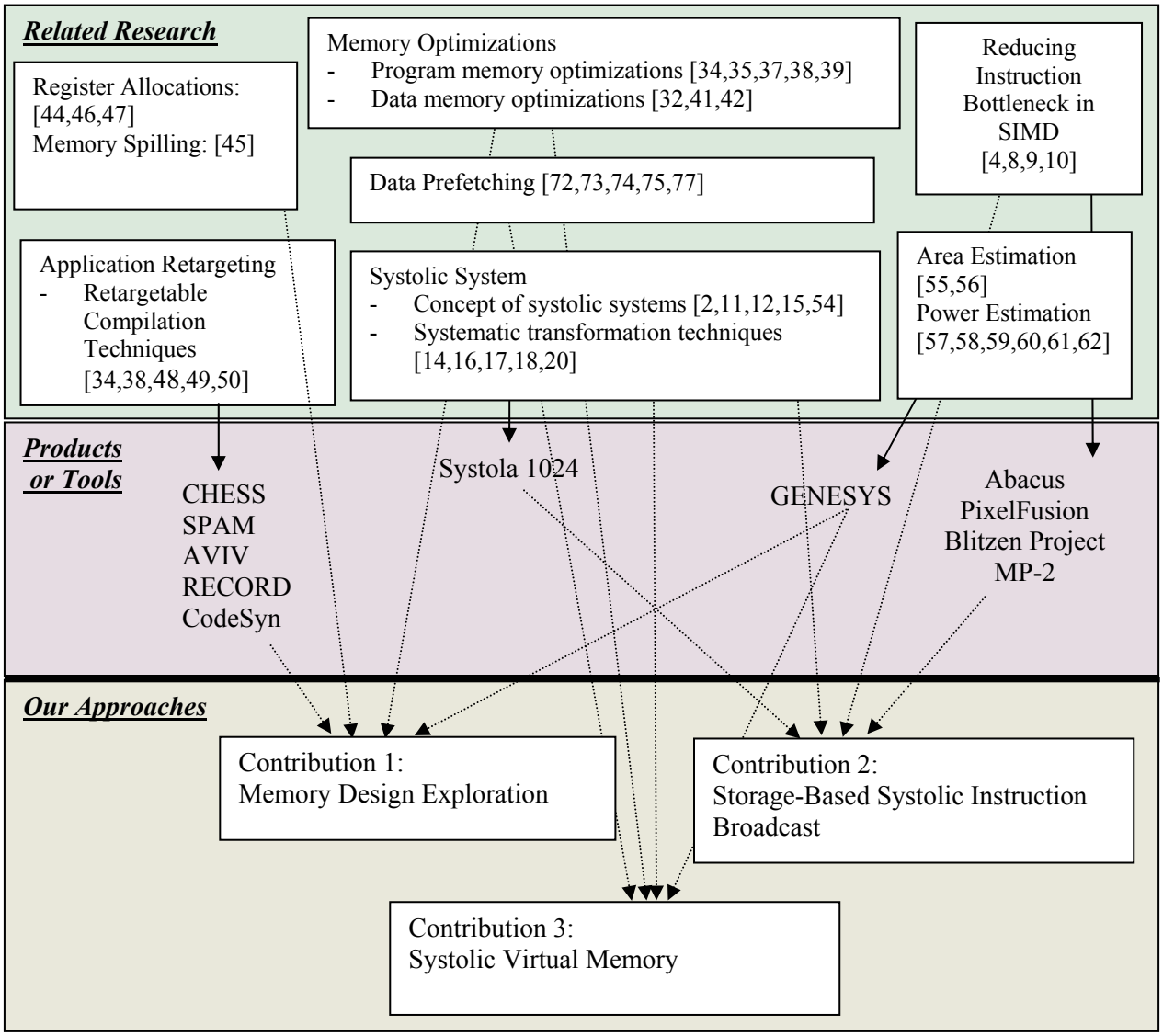


Figure 2: Relations among related research and our approaches in three contributions.

1.4. Thesis Contribution Summary

This outline summarizes the contributions presented in this thesis.

Contribution 1: Efficient Storage Usage in Embedded SIMD Systems

An analysis method for assessing storage needs and costs of a given application automatically retargeted across a spectrum of storage configuration designs was developed. Using this technique, a SIMD processing element achieves optimal area and energy efficiency with a register file containing between 8 and 12 words for given workload. This configuration is between 15% and 25% more area and energy efficient than other memory configurations being considered.

Contribution 2: Systolic Instruction Broadcast for Embedded SIMD Architectures

Systolic instruction broadcast is a high performance and area efficient instruction broadcasting scheme with short-wire interconnects by eliminating of wire latency bottleneck found in global instruction broadcast. In this contribution, we simulated systolic instruction broadcast in three approaches – software method, 2-write port register file method, and bypass method. Each method can result different area efficiencies based on the fraction of communications over a given set of instructions. In our evaluations, due to the system's short clock cycle time and scheduler, a speedup in system performance of up to 7.5 can be achieved by the year 2010. In addition, speedup of area efficiency also can be achieved up to 7.2 for a given workload.

Contribution 3: Systolic Virtual Memory

The ability of minimizing off-chip memory access latency while maximizing access frequency by scheduling techniques along with data prefetch techniques in systolic virtual memory mechanism was evaluated using our SIMD-systolic architecture simulator. Results show that, systolic virtual off-chip memory with shared address space can achieve over 50% higher area efficiency than that of an on-chip only system for a matrix multiplication application.

Contribution 1: Efficient Storage Usage in Embedded SIMD Systems
• Created methods and tools that retarget assembly language applications to different on-chip memory configurations. • Developed a memory configuration evaluation framework for performance, technology-model-based costs (area and energy), and area and energy efficiency. • Exercised and evaluated techniques for a selected application set.
Contribution 2: Systolic Instruction Broadcast for Embedded SIMD Architectures
• Evaluated three approaches (software-only, 2-write port registers and hardware bypass) for systolic instruction broadcasts to support instruction execution at local interconnect clock frequency projections. • Developed instruction reordering scheduler that minimizes execution time penalties for software hazard avoidance methods. • Incorporated implementation and technology models for hazard avoidance methods to evaluate area cost for different technology generations. • Evaluated system performance and area efficiency for hazard avoidance methods for a high-communication application workload.
Contribution 3: Systolic Virtual Memory
• Developed a scheme for utilizing a combination of local PE and off-chip memory in an embedded SIMD system. • Adapted a linear mapping algorithm for systolic off-chip memory prefetch scheduling. • Defined a VLIW-style SIMD instruction format and controller modification to support systolic virtual memory. • Evaluated technique using both high memory synthetic and kernel applications. • Incorporated technology and memory cell implementation models for memory size, area, and area efficiency evaluation.

1.5. Thesis Outline

The next three chapters of this dissertation discuss the methodology and results of the primary contributions. Chapter 2 presents an analysis method for assessing storage needs and costs of a given application automatically retargeted across a spectrum of storage configuration designs. Chapter 3 presents techniques for short-wire instruction broadcast to eliminate the wire latency bottleneck found in global instruction broadcast. Chapter 4 presents support for off-chip dense memory, called systolic virtual memory. The final chapter, Chapter 5, summarizes the results of this work and discusses future work. The appendix provides brief details about our baseline architecture, the SIMPil system.

CHAPTER 2

Efficient Storage Usage in Embedded SIMD Systems

2.1. Summary

Operand storage consumes a significant fraction of silicon area in today's processors. For embedded systems, resources are often limited and cost is critical. In an effort to introduce large-scale parallelism into embedded systems, new techniques are required to evaluate the effectiveness of each level of the storage hierarchy in order to achieve optimal efficiency in a highly replicated processing node design. This chapter presents a technique for analyzing storage performance and efficiency for a given application workload. It takes a two-prong approach: a) an automated retargeting technique is used in analyzing the storage requirements of a program over a range of storage configurations, and b) cost is estimated in terms of energy, and area efficiency for a given workload and storage configuration. Together these are used to explore storage configurations by analyzing a given workload under a range of different storage configurations with respect to performance, energy consumption, and chip area costs. Using this technique, a SIMD processing element achieves optimal area and energy efficiency with a register file containing between 8 and 12 words. This configuration is between 15% and 25% more area and energy efficient than other memory configurations being considered.

2.2. Introduction

Energy and area efficiency are critical metrics for embedded systems where battery life and cost are the central product parameters. Because storage (on-chip memory, caches, and register files) typically occupies half the chip area [3] and consumes a significant fraction of chip energy, exploring designs for effective storage utilization is vital.

This chapter presents a two-prong approach to support this design exploration. One, an automated retargeting technique is used in analyzing storage requirements of programs over a

range of storage configurations. This type of retargeting is too expensive and labor-intensive to perform manually during design exploration, particularly for hand-coded assembly programs that are optimized for specific embedded processor memory designs. Second, cost estimation is performed to assess the energy and area efficiency based on a given workload and storage configuration. This is used to extract a characterization of storage costs for a given program. Three factors, performance, energy, and area are used to decide an optimal storage configuration for a given workload.

In this chapter related research is presented first, followed by a description of our approach to explore the memory design space. Finally, results are given along with conclusions.

2.3. Related Work

2.3.1. Application Retargeting

Application retargeting is often done by retargetable compilers [33,34,35,38,48,49,50]. A significant amount of research in this area has been performed for embedded processors, including CHESS [35], SPAM [34], AVIV [49], RECORD [50], and CodeSyn [48]. Since a single retargetable compiler is sufficient for different configurations, retargetable compilers are gaining popularity as reconfigurable architectures are emerging that can be tailored to a given workload. These compilers take high-level programs and a target processor model description as input to generate assembly programs for the target processor. However, real-time embedded systems often run hand-coded assembly programs for efficiency. As a result, retargetable compilers cannot be utilized in the design phase.

Rewriting assembly programs by hand for new target architectures requires a massive modification effort, particularly since they are notoriously less portable and maintainable [51]. Thus, manually rewriting assembly programs during design exploration is not feasible. For these reasons, the presented technique applies automated application retargeting techniques to assembly programs to adapt them for different storage configurations.

These retargeting techniques extend traditional register allocation and memory optimization techniques described in the next two sections.

2.3.2. Register Allocation

One of the back-end compilation processes is register allocation [44], usually using a common graph-coloring algorithm [45,46,47]. As a standard part of compilation processes,

register allocation takes an internal representation (IR) as input to allocate a given set of registers to variables. Processes in register allocations proposed in [45,46] are depicted in Figure 3 (Chatin's register allocator) and Figure 4 (Brigg's register allocator), respectively, and each step in the process is described as follows.

- **Renumber:** Assign a unique name to each symbolic register during its live ranges.
- **Build:** Build the interference graph for the renamed registers based on the live ranges.
- **Coalesce:** Delete "copy" instructions if the destination and source live ranges do not interfere.
- **Spill Costs:** Compute the spill cost for each live range by estimating the weighted number of loads, stores and other instructions needed to spill them.
- **Simplify:** Recursively remove unconstrained nodes from the graph and push them onto a coloring stack. If there are only constrained nodes in the graph, remove the nodes, mark them for spilling, and continue.
- **Spill Code:** Insert spill code for marked nodes.
- **Color:** Pop all the nodes off the coloring stack and give each node a color different from its neighbors.

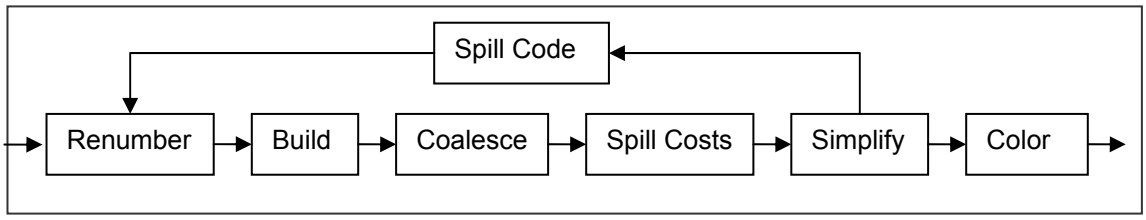


Figure 3: Chatin's register allocator [45]

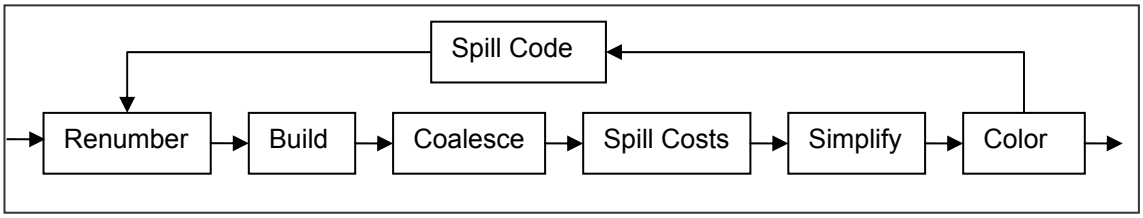


Figure 4: Brigg's register allocator [46,47]

Brigg's allocator modifies Chatin's allocator such that registers are allocated for variables more optimistically by deferring a decision of node colorability until the last stage, 'Color'.

Our technique extends Chatin's allocator to support register allocation technique across a range of file and memory sizes. Typically, register allocation is performed on an internal representation (IR) just before code generation. However, in our research, a register reassigning process takes an assembly program as input and retargets it to operate within a given register file limit. If an assembly program can run with a given number of registers, it is rewritten automatically to minimize the number of registers used in an application. Otherwise, register values will be spilled into memory based on a cost model [47].

2.3.3. Memory Optimization

Memory optimization techniques for embedded systems have been mainly developed to decrease code size, such as addressing optimization [34,48], mode optimization [40], and code compaction [37]. These techniques utilize special architectural features for particular embedded systems to minimize code size. Program memory optimization tends to be simpler than data memory optimization since data memory optimization must compute an effective address of data which is sometimes difficult or impossible to do unambiguously at compile time. Data memory optimization techniques that reduce the size of data memory are considered by placement and indexing of an array, which usually takes a large fraction of data in multimedia applications [41,42]. There are two main allocation strategies for array data – static and dynamic strategies [42]. In this chapter, we take a dynamic memory allocation strategy based on live ranges of memory words, with the assumption that the effective address can be determined at compilation time. Since compilation time in embedded systems is less critical than that in general systems, we compute all effective addresses at compile time. In addition, since our approach is applied to assembly programs, there is no difference between array data and non-array data.

2.4. Approach: Application Retargeting for Different Memory Configurations

The overall retargeting procedure, depicted in Figure 5, is discussed in this section. A significant number of applications for embedded processors are still hand-coded to meet real-time constraints. Since rewriting assembly code for several different configurations is error-prone and laborious, the development of automatic retargeting techniques for assembly programs is necessary for efficient design exploration. These assembly programs can be either hand-coded or generated by compilers. Once applications are retargeted, a simulator is used to run them to estimate their performances in executed clock cycles under the assumption that each instruction takes one clock cycle to execute. Since we are assuming that both on-chip memory and registers can be accessed in one clock cycle, differences in execution clock cycles are from the overhead of register spilling.

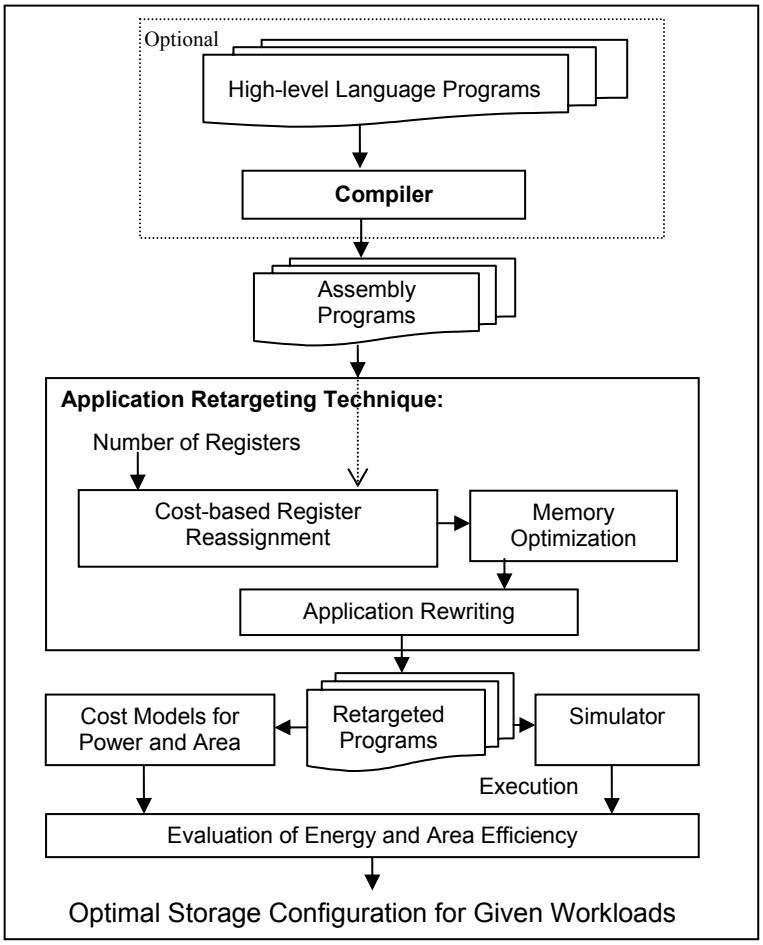


Figure 5: Overall framework for finding optimal storage configurations by application retargeting.

There are two main steps to retarget applications for given storage requirements. The first step is register reassignment, followed by memory optimization.

2.4.1. Register Allocation

Register allocation techniques are used to generate register-optimized applications with a limited number of registers by analyzing variable lifetimes. If there is a lack of registers, register spilling techniques [47] are applied to use memory instead of registers until a resulting application can run on the given number of registers. Figure 6 shows the register allocation module used in our approach. This register allocation technique is composed of processes in Chaitin's register allocator [45]. However, a typical register allocator usually takes the intermediate representation (IR) as an input of the back-end compilation process, while our approach takes either hand-written or compiler-generated assembly programs. In addition, our output programs can be run on target architectures directly. This can maximize code reusability.

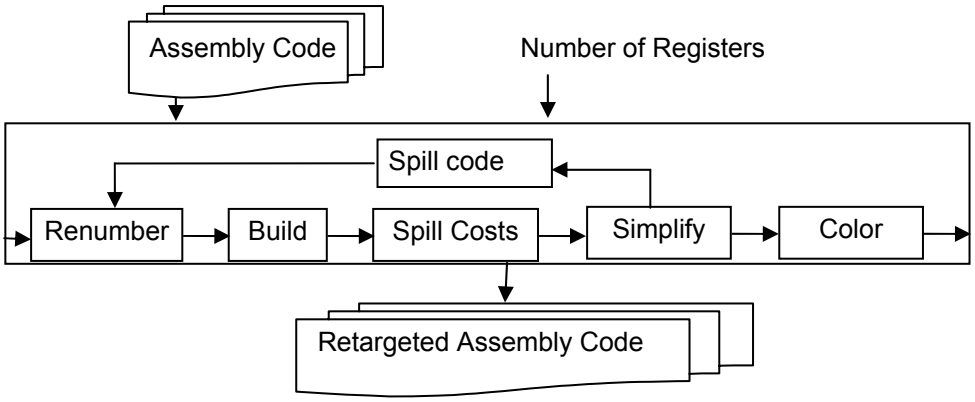
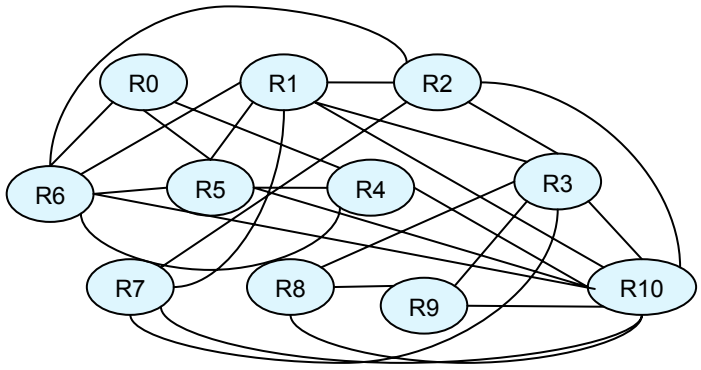


Figure 6: Application retargeting module with register reassignment

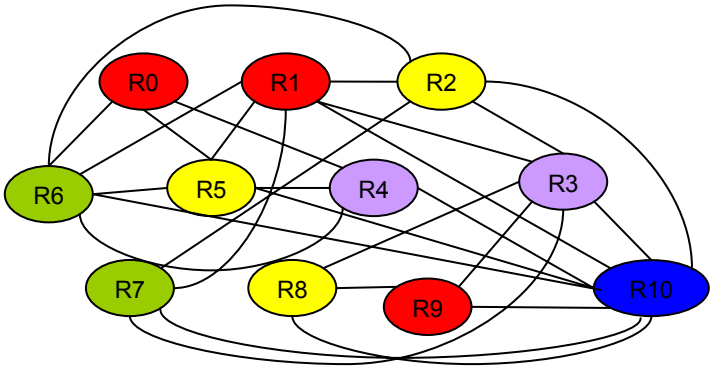
Since spilling costs are computed based on register usage, less frequently used register values are spilled. An example of register reassignment based on lifetimes is depicted in Figure 7. Assembly programs are written in the SIMPil assembly language and the description of each instruction is explained in the 'Comment' column in Figure 7. As in Chatin's allocator, an interference graph is built such that each register is represented by a node in the graph, and edges are drawn between nodes when there is an overlap in the registers' lifetimes. The resulting graph is shown in Figure 7 (b) and the colored graph is shown in Figure 7 (c). Finally, register reassigned assembly code is shown in Figure 7 (d).

Source Code		Live Ranges of Registers										
Code	Comment	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Loadi 10	R0 ← 10											
Addi R4, R0, 0	R4 ← R0 + 0											
Addi R5, R0, 1	R5 ← R0 + 1											
Addi R6, R0, 2	R6 ← R0 + 2											
Addi R10, R0, 3	R10 ← R0 + 3											
Load R1, R4	R1 ← MEM[R4]											
Load R2, R5	R2 ← MEM[R5]											
Load R3, R6	R3 ← MEM[R6]											
Sub R7, R1, R2	R7 ← R1 – R2											
Addi R8, R2, 0	R8 ← R2 + 0											
Slti R7, 0x01	If (R7 < 0) then sleep											
Addi R8, R1, 0	Else R8 ← R1 + 0											
Wakeupi 0x01	Wakeup											
Sub R9, R3, R8	R9 ← R3 – R8											
Slti R9, 0x02	If (R9 < 0) then sleep											
Addi R8, R3, 0	Else R8 ← R3 + 0											
Wakeupi 0x02	Wakeup											
Store R10, R8	MEM[R10] ← R8											

(a) Source code and live ranges of registers.



(b) Interference graph.



(c) Graph coloring with colors = {R, G, B, Y, V}.

Source Code	
Register in Original Source Code	Reassigned Register
R0	R0
R1	R0
R2	R3
R3	R4
R4	R4
R5	R3
R6	R1
R7	R1
R8	R3
R9	R0
R10	R2

Source Code	
Original Code	Reassigned Code
Loadi 10	Loadi 10
Addi R4, R0, 0	Addi R4, R0, 0
Addi R5, R0, 1	Addi R3, R0, 1
Addi R6, R0, 2	Addi R1, R0, 2
Addi R10, R0, 3	Addi R2, R0, 3
Load R1, R4	Load R0, R4
Load R2, R5	Load R3, R3
Load R3, R6	Load R4, R1
Sub R7, R1, R2	Sub R1, R0, R3
Addi R8, R2, 0	Addi R3, R3, 0
Slti R7, 0x01	Slti R1, 0x01
Addi R8, R1, 0	Addi R3, R0, 0
Wakeupi 0x01	Wakeupi 0x01
Sub R9, R3, R8	Sub R0, R4, R3
Slti R9, 0x02	Slti R0, 0x02
Addi R8, R3, 0	Addi R3, R4, 0
Wakeupi 0x02	Wakeupi 0x02
Store R10, R8	Store R2, R3

(d) Register reassigning for each color: R→ R0, G→R1, B→R2, Y→R3, V→R4.

Figure 7: Example of register reassignment.

The source operand and destination operand can use the same register in this example. Thus if two lifetimes are overlapped only in one instruction, that is a defined instruction of one register value is also the last-use instruction of another register value, the same register can be allocated for these two registers. For example, R1 and R4 in Figure 7 (a) can be assigned to one register. As a result, there is no edge between these two registers in an interference graph that represents the interference between variables (registers in our case). This example shows an instruction level

register allocation not a basic block level. Since our approach is applied to basic blocks based on the control flow of applications, there are possible false overlaps of lifetimes of registers when two registers are used in the same basic block. However, control flow analysis line by line requires too much memory space to keep that information throughout the compilation phases or retargeting processes.

2.4.2. Memory Optimization

After register reassignment, memory accesses in applications are optimized to reduce memory requirements. By computing memory variable lifetimes, the same memory location can be reused for different memory data. The processes in this technique are depicted in Figure 8.

The memory reassigning phase also utilizes variable lifetime. For a given assembly program, lifetimes for each memory variable are computed for each basic block. After that, memory words are identified that can share the same memory location with other memory words, where the lifetimes of these memory words do not interfere. Then, memory addresses for selected data words are reassigned in a given application to share the memory location as much as possible. Finally, the retargeting module rewrites the application with reassigned memory addresses for memory data to achieve maximum reusability of memory locations, resulting in lower (or at least same, in the worst case) memory requirements.

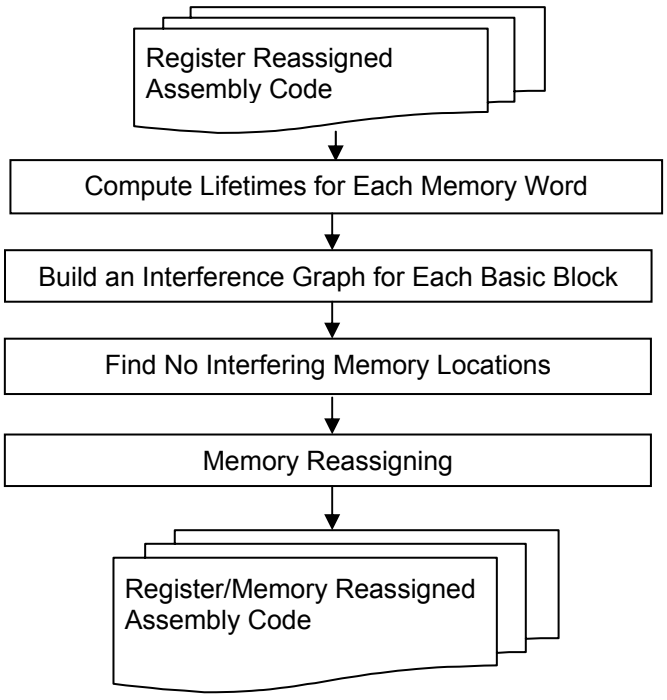
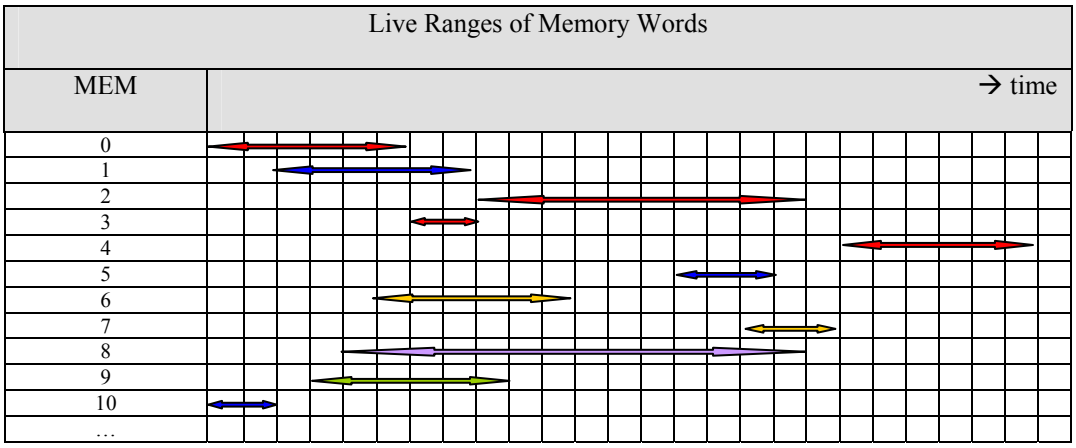
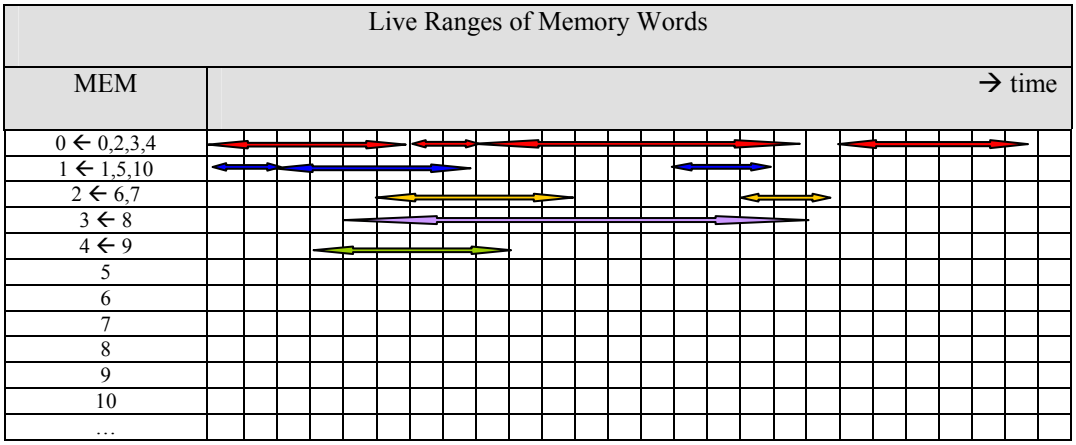


Figure 8: Application retargeting module with memory optimization.

An example of memory optimization is depicted in Figure 9. Based on lifetimes of each memory word, memory locations can be reused to reduce the memory requirement for a given application. In this example, the number of memory words used in the original program is 11 and memory requirement for a retargeted code is 5 words.



(a) Original memory uses and lifetimes.



(b) Reassigned memory words.

Figure 9: Example of memory optimization based on the lifetimes.

Figure 10 shows the results of the memory optimization phase for four different applications. A brief description of each application is given in Table 4.

