Stream Oriented Computations in Dataflow Execution Model and Application to String Matching Problems

By

JIN HWAN PARK

Master of Science

The Ohio University

Athens, Ohio

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1998

Thesis
1998D
P235s

COPYRIGHT

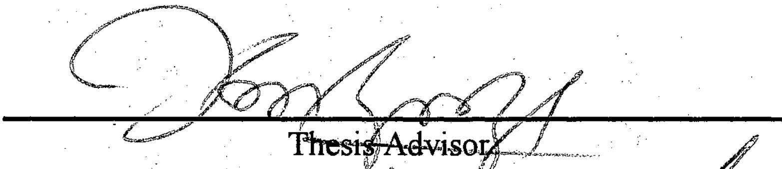
By

Jin Hwan Park

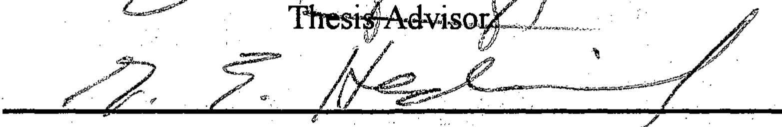
December, 1998

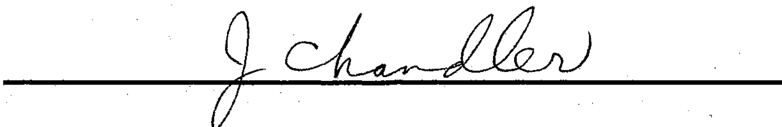
Stream Oriented Computations in Dataflow Execution Model and Application to String Matching Problems

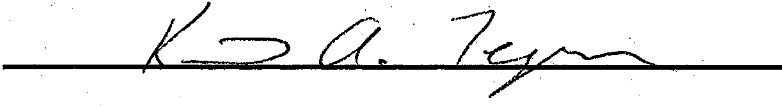
Thesis Approved:

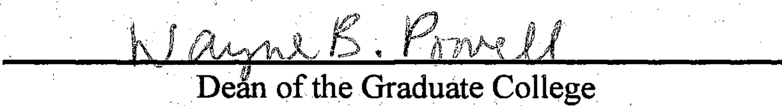


Thesis Advisor









Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my deep gratitude and sincere appreciation to my major advisor Professor K. M. George for his supportive suggestions, constructive comments, and critical insights throughout this thesis. My appreciation extends to my other committee members Professor J. P. Chandler, Professor G. E. Hedrick, and Dr. K. A. Teague for their valuable suggestions and encouragement during the course of this project.