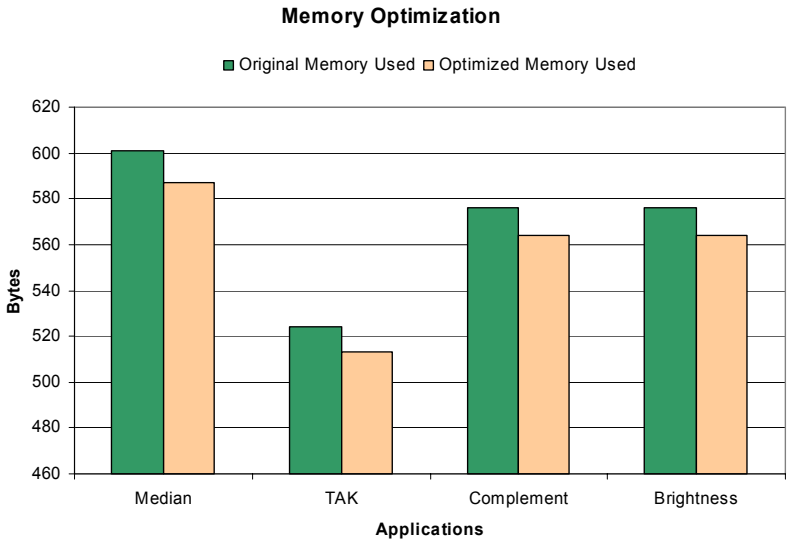


Figure 10: Results of memory optimization: memory words in byte used in the original program and in the optimized program for median filtering, TAK, complement, and brightness slicing image processing applications.

Results show that 2% of memory requirements can be reduced by the memory optimization phase. Since our approach is performed in compilation time based on application retargeting, results from our approach are not good enough as a run-time memory optimization. In addition, patterns of memory accesses in a set of workloads make it hard to share the memory locations based on lifetimes. The following description is a typical algorithm pattern of multimedia applications that also shows typical memory access patterns.

Algorithm:

A. Get an image data from I/O.
B. Store an input image (IMG_{IN}) in a memory (array).
C. Do some operations.
D. Load image data into registers from memory.
E. Process some operations on image data in registers
F. Store register values back to memory as a result (IMG_{OUT}).

Since step B, D and F are usually repeated for video processing applications, nested loops are commonly used for these steps (often D and E are in the same loop). Based on this algorithm, image array data, IMG_{IN} is defined in step B and is used at step D. If IMG_{IN} is no longer used in

an application, IMG_{IN} is dead in step D. Thus all elements in IMG_{IN} tend to overlap their lifetimes, which prevents sharing the same memory locations among memory data. In addition, resulting image data, IMG_{OUT}, tends to be stored in the same location as IMG_{IN}. As a result, only intermediate data uses, that are a small portion of memory used in a given application, have a chance to be optimized, resulting in only a 2% decrease in memory requirements.

2.5. Energy and Area Estimation for Storage

In VLSI systems, since increased of chip area may result in higher chip cost, area is a critical design parameter in embedded systems. However, tradeoffs between chip area and system performance should be considered. Thus area efficiency is modeled to consider chip area and performance together. In addition, energy consumption is also an important issue in embedded systems design, due to limited battery life. Thus area efficiency and energy efficiency should be considered during system design exploration.

There is on-going research to estimate energy consumption and chip area, reported in [63]. This is based on a generic system simulator referred to a GENESYS [63] whose structure is shown in Figure 11. The cost model used in this system is an empirical model to estimate energy consumption and die size considering interconnections and technology factors. GENESYS has seven different levels of inputs to generate a variety of chip features as outputs that are shown in Figure 11. GENESYS also provides a library of input files for commercial general-purpose architectures, such as the Pentium processor and a set of values derived from the International Technology Roadmap for Semiconductors (ITRS). The cost model used in this research involves primarily the system architecture inputs which are divided into three parts – architecture parameters, CPI parameters and gate parameters. Over these parameters, the Rent's parameters (constant, internal exponent, and external exponent), number of logic transistors, activity factor, word and bus size, address space, gate fan-in and –out, and gate utilization, etc are set. All these values will directly feed into GENESYS to estimate energy consumption and die size. Energy consumption, die size, and performance are used to compute the area efficiency and energy efficiency for each given input parameters depending on storage configurations.

Energy and area estimations are used in early stage decisions for architecture designs. In our research, after a storage-oriented application retargeting phase, we examine the cost of storage configurations for each workload. By doing this, we can decide the optimal on-chip memory configuration in terms of energy and area efficiencies during design explorations.

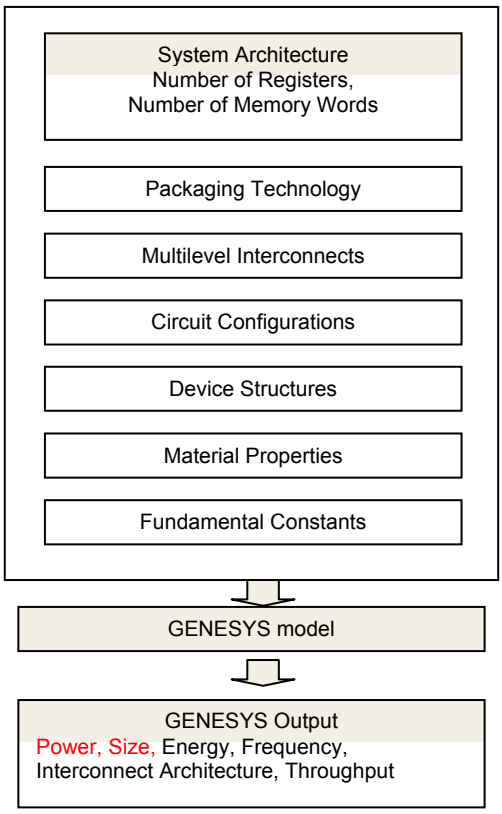


Figure 11: GENESYS structure [63].

2.6. Validation and Evaluation

An application retargeting technique has been developed to target different memory configurations in memory design exploration. Architecture simulators are used to assess performance and memory usage, and analysis tools are chosen to evaluate the efficiency of target systems in terms of energy and area efficiencies. These simulators provide validation of our approach and also show the impact of architectural changes in terms of memory size.

The evaluation focuses on changes in memory size (register file size and corresponding required memory words) for a given workload in the DSP area. Retargeted applications based on changes in register file size are evaluated by measuring execution time, code size, energy consumption, and chip area. Memory optimization techniques are used in our research to minimize memory requirements. Results from memory optimizations are evaluated by comparing memory requirements of retargeted applications and original input applications.

Results are verified by comparing the results of retargeted applications with varying register file sizes with that of the original input application. In addition, memory contents are also

checked for the correctness of the retargeted applications by dumping memory words. The verification steps are depicted in Figure 12.

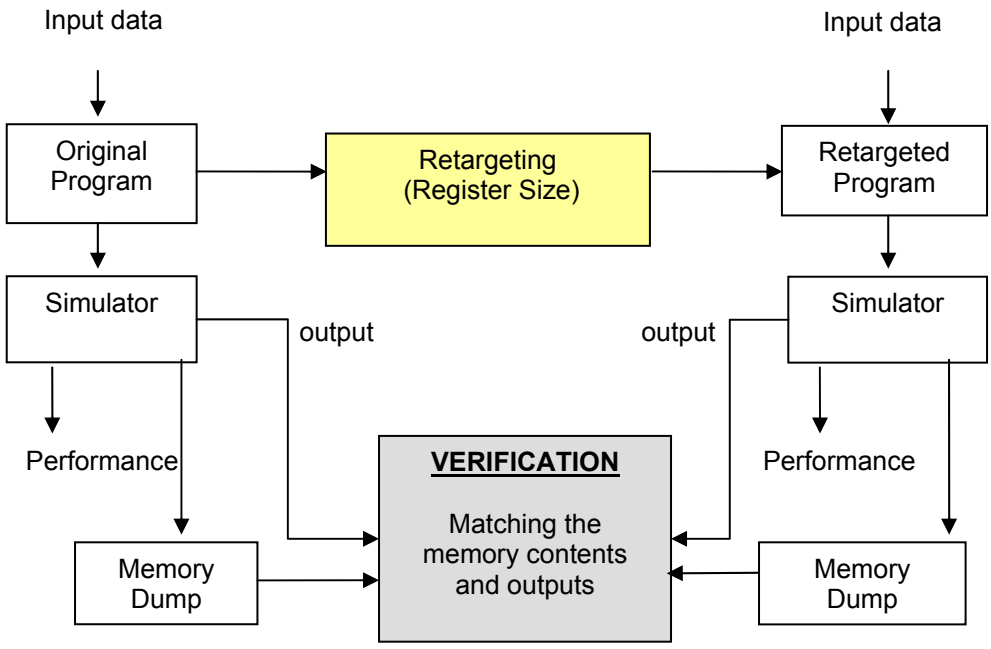


Figure 12: Verification steps.

2.7. Metrics and Analysis

Estimated energy and area must be considered along with performance. Due to tradeoffs between area, energy consumption, and performance, it is necessary to explore many design configurations to decide an optimal storage configuration. Table 3 shows evaluation metrics used in our approach to analyze efficiencies of given storage configurations for a set of workloads.

Table 3: Metrics for experiments.

Analysis	Metrics
Code Size	Increase in code size for application retargeting (in instructions)
Execution Time	Clock Cycle Time (in seconds)
Energy Efficiency	Performance divided by Power Dissipation (in J/sec)
Area Efficiency	Performance divided by Die Size (in instructions/sec/cm^2)

Our metrics for analyses are listed in Table 3 and the following formulas are given to compute efficiency factors considered in the analyses.

Storage Area Efficiency = $\frac{Performance}{Storage\_Area}$ where performance is measured per clock cycle and storage area is the sum of the area for spilling memory (code and data memory) and the area for the register file in mm^2.

Overall Area Efficiency = $\frac{Performance}{DieSize}$ where performance is measured per clock cycle and die size is chip area in mm^2.

Energy Efficiency = $\frac{Performance}{Power\_Dissipation}$ where performance is measured per clock cycle and power dissipation is measured for an overall chip in Watt.

Normalized area and energy efficiency is compared for each workload. Normalized storage area efficiency is computed for the total storage area, which is the sum of register area and memory area. Based on this, normalized storage area efficiency is computed as follows:

Normalized Storage Area Efficiency = $\frac{(Performance / StorageArea(new\_application))}{(Performance / StorageArea(original\_application))}$

where storage area is the sum of register area and memory area.

Similarly, normalized energy efficiency is computed as follows:

Normalized Energy Efficiency = $\frac{(Performance / PowerDissipation(new\_application))}{(Performance / PowerDissipation(original\_application))}$

To determine the best configuration of storage, all efficiency factors for energy efficiency and area efficiency will be averaged with an assumption that all applications have an equal importance in a given system.

2.8. Results and Analyses

In this section, our experimental results are presented and discussed. In Section 2.8.1, an example Vector Quantization (VQ) application written in the SIMPil assembly language is described to show the effect on code size and execution time in clock cycles for different memory configurations. The following section shows the results and analyses from several assembly applications generated by gcc compiler.

2.8.1. Example Application: Vector Quantization (VQ) Encoding

Vector Quantization (VQ) is commonly used for data compression in speech, image and video coding, and speech recognition [64]. VQ exploits a spatial correlation existing between neighboring signals. It quantizes a group of signals together and operates directly on image blocks to compare an image block and codeword vector in a given codebook. The index of the codeword vector having minimum distortion is transmitted instead of a full image block. One of popular distortion measurements is the Euclidean distance (d) between two vectors.

d (input, codeword) = || input – codevector ||^2 $= \sum_{i=0}^{N} (Ci - input)^2$ where Ci is the ith code vector and codebook size is N.

Since each input image block is replaced with the index of a codeword in a codebook, encoding (or compression) in VQ can achieve a compression factor of index in bits / image blocks in bits. The decoding (decompression) step is a reverse of an encoding step. It takes a transmitted index and replaces it with the corresponding codeword, using the same codebook used in the encoding step. Figure 13 depicts the overall VQ process.

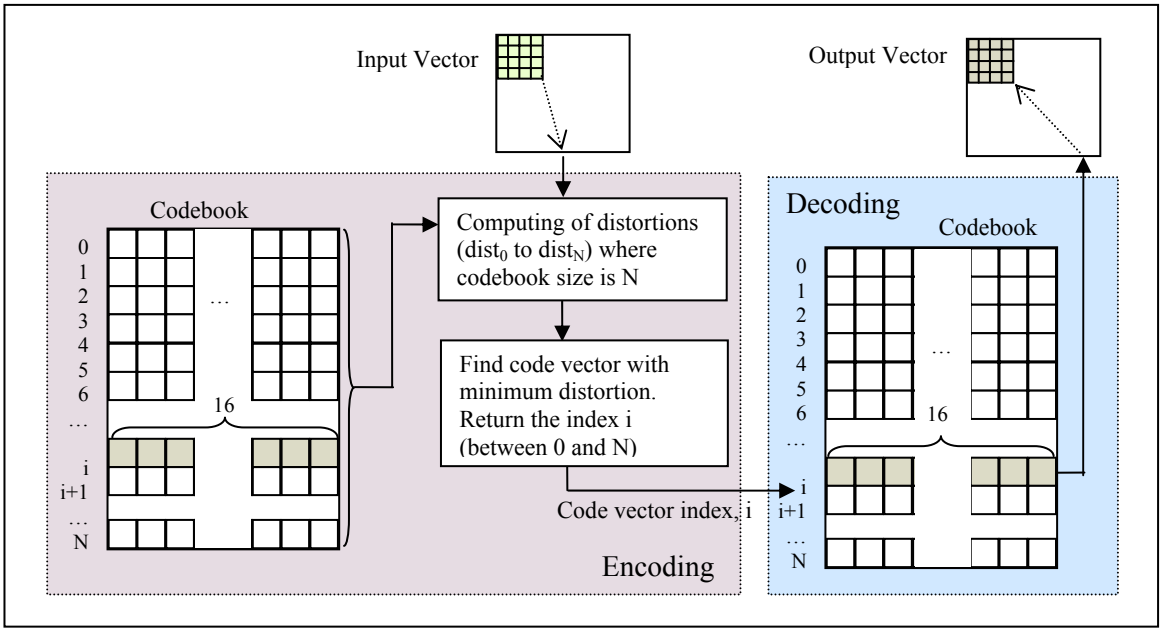
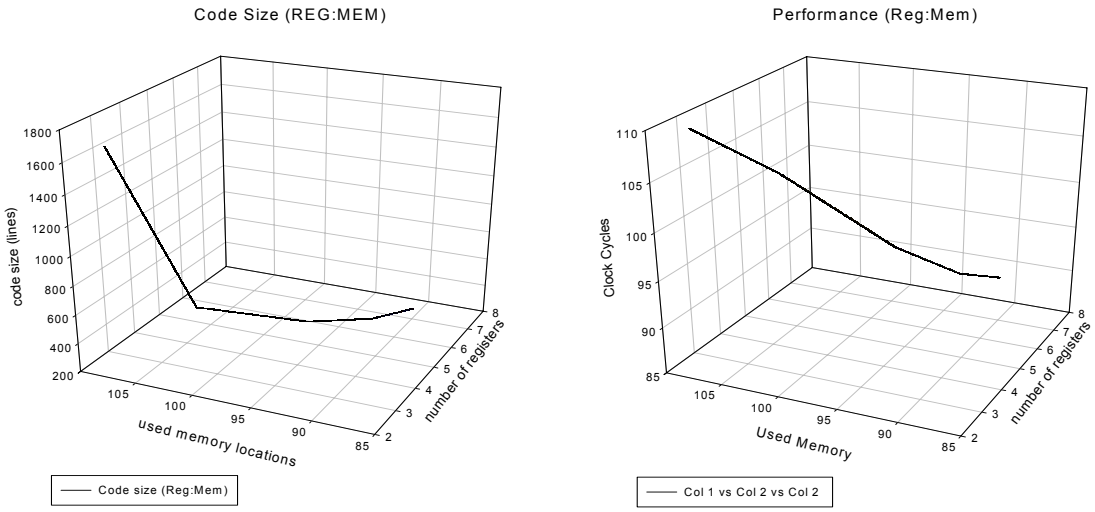


Figure 13: Overall VQ processes.

We consider the VQ encoding process due to its high computing-demand characteristics as well as high memory requirements. The code sizes and performance for each on-chip memory configuration are depicted in Figure 14.



(a) Code size.

(b) Clock cycles.

Figure 14: Results of code sizes and execution time in clock cycles for retargeting VQ application to different storage configurations.

2.8.2. Experimental Results

Table 3 shows evaluation metrics applied over a given set of storage configurations. In our experiments, clock frequency is assumed to be 500MHz and word size is 32 bits. Both instruction width and register width are one word. Graphs in this section represent specific storage configurations on a horizontal axis. These correspond to a number of registers and a number of memory words as shown in Table 5 for each application in Table 4.

Table 4: Description of selected workloads.

Application	Description
Median Filtering	Each pixel in an image is examined to find the median-ranked brightness value of the pixels in a certain-sized window surrounding the pixel.
TAK	A popular benchmark for recursive function calling, this function is defined as: tak(x y z) = z, if y >= x; tak(x,y,z) = tak (tak (x-1) y z) (tak (y - 1) z x) (tak (z - 1) x y), otherwise.
Complement Image	Each image pixel is logically complemented.
Brightness Slicing	If input pixel value is between two given threshold values, the output pixel value is set as 255. Otherwise, the value is set to 0.

Table 5: Explored storage configurations.*

	Median Filtering	TAK	Complement	Brightness
7	(r7, m162)	(r7, m141)	(r7, m149)	(r7, m148)
8	(r8, m153)	(r8, m135)	(r8, m141)	(r8, m141)
9	(r9,m152)	(r9,m131)	(r9,m141)	(r9,m141)
10	(r10,m148)	(r10,m130)	(r10,m141)	(r10,m141)
11	(r11,m147)	(r11,m129)	(r11,m141)	(r11,m141)
12	(r12,m147)	(r12,m128)	(r12,m141)	(r12,m141)
13	(r13,m147)	(r13,m128)	(r13,m141)	(r13,m141)
14	(r14,m147)	(r14,m128)	(r14,m141)	(r14,m141)
15	(r15,m147)	(r15,m128)	(r15,m141)	(r15,m141)

*r# means number of registers and m### is the number of memory words required by applications (including register spilling overhead, both instructions and data if any).

The following three tables show experimental results for each configuration depicted in Table 5. Table 6 shows area efficiencies for storage (registers, and memory including an extra memory for register spilling) that is a function of execution time and storage area.

Table 7 depicts overall area efficiencies and Table 7 shows energy efficiencies for each workload.

Table 6: Storage area efficiency (x 10^{-6}).

Number of Registers	Median Filtering	TAK	Complement	Brightness
7	3.049	0.555	67.581	58.746
8	2.123	0.246	55.198	53.595
9	2.857	0.349	80.519	69.993
10	2.842	0.458	73.509	63.899
11	3.166	0.466	67.581	58.746
12	3.049	0.507	62.498	54.328
13	2.834	0.555	58.089	50.495
14	2.646	0.518	54.226	47.137
15	2.479	0.486	50.813	44.170

Table 7: Area efficiency (x 10^{-6}).

Number of Registers	Median Filtering	TAK	Complement	Brightness
7	0.155	0.030	3.212	2.792
8	0.143	0.020	3.013	2.723
9	0.153	0.023	3.289	2.859
10	0.154	0.025	3.250	2.825
11	0.156	0.027	3.212	2.792
12	0.155	0.028	3.175	2.760
13	0.153	0.030	3.139	2.728
14	0.151	0.030	3.103	2.697
15	0.150	0.029	3.068	2.666

Table 8: Energy efficiency (x 10^{-6}).

Number of Registers	Median Filtering	TAK	Complement	Brightness
7	0.112	0.022	2.313	2.011
8	0.103	0.015	2.181	1.986
9	0.109	0.017	2.361	2.052
10	0.111	0.018	2.337	2.031
11	0.112	0.019	2.313	2.011
12	0.112	0.020	2.291	1.992
13	0.111	0.022	2.269	1.972
14	0.110	0.021	2.247	1.953
15	0.109	0.021	2.225	1.934

Figure 15 shows code size increases relative to that of the original application. The increase in code size is due to memory instructions added for register spilling. The amount of code size increase tells how many spilling instructions are inserted for register spilling, while execution times indicates how often these spilling codes are executed at run time. Thus, if spilling instructions are added inside a loop, the ratio of execution time increase will be larger than that of

code size increase. Based on experimental results shown in Figure 16, additional instructions for spilling tend not to be used repeatedly since the ratios of increased code size and execution times are similar. Particularly, TAK application has a greater increase in execution times when there are spilling codes, which means either TAK has more reusable code or more spilling instructions are placed inside loops relative to the other applications.

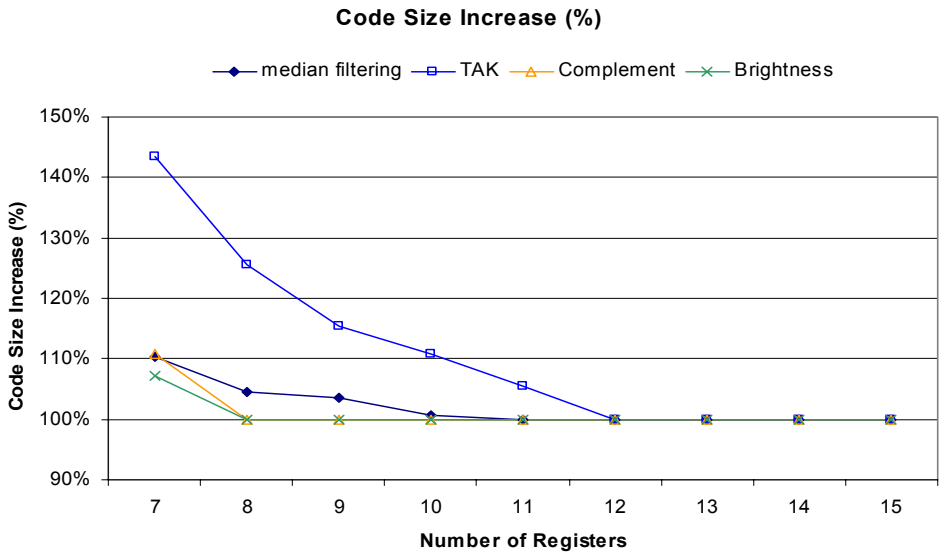


Figure 15: Code size increase.

To compare efficiencies for each workload, normalized efficiencies for area and energy consumption are computed. Normalized storage area efficiency is shown in Figure 17 which is computed as follows:

$$\text{Normalized Storage Area Efficiency} = \frac{(Performance / StorageArea(new\_application))}{(Performance / StorageArea(original\_application))}$$

where storage area is the sum of register area and memory area.

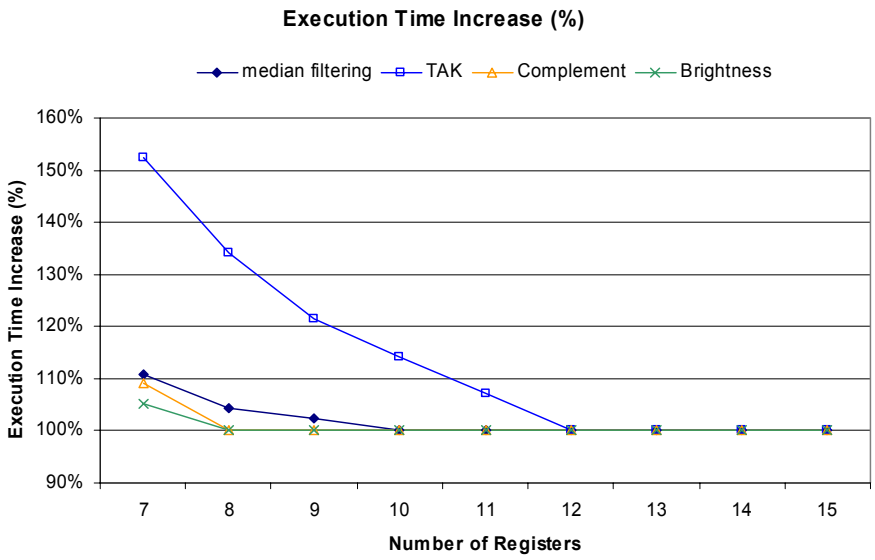


Figure 16: Execution time increase.

Normalized storage area efficiency shows the relative storage area efficiency for each storage configuration normalized to the original configuration of given workloads.

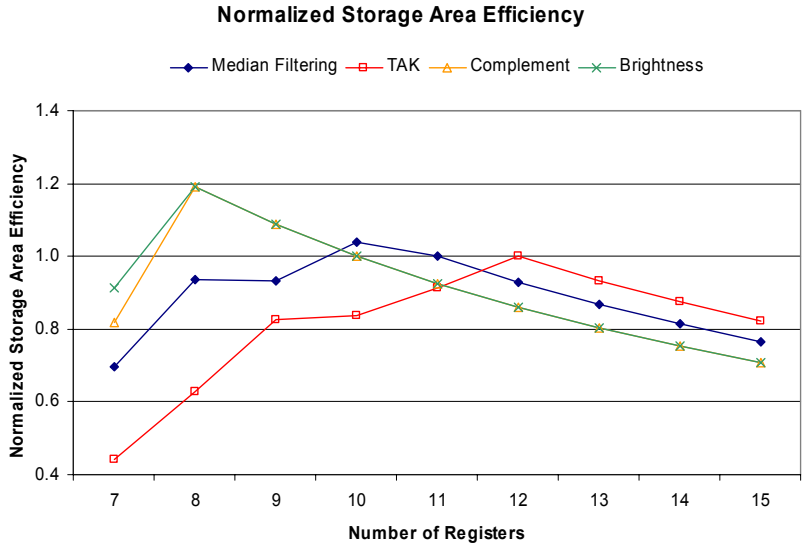


Figure 17: Normalized storage area efficiency for selected workloads.

As shown in Figure 17, storage configurations having best storage area efficiencies are different for each workload. Experimental results show that storage configurations, (r10, m148) for median filtering, (r9, m131), (r12, m128) for TAK, and (r8, m141) for complement and brightness slicing applications are the best configurations in terms of storage area efficiency.

Similarly, normalized energy efficiency is computed as follows:

$$\text{Normalized Energy Efficiency} = \frac{(Performance / PowerDissipation(new\_application))}{(Performance / PowerDissipation(original\_application))}$$

For energy efficiency, an overall chip power dissipation is computed. Figure 18 shows computed normalized energy efficiencies.

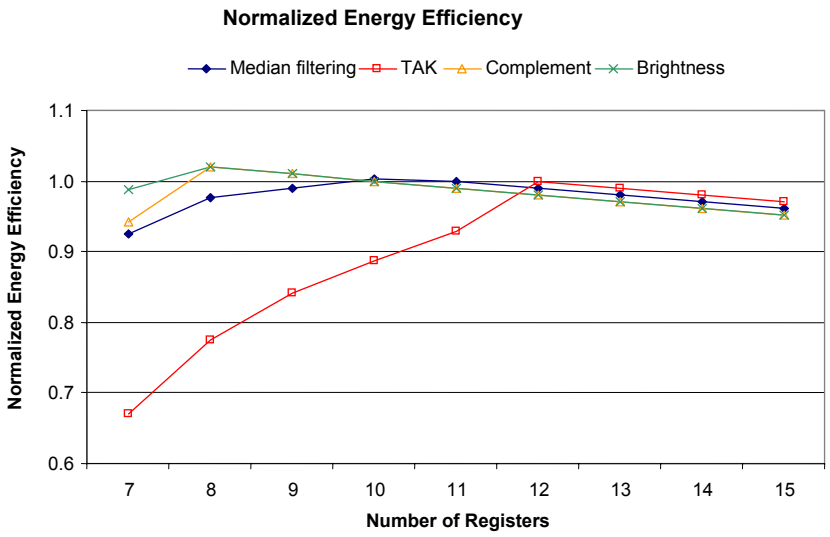


Figure 18: Normalized energy efficiency.

As shown in Figure 18, the best configurations for each workload in terms of energy efficiency are (r10, m148) for median filtering, (r7, m149) for complement, (r7, m148) for brightness slicing application, and (r12, m128) for TAK application.

To determine the best configuration of storage, we average all efficiency factors for energy efficiency and area efficiency with an assumption that all applications are equally important in a given system. Figure 19 shows the average of normalized storage area efficiencies and Figure 20 shows the average of normalized energy efficiencies for a set of applications. The figures indicate

that (r8, m153) is the optimal configuration for given workloads in terms of area efficiency and (r12, m147) is the optimal configuration in terms of the energy efficiency.

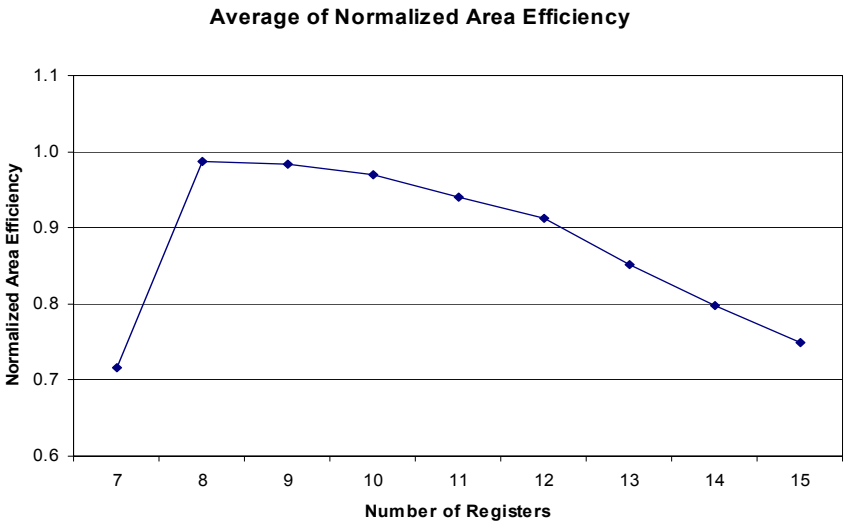


Figure 19: Average of normalized storage area efficiency.

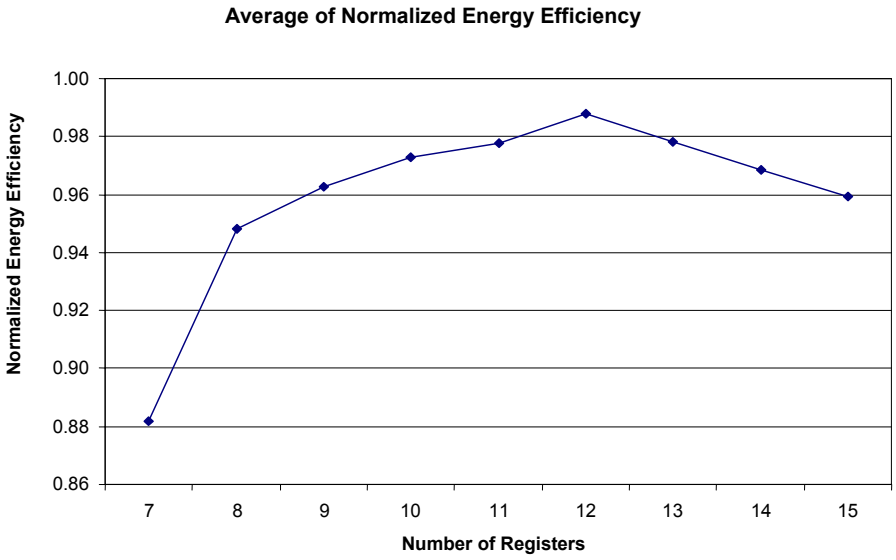


Figure 20: Average of normalized energy efficiency.

2.9. Chapter Conclusion

This chapter presented an analysis method for assessing storage needs and costs of a given application automatically retargeted across a spectrum of storage configuration designs. It demonstrates how an optimal configuration for a given workload can be chosen early in the design phase based on estimates of energy and area efficiency as well as performance. Using this technique with a simple processing element showed variations of area and energy efficiency of 15% to 25%.

This technique is part of an overall strategy to increase the effectiveness of storage in embedded SIMD architectures. Additional techniques in Chapter 4 allow memory to be divided between on-chip local memory and off-chip dense memory arrays. The next chapter introduces a systolic instruction broadcast scheme that helps enable this memory structuring technique.

CHAPTER 3

Systolic Instruction Broadcast for Embedded SIMD Architectures

3.1. Summary

Traditional SIMD execution employs simultaneous global broadcast and execution of instructions. Direct implementation of this definition leads to performance degradation as broadcast wire delay becomes increasing significant in clock cycle time. In this chapter, previous techniques for eliminating long broadcast wires are extended and evaluated to achieve greater performance as SIMD array sizes increase and technology feature sizes shrinks. A systolic instruction broadcast technique is defined. Three methods to resolve inherent data dependency conflicts are defined and evaluated including a scheduling algorithm that attempts to reorder intra-block instructions to eliminate delays. An architectural simulator, implementation area estimates, and a technology modeling tool incorporating ITRS projections are used to evaluate the effectiveness of each method in terms of performance and area efficiency. Using these techniques, a 2010 technology SIMD area is projected to benefit from a 6x increase in performance and a 7x increase in area efficiency compared to a traditionally implemented system.

3.2. Introduction

Interconnection is a critical bottleneck to increasing performance of high-speed integrated circuits [66]. For embedded SIMD architectures, the greatest impact of interconnect is in instruction broadcast where each processing element (PE) simultaneously receives globally broadcast instructions from a central array controller. The delay associated with this chip-crossing broadcast will directly impact clock cycle times as technology advances. The causes of this degradation are found in the underlying technology.

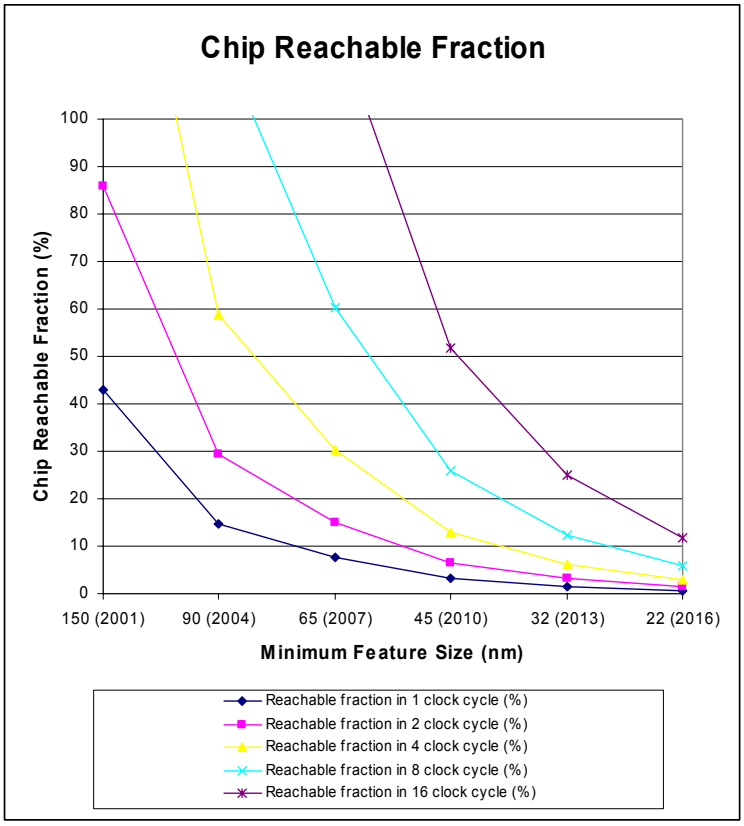


Figure 21: Reachable fraction of a chip for future VLSI technology (from ITRS).

Global wiring has been shown to cause poor scalability and long propagation delays [66, 67]. Figure 21 is from the 2001 International Technology Roadmap for Semiconductors (ITRS) [68]. This plot gives projections of the reachable fraction of a chip in a given number of clock cycles, showing that more clock cycles are needed as VLSI technology advances. In addition, continual scaling of global interconnect with increasing die size may limit the attainable clock frequencies in microprocessors [67]. Thus, it is necessary to design an alternative scheme to the global instruction broadcast in SIMD architectures. Eliminating simultaneous instruction broadcast can increase system throughput by allowing increased clock frequencies as technology advances.

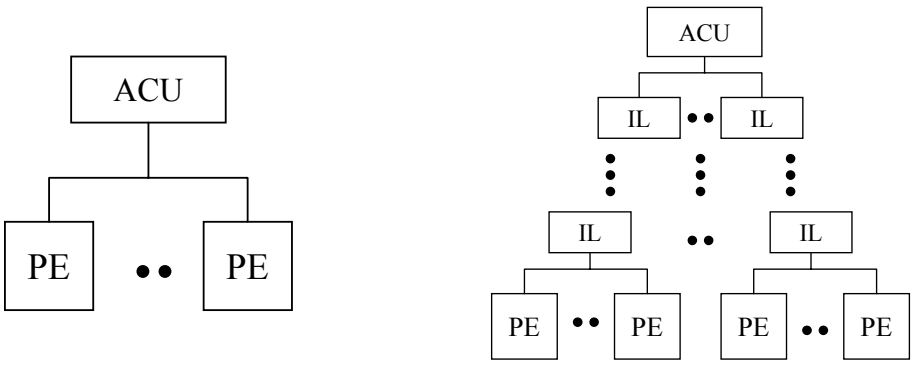
This chapter develops and evaluates a systolic instruction broadcast technique that eliminates long interconnect associated with simultaneous instruction broadcast. Systolic instruction broadcast creates the potential for data hazards during inter-PE communications. Three methods to resolve these hazards are introduced and evaluated. First, a software-only approach is presented where instructions are reordered to fill delay slots necessary to avoid the hazard. An intra-block instruction reorder scheduler is defined and implemented for the existing

register file design. A second method to resolve hazards includes the same scheduler, but now assuming an enhanced 2-read, 2-write port register file. The third method presented incorporates full hardware bypass techniques to eliminate hazards with no instruction reordering. These methods are compared using several high communication convolution kernels executing on a low pixel per processor ratio. A more complex application, median filtering with PPE = 16, is also added for comparison. These programs are simulated for each of the three methods of hazard elimination and projected performance is compared for different technology generations. A model of implementation cost (area) is then presented for each method. Using this and performance data, the area efficiency of each method is considered for future VLSI technologies using ITRS projections.

3.3. Related Work

3.3.1. Pipelined Instruction Broadcast

Pipeline instruction broadcast was first introduced in 1996 [4]. This technique uses a k-ary tree, where inside nodes are instruction latches (ILs) and leaves are PEs. This main concept is depicted in Figure 3. Pipelined instruction broadcast can reduce the instruction propagation delay by shortening the effective bus length. By setting cycle time equal to the time required to drive the load, we can determine the fanout of each node in the tree. Once fanout is determined, the depth of the tree is determined for the number of PEs.



(a) Global broadcast (b) Pipelined broadcast

Figure 22: Two methods for delivering instructions to PEs [4].

The pipeline instruction broadcast mechanism implemented is based on performance requirements, the number of PEs, and the target technology. As technology or system

configuration changes, the tree structure must be revised in terms of fanout and pipelining depth. The required number of latches in a tree becomes significant for large processor arrays. Thus, a more scalable technique is required for future systems.

3.3.2. Instruction Systolic Architecture (ISA)

This section discusses instruction systolic architecture (ISA), Systola 1024 (ISATEC Co.), which incorporates systolic instruction distribution [27,28,29,30]. The overall architecture is depicted in Figure 23. This architecture is based on a SIMD architecture that applies a single operation over different data in many PEs. Thus control flow in an instruction systolic architecture is defined by the instruction moving through the entire processor array as in Figure 24.

For flexibility, Systola 1024 uses a row and a column selector to execute the instruction on each PE, based on the values in these selectors. A moving instruction is executed on a PE when both values from selectors are 1. Thus by considering different selections of these values, each PE can execute a different set of instructions. Unfortunately, Systola 1024 does not support the conditional jump instruction, which is necessary to program a greater variety of algorithms. Systola 1024 has two dedicate communication registers in each PE which are shared by the neighboring PEs. The value in this register can be seen by its four neighbors – North, South, East and West. The structure of the communication register is shown in Figure 25.

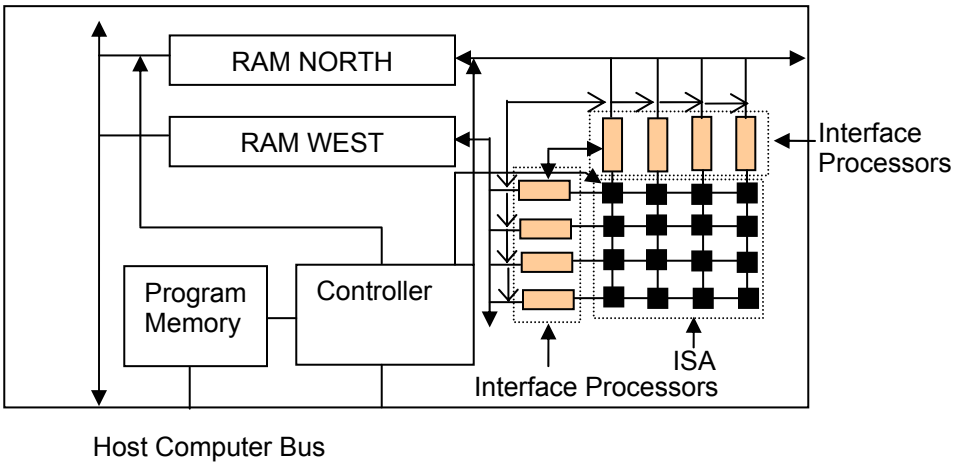


Figure 23: The architecture of Systola 1024 [30].

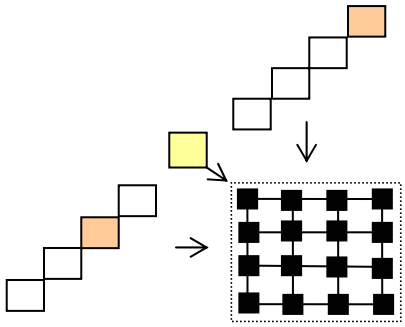


Figure 24: Control flow in an instruction systolic array [30].

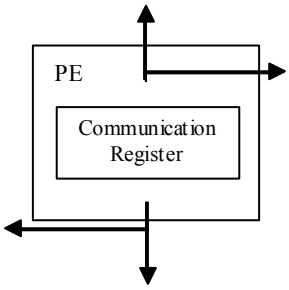


Figure 25: The structure of the dedicated communication register in Systola 1024 [30].

As shown above, the Systola system is controlled by moving instructions from the central controller through entire processor array. Thus the careful writing of applications is critical for correct functionality. However, this is not trivial since many PEs can perform different jobs at any time. In the Systola system, the existence of two selectors – column and row – simplifies the control by sacrificing the overall performance. Since performance degradation can be reduced by proper scheduling of instructions, scheduling efficiency becomes important. The complexity of this job resides in timing constraints in terms of instruction arrival time and data available time. Thus an automatic scheduler, incorporating data sequencing is necessary to increase the programmability of applications on this kind of architecture while ensuring the correctness of the applications. However, the Systola system is not supported by an automatic scheduler or data sequencer during application developments.

3.3.3. Systolic Instruction Broadcast

In this section, a nontraditional architectural approach, introduced in [1], is presented that can minimize wire delay produced by long broadcasting buses. It can reduce instruction cycle

time and improve the overall performance significantly. Short-wire instruction broadcast uses an approach similar to ISA [30]. Instructions are propagated to neighboring PEs systolically in each clock cycle [1]. The longest distance traversed by any instruction in a single clock cycle is limited to the maximum distance between any two PEs. This approach exploits locality and delivers instructions efficiently. Hence, a nontraditional instruction broadcasting mechanism reduces instruction cycle time. Instruction cycle time is no longer determined by long instruction broadcasting wire delays, but instead by the critical path among PEs.

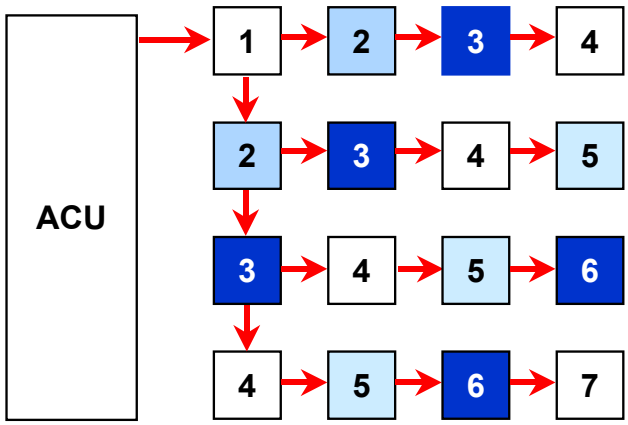


Figure 26: A 4x4 mesh of PEs showing how instructions are pumped from an ACU to PEs [1].

An array control unit (ACU) fetches instructions from instruction memory and decodes them, broadcasting instructions initially to a node located in one of four corners of the mesh array (the upper-left corner in this case). Then, each PE pumps these instructions to its neighboring PE through instruction channels, as illustrated in Figure 26. The numbers in each PE indicates the clock cycle at which a node gets a particular instruction. As shown in the figure, the number increases from left to right and from top to bottom. It takes (2N-1) clock cycles to reach a node located in the lower right corner of mesh, where N^2 is the number of nodes in the mesh network.

This approach is more scalable than pipelined instruction broadcast [1,4]. Since pipelined instruction broadcast is a tree based instruction broadcast mechanism, the tree structure has to be changed as the number of PEs is changed. However, because short-wire broadcast uses a 2D approach, it is possible to add more PEs without modifying the existing instruction broadcasting network by facilitating network expansion.

Despite the overhead to implement this method, the short clock cycle time may compensate for the increased clock counts. To maximize the efficiency of systolic instruction broadcast SIMD architectures, effort should be placed on scheduling to minimize the delays due to the

synchronization of the PEs. However, there has been no research until now on compilation-based scheduling. Thus we develop compile-time scheduling algorithms, which can maximize the efficiency of our architecture. In addition, we present a data forwarding hardware method to eliminate delays due to communication between neighboring PEs and data dependencies.

3.4. Approach: Systolic Instruction Broadcast Architecture

An overall target architecture of the SIMD-systolic system is depicted in Figure 27, for a 4 x 4 processor array.

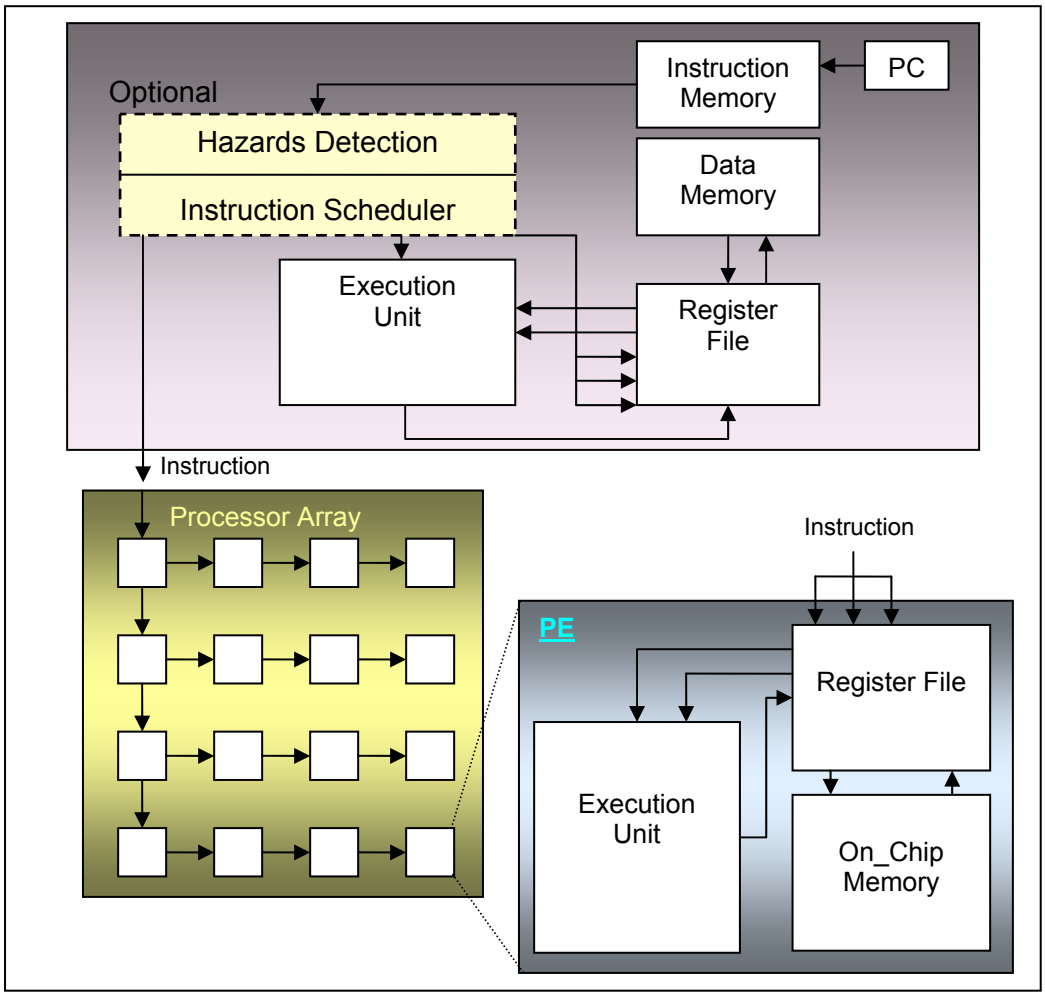


Figure 27: The overall framework of the SIMD-systolic architecture.

In this system, distributed instructions arrive at each node in a staggered pattern as in Figure 26. The SIMD-systolic system simplifies instruction distribution, shortens total wire demand, and smoothes node bandwidth demands while increasing instruction latency time. It takes (2N-1)

clock cycles to reach a node located in the lower right corner of the mesh, where N^2 is the number of nodes in the mesh network. However, the SIMD-systolic system can improve instruction throughput. A similar technique is used in the Systola 1024 system [30] discussed in Section 3.3.2. The differences between Systola and our approach lie in instruction scheduler and data sequencer. The next two sections will describe how to extend SIMD architectures to support systolic instruction broadcast efficiently.

3.4.1. Data and Structural Hazards

Under the systolic instruction broadcasting mechanism, no changes are necessary for instructions that process local data. However, instructions that communicate with the neighboring PEs need to check data dependencies and resource conflicts to ensure the correctness of program results. In this section, we describe how these problems are handled in our approach.

3.4.1.1. Nearest Neighbor Communication

First, we explain how PEs communicate with each other to pass the data to their neighbors. In our system, a North-East-West-South (NEWS) network is used for communication. Figure 28 shows how the PEs communicate each other using a NEWS network.

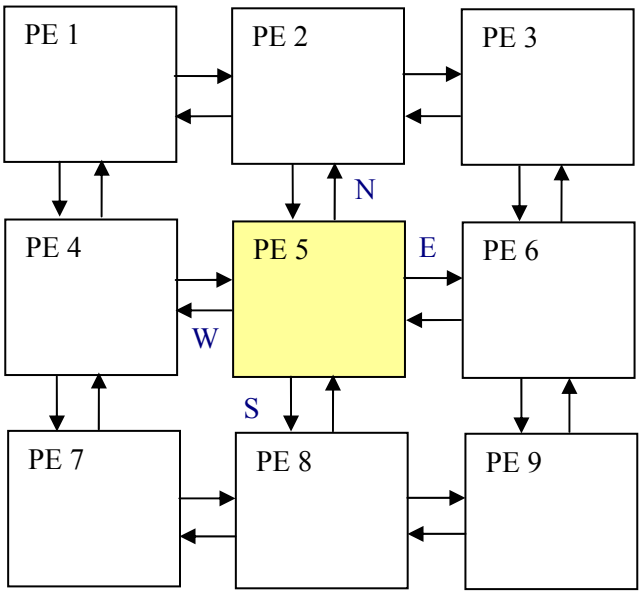


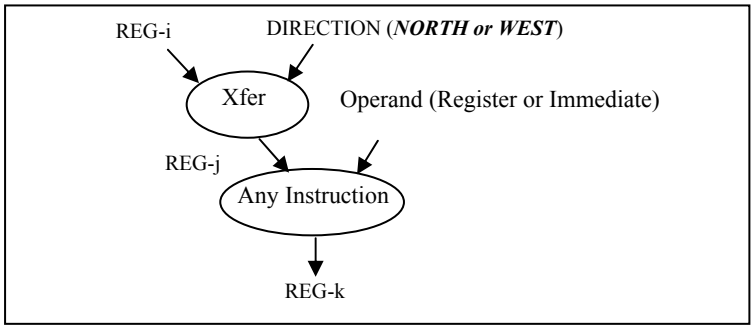
Figure 28: Neighboring PEs showing how data is transferred.

The communication instruction (‘*XFER*’ in SIMPil) uses four directions – north, east, west, and south - to pass/get the data from neighbors. This instruction can be executed in one clock

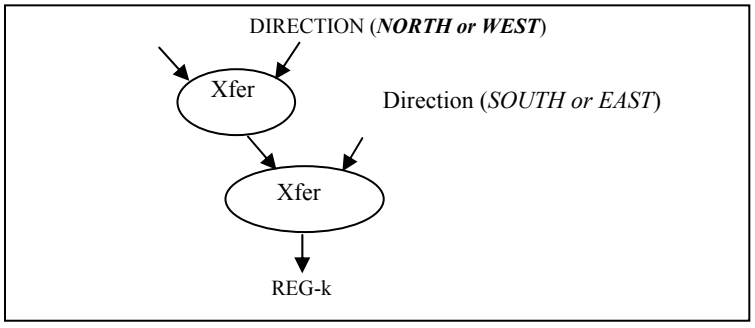
cycle and the transferred data can be copied from/to the register file of neighboring PEs directly without buffering. The directions of communication are given explicitly in each *XFER* instruction.

3.4.1.2. Data and Structural Hazard Analyses

For the given systolic instruction broadcast (Figure 26) and communication mechanisms (Figure 28), SIMD-systolic system should be stalled 1) when communication instructions are executed in an opposite direction to that of systolic instruction broadcast or 2) when neighboring PEs attempt to use the same data channel. The former case is referred as a data hazard and the latter is called a structural hazard. The conditions of these hazards are shown in Figure 29 with the given pattern of systolic instruction broadcast depicted in Figure 26. The solution to these problems is reordering of instructions so that conflicted instructions are not scheduled consecutively. If instruction reordering cannot eliminate conflicts, no-op delays are inserted to resolve the hazards.



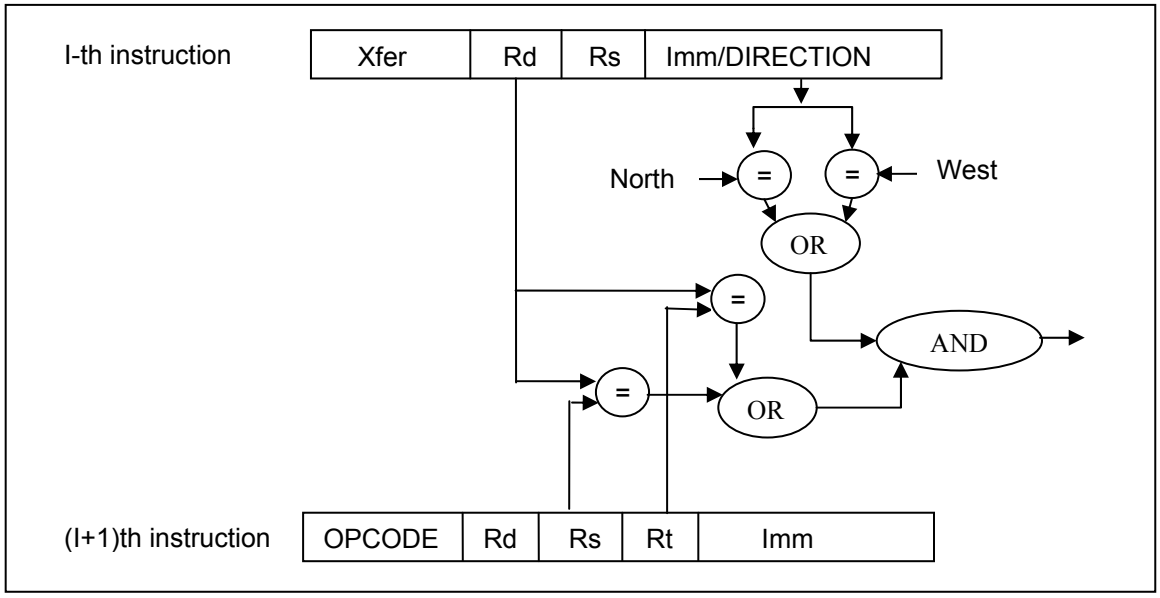
(a) Data hazard.



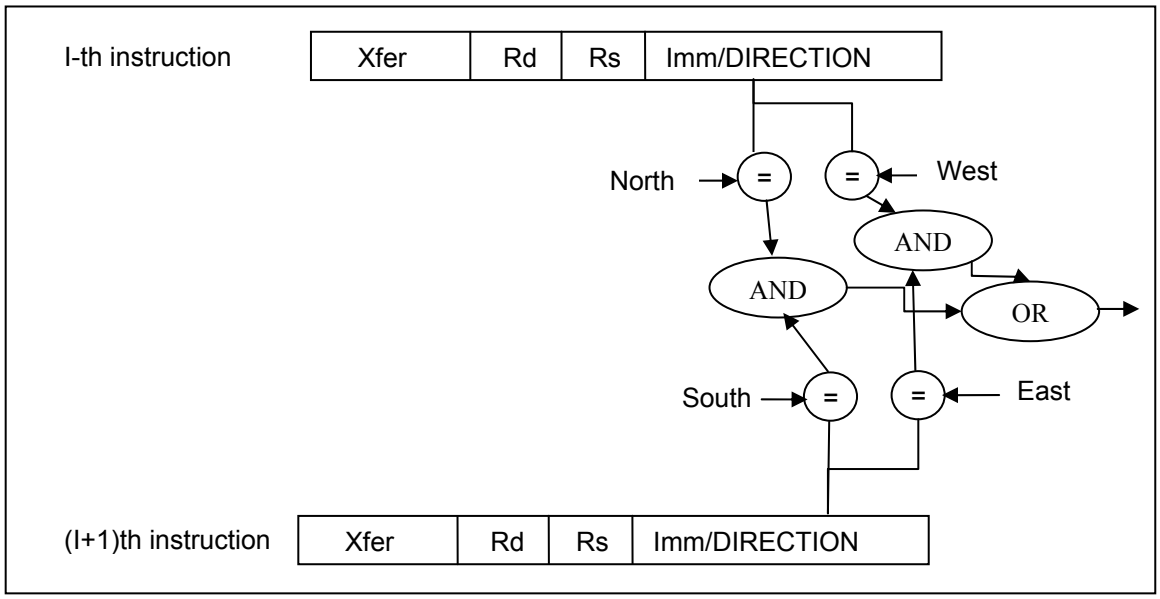
(b) Structural hazard.

Figure 29: Conditions causing delays in systolic instruction broadcast.

Data and structural hazards also can be detected in hardware logic as depicted in Figure 30.



a) Data hazard detection logic.



b) Structural hazard detection logic.

Figure 30: Hazard detection logic.

3.4.2. Implementation of Systolic Instruction Broadcast

In this section, we show three approaches to implementing systolic instruction broadcast – a basic software approach, a two write-port register file approach, and a bypass hardware implementation. Software approaches (the first two approaches in the following sections) should employ instruction scheduling techniques to eliminate as many delays as possible that result from the systolic instruction broadcast. Instruction scheduling techniques will be described separately in Section 3.4.2.3 after explaining the basic mechanisms of the two software methods.

3.4.2.1. Software Approach

Our first approach is a software method to implement the systolic instruction broadcast by utilizing an instruction scheduler. As shown in Section 0, systolic instruction broadcast can cause some nop-delays due to data dependencies and resource conflicts resulting from communication instructions. In addition, a communication instruction with opposite direction to the instruction broadcast will write the transmitted value to the register file at cycle time, t+1 where t is the instruction arrival time on that node. Thus the consecutive instruction which attempts to write the data to the register file cannot proceed if there is one write port in the register file. The code in Figure 31 shows an example of this case.

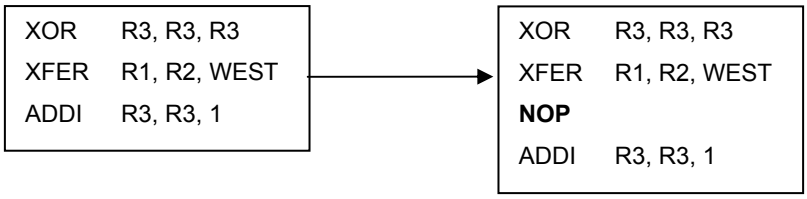


Figure 31: Sample code that introduces a delay due to a write following a communication instruction.

In addition, the clock cycle time has been split into a RD cycle for a register read and a WR cycle for a register write as in Figure 11. By splitting the clock cycle, transferred data can be written after the transmitted data has been read. Thus, it is possible to transfer the data to neighboring PEs using the NEWS network in one clock cycle without buffering. Figure 33 shows the timing of data and instruction arrivals as an example.

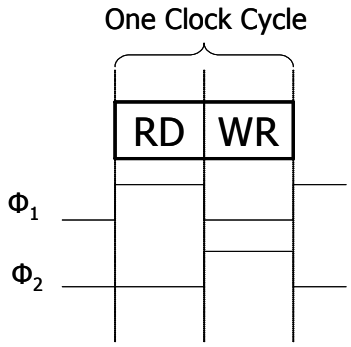


Figure 32: Clock cycle splitting – RD for a data read from register file and WR for a data write to register file.