TABLE OF CONTENTS

Chapter **Page**

I. **INTRODUCTION** 1

1.1 Overview of Application to String Matching Problems 3

1.2 The Organization of the Thesis 5

II. **PRELIMINARIES** 7

2.1 Dataflow Execution Models: Implicit Parallelism 7

2.1.1 Classifications 9

2.1.1.1 Static/dynamic classification 9

2.1.1.2 Micro/Macro classification 12

2.1.2 Prototypes and Implementations 13

2.2 Representation of Streams in Dataflow Execution Models 13

III. **HISTORY SENSITIVE COMPUTATION PROBLEM** 18

3.1 Primitive Models 18

3.2 Approaches 20

3.2.1 Recursive Scheme 20

3.2.2 Iterative Scheme 23

3.2.3 Automata Based Scheme 25

IV. **PROPOSED APPROACH: THE MEMORYLESS SCHEME** 28

4.1 Accumulator Based Model 28

Chapter	Page

4.2 Working-Set Based Model 30

4.3 Language Constructs 34

V. COMPARISON 36

5.1 Iterative Scheme 38

5.2 Recursive Scheme 38

5.3 Automata Based Scheme 39

5.4 The Memoryless Scheme 39

VI. EXPLOITING MAXIMUM PARALLELISM 40

6.1 Explicit Parallelism in Accumulator Based Model 41

6.1.1 Forwarding Mechanism (PACC) 42

6.2 Explicit Parallelism in Working-Set Based Model 47

6.2.1 Alternative Representation of WS Block 47

6.2.2 Forwarding Mechanism (PWS) 50

6.3 Performance Measurements 55

VII. APPLICATION TO PARALLEL PREFIX COMPUTATION 61

7.1 Prefix Computation Problem 61

7.2 Implementation Methodologies 62

VIII. APPLICATION TO PARALLEL STRING MATCHING 65

8.1 String Matching Problem 65

8.2 Problem Definitions and Related Work 67

8.2.1 K-differences Problem 68

Chapter **Page**

8.2.2 K-mismatches Problem 69

8.2.3 Exact Matching Problem 69

8.2.4 Need for Special Purpose Architecture 70

8.2.5 Hardware Approaches 71

8.2.5.1 K-differences problem 71

8.2.5.2 Exact matching problem 74

8.3 Design Strategy 75

IX. K-DIFFERENCES PROBLEM 78

9.1 Edit Distance Computation 78

9.2 Dataflow Scheme: Implicit Parallelism 81

9.3 Performance 87

9.4 Implementation Methodologies 89

X. K-MISMATCHES PROBLEM 91

10.1 Edit Distance Computation with K-mismatches 91

10.2 Dataflow Scheme: Implicit Parallelism 93

10.2.1 Hierarchical Scheme 94

10.2.2 Linear Scheme: Linear representation of the computation part 98

10.2.3 Broadcasting scheme 103

10.2.4 Performance 106

10.3 Parallelization with Multiple Streams: Explicit Parallelism 109

10.3.1 Parallel Approaches for Hierarchical and Linear Schemes 111

Chapter **Page**

10.3.2 Parallel Broadcasting Scheme 116

10.3.3 Performances 126

10.4 Implementation Methodologies 133

10.4.1 Hierarchical Scheme 133

10.4.2 Linear Scheme 135

10.4.3 Broadcasting Scheme 137

XI. EXACT MATCHING PROBLEM 139

11.1 Alternative Schemes 139

XII. CONCLUSION 143

REFERENCES 147

LIST OF TABLES

Table **Page**

1. Characteristics of string matching subproblems 67

2. Time complexities of schemes for k-mismatches (and exact matching) problem .. 132

LIST OF FIGURES

Figure		Page
1.	Static / dynamic dataflow computers	11
2.	Static and dynamic representations of stream	16
3.	Recursive scheme of history sensitive computations	21
4.	Redundant memory accesses in working-set (window size =3)	24
5.	I-Structure memory	25
6.	Generic actions of automata based scheme	26
7.	Memoryless scheme for the accumulator based model	29
8.	Memoryless scheme for the working-set based model in static dataflow environment	30
9.	Snap shots of working-set based model with window size 3	31
10.	Memoryless scheme for the working-set based model in dynamic (tagged token) dataflow environment	32
11.	The MAV problem with the Memoryless scheme in dynamic (tagged token) dataflow environment	33
12.	Dataflow graphs for Fibonacci number generation	37
13.	Input/output arrangements of forwarding mechanism (PACC(d))	42
14.	Concept of forwarding mechanism with degree d=3 case (PACC(3))	43
15.	Optimized forwarding mechanism (PACC) with binary operators	45

Figure		Page

16. Forwarding mechanism (depth 3 case) in dynamic (tagged token) dataflow environment 46

17. Alternative scheme for the working-set based model (window size 3 case) 48

18. Input/output arrangements of the forwarding mechanism PWS(d*dm) 50

19. Concept of the forwarding mechanism 51

20. Examples of forwarding mechanisms generated by Algorithm-1 53

21. Forwarding mechanism (depth = 3, window size = 3 case) for the working-set based history sensitive problems in static dataflow environment 54