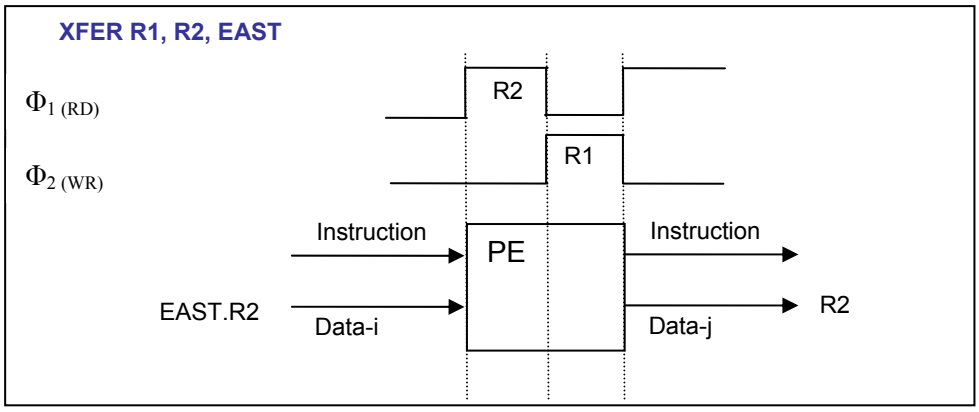


Figure 33: An example of an execution of a communication instruction with split clock cycle

3.4.2.2. Two Write-port Register File Method

Our second approach has the same basic concept as the first approach except multiple write ports (i.e., two in this case) exist in a register file. By having two register write ports, two simultaneous writes to register file – one for transferred data from neighboring PE and the other for the execution result from a current instruction – can be achieved without any stall as depicted in Figure 34.

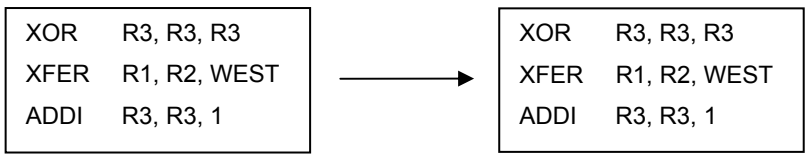


Figure 34: Sample codes that eliminate a delay due to a following write after communication instruction by having two write register ports.

The software method is supported by an instruction scheduler to reduce instruction stalls by replacing nop-delays with meaningful instructions without affecting application results. Instruction scheduling techniques implemented for a SIMD-systolic system are described in following section, Section 3.4.2.3.

3.4.2.3. Instruction Scheduling Techniques for Systolic Instruction Broadcast

The overall framework of our instruction scheduler is shown in Figure 35. It consists of four main tasks – hazard detection and resolution, simultaneous writes check, data flow analysis, and delay reductions, which are described as follows.

- **Hazard Detection and Resolution**:

First, the instruction scheduler checks hazards (data hazards and structural hazards) based on the instruction patterns shown in Figure 29. If hazards are detected, delays are added to prevent them in this step.

- **Simultaneous Register Writes Check** (Software Approach Only):

This step is necessary only for a software approach. Since, there is one write port in a register file, simultaneous writes must be avoided. Thus only instructions that do not attempt writing results to a register file can proceed consecutively to the communication instruction. Otherwise, delays are added to prevent simultaneous register writes.

- **Data Flow Analysis**:

The first two steps may produce some amount of delays to ensure the correctness of application results. However, delays should be minimized while preserving application correctness. In our instruction scheduler, data dependencies are analyzed to select a candidate instruction that does not have any dependency with other instructions between 'NOP' and the candidate instruction.

- **Delay Reduction**:

Based on data flow analysis, delays are minimized by replacing 'NOP' instructions with other meaningful instructions. Since candidate instructions are chosen based on dependency information, replacing delays with such instructions does not affect application results while achieving performance improvement.

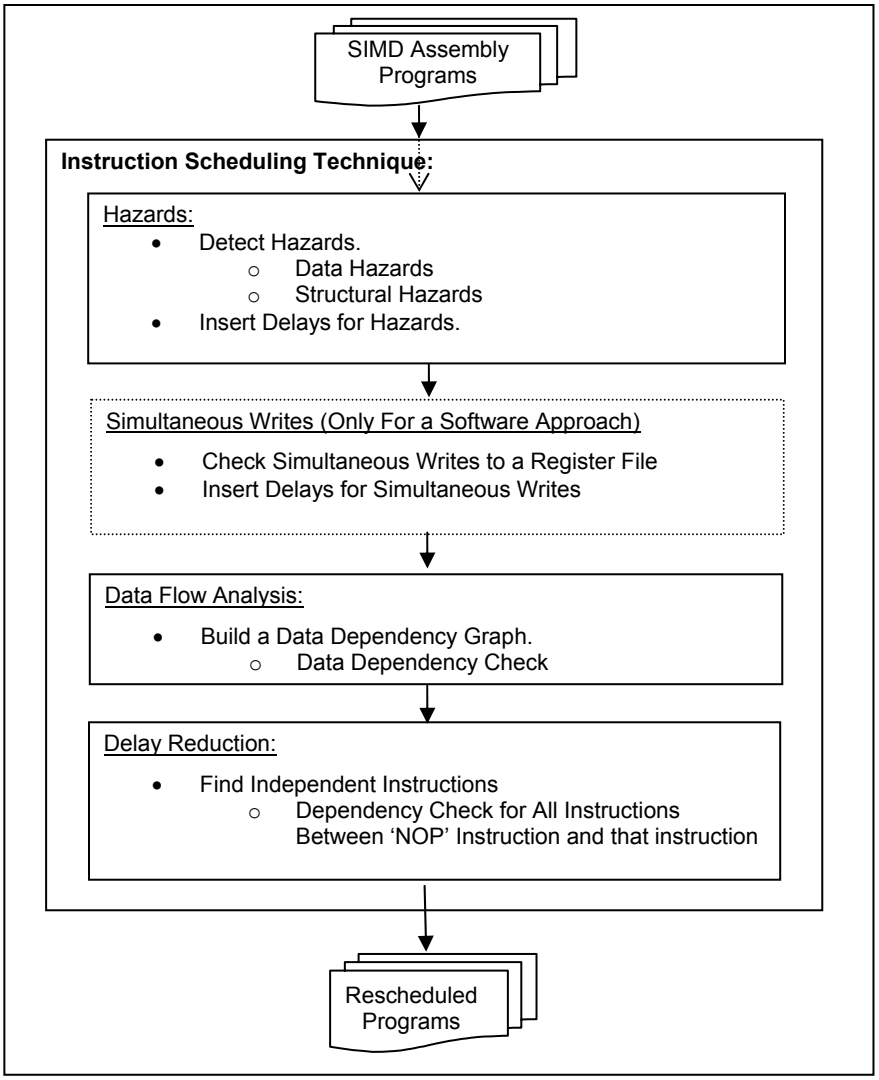


Figure 35: Framework of an instruction scheduler for systolic instruction broadcast.

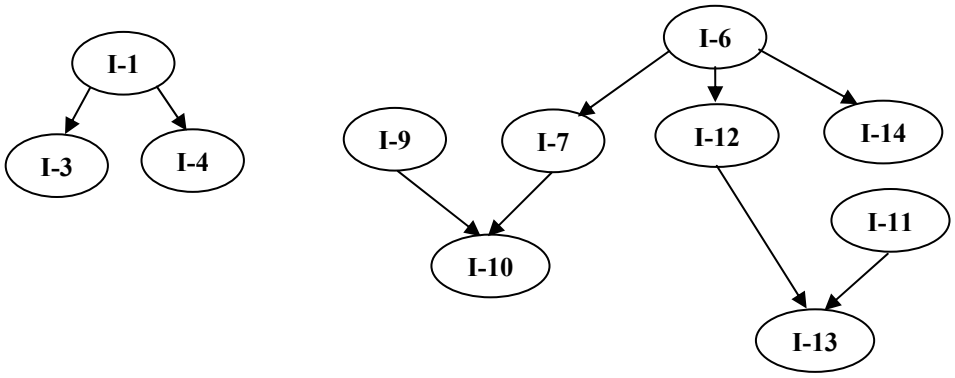
Examples of instruction scheduling techniques for the software approach and the two write-port register file method are illustrated in Figure 36 and Figure 37, respectively.

Source Code		
Code	Comment	ETC.
Xfer R2, R1 , NORTH	R2 ← R1@SOUTH	Data Hazard
Addi R3, R2, 2	R3 ← R2 + 2	
Xfer R4, R2, WEST	R4 ← R2@EAST	Simultaneous Writes to a Register File
Addi R6, R5, 2	R6 ← R5 + 2	
Xfer R7, R6, WEST	R7 ← R6@EAST	Structural Hazard
Xfer R9, R8, EAST	R9 ← R8@WEST	
Store R9, R7	MEM[R9] ← R7	
Addi R11, R10, 1	R11 ← R10 + 1	
Subi R12, R6, 2	R12← R6 – 2	
Add R13, R11, R12	R13 ← R11 + R12	
Add R14, R6, R8	R14 ← R6 + R8	

(a) Original example source code.

Source Code			
No.	Code	Comment	ETC.
I-1	Xfer R2, R1 , NORTH	R2 ← R1@SOUTH	Delay Insertion for Data Hazard
I-2	NOP	Delay	
I-3	Addi R3, R2, 2	R3 ← R2 + 2	
I-4	Xfer R4, R2, WEST	R4 ← R2@EAST	Delay Insertion for Simultaneous Writes to a Register File
I-5	NOP	Delay	
I-6	Addi R6, R5, 2	R6 ← R5 + 2	
I-7	Xfer R7, R6, WEST	R7 ← R6@EAST	Delay Insertion for Structural Hazard
I-8	NOP	Delay	
I-9	Xfer R9, R8, EAST	R9 ← R8@WEST	
I-10	Store R9, R7	MEM[R9] ← R7	
I-11	Addi R11, R10, 1	R11 ← R10 + 1	
I-12	Subi R12, R6, 2	R12← R6 – 2	
I-13	Add R13, R11, R12	R13 ← R11 + R12	
I-14	Add R14, R6, R8	R14 ← R6 + R8	

(b) Example code after delay insertion to prevent hazards and simultaneous register writes.



(c) Data dependency graph of a given program.

No.	Code
I-1	Xfer R2, R1 , NORTH
I-2	NOP
I-3	Addi R3, R2, 2
I-4	Xfer R4, R2, WEST
I-5	NOP
I-6	Addi R6, R5, 2
I-7	Xfer R7, R6, WEST
I-8	NOP
I-9	Xfer R9, R8, EAST
I-10	Store R9, R7
I-11	Addi R11, R10, 1
I-12	Subi R12, R6, 2
I-13	Add R13, R11, R12
I-14	Add R14, R6, R8

(I-2) ← (I-4)

No.	Code
I-1	Xfer R2, R1 , NORTH
I-4	Xfer R4, R2, WEST
I-3	Addi R3, R2, 2
I-6	Addi R6, R5, 2
I-7	Xfer R7, R6, WEST
I-8	NOP
I-9	Xfer R9, R8, EAST
I-10	Store R9, R7
I-11	Addi R11, R10, 1
I-12	Subi R12, R6, 2
I-13	Add R13, R11, R12
I-14	Add R14, R6, R8

(I-5) is automatically eliminated; (I-8) ← (I-11).

(d) Candidate selections based on data dependencies.

No.	Code
I-1	Xfer R2, R1 , NORTH
I-4	Xfer R4, R2, WEST
I-3	Addi R3, R2, 2
I-6	Addi R6, R5, 2
I-7	Xfer R7, R6, WEST
I-11	Addi R11, R10, 1
I-9	Xfer R9, R8, EAST
I-10	Store R9, R7
I-12	Subi R12, R6, 2
I-13	Add R13, R11, R12
I-14	Add R14, R6, R8

(e) Rescheduled program after instruction scheduling to minimize delays. (In this case, all delays are removed.)

Figure 36: An example of instruction scheduling technique for software approach.

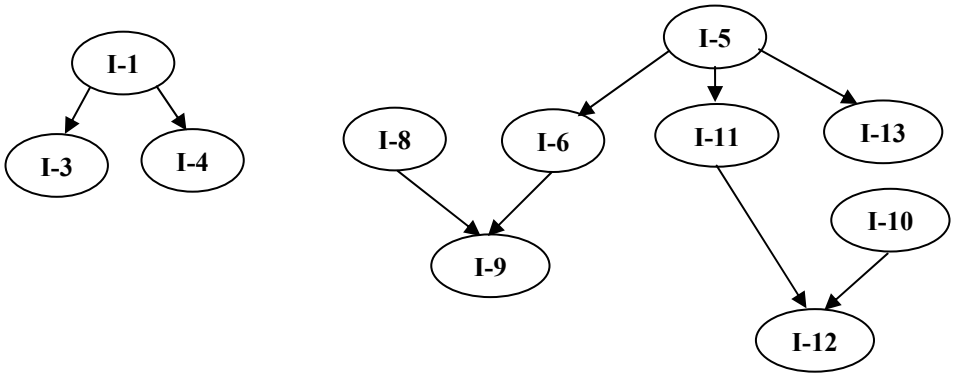
In Figure 37, the example program is the same as in Figure 36, except there is no delay between two instructions that produce simultaneous writes since the register file has two write ports.

Source Code		
Code	Comment	ETC.
Xfer R2, R1 , NORTH	R2 ← R1@SOUTH	Data Hazard
Addi R3, R2, 2	R3 ← R2 + 2	
Xfer R4, R2, WEST	R4 ← R2@EAST	
Addi R6, R5, 2	R6 ← R5 + 2	
Xfer R7, R6, WEST	R7 ← R6@EAST	Structural Hazard
Xfer R9, R8, EAST	R9 ← R8@WEST	
Store R9, R7	MEM[R9] ← R7	
Addi R11, R10, 1	R11 ← R10 + 1	
Subi R12, R6, 2	R12← R6 – 2	
Add R13, R11, R12	R13 ← R11 + R12	
Add R14, R6, R8	R14 ← R6 + R8	

(a) Original example source code.

Source Code			
No.	Code	Comment	ETC.
I-1	Xfer R2, R1 , NORTH	R2 ← R1@SOUTH	Delay Insertion for Data Hazard
I-2	NOP	Delay	
I-3	Addi R3, R2, 2	R3 ← R2 + 2	
I-4	Xfer R4, R2, WEST	R4 ← R2@EAST	
I-5	Addi R6, R5, 2	R6 ← R5 + 2	
I-6	Xfer R7, R6, WEST	R7 ← R6@EAST	Delay Insertion for Structural Hazard
I-7	NOP	Delay	
I-8	Xfer R9, R8, EAST	R9 ← R8@WEST	
I-9	Store R9, R7	MEM[R9] ← R7	
I-10	Addi R11, R10, 1	R11 ← R10 + 1	
I-11	Subi R12, R6, 2	R12← R6 – 2	
I-12	Add R13, R11, R12	R13 ← R11 + R12	
I-13	Add R14, R6, R8	R14 ← R6 + R8	

(b) Example code after delay insertion to prevent hazards.



(c) Data dependency graph of a given program.

No.	Code
I-1	Xfer R2, R1 , NORTH
I-2	NOP
I-3	Addi R3, R2, 2
I-4	Xfer R4, R2, WEST
I-5	Addi R6, R5, 2
I-6	Xfer R7, R6, WEST
I-7	NOP
I-8	Xfer R9, R8, EAST
I-9	Store R9, R7
I-10	Addi R11, R10, 1
I-11	Subi R12, R6, 2
I-12	Add R13, R11, R12
I-13	Add R14, R6, R8

(I-2) ← (I-4)

No.	Code
I-1	Xfer R2, R1 , NORTH
I-4	Xfer R4, R2, WEST
I-3	Addi R3, R2, 2
I-5	Addi R6, R5, 2
I-6	Xfer R7, R6, WEST
I-7	NOP
I-8	Xfer R9, R8, EAST
I-9	Store R9, R7
I-10	Addi R11, R10, 1
I-11	Subi R12, R6, 2
I-12	Add R13, R11, R12
I-13	Add R14, R6, R8

(I-7) ← (I-10)

(d) Candidate selections based on data dependencies.

No.	Code
I-1	Xfer R2, R1 , NORTH
I-4	Xfer R4, R2, WEST
I-3	Addi R3, R2, 2
I-5	Addi R6, R5, 2
I-6	Xfer R7, R6, WEST
I-10	Addi R11, R10, 1
I-8	Xfer R9, R8, EAST
I-9	Store R9, R7
I-11	Subi R12, R6, 2
I-12	Add R13, R11, R12
I-13	Add R14, R6, R8

(e) Rescheduled program after instruction scheduling to minimize delays. (In this case, all delays are removed.)

Figure 37: An example of instruction scheduling technique for two write-port register file approach.

3.4.2.4. Hardware Approach

Our third approach is a hardware solution. This approach utilizes the bypass (or forwarding) logic to pass the results to the dependent instruction which is executed in a neighboring PE. Figure 38 shows how the bypass logic is implemented for two, neighboring PEs.

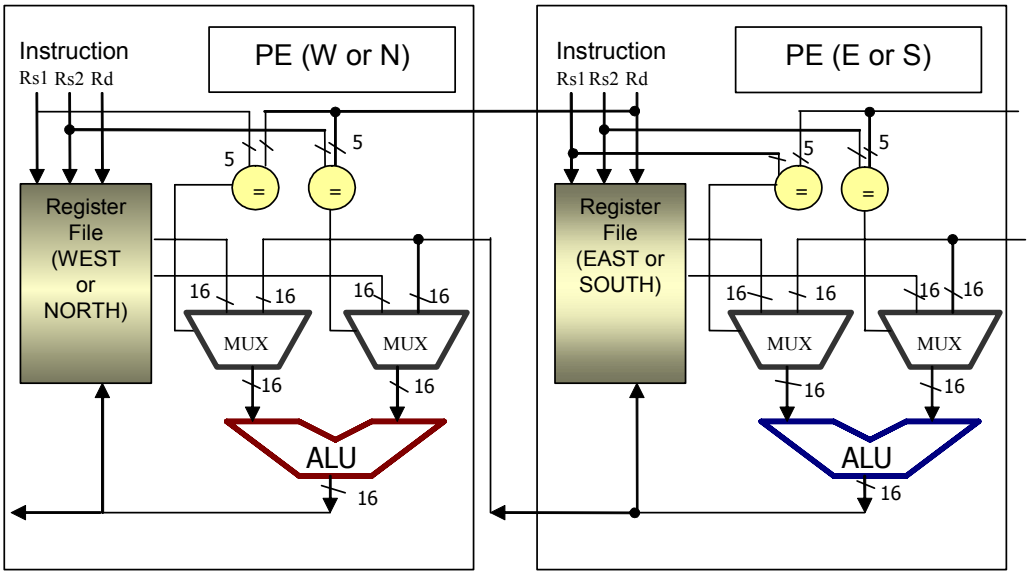


Figure 38: Bypass logic.

For one PE, results should be forwarded from two neighboring PEs (SOUTH PE and EAST PE). Thus additional hardware requirements to implement this bypass logic should be double resulting in four 32x16 MUXES, and four 5bit comparators. In addition, the register file has two write ports. As a result, the hardware solution can eliminate delays resulting from a systolic instruction broadcast mechanism with a reasonably small amount of hardware.

3.5. Results and Analysis

3.5.1. Metrics

Systolic instruction broadcast mechanism is evaluated through behavioral simulation and technology analysis. The metrics for the analysis are shown in Table 9.

Table 9: Metrics for experiments.

Analysis	Metrics
Clock Count Penalty	The number of delays due to systolic instruction broadcast in clock cycles
System Performance in Sustained Throughput	(The number of dynamic instructions + Delays) x Clock Cycle Time (= 1/ Clock Frequency) in giga-operations per second (GOPs)
Hardware Overhead in Number of Transistors	Additional Number of Transistors to support a given approach
Hardware Overhead for Register File	Additional Area for Two Write Ports in Register File in mm^2
Area Efficiency	Performance Over Die Size in $GOPs/cm^2$

The following describes how these metrics are measured in our approach.

- **Clock Count Penalty** (Systolic Instruction Broadcast) = Delays (Data Hazard) + Delays (Structural Hazard) [+ Delays (Simultaneous Register Writes)] where 'Delays (Data Hazard)' is the number of delays due to data hazards, 'Delays (Structural Hazard)' is the number of delays due to structural hazard, and 'Delays (Simultaneous Register Writes)' is the number of delays due to simultaneous writes to the register file.
- **System Performance (Sustained Throughput)** = $IPC \cdot U \cdot f \cdot N_{PE}$ where IPC is number of executed instructions per cycle, U is system utilization factor, f is a system clock frequency, and N_{PE} is the number of PEs in a given system.
- **Hardware Overhead in Number of Transistors** = Additional Number of Transistors to support a given approach.
- **Hardware Overhead for Register File** = Additional Chip Area to support a two write-port register file.

- **Area Efficiency** = $\frac{Performance}{DieSize}$ where performance is sustained throughput and die size is chip area in cm^2.

Systolic instruction broadcast can increase clock frequencies by reducing worst-case wire length. However, it may result in increased no-op delays and increased execution clock cycles. Thus to evaluate systolic instruction broadcast mechanism, the number of executed clock cycles should be considered along with clock frequencies.

3.5.2. SIMPil Applications

To evaluate the set of architectural design choices implemented in the SIMD-systolic systems, an application test suite has been simulated using a SIMD-systolic simulator that we built. The applications are selected to evaluate the impact of systolic instruction broadcast method. These well-known applications, median filtering and convolution in image processing area, are described briefly as follows.

- **Median Filtering.** Median filtering (MF) is used to remove binary noise from an image while preserving spatial resolution. A 3x3 window is used for this implementation. The algorithm works by replacing each pixel in the image with the median value in the window.
- **Convolution [**71**].** Spatial convolution is used to implement spatial filters. Spatial convolution is the method used to calculate what is going on with the pixel brightness around the pixel being processed. The equation for the spatial convolution process is as follows.

  O(x, y) = aI(x–1, y–1) + bI(x, y–1) + cI(x+1, y–1) + dI(x–1, y) + eI(x, y) + fI(x+1, y) + gI(x-1, y+1) + hI(x, y+1) + iI(x+1, y+1) where nine convolution coefficients are defined and labeled, as below:

  a b c
  d e f
  g h i

  The equation is applied for each pixel in an input image, creating corresponding output pixels.

3.5.3. Clock Count Penalty

Short-wire instruction broadcast may improve system performance because attainable clock frequency to technologies can be raised. As mentioned earlier, there can be some penalty

associated with each approach. However, the instruction scheduler can reduce penalty clock cycles by replacing them with meaningful work. Table 10 shows the number of executed clock cycles of the given applications for a typical SIMD system (described as 'Original' for tables and graphs from now on), and for SIMD-systolic systems using the three different approaches. In this table, Method-1 is a software approach with one write-port register file, Method-2 is a two write-port register file method, and Method-3 is a hardware solution with bypass logic. Numbers of instructions issued by the controller (including 'nop' instructions) are given in Table 10 for each method. The software methods (Method-1 & 2) are each depicted in two columns with the instruction scheduling technique and without it. Clock count penalty resulting from hazards for both methods and simultaneous register writes for the first method is shown in Table 11. The baseline system uses global instruction broadcast, which assumes all neighboring communication instructions complete in a single clock cycle.

Table 10: Number of instructions issued by controller.

Application	Original	Method-1		Method-2		Method-3
		W/O Scheduling	W/ Scheduling	W/O Scheduling	W/ Scheduling	
Median Filtering (PPE16)	13,509	13,534	13,510	13,521	13,509	13,509
Convolution (3x3) (PPE1)	39	50	41	45	41	39
Convolution (5x5) (PPE1)	119	162	135	151	121	119
Convolution (7x7) (PPE1)	275	384	305	360	287	275

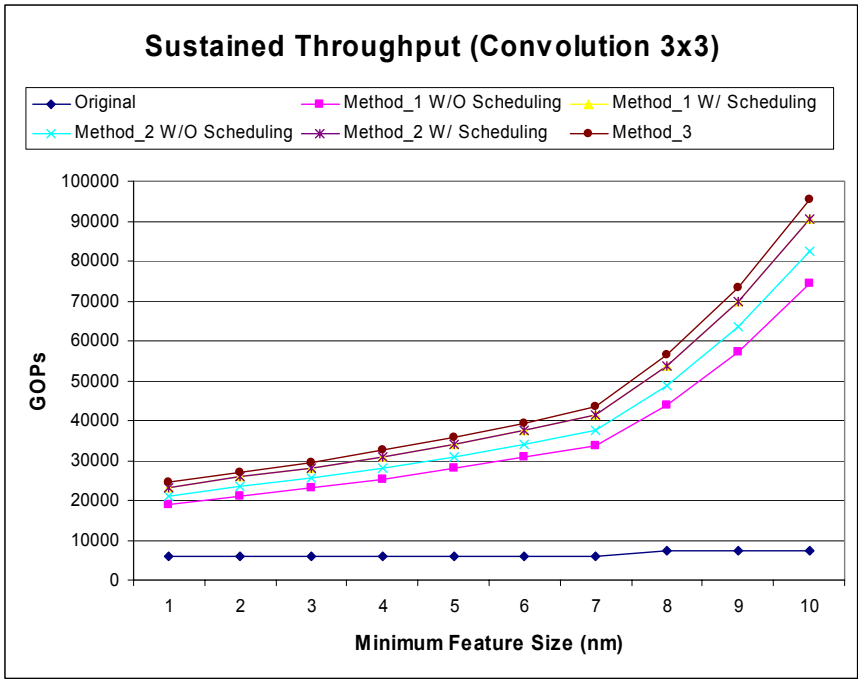
Table 11: Clock count penalties (delay cycles) of SIMD-systolic systems in three approaches.

Application	Method-1		Method-2		Method-3
	W/O Scheduling	W/ Scheduling	W/O Scheduling	W/ Scheduling	
Median Filtering (PPE16)	25	1	12	0	0
Convolution (3x3) (PPE1)	11	2	6	2	0
Convolution (5x5) (PPE1)	43	16	32	2	0
Convolution (7x7) (PPE1)	109	30	85	12	0

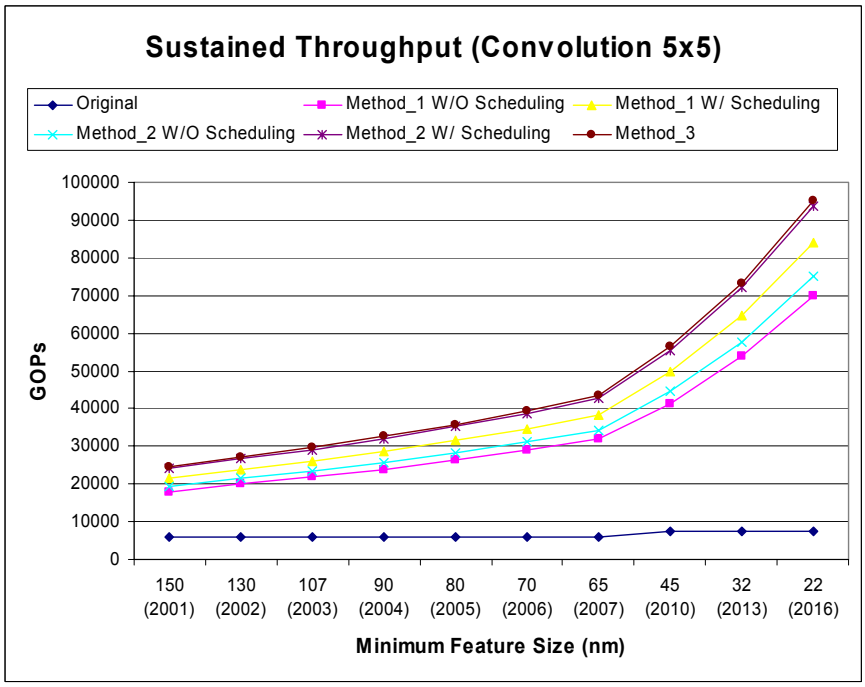
3.5.4. System Performance

Since short-wire instruction broadcast can operate at a higher clock frequency than global instruction broadcast, clock count penalty can be compensated for or even outperformed. In this section, the impact of the systolic instruction broadcast method on system performance is described in terms of sustained instruction throughput projected to technology changes. The performance of SIMD-systolic systems is computed based on a lower bound of a local PE operational clock for the future technologies, and that of typical SIMD systems is based on across-chip clock frequencies projected by ITRS [68]. Figure 39 shows the sustained instruction throughput for each application where the utilization factor is assumed to be 90%.

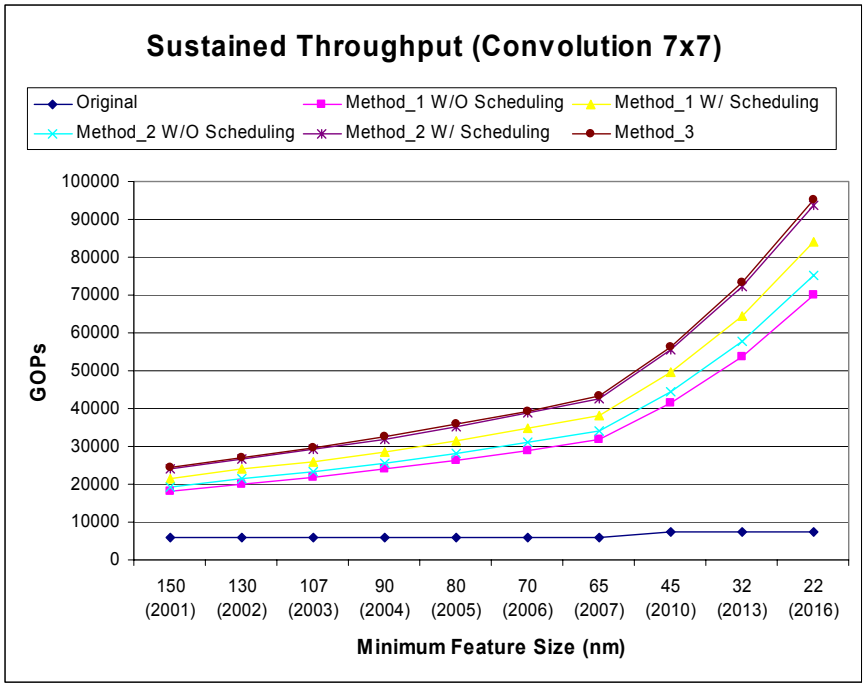
In this section, to show the impact of percentage of communication instructions for a program, we evaluate the convolution applications with three different mask sizes, which are 3 x 3, 5 x 5, and 7 x 7. As mask size increases, the number of neighboring pixels, that is required to compute the output pixel, increases. In our applications, 31%, 50%, and 61% of overall issued instructions are communication instructions and half the communications involve NORTH and WEST communications for each implementation (3 x 3 convolution, 5 x 5 convolution, and 7 x 7 convolution). As a result, as mask size increases, the number of communication instructions increases. This is shown in Table 10 and Table 11, the delay ratio, (clock count penalty / number of overall issued instructions), is increased as mask size is increased. Figure 39 shows projected sustained throughput of convolution applications with three different mask sizes for on-chip clock frequencies and off-chip clock frequencies where projected clock frequencies for low-cost systems are used in the experiments [68]. Lower bounds of on-chip clock frequencies are used to plot this figure. Figure 39 shows huge differences between sustained throughput using on-chip clock frequencies and that under off-chip clock frequencies. Even if we use lower bounds of on-chip clock frequencies, there are still big differences from that of off-chip. In addition, performance improvement can be achieved with our scheduler. Figure 40 shows the normalized sustained throughput relative to the typical SIMD system. This figure shows that SIMD-systolic system can achieve much higher (up to 7.6 times) sustained throughput for hardware method (without delay) due to short clock cycle time, up to 7.5 times of original throughput for two write-port register file with scheduler (with delays), and up to 7.2 times for a software method with scheduler for a given workload.



(a) Sustained throughput of 3 x 3 convolution application for on-chip clock frequencies and off-chip clock frequencies.



(b) Sustained throughput of 5 x 5 convolution application for on-chip clock frequencies and off-chip clock frequencies.



(c) Sustained throughput of 7 x 7 convolution application for on-chip clock frequencies and off-chip clock frequencies.

Figure 39: Projected system performance in sustained system throughput.

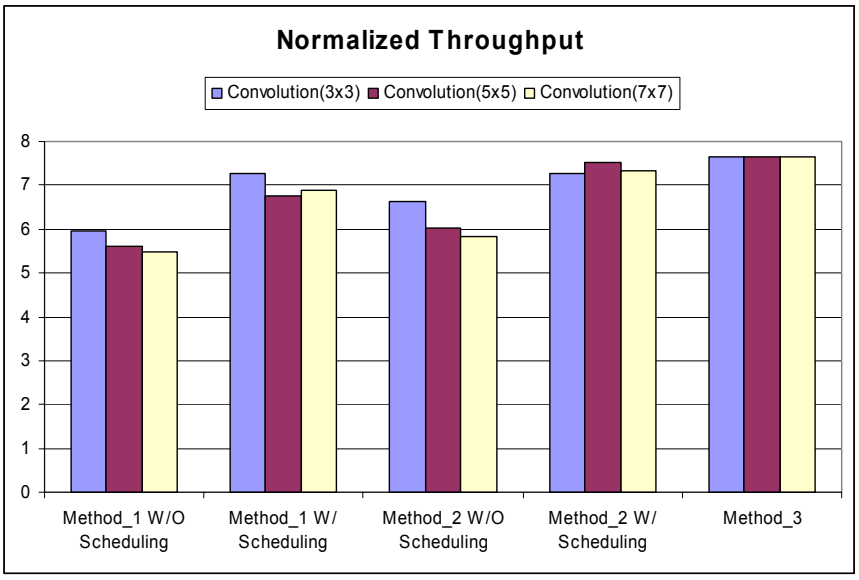


Figure 40: Normalized sustained throughput.

3.5.5. Hardware Overhead

- **Bypass Logic**

To support bypass logic for our hardware approach described in Section 3.4.2.4, four 32x16 MUXES and four 5-bit comparators are needed. To compute an additional number of transistors to implement bypass logic, we assume that we use 6-transistor 2x1 MUX and 8-transistor 2-input XNOR gates. Since a 32x16 MUX can be implemented by sixteen 2x1 MUXes, 384 (= 4 x 16 x 6) transistors are additionally required for the MUXes. In addition, since a 2-input 5-bit comparator is implemented by five 2-input XNOR gate, 160 (= 4 x 5 x 8) transistors are added for comparators. As a result, 544 transistors are required for one PE to support bypass logic in our hardware approach. Based on the datasheet of our base architecture, SIMPil16, the total number of transistors used in one PE is 38,590. As a result, to support bypass logic a 1.4% overhead in transistor count for one PE is observed. Based on average area for one transistor in SIMPil, especially for the logic part (other than memory and register file), the increased number of transistors under 0.8μm technology occupies 0.11 mm^2 resulting in a 1.7% area increase.

- **Register File**

Except for the first software approach, we use two write-port register file for simultaneous register writes. Additional write ports in the register file results in increased of register file size. Base architecture uses two read ports and one write ports. This is shown in Figure 41 based on the register file model in [69].

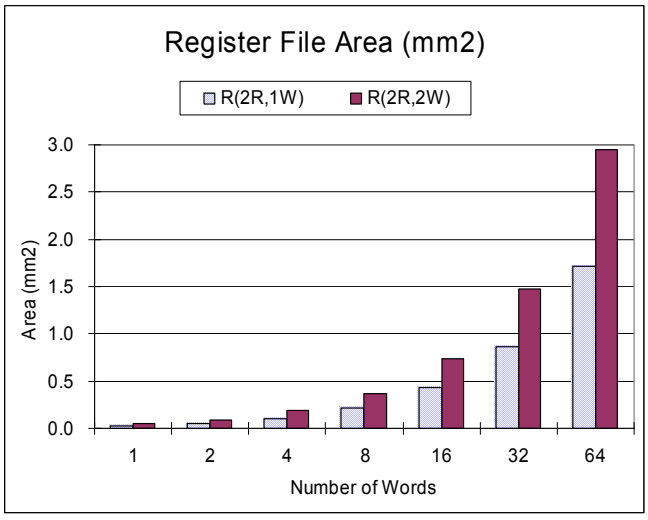
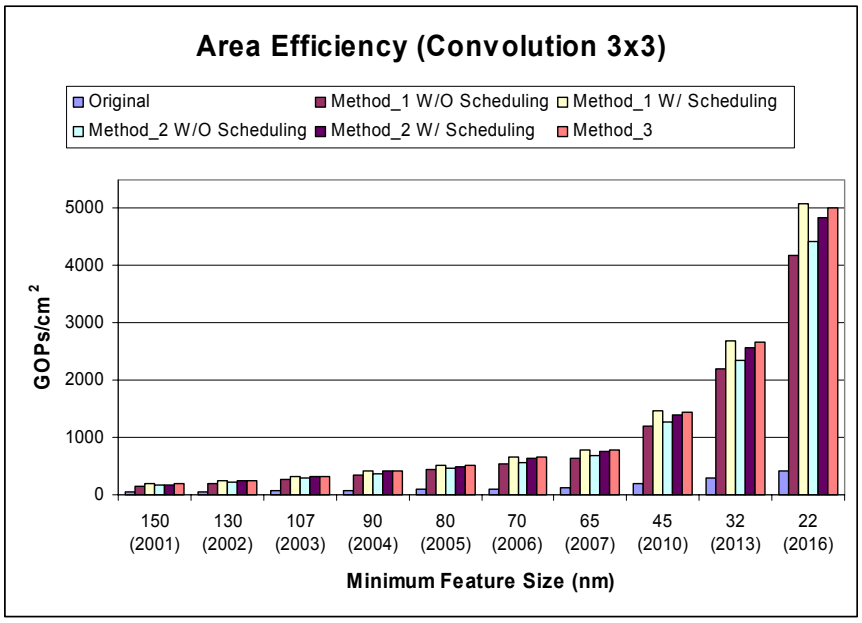


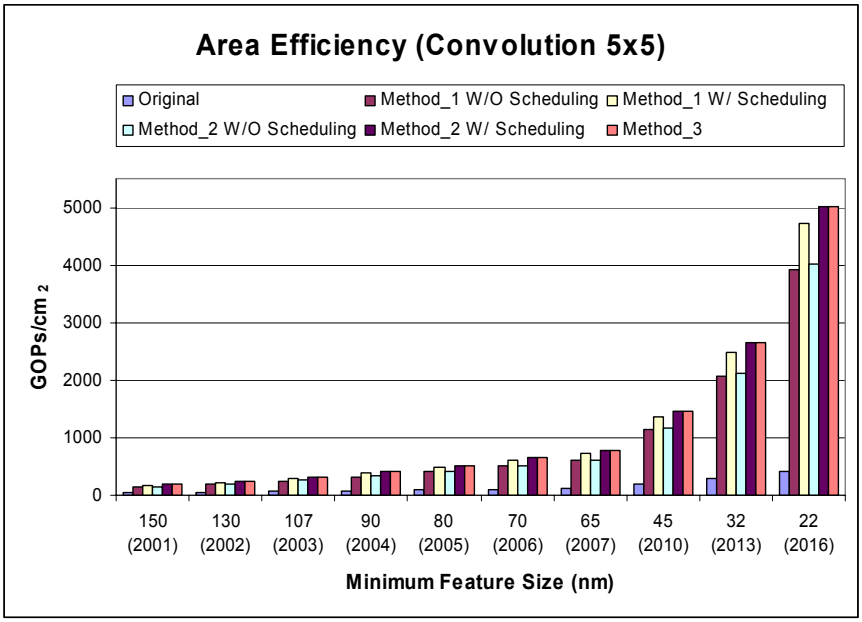
Figure 41: Register file size increase for an additional write port: R(2R,1W) – 2 read-port, 1 write-port register file; R(2R,2W) – 2 read-port, 2 write-port register file.

3.5.6. Area Efficiency

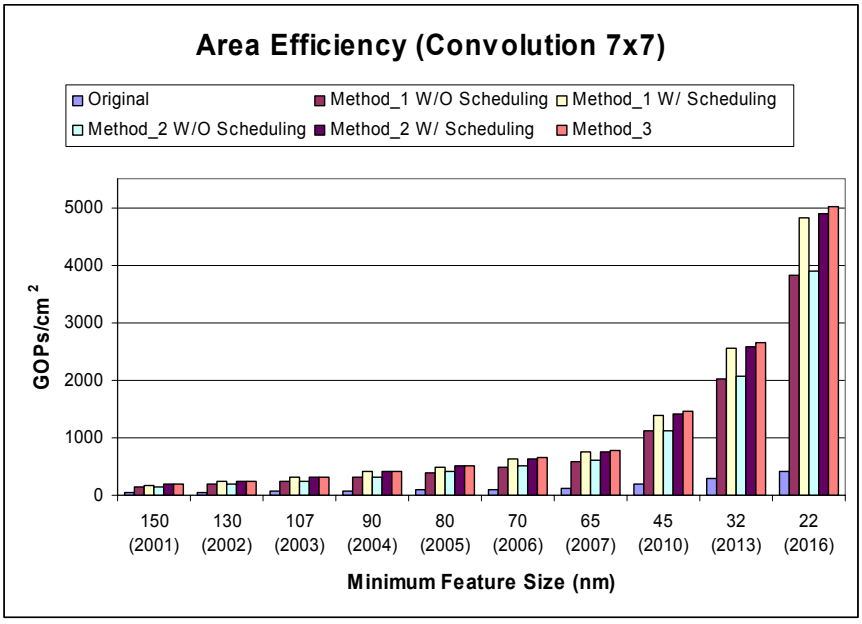
Area efficiency considers die size and performance together. Shorter instruction broadcast wires allow clock frequencies to operate near ITRS projected local interconnect levels. Thus the area efficiency for systolic instruction broadcast systems will be significantly higher than the baseline architecture. Figure 42 shows area efficiencies corresponding to the sustained throughput in Figure 39. As shown in Figure 42-(a), for programs with low levels of inter-PE communication, the software method can achieve the highest area efficiency. However, as communications between neighboring PEs increases, higher efficiency is obtained with added hardware. As a result, hardware implementation (bypass logic) achieves the highest area efficiency in Figure 42-(c) where applications spend the most time to communicate with neighboring PEs. In addition, the scheduler can effectively improve the area efficiency for both software methods. In any case, systolic instruction broadcasted systems have much higher area efficiency (up to 7.2) relative to the baseline SIMD system as shown in Figure 43.



(a) Area efficiency of 3 x 3 convolution application for each system.



(b) Area efficiency of 5 x 5 convolution application for each system.



(c) Area efficiency of 7 x 7 convolution application for each system.

Figure 42: Area efficiencies in (GOP/mm^2).

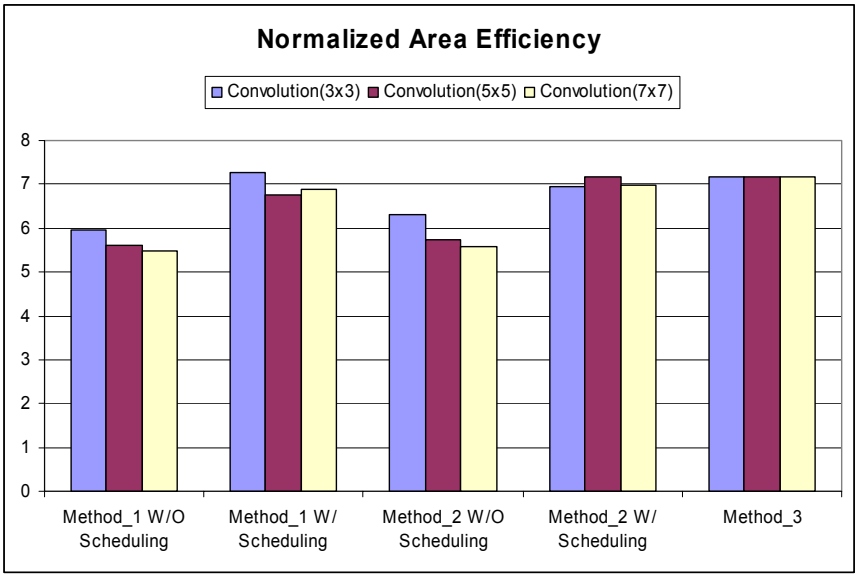


Figure 43: Normalized area efficiency projected in year 2010.

3.6. Chapter Conclusion

A high performance and area efficient instruction broadcasting scheme with short-wire interconnects was presented. Short-wire instruction broadcast overcomes the wire latency bottleneck found in global instruction broadcast. Three systolic instruction broadcast methods have been presented and evaluated. For software methods, the scheduler improves the area efficiency for a given workload. We can choose the method for a set of particular applications by analyzing area efficiencies. In any case, systolic instruction broadcasted systems have much higher area efficiency (up to 7.2) relative to the typical SIMD system.

In the next chapter, the temporal and special instruction execution order resulting from systolic instruction broadcast is used to enable a low overhead off-chip memory access scheme.

CHAPTER 4

Systolic Virtual Memory

4.1. Summary

While local PE memory provides the lowest access latency and highest bandwidth to the PE's datapath, local memory cells are far larger per bit than in dedicated dense memory array chips that benefit from specialized processes and amortized support circuitry. A monolithically integrated SIMD PE array would be significantly less expensive in area and cost if a portion of local PE memory could be relegated to off-array dense memory chips. However limited memory access bandwidth and increased access times pose obstacles to this approach. The staggered spatial execution resulting from the systolic instruction broadcast technique presented in the last chapter offers a new opportunity for utilizing off-array dense memory chips. This chapter presents a systolic off-array memory access scheme called Systolic Virtual Memory (SVM) where off-array addresses are scheduled to allow instruction and data operand rendezvous at the PE using a separate instruction and memory delivery network. A scheduling algorithm is presented and new controller hardware is described. Several large data memory kernels and synthetic test programs are used to evaluate the proposed system. Area models for storage alternatives are developed and the systolic virtual memory technique is compared to a local memory-only system in terms of performance, area, and area efficiency. For a high memory access kernel (matrix multiplication), this technique offers a 30% - 50% reduction in memory area and 20% - 50% increase in area efficiency for only a 20% execution time penalty.

4.2. Introduction

Local PE memory arrays provide high access, low latency operand storage. They also bring high area costs in an often expensive high-speed digital VLSI process. Dense off-array memory chips offer large data storage arrays at very low cost. This chapter considers an approach to combining these two storage mediums in a scheme similar to virtual memory that combines fast semiconductor memory with dense magnetic disk storage. High reference locality in the register file and local memory and a load/store instruction set results in a manageable number of off-array memory accesses. However SIMD's typical synchronous instruction execution concentrates off-array accesses overwhelming the memory access bandwidth of commercially available dense memory chips. Systolic instruction broadcast staggers instruction execution enabling systolic off-array memory prefetching. When memory addresses are not data dependent, accesses can be prefetched in advance so that requested memory access can arrive at the appropriate PE simultaneous with the instruction. This organization is illustrated in Figure 44. In this example, each PE in a 4 x 4 SIMD array requests a different memory location in off-array memory. Data access begins 4 cycles before the first node executes the systolic load instruction. This allows data to travel from the bottom edge of the array to rendezvous with the load instruction on a particular PE. Figure 45 shows an example where instructions and corresponding data rendezvous in each PE at a different clock cycle. The architecture is simplified in this example and it is assumed that the first instruction is executed at the first clock cycle at the first PE.

This chapter continues with a summary of related research that serves as the foundation for this work. Then the details of systolic virtual memory are presented including required modification to the controller and instruction set. A VLIW style instruction is defined where an independent PE and memory controller operation is bundled at each code location. A scheduling algorithm using linear mapping is defined and implemented. Then several kernels that require significant PE storage are simulated and evaluated. The evaluation metrics are execution time, delay cycles, required memory and corresponding memory area, and area efficiency. The presented systolic virtual memory system is compared to an all-local memory system. Although the SVM system requires 20% longer execution time, it reduces storage area by as much as 50%, with a similar improvement in area efficiency.

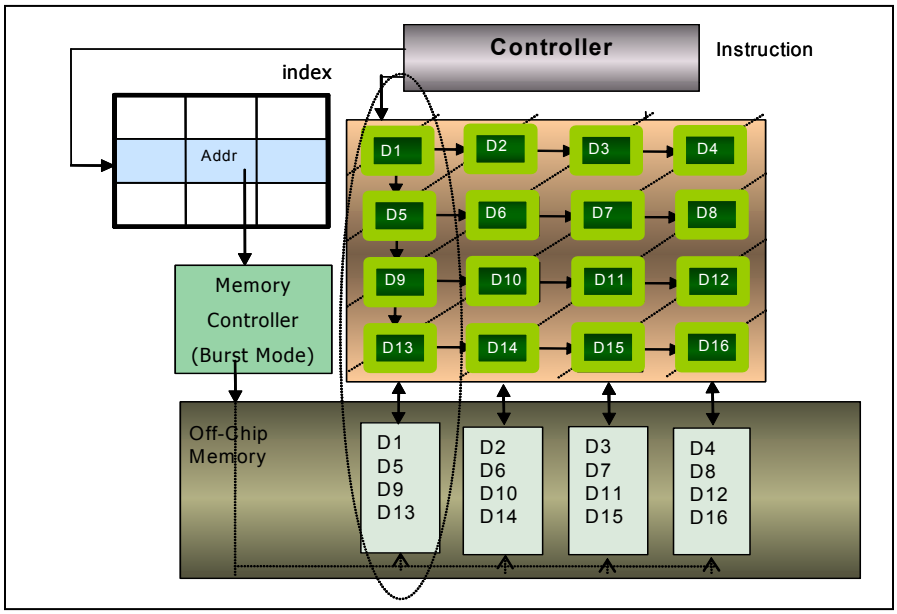


Figure 44: Mechanism of systolic instruction broadcast and systolic data movement.

4.3. Related Work

4.3.1. Linear Mapping Technique

Linear mapping techniques are widely used to design systolic systems for specific applications [14,16,17,19]. This technique can determine how operands are distributed through a processor array, including their speed and direction. Definitions used in the linear mapping are shown in Table 12 followed by a description of transformation methods.

Table 12: Definitions for linear mapping method.

Term	Definitions
Dependence Graph (DG)	A directed graph that shows the dependences of computations in an algorithm where the nodes in DG represent computations and edges represent the precedence constraints among nodes.
Regular DG	DG which has the same directional edges at all nodes in the DG.
Projection vector (or iteration vector), $d^T = (d_1, d_2)$	Two nodes that are displaced by d or multiples of d are executed by the same processor.
Processor space vector, $p^T = (p_1, p_2)$	Any node with index $I^T = (i, j)$ would be executed by processor $p^T I = (p_1, p_2)(i\ j)^T$.
Scheduling vector, $s^T = (s_1, s_2)$	Any node with index I would be executed at time, $s^T I$.
Hardware Utilization Efficiency, $HUE = 1/\|s^T d\|$	Two tasks executed by the same processor are spaced $\|s^T d\|$ time units apart.

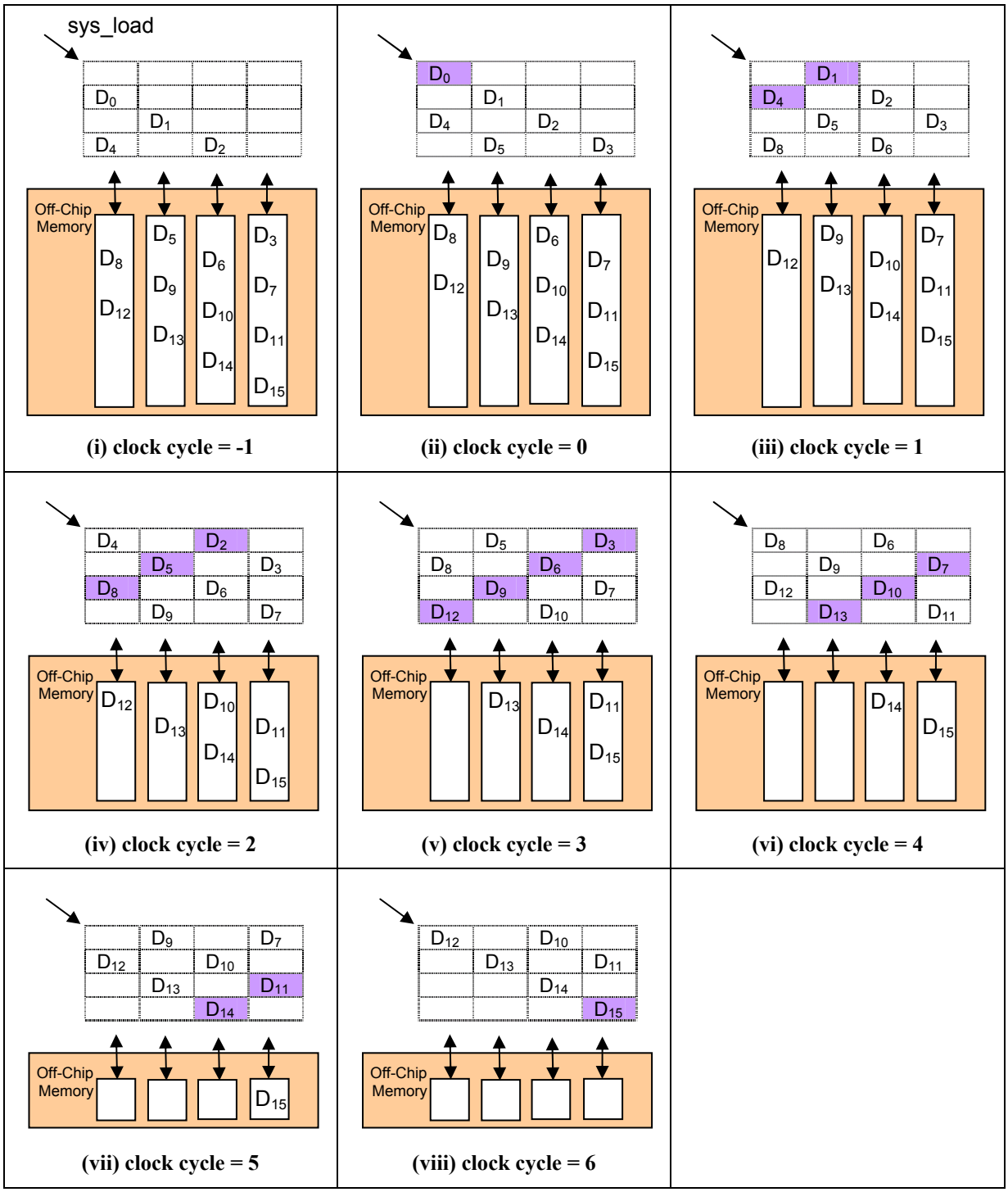


Figure 45: An example of data movement with systolic load instruction.

Transformations based on the dependence graph (DG) are performed by the following steps.

- Step 1: Build a regular dependence graph which is a space representation.
- Step 2: Transform by mapping DG from space to space-time representation.
- Step 3: Design various systolic systems for a given problem by selecting different sets of vectors: projection vector, processor space vector and scheduling vector.

We can choose one system based on the hardware utilization efficiency. The sets of vectors chosen should fulfill constraints to preserve the correctness of the designed system. The following descriptions are details of each step in the transformations.

Step 1: Build a regular Dependence Graph

The DG is built by creating a new node whenever a new computation is necessary in an algorithm. No node is ever reused on a single computation basis. In a regular DG, the presence of an edge in a certain direction at any node in the DG represents the presence of an edge in the same direction at all nodes.

Step 2: Transform by Mapping DG from space to space-time representation

A regular DG is a spatial representation which typically corresponds to 0 time instance. Thus to assign time instances to all computations, a mapping technique that transforms a space representation to a space-time representation is necessary. In this transformation, each node is mapped to a certain PE and also scheduled to a certain time instance. This mapping technique can map an N-dimensional DG to a lower dimensional systolic array. The transformation is based on several basic vectors described in Table 12 and is used to design many systolic systems for a specific algorithm.

Step 3: Selection of Basic Vectors

The selection of basic vectors is restricted by the following constraints to preserve the correctness of the designed system.

- Orthogonality of processor space vector and projection vector: If point *A* and *B* differ by the projection vector, i.e., $I_A - I_B$ is same as d, then they must be executed by the same processor. In other words, $p^T I_A = P^T I_B$. This leads to $p^T (IA - IB) = 0 \Rightarrow p^T d = 0$.
- Processor mapping*:* If *A* and *B* are mapped to the same processor, then they cannot be executed at the same time, i.e., $s^T I_A \neq s^T I_B$, i.e., $s^T d \neq 0$.
- Edge mapping: If an edge *e* exists in the space representation or DG, then an edge $p^T e$ is introduced in the systolic array with $s^T e$ delays.

The transformation from a space representation to a space-time representation is done by interpreting one of the spatial dimensions as a temporal dimension. For a two-dimensional (2D) DG, the general transformation is described by $i' = t = 0$, $j' = p^T I$ and $t' = s^T I$, or equivalently,

$$\begin{bmatrix} i' \\ j' \\ t' \end{bmatrix} = T \begin{bmatrix} i \\ j \\ t \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ p_1 & p_2 & 0 \\ s_1 & s_2 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ t \end{bmatrix}.$$

In the space-time representation, the j' axis represents the processor axis and t' represents the scheduling time instance.

The linear mapping technique is for the regular DG. However, a SIMD-systolic system cannot be represented by a regular DG since the borders of the processor array have irregularity in the instruction broadcasting. In addition, each PE in a SIMD-systolic system can operate the different functionalities controlled by the broadcasted instruction. Thus, instruction arrival timing also should be considered in SIMD-systolic system design. Consequently, the basic linear mapping method must be extended for use in a SIMD-systolic system design for a particular application. The architectural information should be considered to verify the correctness-preserving design for a given application. This architectural decision is based on many aspects, including technological issues such as data bandwidth limited by the pin counts.

Following section describes related techniques to hide off-chip memory access latency by data prefetching.

4.3.2. Data Prefetching Technique

The performance gap between CPU and memory systems is well-known. Figure 46 shows the increasing performance gap between processor and memory extracted from ITRS 2001 [68].

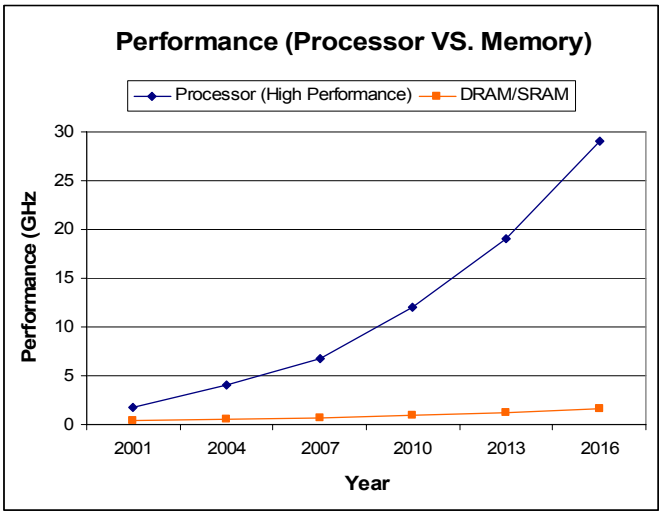


Figure 46: Performance gap between processor and DRAM (from ITRS).

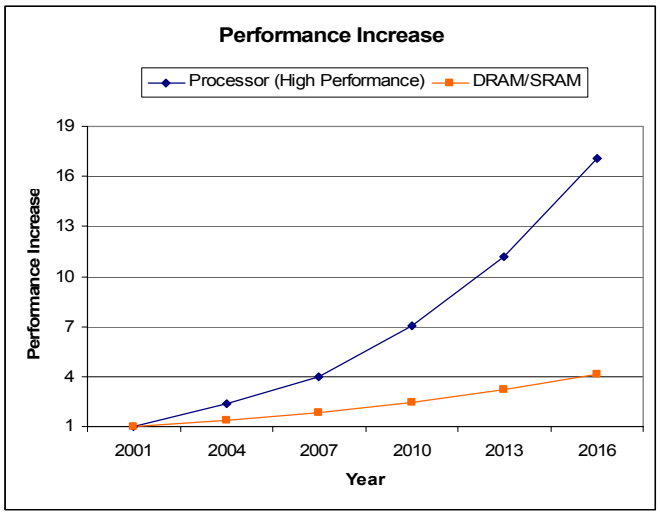


Figure 47: Performance increase rates for processor and DRAM (from ITRS).

By normalizing to the performance in year 2001, the performance gap is illustrated in Figure 47. Processor performance increases significantly faster than memory, resulting in an increasing mismatch between these system components in the future. Thus it is evident that an efficient handling of memory access is necessary to avoid a memory access performance bottleneck.

By supporting memory hierarchies, it is possible to reduce the latency of main memory accesses for frequently used data. However, access locality is not always present. For example, scientific and multimedia applications spend more than half their execution time stalled on memory requests [72]. In situations where cache misses occur, normally data fetch from main memory is initiated on demand. As a result, execution of applications having little data locality will be stalled frequently to wait until a requested cache block is fetched from main memory. To overcome this problem, data prefetch techniques have been proposed and a survey on that topic is found in [73]. Since data prefetching can be overlapped with processor computations by issuing a fetch to the memory system in advance of an actual memory reference, main memory access latency can be hidden. Data prefetching techniques can be implemented in software methods, hardware methods, or hybrid methods. Software prefetching has been widely used in many contemporary microprocessors such as PowerPC, HP PA-8000, and MIPS R10000. Usually software prefetching is supported by adding ‘fetch’ instructions in a given program. They are placed relative to the corresponding ‘load’ or ‘store’ instructions, a technique known as prefetch scheduling. Data prefetching is typically useful inside loops that perform computation on large arrays because this type of computations is common in scientific or multimedia applications. In addition, since this type of data has little data locality, cache memory cannot reduce the memory

latency time effectively. Thus, data prefetching can be utilized in this case with highly predictable patterns of data access. Software prefetching techniques introduce fetch instruction execution overhead. Alternative hardware prefetching techniques do not impose this overhead, but do require additional hardware. There have been several approaches in hardware prefetching, such as sequential prefetching [74], and prefetching with arbitrary strides [75]. Sequential prefetching can be implemented with relatively simple hardware. However, this method results in poor prefetching performance where irregular or strided access patterns are observed. Thus, other prefetching approach to support arbitrary strides has been proposed [75]. This method utilizes a reference prediction table (RPT) to hold most recently used memory operations. Stride information is computed at run time. Once this information is recorded in the RPT, the next effective address of that memory instruction is simply computed as (current effective address + stride).

Software data prefetching techniques perform better than hardware methods on irregular memory access patterns, but, unlike hardware methods, compilation effort and fetch instruction overhead are factors. Hybrid data prefetching, which integrates software prefetching and hardware prefetching techniques, has been proposed since neither of these approaches is superior in all cases [77]. Our approach is a hybrid data prefetching technique, as will be discussed in Section 4.4.5.

4.4. Approach: SIMD-systolic System with Systolic Virtual Memory

4.4.1. Systematic Design Approach

As the complexity of VLSI systems increases, formal design methods are required to guarantee correct behavior of systems. Our approach employs linear mapping on regular dependence graphs as described in Section 4.3. This design methodology is extended to consider the timing between data movements and systolically broadcasted instructions in SIMD-systolic systems. The overall design methodology we use to verify the SIMD-systolic system for an application-specific way is depicted in Figure 48.

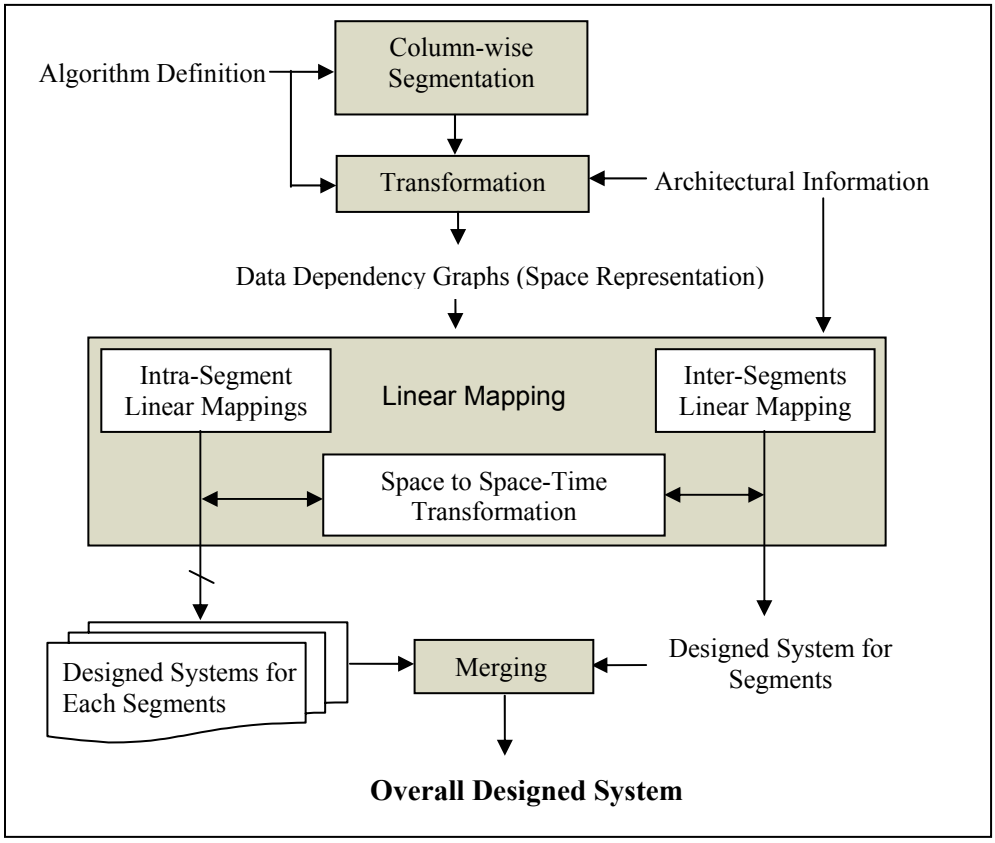


Figure 48: The overall framework of the systematic design method for a SIMD-systolic system.

Linear mapping techniques are extended in our research. The basic method works with the regular dependency graph in which data moves in the same direction for all PEs. This method assumes a pure systolic system that does not need any control information because all PEs can execute the fixed functionality defined by applications at any time. However, a SIMD-systolic system is a programmable system, in which all PEs are controlled by systolically broadcasted instructions and the directions of data movement are fixed from south to north because off-chip memory modules are connected to the PEs in south border. This connection of memory modules is well-harmonized with the systolic instruction broadcast. Thus architectural information is considered in our design methodology, unlike the original mapping techniques which are based solely on the description of algorithms. In addition, instructions are also treated as a kind of data to harmonize scheduling of instructions and sequencing of off-chip data. An example of an extended method is shown in Section 4.4.2.

4.4.2. Case Study of Extended Mapping Techniques: Vector Quantization

The basic description of the VQ image compression application is described in Section 2.8.1. The formal representation of the VQ encoding process can be expressed as follows.

Let $X = [x_0\ x_1\ x_2\ \ldots\ x_{k-1}]$ be an input vector of dimension k, N be the number of code vectors, and $Ci = [c_{i,0}\ c_{i,1}\ c_{i,2}\ \ldots\ c_{i,k-1}]$ be the i-th codeword of dimension k, where i = 0, 1, 2, …, N-1. This step involves the measuring of the N distortions, d_i, where i = 0, 1, 2… N-1, and selecting the codeword index i for which d_i is the minimum distortion.

During this encoding step, exhaustive search is necessary, which is computationally expensive. There are two approaches to dealing with the search costs: using a sub-optimal vector quantizer in a heuristic way and utilizing the multiprocessors. We takes the second approach by developing an efficient system design technique to perform compute-intensive algorithms efficiently on a SIMD-systolic architecture.

A space representation, DG for VQ encoding can be built by incorporating the set of basic vectors, P^T, S^T and d, based on the architectural information, such as the method to get the input data. Since our target architecture is a focal plane architecture, image data can be moved directly from the focal plane to each node in the processor array at once. This results in a DG with the input data X which stays in each node as depicted in Figure 49.

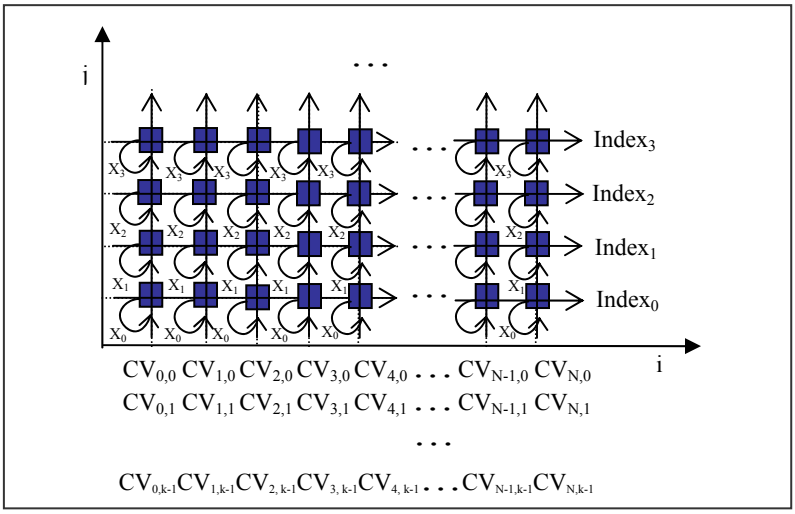


Figure 49: DG for VQ application.

The input data X_i is assigned to the i^{th} processor in the target architecture. Nodes in the DG with the same input data will be assigned to the same PE which results in $P^T = [0, 1]$. Since $P^Td = 0$ and $S^Td \neq 0$, the d_2 should be 0 and S_1 and d_1 should not be 0 where $S^T = [S_1, S_2]$ and $d^T = [d_1, d_2]$. Since instructions are systolically broadcast and nodes with the same j value will be assigned

to the same PE, the schedule time of each PE should be the same as the j value, assuming the encoding operation can be executed for one code vector in unit time. The schedule time of each node is decided by $S^T I$. Based on this observation, for a code vector size of 16, the node (i, j) will be scheduled at (16 x i + j). As a result, the scheduling vector is $S^T = [16, 1]$. Consequently, the selection of a set of basic vectors are $P^T = [0, 1]$, $S^T = [16, 1]$ and $d^T = [1, 0]$. Transformations are listed for each edge in Table 13.

Table 13: Transformation table for each edge.

Edge e	$P^T e$	$S^T e$
CV (0, 1)	1	1
X (0, 0)	0	0
Index (1, 0)	0	16

The space-time representation can be created by the following computation.

$$\begin{bmatrix} i' \\ j' \\ t' \end{bmatrix} = T \begin{bmatrix} i \\ j \\ t \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ p_1 & p_2 & 0 \\ s_1 & s_2 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ t \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 16 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ 0 \end{bmatrix}.$$

As a result, the processor axis is j' = j and time axis is t' = 16i + j. The resulting systolic system is shown in Figure 50 and the corresponding space-time representation is depicted in Figure 51.

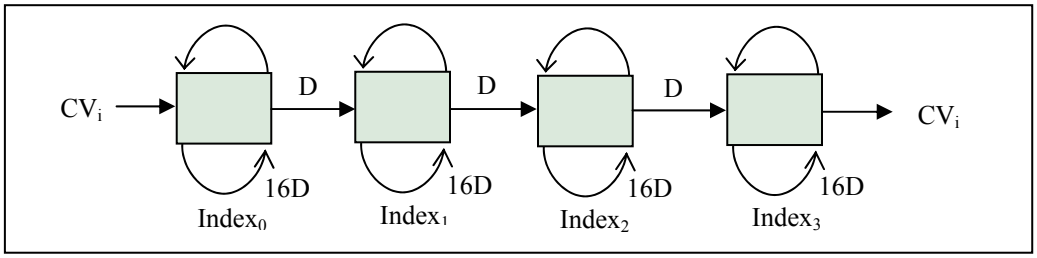


Figure 50: One-Column system designed for VQ based on transformations in Table 13.

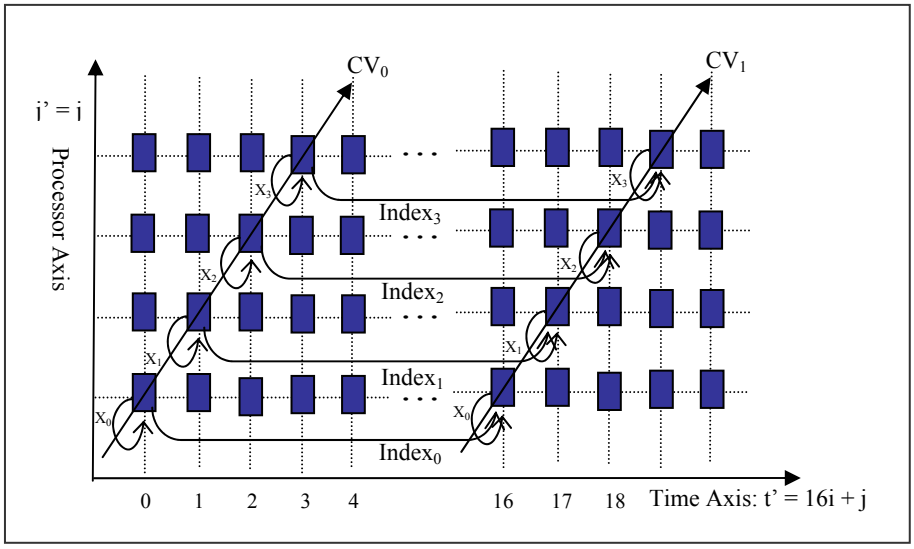


Figure 51: Space-time representation.

The space representation in Figure 49 corresponds to one column of PEs, also called a segment, of the PE array. We need to consider the column-to-column design using the same concept. Mappings for inter-segments need to consider only systolic instruction broadcast. Thus we can easily draw the system at the segment level as in Figure 52.

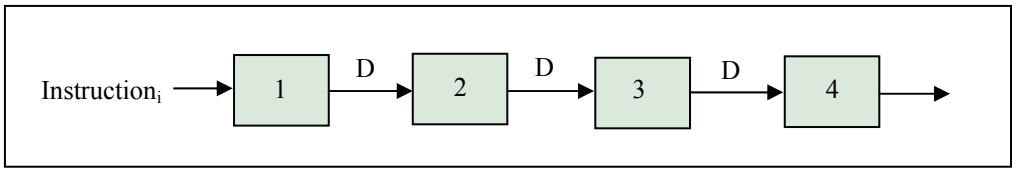


Figure 52: Segment-level system designed for VQ.

This can be seen as a high-level view of a designed system since one node is corresponding to the set of PEs in a column. By combining Figure 50 and Figure 52, we can show the overall system view of a SIMD-systolic system for VQ. Since each segment node will be scheduled in consecutive time slots, the space-time representation can be expressed by sliding one step towards the time axis. The simplified representation is given in Figure 53. Each segment box is corresponding to the space-time representation given in Figure 51.

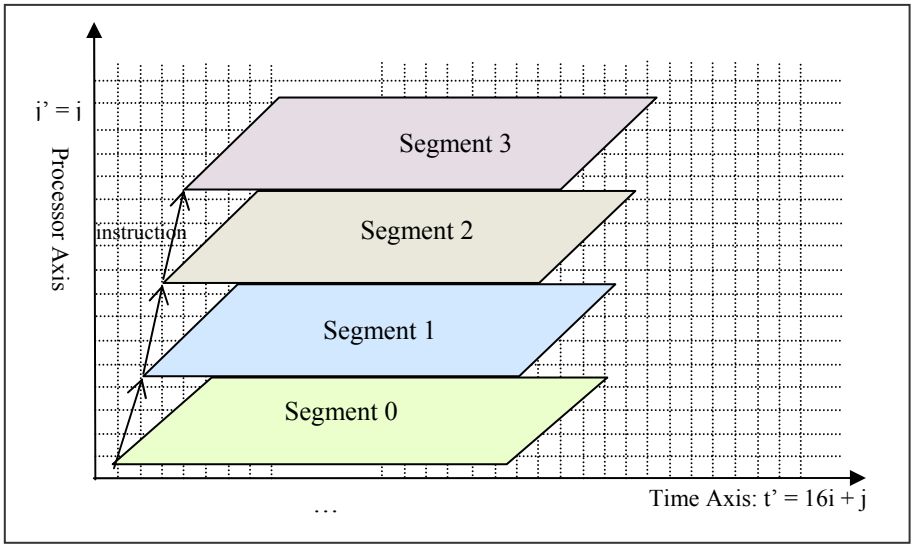


Figure 53: High level view of space-time diagram assuming 4 x 4 processor array.

4.4.3. Memory Operations in SIMPil Architecture

This section describes the kinds of memory operations that are supported in our target system. Our target architecture is based on SIMPil16 with an off-chip memory interface. SIMPil architecture can support three types of memory addressing modes: immediate addressing (direct addressing), controller register indirect addressing, and PE local register indirect addressing.

Table 14 shows the possible memory addressing modes for each type of memory operation. Basically, memory operations are 'load' and 'store'. There are three types of memory operations depicted in Table 14 based on which memory words are loaded from and stored to.

Table 14: Memory addressing modes for each type of memory operation.

Memory Operation	Controller	PE Local	OFF_CHIP Memory	
			Shared	Private
Load	Immediate CREG Indirect	Immediate CREG Indirect PREG Indirect	Immediate CREG Indirect (same for all PEs in a column)	Immediate CREG Indirect
Store	Immediate CREG Indirect	Immediate CREG Indirect PREG Indirect	N/A	Immediate CREG Indirect

4.4.4. Systolic Virtual Memory (SVM): SIMD-systolic System with Off-Chip Memory Accesses

This section describes how the SIMD-systolic system supports off-chip memory accesses. The two kinds of off-chip memory operations are systolic load ('sys_load') and systolic store ('sys_store'). Typically, off-chip memory access results in much higher latency, which becomes a performance bottleneck. In addition, due to the limited off-chip memory bandwidth, sufficient data cannot be provided for all PEs in relatively large SIMD systems. In our research, by leveraging from the staggered instruction execution of systolic instruction broadcast, off-chip memory access is efficiently handled with limited memory bandwidth. Systolically broadcasted 'sys_load' instructions will rendezvous with the data in each PE, which is simultaneously being moved systolically from the bottom to the top in a processor array. Off-chip memory latency will be hidden by data prefetching techniques. This management of off-chip memory gives the illusion of a large on-chip memory. Since this is conceptually similar to the virtual memory [79] found in most contemporary architectures [78], we refer to our mechanism as systolic virtual memory (SVM).

Section 4.4.4.1 describes the mechanism to support the rendezvous of a systolic load instruction with off-chip memory data. Section 4.4.4.2 discusses the corresponding systolic store mechanism.

4.4.4.1. Systolic Load

Figure 54 shows the mechanism for a systolic load operation for one column of a 4 by 4 processor array. The address table in this figure is used to support data prefetching, which will be discussed later. Based on the mechanism shown in Figure 54, off-chip memory is placed at the bottom of processor array and data is moved from the bottom to the top in a processor column. An example of systolic data load is illustrated in Figure 45.

4.4.4.2. Systolic Store

Figure 55 shows the mechanism for a systolic store operation for one column of a 4 by 4 processor array. Data to be stored in off-chip memory is moved from each PE. An example is illustrated in Figure 56. A FIFO store buffer is used to prevent unnecessary stalls resulting from the simultaneous memory access. Off-chip memory load operations look in this store buffer first in case there is data which has not yet been written to the off-chip memory.

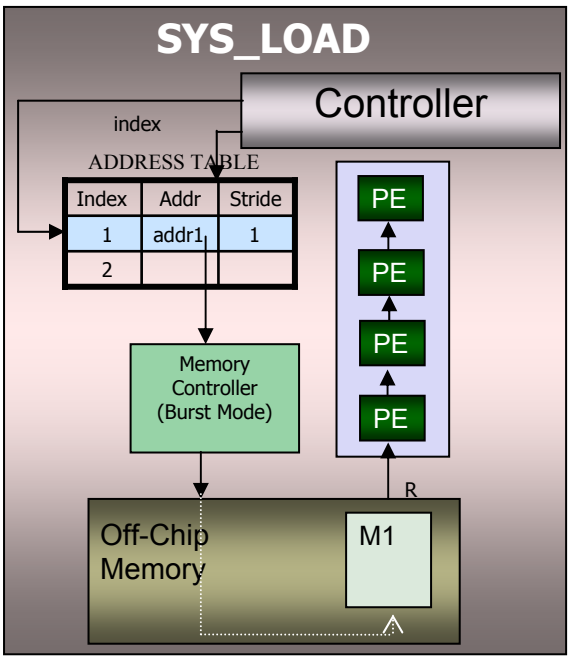


Figure 54: Systolic virtual memory: systolic load mechanism.

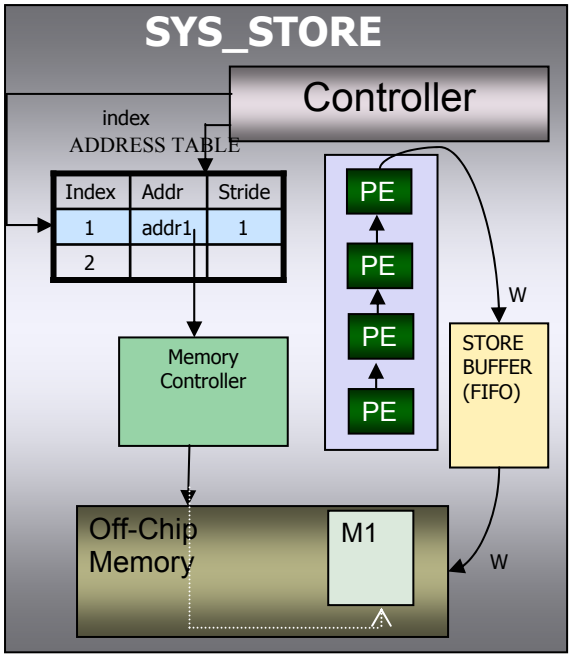


Figure 55: Systolic virtual memory: systolic store mechanism.

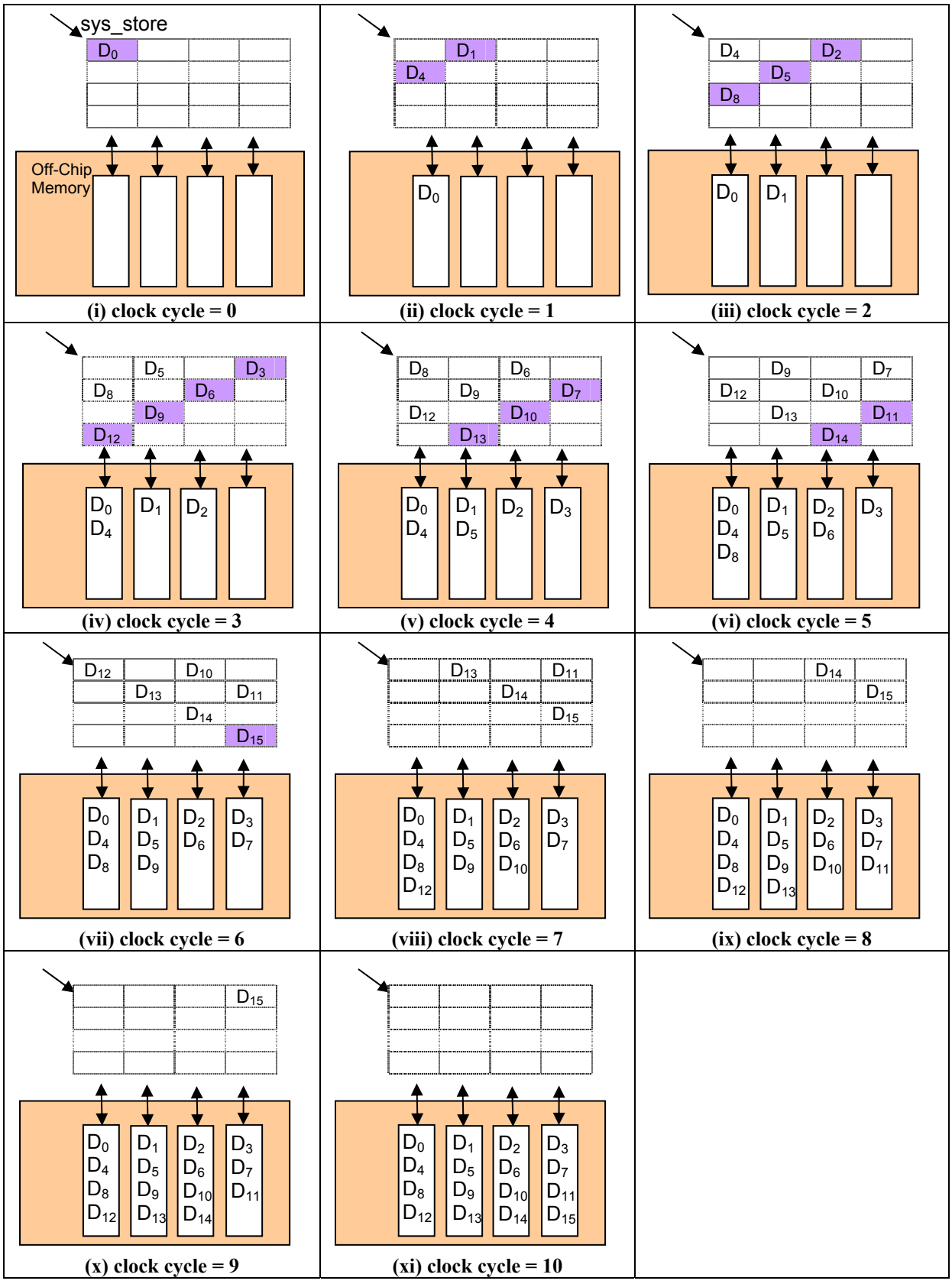


Figure 56: An Example of Systolic Store.

Using these mechanisms for systolic load and store, the next section describes the data prefetch technique used to hide the off-chip memory access latency and the traverse time.

4.4.5. Data Prefetch

A hybrid data prefetch technique is used in our research. As in software data prefetching, our method inserts 'prefetch' instructions explicitly into the programs. In addition, a hardware-based address table is used. This is initialized at compile time and updated at runtime using hardware logic. The information in the address table is used to reference the off-chip memory word.

4.4.5.1. Prefetch Instruction

Our approach to inserting and executing prefetch instructions strives to reduce overhead. Because one systolic load instruction is executed in all PEs, the number of prefetch instructions would normally equal the number of PEs in a processor array. However, the behavior of each column in a processor array is identical except that there is a one cycle delay between columns. Since PEs in a same row will occupy the same location in each memory module, the number of added prefetch instructions can be reduced to the number of rows in a processor array. However, even with this reduction in the number of prefetch instructions, a considerable amount of instruction overhead remains. As shown in Figure 57, a sample program with 2 instructions is expanded by 15 additional instructions: 8 'prefetch' instructions and 7 'nop' instructions to control the data arrival time. We reduce this overhead by overlapping prefetches with the execution of other instructions using an extended long instruction format.

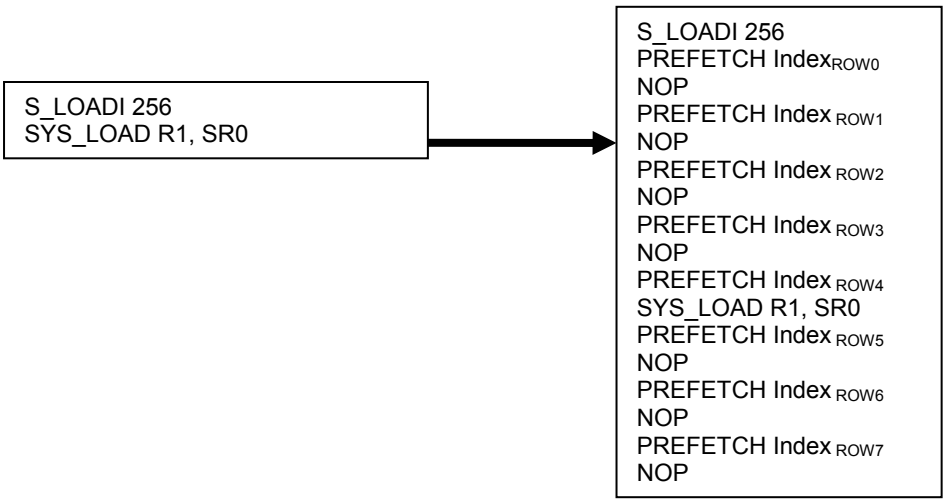


Figure 57: An example of software data prefetching using the 'prefetch' instruction.

Figure 58 shows the extended instruction format which includes a PE instruction and an off-chip memory instruction field. At this time, this field is used only for 'prefetch' instruction, but may be used for other types of instructions in the future. The 'prefetch' instruction's format is 'prefetch index' where index is used to reference the address table. The extended instruction format makes it possible to execute the 'prefetch' instruction with other PE instruction at the same time. Figure 59 shows the instructions resulting from the 'sys_load R1, SR0' instruction executing on an 8 by 8 processor array. As depicted in this figure, the PE instruction and prefetch instruction run independently from each other. Thus, all slots, except the 8th PE instruction slot can be filled with other useful instructions (if any), independent of the existence of prefetch instructions in the off-chip memory instruction field. Thus the instruction overhead resulting from prefetch instruction insertion will be effectively reduced.

Figure 60 illustrates how data is systolically loaded from the off-chip memory through the column of an 8 x 8 processor array in time.

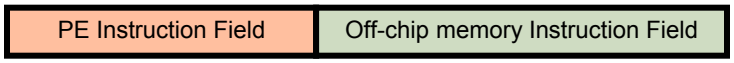


Figure 58: Extended instruction format for data prefetching.

	PE Instruction	Off-Chip Memory Instruction
1		Prefetch Index$_{ROW0}$
2		
3		Prefetch Index $_{ROW1}$
4		
5		Prefetch Index $_{ROW2}$
6		
7		Prefetch Index $_{ROW3}$
8	A: mload R1, SR0	
9		Prefetch Index $_{ROW4}$
10		
11		Prefetch Index $_{ROW5}$
12		
13		Prefetch Index $_{ROW6}$
14		
15		Prefetch Index $_{ROW7}$

Figure 59: An example of an extended instruction format: 'sys_load R1, SR0.'

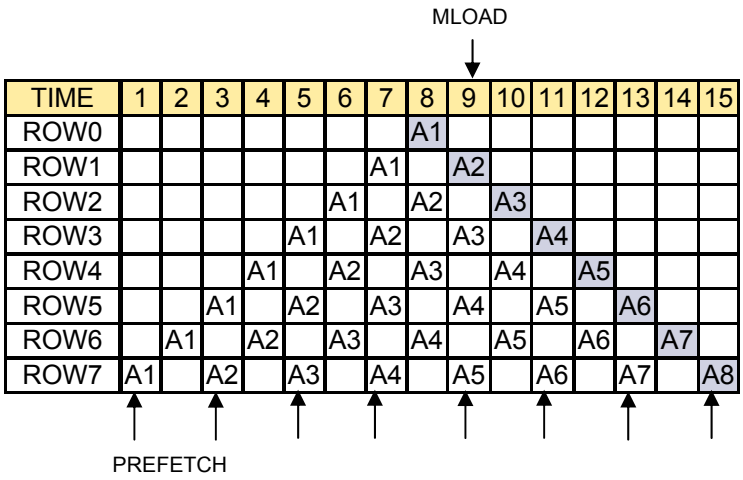


Figure 60: An example of data movement in one column of an 8 x 8 processor array for a 'sys_load.'

4.4.5.2. Address Table

The address table used in our data prefetch method is depicted in Figure 61. There are three columns in this table – index, base address, and stride. The index field is not an explicit column but it is used for referencing of items in the table. The base address field is used to compute the effective address of the memory operation along with a stride field. The base address is the address of prefetched data for the first row of a processor column. This address is set at compile time and incremented automatically at run time to be used for the PE in the next row. The 'stride' field is useful where the systolic load instruction is inside the loop. This value is used to update the base address after the last prefetch instruction corresponding to one systolic load in a loop (i.e., the prefetch instruction for a PE in the last row for a particular systolic load) is executed.

Index	Base Address	Stride
1	0	Don't Care
2	16	1

Figure 61: Address table.

Figure 62 shows an example of data prefetching using the extended instruction format for a prefetch instruction with the address table shown in Figure 61. Conceptually, the effective address for each PE is computed as (Base Address + Row Number for a Particular PE). Since the row number is incremented each time, the base address is automatically incremented by one in the implementation. In this example, all indexes for PEs are 1 which references the first row, which has the (base address = 0).

	PE Instruction (PE0)	Off-Chip Memory Instruction	Memory Address
1		Prefetch 1	0 ← [BaseAddress(=0) + RowNumber(=0)]
2			
3		Prefetch 1	1 ← [BaseAddress(=0) + RowNumber(=1)]
4			
5		Prefetch 1	2 ← [BaseAddress(=0) + RowNumber(=2)]
6			
7		Prefetch 1	3 ← [BaseAddress(=0) + RowNumber(=3)]
8	A: mload R1, SR0		
9		Prefetch 1	4 ← [BaseAddress(=0) + RowNumber(=4)]
10			
11		Prefetch 1	5 ← [BaseAddress(=0) + RowNumber(=5)]
12			
13		Prefetch 1	6 ← [BaseAddress(=0) + RowNumber(=6)]
14			
15		Prefetch 1	7 ← [BaseAddress(=0) + RowNumber(=7)]

Figure 62: An example of data prefetching using the address table given in Figure 61.

The data prefetching technique was devised to reduce the off-chip memory latency. However, so far, we have been considering how to sequence the systolically moved data from off-chip memory to rendezvous with a systolically distributed instruction. As a result, data prefetching technique is used to hide the data traverse time resulting from the systolic distribution of data. To hide memory access latency as well as data traverse time, the 'prefetch' instruction should be issued earlier than that shown in previous examples. In particular, if an 'mload' instruction arrives at time T_{MLOAD} for a particular PE , a 'prefetch' instruction should be issued at time $T_{PREFETCH} = T_{MLOAD} - (T_{TRAVERSE} + T_{MEM})$, where.$T_{TRAVERSE}$ is data traversal time from the bottom of the array to a particular PE, and T_{MEM} is the off-chip memory access time in clock cycles. The resulting sequence of instructions is depicted in Figure 63 for an 8 by 8 processor array where the first prefetch occurs at cycle time 1 and T_{MEM} is 6 clock cycles. Based on this information, T_{MLOAD} becomes 14 in this case.

PE_{ROW0}: $T_{PREFETCH\_0} = T_{MLOAD\_0} - (T_{TRAVERSE\_0} + T_{MEM}) = T_{MLOAD} - (7 + 6) = 1$

PE_{ROW1}: $T_{PREFETCH\_1} = T_{MLOAD\_1} - (T_{TRAVERSE\_1} + T_{MEM}) = 15 - (6 + 6) = 3$

PE_{ROW2}: $T_{PREFETCH\_2} = T_{MLOAD\_2} - (T_{TRAVERSE\_2} + T_{MEM}) = 16 - (5 + 6) = 5$

PE_{ROW3}: $T_{PREFETCH\_3} = T_{MLOAD\_3} - (T_{TRAVERSE\_3} + T_{MEM}) = 17 - (4 + 6) = 7$

PE_{ROW4}: $T_{PREFETCH\_4} = T_{MLOAD\_4} - (T_{TRAVERSE\_4} + T_{MEM}) = 18 - (3 + 6) = 9$

PE_{ROW5}: $T_{PREFETCH\_5} = T_{MLOAD\_5} - (T_{TRAVERSE\_5} + T_{MEM}) = 19 - (2 + 6) = 11$

PE_{ROW6}: $T_{PREFETCH\_6} = T_{MLOAD\_6} - (T_{TRAVERSE\_6} + T_{MEM}) = 20 - (1 + 6) = 13$

PE_{ROW7}: $T_{PREFETCH\_7} = T_{MLOAD\_7} - (T_{TRAVERSE\_7} + T_{MEM}) = 21 - (0 + 6) = 15.$

	PE Instruction (PE0)	Off-Chip Memory Instruction	Memory Address
1		Prefetch 1	0 ← [BaseAddress(=0) + RowNumber(=0)]
2			
3		Prefetch 1	1 ← [BaseAddress(=0) + RowNumber(=1)]
4			
5		Prefetch 1	2 ← [BaseAddress(=0) + RowNumber(=2)]
6			
7		Prefetch 1	3 ← [BaseAddress(=0) + RowNumber(=3)]
8			
9		Prefetch 1	4 ← [BaseAddress(=0) + RowNumber(=4)]
10			
11		Prefetch 1	5 ← [BaseAddress(=0) + RowNumber(=5)]
12			
13		Prefetch 1	6 ← [BaseAddress(=0) + RowNumber(=6)]
14	A: mload R1, SR0		
15		Prefetch 1	7 ← [BaseAddress(=0) + RowNumber(=7)]

Figure 63: An example of data prefetching.

Our data prefetching techniques are used to hide both data traverse time in SVM mechanism and off-chip memory latency time. Also, by extending the instruction format, we can minimize the instruction overhead of the additional prefetch instructions. However, to match the systolically pumped data from off-chip memory with the systolically broadcast instructions, memory load instructions (‘mload’) should be delayed where there is a channel conflict. Section 4.4.6 describes techniques for minimizing these delays, illustrating with particular examples.

4.4.6. Instruction Scheduling

Data prefetching technique can hide the data traverse time and memory access latency as described in the previous section. However, we need to consider resource constraints, such as limited memory bandwidth and number of channels, since they can cause delays if there is a conflict. With a given number of physical channels for systolic load (1 in this case), the number of consecutive systolic load instructions should not exceed 2 to avoid delays from the instructions. Thus where more than 2 consecutive systolic load instructions are broadcasted, delays should be inserted between the second ‘sys_load’ instruction and the third ‘sys_load’ instruction. The scheduler and data sequencer play important roles in minimizing the performance degradation as well as preserving the application correctness. Figure 64 shows an example of two consecutive systolic load instructions executed under the same assumption as in Figure 63.

Sample Program:

```
SYS_LOAD R1, SR1
SYS_LOAD R2, SR2
```

	PE Instruction	Off-Chip Memory Instruction
1		Prefetch 1
2		Prefetch 2
3		Prefetch 1
4		Prefetch 2
5		Prefetch 1
6		Prefetch 2
7		Prefetch 1
8		Prefetch 2
9		Prefetch 1
10		Prefetch 2
11		Prefetch 1
12		Prefetch 2
13		Prefetch 1
14	A: mload R1, SR1	Prefetch 2
15	B: mload R2, SR2	Prefetch 1
16		Prefetch 2

TIME	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
PE(ROW0)								A1	B1							
PE(ROW1)							A1	B1	A2	B2						
PE(ROW2)						A1	B1	A2	B2	A3	B3					
PE(ROW3)					A1	B1	A2	B2	A3	B3	A4	B4				
PE(ROW4)				A1	B1	A2	B2	A3	B3	A4	B4	A5	B5			
PE(ROW5)			A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6		
PE(ROW6)		A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	
PE(ROW7)	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8

Figure 64: An example of consecutive systolic load instructions.

As shown in Figure 64, the data channel is fully occupied at some point by the systolically moved data for two consecutive sys_loads. This saturation also can be seen in the off-chip memory operation field of the sequence of instructions in Figure 64 such that all the fields are occupied by the prefetch instructions. For the first systolic load instruction, all odd numbered off-

chip memory instruction fields are used by prefetch instructions, and for the second systolic load instruction, all even numbered off-chip memory instruction fields are occupied. Likewise, the case of two consecutive systolic stores is depicted in Figure 65.

SAMPLE PROGRAM:

```
SYS_STORE SR1, R1
SYS_STORE SR2, R2
```

	PE Instruction	Off-Chip Memory Instruction
1	mstore SR1, R1	
2	mstore SR2, R2	
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8
	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8	
		A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8		
			A4	B4	A5	B5	A6	B6	A7	B7	A8	B8			
				A5	B5	A6	B6	A7	B7	A8	B8				
					A6	B6	A7	B7	A8	B8					
						A7	B7	A8	B8						
							A8	B8							

Figure 65: An example of consecutive systolic store instructions.

Systolic memory operations can be executed without delays due to the limited bandwidth. Since one word can be loaded from the off-chip memory at once, the third systolic memory

operation must wait until the first two systolic memory instructions are executed through the processor array. This case is depicted in Figure 66.

SAMPLE PROGRAM:

```
I1: SYS_LOAD R1, SR1
I2: SYS_LOAD R2, SR2
I3: SYS_LOAD R3, SR3
```

DELAYS (14 = 2N -2)

							I1	I2														I3
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
							A1	B1															C1
						A1	B1	A2	B2													C1	C2
					A1	B1	A2	B2	A3	B3											C1	C2	C3
				A1	B1	A2	B2	A3	B3	A4	B4									C1	C2	C3	C4
			A1	B1	A2	B2	A3	B3	A4	B4	A5	B5							C1	C2	C3	C4	C5
		A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6					C1	C2	C3	C4	C5	C6
	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7			C1	C2	C3	C4	C5	C6	C7
A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8	C1	C2	C3	C4	C5	C6	C7	C8

Figure 66: An example of three consecutive systolic load instructions.

As in Figure 66, delays are added to wait until the data channel is available for the third systolic load instruction. The number of nop delays is (2N – 2) for an N x N processor array. The delays should be minimized for the efficiency of the SIMD-systolic system. Thus an instruction scheduling technique based on data flow analysis is used to replace delays with useful instructions. The overall framework of our instruction scheduler for SVM is shown in Figure 67. It consists of three main tasks – channel conflict detection and resolution, data flow analysis, and delay reductions.

- **Channel Conflict Detection and Resolution**:

  The instruction scheduler checks if there is a channel conflict among the systolic memory operations. For different combinations of off-chip memory instructions, there are different numbers of necessary distances among the memory operations due to the limited bandwidth and channel availability. Thus if the instruction scheduler detects any channel conflict, nop-delays are inserted to resolve it.

- **Data Flow Analysis**:

  As in the previous chapter, to minimize the delays produced by the instruction scheduler and to preserve application correctness, data dependencies are analyzed to select a

candidate instruction to replace the delays. The candidate instructions should not have any dependency with other instructions between 'NOP' and the candidate instruction.

- **Delay Reduction**:

  Based on data flow analysis, delays are minimized by replacing 'NOP' instructions with other meaningful instructions. Since candidate instructions are chosen based on dependency information, replacing delays with such instructions does not affect application results but does improve performance.

This instruction scheduler can be unified with that for systolic instruction broadcast described in the previous chapter. Since the example of instruction scheduling based on data dependencies are already depicted in CHAPTER 3, the distances among consecutive off-chip memory operations that are used to detect the channel conflict in the instruction scheduler are defined instead. This information is described in Table 15.

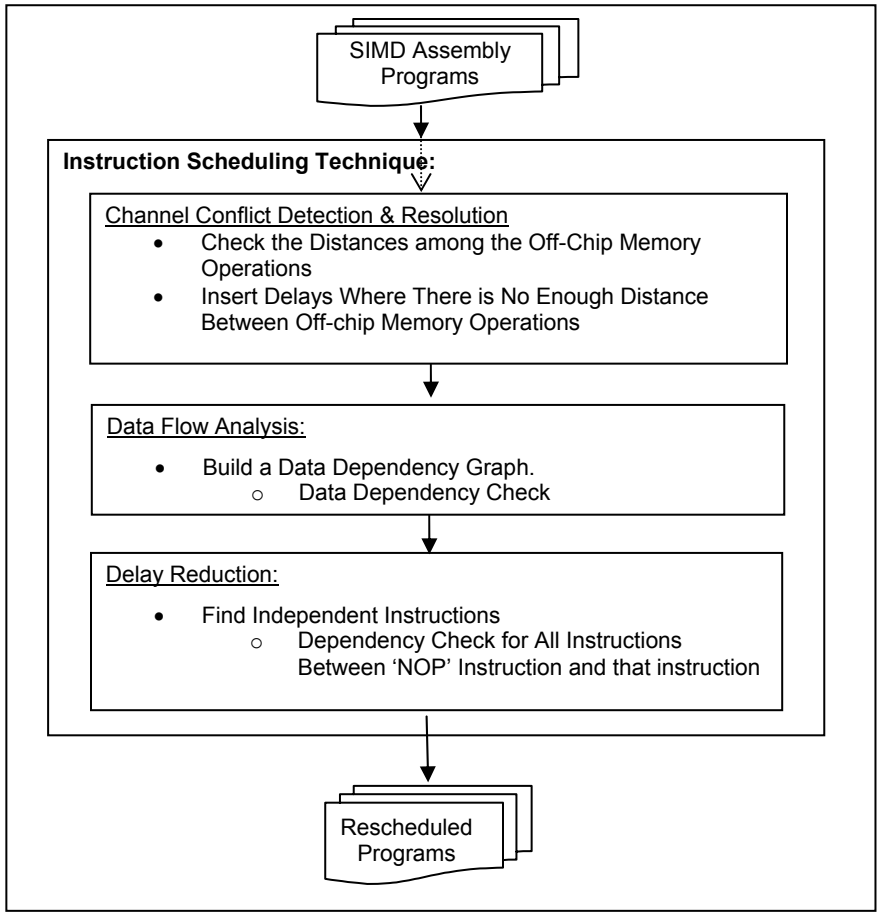


Figure 67: Framework of an instruction scheduler for systolic virtual memory.

Table 15: Distances between consecutive off-chip memory operations.

Combinations of Off-Chip Memory Operations (L: Sys_Load, S: Sys_Store)	Distance	Combinations of Off-Chip Memory Operations (L: Sys_Load, S: Sys_Store)	Distance
L-L	1	L-L-L-L	1, 2N-1, 1
L-S	1	L-L-L-S	1, 2N-1, 1
S-S	1	L-L-S-L	1, 1, 2N-1
S-L	1	L-L-S-S	1, 1, 1
L-L-L	1, 2N-1	L-S-L-L	1, 2N-1, 1
L-L-S	1, 1	L-S-L-S	1, 2N-1, 1
L-S-L	1, 2N-1	L-S-S-L	1, 1, 2N-1
L-S-S	1, 1, 1	L-S-S-S	1,1, 2N-1
S-L-L	1, 2N-1	S-L-L-L	1, 2N-1, 1
S-L-S	1, 2	S-L-L-S	1, 2N-1, 1
S-S-L	1, 2N-1	S-L-S-L	1, 2, 2N-1
S-S-S	1, 2N-1	S-L-S-S	1, 2, N
		S-S-L-L	1, 2N-1, 1
		S-S-L-S	1, 2N-1, 1
		S-S-S-L	1, 2N-1, 1
		S-S-S-S	1, 2N-1, 1

Table 15 shows only distances between consecutive off-chip memory operations; the instruction scheduler should be able to check the channel conflicts for the general cases. For example, consecutive systolic loads can be executed without delay but if there is another instruction between two systolic loads, the instruction should be rescheduled to prevent channel conflict.

4.5. Results and Analysis

Systolic virtual memory mechanism is evaluated through behavioral simulations and technology analysis for a 16 x 16 processor array. The metrics for the analysis are shown in Table 16.

Table 16: Metrics for experiments.

Analysis	Metrics
Channel Utilization	Percentage of used channels over overall available channels
Performance	Normalize execution time depending on the number of independent instructions
Clock Count Penalty	The number of delays due to systolic virtual memory
Memory Requirement	Number of memory words required by a given application
Memory Area Efficiency	Performance divided by required memory area

The following describe how these metrics are measured in our approach.

- **Channel Utilization** = (Used number of channels / Total number of available channels) in a given time.
- **Performance** = Normalized execution time relative to the execution time of application with only on-chip memory operations where the number of independent instructions is varying.
- **Memory Requirement**
  - Number of words: Required number of memory words for a given application.
  - Memory area: Required memory area when a certain type of memory is used for a system.
- **Memory Area Efficiency** = $\frac{Performance}{MemoryArea}$ where performance is IPC and memory size is the required memory area in mm^2.

4.5.1. Channel Utilization

In this section, channel utilization is examined to show how on-chip systolic communication network is utilized for a given set of memory operations. Figure 68 shows the utilization of the on-chip systolic communication network for a certain number of consecutive memory operations. In this figure, simulated memory operations are all the same type of operations.

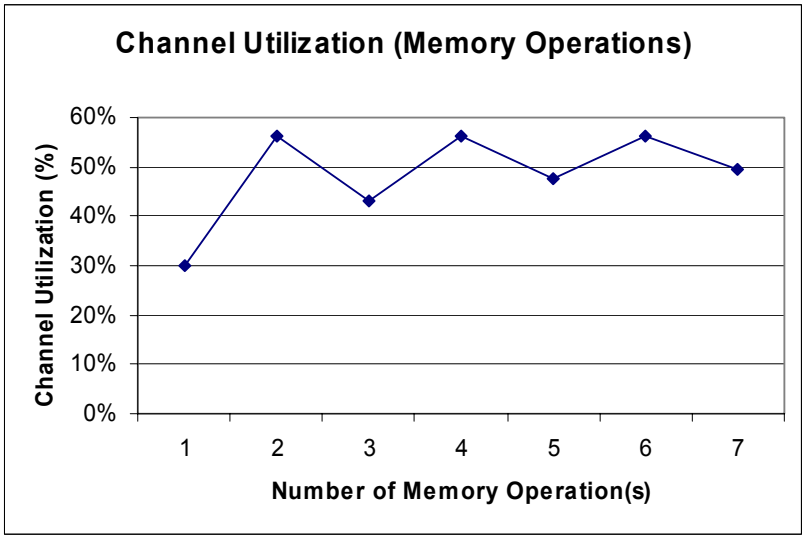


Figure 68: Channel utilization for a given number of consecutive memory operations.

Two consecutive off-chip memory operations (e.g., sys_load & sys_load or sys_store & sys_store) can execute without delay because there is no conflict in channel usage, for a given time to move the data through the network. Because of this data channel utilization is better than that for an odd number of off-chip memory operations. However, as the number of memory operation increases, the impact of the overhead resulting from the channel conflicts will be decreased as is depicted in Figure 68.

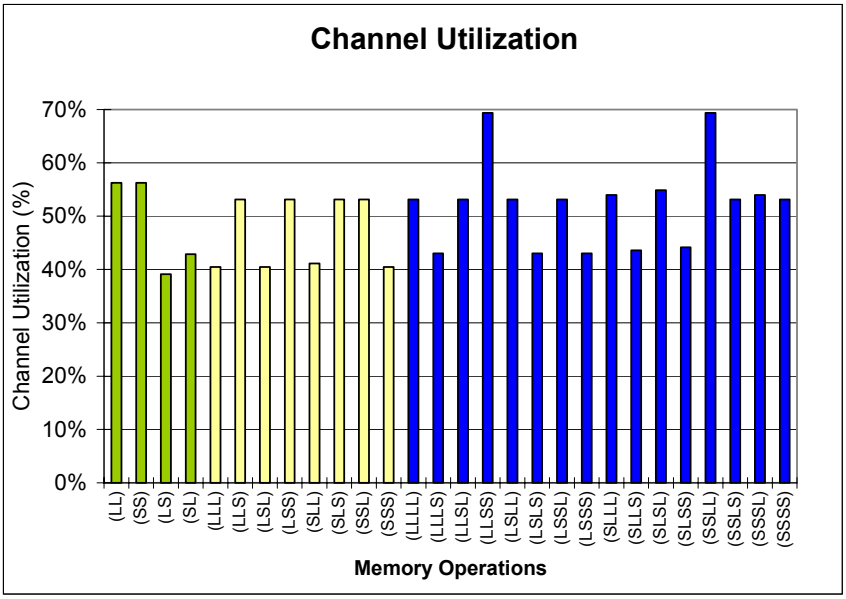


Figure 69: Channel utilization for a given combination of memory operations. (L:sys_load, S: sys_store)

Figure 69 shows the channel utilization of the systolic network for different combinations of consecutive off-chip memory operations. Channel utilization varies based on how the data moves through the network, for a given combination of memory operations. In this figure, the combinations (LLSS) and (SSLL) have the best utilizations where S is the systolic store and L is the systolic load instruction.

4.5.2. System Performance

System performance in terms of Instructions per Cycle (IPC) is shown in this section. Since he instruction scheduler replaces the delays resulting from the channel conflict with the useful instructions in a program, if there are enough instructions to be replaced, off-chip memory access would be free as depicted in Figure 70. In the simulations, we are using a 16 x 16 processor array

and delays from off-chip memory is 30 (= 2N – 2 = 2 x 16 – 2), where N is 16 in this case. Thus if the number of independent instructions which are candidates to be replaced with delay instructions by the scheduler, is same as the number of delays (30, in this case), free off-chip memory access is achieved. Even though there are not enough instructions to be replaced with the delays, massive data parallelism will still result in much better performance, relative to the non data parallel implementation. For example, if we have only one independent instruction which results in 8 times the execution time, we can still achieve a factor of 32 (= 256 / 8) performance gain compared to the non-data parallel systems, because of the concurrency of 16 x 16 PEs.

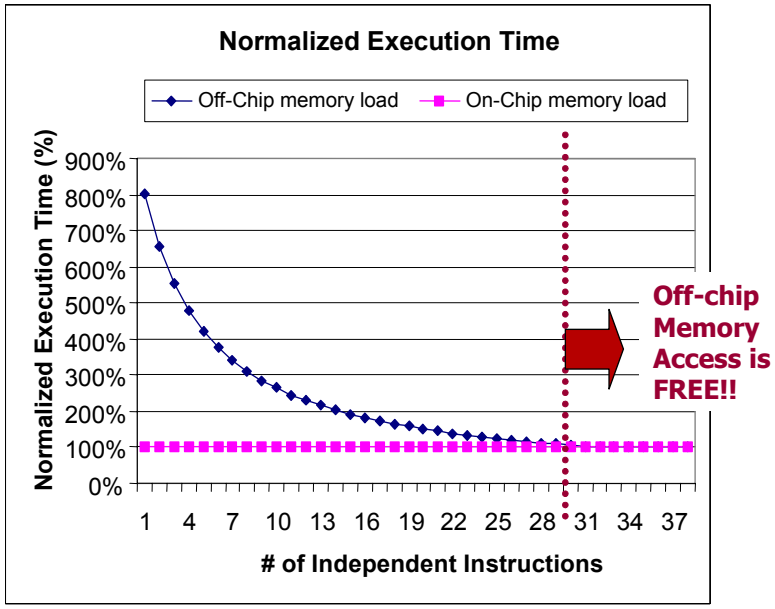


Figure 70: Normalized execution time of off-chip memory operation relative to the on-chip memory operation.

4.5.3. Clock Count Penalty

In this section, the overhead of systolic virtual memory is shown in terms of clock count penalty. This term describes at what cost in extra delays the systolic virtual memory can be utilized. As shown in Figure 71, in REGION 1 (where clock count penalty is less than 2N – 2 = 30) the penalty decreases linearly as the number of independent instructions increases. And in Region 2 (where clock count penalty is greater than or equal to 30), zero-penalty off-chip memory access can be obtained due to instruction scheduling.

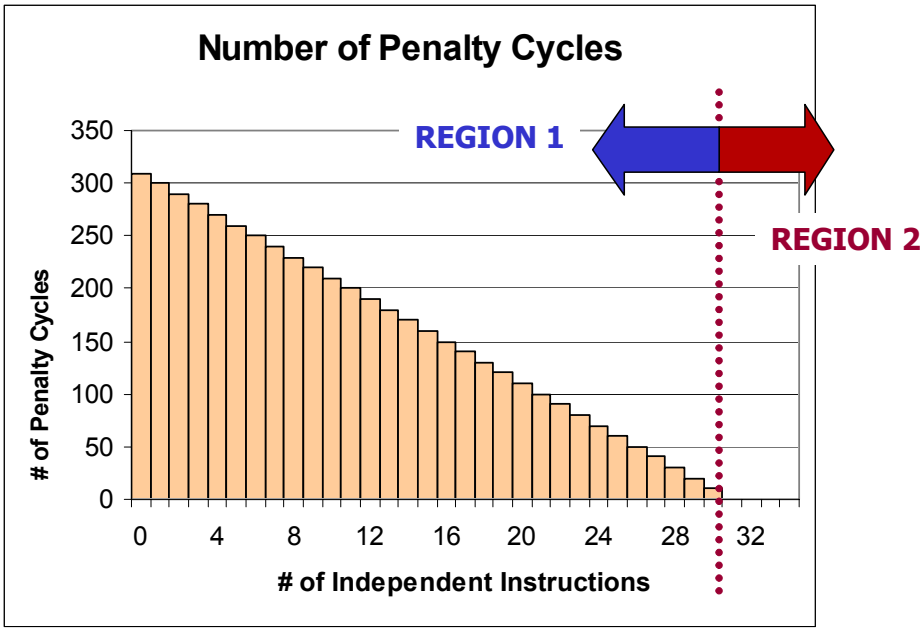


Figure 71: Clock count penalty for systolic virtual memory.

4.5.4. Application: Matrix Multiplication

In this section, we simulate a matrix multiplication application to analyze the memory requirement in terms of number of words and memory area, execution time where the off-chip memory operations are used in an application, and area efficiency for two types of implementations of this application. Based on the results, we decide which implementation for a given application is optimal in terms of area efficiency.

For systolic virtual memory access, we can use two types of memory space, which are private address space and shared memory address space. This is defined in an application to select the implementation method by an application programmer. Figure 72 shows how the matrices A and B are placed to be multiplied in the system where shared memory address space is used. For illustration, a 4 x 4 processor array is used. In this implementation, a column of B matrix is shared by the PEs in a same column.

Figure 73 shows how the matrices A and B are placed to be multiplied in the system where private memory address space is used. Each column of matrix B is duplicated in all PEs in the same column of the processor array. For these two implementations, execution time, memory requirement, and area efficiency are analyzed to decide the optimal implementation of a given application.

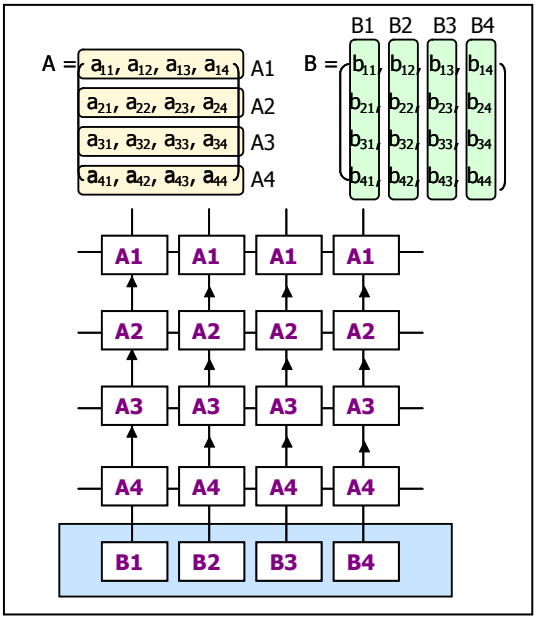


Figure 72: Matrix multiplication application with shared memory address space.

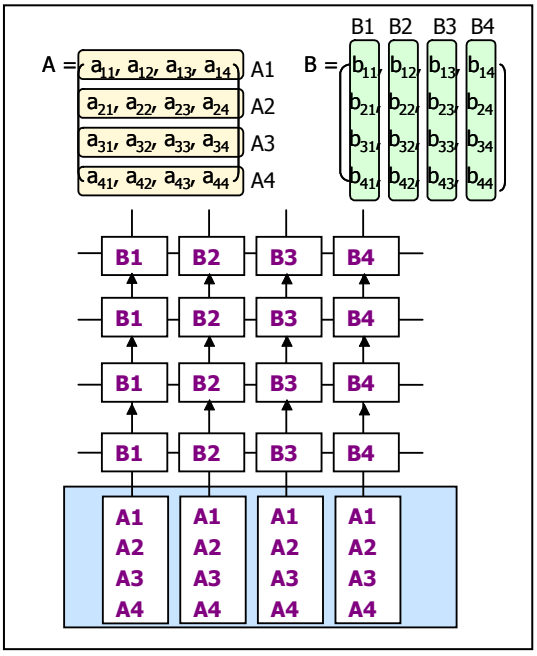


Figure 73: Matrix multiplication application with private memory address space.

4.5.4.1. Matrix Multiplication: Execution Time

We implemented three versions of the matrix multiplication application. The first one is implemented using only on-chip memory, the second one is implemented using off-chip memory instructions with private memory model, and the last one is implemented using off-chip memory with shared address space. To obtain the execution times in clock cycles, we simulate these three types of applications using a behavioral simulator developed for this research. Figure 74 shows the normalized execution time of the first application which uses only on-chip memory. As depicted in this figure, off-chip memory access can result in over 20% of execution time overhead independent of the type of address space.

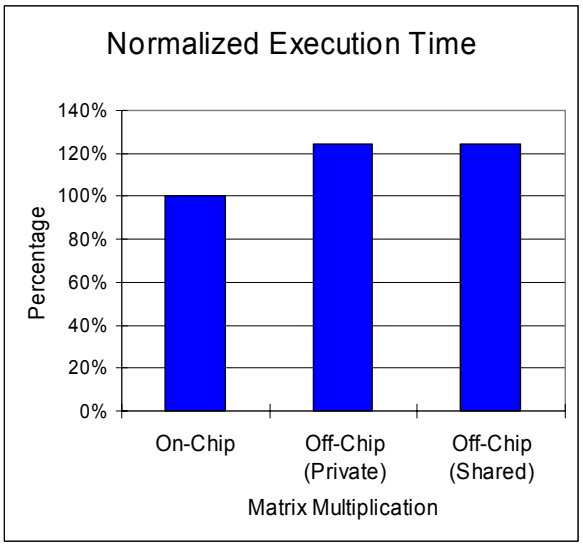


Figure 74: Matrix multiplication: normalized execution time.

4.5.4.2. Memory Requirement

In this section, the results of memory requirements to run the given applications are described in terms of the number of memory words and the required memory area for certain types of memory. Before we show the results for the applications, it is interesting to show the relationship of the number of transistors and occupied area to a given number of memory words for different types of memories, such as DRAM, SRAM, and register file.

Figure 75 and Figure 76 show that required memory area for a given number of memory words is ordered as [Area (Register File) > Area (SRAM) > Area (DRAM)]. Thus the same amount of memory can be implemented with different area costs by choosing different types of memory. To see this impact, first we show the memory requirements in memory words for each implementation of matrix multiplication in Figure 77.

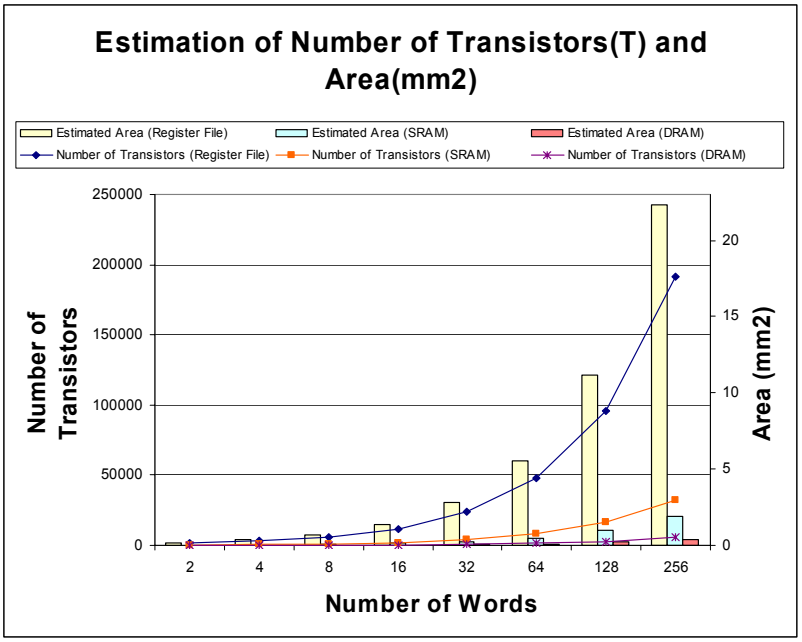


Figure 75: Estimation of the number of transistors and area for each type of memories: DRAM, SRAM, and register file.

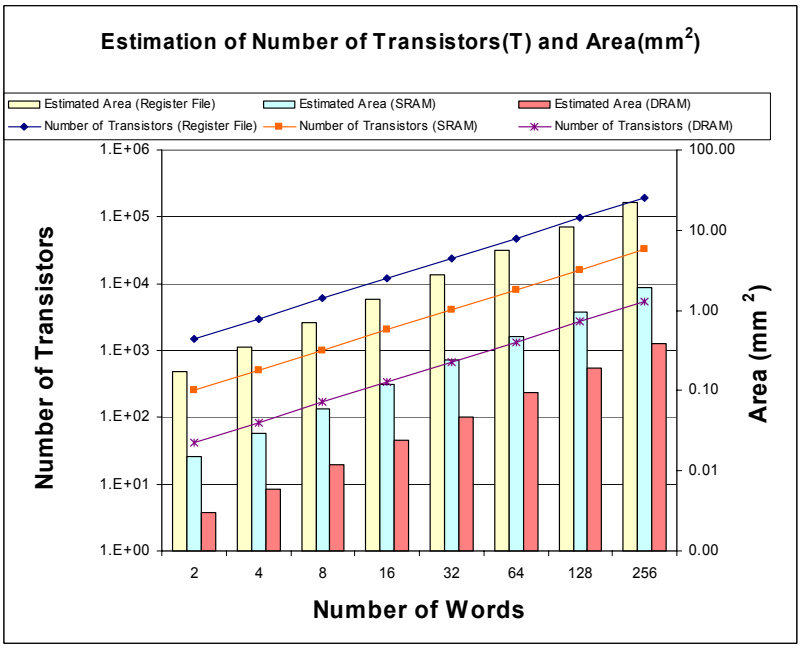


Figure 76: Estimation of the number of transistors and area for each type of memories: DRAM, SRAM, and register file (log scale).

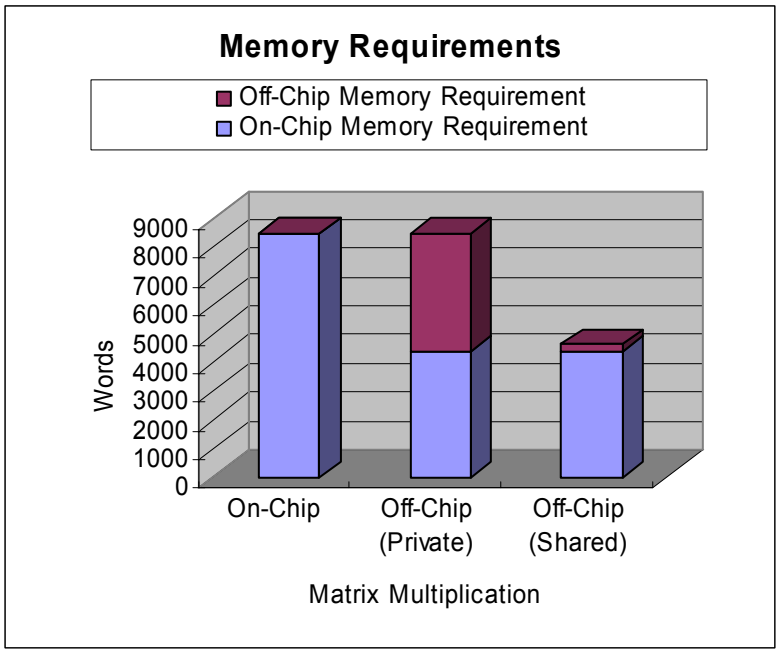


Figure 77: Memory requirements for each implementation in the number of memory words.

Since, matrix B is shared by storing it in shared off-chip memory space, the memory requirement for a matrix multiplication application using shared off-chip memory space is minimal in this simulation. For each memory requirement for each implementation as in Figure 77, we estimate the required memory area for two cases. The first case is that DRAM is used for the off-chip memory, and SRAM is used for the on-chip memory. The second case is that both memories (on-chip and off-chip) are implemented by SRAM. The former is depicted in Figure 79 and the latter is depicted in Figure 78.

As shown in Figure 77, the number of required memory words is same for the first two implementations – on-chip and off-chip (private). However, required memory area is much less for the second implementation where the off-chip memory is implemented by DRAM technology as shown in Figure 79.

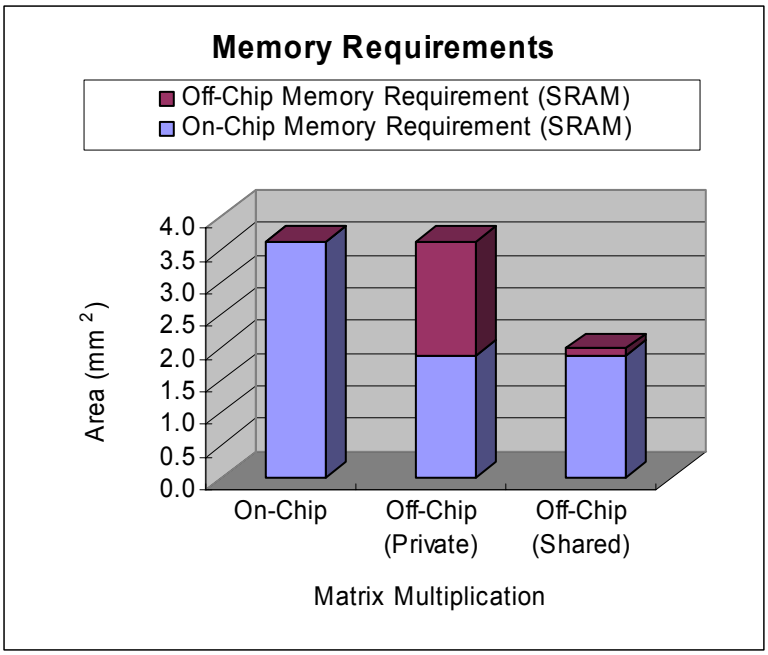


Figure 78: Memory requirements in area where both memories (on-chip & off-chip) are implemented by SRAM.

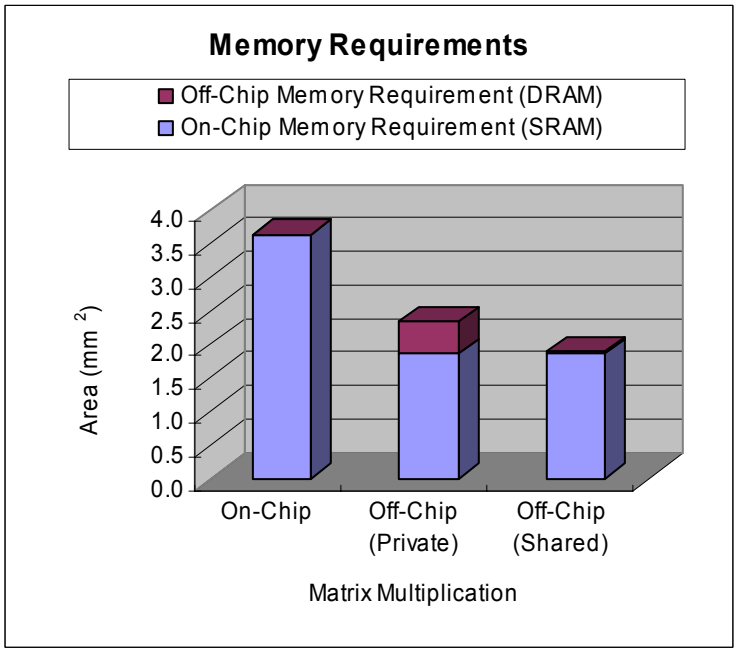


Figure 79: Memory requirements in area where off-chip memory is implemented by DRAM and on-chip memory is implemented by SRAM.

4.5.4.3. Memory Area Efficiency

As mentioned earlier, memory area efficiency is computed as $\frac{Performance}{MemoryArea}$ where Performance is 1 / (Execution Time in Clock Cycles) and MemoryArea is the estimated area shown in Figure 79. To compute the area efficiency, DRAM dense memory is used for the off-chip memory.

Figure 80 shows the normalized area efficiency where this value is 1 for the implementation that is using only on-chip memory. This figure shows that implementations with systolic off-chip memory are better in terms of area efficiency compared to the application with only on-chip memory. In addition, between two implementations with systolic virtual memory, the third implementation which uses shared address memory space can achieve much better area efficiency for the matrix multiplication application. As a result, systolic virtual off-chip memory with shared address space can achieve over 50% higher area efficiency than that of an on-chip only system for a matrix multiplication application.

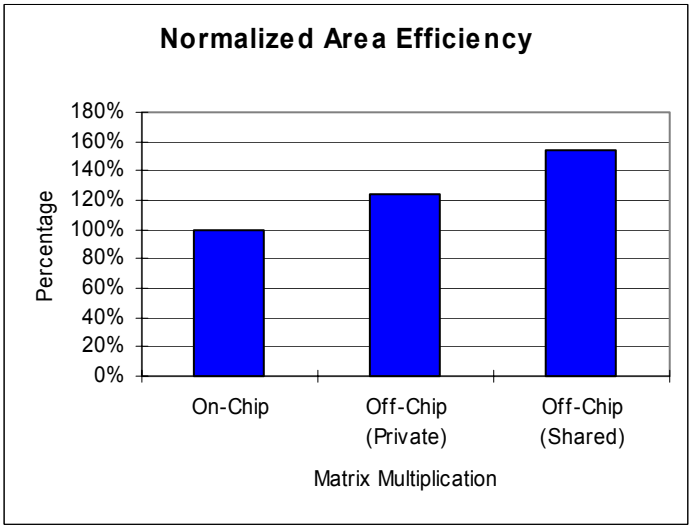


Figure 80: Normalized area efficiency.

4.6. Chapter Conclusion

An area efficient SIMD-systolic system with systolic virtual memory is presented. A data prefetch technique developed in our research extends the instruction format by attaching prefetch information to the local PE instruction. Due to the long instruction format, executions of local PE operation and data prefetch instruction can be overlapped. As a result, our approach to data prefetching with hard-wired address table can effectively hide the relatively long off-chip memory latency without prefetch instruction overhead unless there is a channel conflict. In addition, instruction scheduling also minimizes delays resulting from systolic off-chip memory access. To utilize the systolic virtual memory with systolic instruction broadcast, instruction and data should rendezvous at a certain time. The constraints are handled by the instruction scheduler in our approach. To analyze the effectiveness of our system, we implemented a matrix multiplication application in three different versions. By analyzing the area efficiency, we can determine that systolic virtual off-chip memory with shared address space can achieve over 50% higher area efficiency than that of on-chip only system for a matrix multiplication application.

CHAPTER 5

Concluding Remarks

5.1. Conclusions

Contribution 1: Efficient Storage Usage in Embedded SIMD Systems

An analysis method for assessing storage needs and costs of a given application automatically retargeted across a spectrum of storage configuration designs was developed. Using this technique, a SIMD processing element achieves optimal area and energy efficiency with a register file containing between 8 and 12 words for given workload. This configuration is between 15% and 25% more area and energy efficient than other memory configurations being considered.

Contribution 2: Systolic Instruction Broadcast for Embedded SIMD Architectures

Systolic instruction broadcast is a high performance and area efficient instruction broadcasting scheme with short-wire interconnects by eliminating of wire latency bottleneck found in global instruction broadcast. In this contribution, we simulated systolic instruction broadcast in three approaches – software method, 2-write port register file method, and bypass method. Each method can result different area efficiencies based on the fraction of communications over a given set of instructions. In our evaluations, due to the system's short clock cycle time and scheduler, a speedup in system performance of up to 7.5 can be achieved by the year 2010. In addition, speedup of area efficiency also can be achieved up to 7.2 for a given workload.

Contribution 3: Systolic Virtual Memory

The ability of minimizing off-chip memory access latency while maximizing access frequency by scheduling techniques along with data prefetch techniques in systolic virtual memory mechanism was evaluated using our SIMD-systolic architecture simulator. Results show

that, systolic virtual off-chip memory with shared address space can achieve over 50% higher area efficiency than that of an on-chip only system for a matrix multiplication application.

5.2. Future Work

Memory design exploration techniques can be extended to analyze the off-chip memory designs as well as on-chip memories. This work requires thorough knowledge of data usage and efficient data placement algorithms. Since the PEs in a column of a SIMD-systolic system share an off-chip memory module, storing data which is shared by PEs in a same column in the off-chip memory will save the required memory area and also achieve high area efficiency. In this research, we assumed that all off-chip data reside in one burst length under burst mode. However, this will be extended to consider the boundary conditions of burst length and reschedule the instructions based on this information. To evaluate the effectiveness of systolic virtual memory, more design explorations are needed such as the replication factor of data stored in off-chip memory, the directions of data distribution from off-chip memory along with the directions of instruction broadcasting, and the placement and the number of off-chip memory modules.

APPENDIX A

SIMPil Architecture

The SIMD Pixel Processor (SIMPil) architecture is a portable, single-chip, focal-plane SIMD processor developed by the Portable Image Computation Architecture Group (PICA) at Georgia Institute of Technology [70]. Each PE has its own local memory, register file, a 4 x 4 photo detector to sample an image, MACC (multiply accumulator), ALU, barrel shifter, sleep unit, decoder, bus driver, and communication unit. The PEs communicate with neighboring PEs using a NEWS network. Since SIMD architecture executes a single instruction over a set of data in each PE, high throughput can be achieved where the data parallelism presents in a given algorithm. Especially, image processing applications have significant amount of data parallelism based on the algorithm. Thus, SIMD architecture can achieve high performance in multimedia application area and the performance is proportional to the number of PEs in a SIMD array. For the evaluation of our research, we extend a behavioral SIMD simulator, which has been developed by the PICA group [70], to support systolic instruction broadcast and systolic memory operations. Figure 81 shows the overall structure of SIMPil16 architecture.

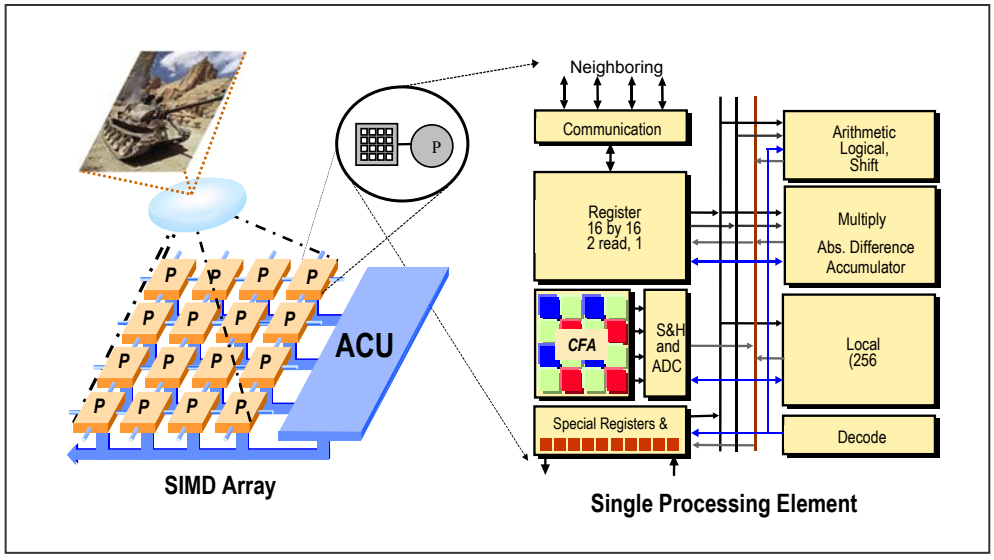


Figure 81: SIMPil microarchitecture.

References

[1] Kee Shik Chung, "ILP-SIMD: An Instruction Parallel SIMD Architecture With Short-Wire Interconnects," *Ph.D. Dissertation, Georgia Institute of Technology*, Atlanta, Georgia, April 2000.

[2] Ralph Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, vol. 23, no. 2, pp. 5-17, 1990.

[3] Wen-Tsong Shiue, Sathish Udayanarayanan and Chaitali Chakrabarti, "Data Memory Design and Exploration for Low Power Embedded Systems", *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 4, pp. 553-568, 2001.

[4] James D. Allen and David E. Schimmel, "Issues in the Design of High Performance SIMD Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 8, pp. 818-929, 1996.

[5] M. Bolotski, R. Amirtharajah, W. Chen, T. Kutscha, T. Simon and T. Knight Jr., "Abacus: A High-Performance Architecture for Vision," *Proceedings of International Conference on Pattern Recognition*, 1994.

[6] PixelFusion, Ltd.: http://www.pixelfusion.com

[7] Eberhard Zehendner, "Simulating Systolic Arrays on MasPar Machines," *Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97)*, New Frontiers of Information Technology, pp. 394-401, 1997.

[8] D. W. Blevins, E. W. Davis and R. A. Heaton, "Blitzen: A highly integrated massively parallel machine," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 150-160, 1990.

[9] MASPAR MP-2: http://csep1.phy.ornl.gov/mp2_guide/mp2_guide.html

[10] Todd E. Rockoff, "SIMD instruction cache," *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 67-75, 1994.

[11] C. E. Leiserson, Area-Efficient VLSI Computation, *ACM Doctoral Dissertation Award Series*, MIT Press, 1983.

[12] H. T. Kung, "Why Systolic Architectures?" *IEEE Computer*, vol. 15, no. 1, pp. 37-46, 1982.

[13] H. T. Kung, "Systolic Communication," *Proceedings of the International Conference on Systolic Arrays*, pp. 695 -703, 1988.

[14] J. A. B. Fortes, K. S. Fu and B. W. Wah, "Systematic Design Approaches for Algorithmically Specified Systolic Arrays," *Computer Architecture: Concepts and Systems*, *Chapter 11,* Elsevier Science Publishing Co., pp. 454-494, 1987.

[15] Kurtis T. Johnson, A. R. Hurson and Behrooz Shirazi, "General-Purpose Systolic Arrays," *IEEE Computer* , vol. 26, no. 11, pp. 20-31, 1993.

[16] J. A. B. Fortes and B. W. Wah, "Systolic Arrays – From Concept to Implementation," *IEEE Computer*, vol. 20, no. 7, pp. 12-17, 1987.

[17] J. A. B. Fortes, K. S. Fu and B. W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *Proceedings of the International Conference on Acoustics, Signal and Speech Processing*, vol. 1, pp. 8.9.1-8.9.5, 1985.

[18] Sek. M. Chai and S. Scott Wills, "Systolic Opportunities for Multidimensional Data Streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 4, pp. 388-398, 2002.

[19] Keshab K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, Wiley, John & Sons Inc., 1998.

[20] D. Lavenier, P. Quinton and S. Rajopadhye, *Advanced Systolic Design in Digital Signal Processing for Multimedia Systems, Chapter 23*, Parhi and Nishitani Eds, Marcel Dekker, Inc., 1999.

[21] S. Y. Kung, VLSI array processors, Prentice-Hall, Inc., 1987.

[22] D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Transactions on Computers*, vol. 35, no. 1, pp. l-12, 1986.

[23] Y. Hwang and Y. Hu, "On Systolic Mapping of Multi-Stage Algorithms," *Proceedings of IEEE Conference on Application Specific Array Processors*, pp. 47-61, 1992.

[24] S.K. Rao and T. Kailath, "Regular Iterative Algorithms and Their Implementation on Processor Arrays," *IEEE Proceedings*, pp. 259-269, 1988.

[25] Hans-Werner Lang, "Transitive Closure on an Instruction Systolic Array," *Proceedings of the International Conference on Systolic Arrays*, pp. 295-304, 1988.

[26] O. Y. de Vel and V. K. Murthy, "Programmable Systolic Arrays for Robotic and Computer Vision Systems," *Proceedings of the 3rd International Conference on Image Processing and its Applications*, pp. 506-510, 1989.

[27] Bertil Schmidth, Manfred Schimmler and Heiko Schroder, "Long Operand Arithmetic on Instruction Systolic Computer Architectures and Its Application in RSA Cryptography," *Proceedings of Euro-Par'98*, pp. 916-922, 1998.

[28] Bertil Schmidth and Manfred Schimmler, "A Parallel Accelerator Architecture for Multimedia Video Compression," *Proceedings of Euro-Par Conference (Euro-Par '99)*, pp. 950-960, 1999.

[29] Bertil Schmidth, Heiko Schroder and Manfred Schimmler, "Protein Sequence Comparison on the Instruction Systolic Array," *Proceedings of 6th International Conference on Parallel Computing Technologies (PaCT)*, pp. 498-509, 2001.

[30] Instruction Systolic Array Related Web pages: http://www.ntu.edu.sg/home/asheiko/Research.htm.

[31] Vivek Tiwari, Sharad Malik and Andrew Wolfe, "Compilation Techniques for Low Energy: An Overview," *Proceedings of the Symposium on Low Power Electronics*, pp. 38-39, 1994.

[32] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. G. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systmes," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149-206, 2001.

[33] N. Dutt, A.Nicolau, H. Tomiyama and A. Halambi, "New directions in compiler technology for embedded systems," *Proceedings of Asia and South Pacific Design Automation (ASP-DAC) Conference*, pp. 409–414, 2001.

[34] G. Araujo, S. Malik and M. Lee, "Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures," *Proceedings of the 33rd Design Automation Conference (DAC)*, pp. 591-596, 1996.

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

[36] C. Liem, P. Paulin and A. Jerraya, "Address calculation for retargetable compilation and exploration of instruction-set architecture," *Proceedings of the 33rd Design Automation Conference*, pp. 597-600, 1996.

[37] R. Leupers, "Novel code optimization techniques for DSPs," *Proceedings of the 2nd European DSP Education and Research Conference*, 1998.

[38] R. Leupers, "Code Generation for Embedded Processors," *Proceedings of the 13th International Symposium on System Synthesis (ISSS)*, pp. 173–178, 2000.

[39] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang and Albert Wang, "Code Optimization Techniques for Embedded DSP Microprocessors," *Proceedings of the 32nd ACM/IEEE Conference on Design Automation Conference*, pp. 599-604, 1995.

[40] S. Liao et al., *Code generation and optimization techniques for embedded digital processors*, Kluwer Academic Publishers, 1995.

[41] E. D. Greef, F. Catthoor and H. D. Man, "Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications," *Parallel Computing*, vol. 23, no. 12, pp. 1811-1837, 1997.

[42] E. D. Greef, F. Catthoor and H. D. Man, "Array placement for storage size reduction in embedded multimedia systems," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 66-75, 1997.

[43] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler Transformations for High-performance Computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345- 420, 1994.

[44] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988.

[45] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, vol. 17, no. 6, pp. 98-105, 1982.

[46] Preston Briggs, Keith D. Cooper, Ken Kennedy and Linda Torczon, "Coloring heuristics for register allocation," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 275-284, 1989.

[47] Preston Briggs, "Register Allocation via Graph Coloring," *Rice University, Center for Research on Parallel Computation (CRPC), Houston, TX , Tech Rep*. TR92-183, 1998.

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

[49] S. Hanono and S. Devadas, "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator," *Proceedings of the 35th Design Automation Conference (DAC)*, pp. 510-515, 1998.

[50] R. Leupers and P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models," *Proceedings of European Design and Test Conference (ED&TC)*, pp. 140-144, 1997.

[51] R. Leupers, "Compiler Design Issues for Embedded Processors," *Design & Test of Computers, IEEE*, vol. 19, no. 4, pp. 51-58, 2002

[52] David A. Barrett and Benjamin G. Zorn, "Using Lifetime Predictors to Improve Memory Allocation Performance," *Proceedings of SIGPLAN '93 Conference on Programming Language Design and Implementation*, vol. 28, pp. 187-196, 1993.

[53] David A. Cohn and Satinder Singh, "Predicting Lifetimes in Dynamically Allocated Memory," *Proceedings of Advances in Neural Information Processing Systems conference*, vol. 9, pp. 939-945, 1996.

[54] Nam Ling and Magdy A. Bayoumi, "The design and implementation of multidimensional systolic arrays for DSP applications," *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP-89)*, vol. 2, pp. 1142-1145, 1989.

[55] Martin C. Herbordt, Jade Cravy and Renoy Sam, "A System for Evaluating Performance and Cost of SIMD Array Designs," *Journal of Parallel and Distributed Computing*, vol. 60, no. 2, pp. 217-246, 2000.

[56] Johannes M. Mulder, Nhon T. Quach and Michael J. Flynn, "An Area Model for On-Chip Memories and its Application," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 2, pp. 98-106, 1991.

[57] Massoud Pedram, "Power Minimization in IC Design: Principles and Applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 3-56, 1996.

[58] N. Kavvadias, A. Zanikopoulos, Ch. Voliotidis, S. Kougia, A. Chatzigeorgiou, N. Zervas and S. Nikolaidis, "Power Exploration of Parallel Embedded Architectures Implementing Data-Reuse Transformations," *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, vol. 2, pp. 781-784, 2001.

[59] Milind B. Kamble and Kanad Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *Proceedings of the International Symposium for Low Power Electronics and Design*, pp. 143-148, 1997.

[60] M. Dasygenis, N. Kroupis, K. Tatas, A. Argyriou, D. Soudris and A. Thanailakis, "Power and Performance Exploration of Embedded Systems Executing Multimedia Kernels," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 3-56, 1996.

[61] Paul E. Landman and Jan M. Rabaey, "Activity-Sensitive Architectural Power Analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 571-587, 1996.

[62] Enrico Macii, Massoud Pedram and Fabio Somenzi, "High-Level Power Modeling, Estimation, and Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 11, pp. 1061-1079, 1998.

[63] J. C. Eble, V. K. De, D. S. Wills and J. D. Meindl, "A Generic System Simulator (GENESYS) for ASIC Technology and Architecture Beyond 2001," *Proceedings of the 9th Annual IEEE International ASIC Conference*, pp. 193-196, 1996.

[64] Nasser M. Nasrabadi and Robert A. King, "Image Coding Using Vector Quantization: A Review," *IEEE Transactions on Communications*, vol. 36, no. 8, pp. 957-971, 1988.

[65] Pamela C. Cosman, Robert M. Gray and Martin Vetterli, "Vector Quantization of Image Subband: A Survey," *IEEE Transactions on Image Processing*, vol. 5, no. 2, pp. 202-225, 1996.

[66] Dennis Sylvester, and Kurt Keutzer, "A Global Wiring Paradigm for Deep Submicron Design," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 19, no. 2, pp. 242-252, 2000.

[67] George A. Sai-Halasz, "Performance Trends in High-End Processors," *Proceeding of IEEE*, vol. 83, no. 1, pp. 20-36, 1995.

[68] International Technology Roadmap for Semiconductor (ITRS), 2001: http://public.itrs.net

[69] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens, "Register organization for media processing," *Proceedings of 6th High-Performance Computer Architecture (HPCA-6)*, pp. 375--386, January 2000.

[70] H. Cat et al., "SIMPil: An OE integrated SIMD architecture for Focal Plane Processing Applications," *Proceedings of 3rd International Conference on Massively Parallel Processing using Optical Interconnections*, pp. 44-52, 1996.

[71] Gregory A. Baxes, *Digital Image Processing – Principles and Applications*, John Wiley & Sons, Inc., 1994.

[72] T. C. Mowry, S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[73] S. P. VanderWiel, and D. J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *IEEE Computer*, pp. 23-30, 1997.

[74] A. J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, pp. 473-530, 1982.

[75] T-F Chen and J-L Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609-623, 1995.

[76] T-F Chen and J-L Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proceedings of 21st Annual International Symposium on Computer Architecture*, pp. 223-232, 1994.

[77] E. H. Gornish, and A. V. Veidenbaum, "An Integrated Hardware/Software Scheme for Shared Memory Multiprocessors," *Proceedings of International Conference on Parallel Processing*, pp. 281-284, 1994.

[78] Bruce Jacob, and Trevor Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, pp. 60-75, 1998.

[79] David A. Patterson, and John L. Hennessy, *Computer Organization & Design – The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., 1998.

Vita

Soojung Ryu was born on October 6, 1971 in Seoul, Korea. She completed her Bachelor's degree in Computer Science at Dong-Duk Women's University, Seoul, Korea, in 1994. She received her Master of Science in Communication and Information Engineering from Korea Advanced Institute of Science and Technology, Seoul, Korea, in 1996. She continued to work in database management area at Korea Advanced Institute of Science and Technology, Seoul, Korea as a research engineer until 1997.

She started her graduate studies in Electrical and Computer Engineering at Georgia Institute of Technology in 1998. She joined the Portable Image Computation Architecture research group in 1999. Under Dr. Scott Wills' and Dr. Linda Wills' supervision, she has been doing research on storage management for embedded SIMD architectures. Her research interests include high performance parallel architectures, portable multimedia embedded systems, image processing architectures, short-wire architectures, storage management techniques. She received her Ph.D. in Electrical Engineering in May 2004 from Georgia Institute of Technology.

She is happily married to Ji-Weon Jeong and they have a wonderful son, Daniel.