22. Degree 3 forwarding mechanism (window size m=3 case) in dynamic (tagged token) dataflow environment 55

23. Dependency graphs in accumulator based model 56

24. Pipeline performances with stream size (N) and pipeline depth (d) 57

25. Parallel prefix sum computator of size 4 (pipeline depth=4) 62

26. Latch arrangements in parallel prefix computator (pipeline depth=5 case) 64

27. System block diagram of general purpose computer 71

28. 2-dimensional structure for string distance computation by Cheng and Fu 72

29. High level dataflow in parallel scheme (m=4 case) 81

30. Behavior snap shots of WS(m) block in the scheme (m=3 case) 82

31. Refined scheme for approximate string matching (m=4 case) 83

32. Refined operation block "Main" 84

33. Dataflow graph (static) for the operation block "min" 85

34. Linear systolic array for the k-differences problem 86

Figure **Page**

35. Data dependencies and paralleism on table D (m=4 case) 87

36. Parallel evaluation of the table D (m=4 case) 88

37. Structure of each PE_i for the k-differences problem 90

38. Data dependencies & conceptual timing on table D' (m=4 case) 95

39. Dataflow scheme for the k-mismatches problem (pattern length m=4 case) 96

40. Evaluation timing of the linear scheme on table D' 99

41. The linear scheme for the k-mismatches problem 99

42. Behavior of each global block (GB) in the computation part 100

43. Refined linear scheme for the k-mismatches problem (m=4 case) 100

44. Linear computation part for the static dataflow environment (m=4 case) 102

45. Concept and dataflow of the broadcasting scheme 103

46. The broadcasting scheme (m=4 case) 104

47. Time analysis on dataflow schemes for the k-mismatches problem 107

48. Evaluation of table D' by the linear and the broadcasting schemes (m=4 case) 109

49. Concept of the parallel hierarchical / linear schemes 111

50. Examples of parallel schemes (hierarchical / linear) 112

51. Parallel string matching for the k-mismatches problem (d=3, m=4 case) 113

52. Outputs from the parallel hierarchical / linear schemes 115

53. Concept of the parallel broadcasting scheme and input arrangement 117

54. Conceptual view of the parallel broadcasting scheme 118

55. Diagonal entries to each computation part (BCP) 120

56. Examples of parallel broadcasting blocks 121

Figure		Page

57. Parallel broadcasting scheme for k-mismatches problem (d=3, m=4 case) 122

58. Inputs to BCP before and after the alignment mechanism 122

59. Outputs from the parallel broadcasting scheme 126

60. Evaluation of table D' by parallel hierarchical scheme (d=3, m=4 case) 127

61. Entries of table D' available on time slices of the parallel hierarchical scheme (d=3, m=4 case) 127

62. Evaluation of table D' by parallel linear scheme (d=3, m=4 case) 128

63. Entries of table D' available on time slices of the parallel linear scheme (d=3, m=4 case) 129

64. Evaluation of table D' by parallel broadcasting scheme (d=3, m=4 case) 130

65. Total execution times on serial and parallel broadcasting schemes 131

66. Entries of table D' available on time slices of the parallel broadcasting scheme (d=3, m=4 case) 132

67. Hierarchical array of cells for the k-mismatches problem (serial scheme with m=4 case) 134

68. The structures of basic cells "S-Eq" and "Eq" 135

69. Systolic array of PEs for serial linear scheme (m=4 case) 135

70. Structure of each PE of the linear scheme 135

71. Latches used in implementation of the linear scheme (m=4 case) 136

72. Systolic array of PEs for serial broadcasting schem (m=4 case) 137

73. Structure of each PE (PE_i) of the broadcasting scheme 138

Figure **Page**

74. Alternative hierarchical scheme for the exact matching problem (serial, m=4 case) ... 141

75. Alternative linear scheme for the exact matching problem (serial, m=4 case) ... 141

76. Alternative broadcasting scheme for the exact matching problem (serial, m=4 case) ... 141

77. Refined dataflow graph (static) for the operation block "And" ... 142

Chapter I

INTRODUCTION

As an attractive parallel computation model, dataflow architectures have been proposed and developed steadily [8, 29, 42, 45, 71, 72]. The goal of the efforts has been exploiting maximum parallelism which is inherent in the asynchrony and functionality principles of the dataflow model. The asynchrony principle implies that an operation (instruction) is executable if all its required operands are available based on data driven mechanism. Since the functionality principle implies that all operations are side-effect-free functions, any set of enabled operations can be executed in parallel. In contrast to the centralized control used in conventional von Neumann computers, distributed control mechanism inherent in the implicit instruction level parallelism provides challenging performance improvements [44, 75, 83]. Thus it makes the dataflow execution model one of attractive parallel architectures.

In spite of the advantages of dataflow execution model, obstacles such as structured data (array) handling problem [11, 38, 39, 59], resource management problem [27, 87], and real time situation like history sensitive computation problem degrade the performance of dataflow computers [2, 4, 14]. The history sensitive computation problem is caused by ahistoric nature of the data driven mechanism. In history sensitive computations, current output depends on both current input and history of inputs. In other words, history sensitive function memorizes the previous inputs and this contradicts the data

driven mechanism. Thus, efficient handling of history sensitive computations in dataflow execution model is unavoidable to preserve high performance parallel processing.

Developing elegant solution for handling the history sensitive problems in dataflow execution model has important role in the digital signal processing. That can provide efficient systolic algorithms for the prefix computation problem and the string matching problem. Those systolic algorithms are parallel hardware solutions to the problems since they can be used to design special purpose VLSI chip for those problems.

In dataflow environment, with stream data type [21, 31, 38, 50] history sensitive computations can be implemented in either static or dynamic way. In static scheme, history sensitive functions operate on indirect-access (stored in memory) stream data structures iteratively or recursively. The history of inputs are reserved in stored stream data structure itself. In contrast, dynamic scheme does not use stored stream data structure. Individual tokens flow as a stream and history sensitive functions receive this dynamic stream as input. The automata based approach proposed in [14] is an example of this scheme. Since dynamic scheme does not use static stream (stored in memory), it has to provide other mechanisms to keep the required history of inputs. In fact, the automata based approach also uses memory spaces for maintaining the required history. These implementation schemes are briefly reviewed with analysis of their shortcomings in Chapter III. In addition to their major shortcomings, existing schemes use memory references during computations and these incur memory management overhead. Thus, they take one step back to the von Neumann model and do not meet the principles of dataflow model. Therefore, researchers have been motivated towards the design of

elegant solutions to the history sensitive computation problems which meet the principles of the dataflow model and provide high performance.

The first major concern of this dissertation is presenting a new approach in a pure dataflow way to handle history sensitive computations in dataflow execution model. We call this a memoryless scheme. The pure dataflow way means that it does not use memory locations to store the stream data structure or to keep the history of inputs needed in computation. Only tokens flow dynamically along the arcs of dataflow graph and there are no memory references. Thus, it is free from memory operations. Based on multiple streams, parallelization mechanisms for the memoryless scheme are also presented.

The other major concern is applying the memoryless scheme to real problems including the prefix computation problem and the string matching problem. The memoryless scheme with parallelization mechanisms provides high performance dataflow and hardware solutions to the problems.

1.1 Overview of Application to String Matching Problems

Later chapters of the dissertation present parallel solutions to string matching problems. The solutions are based on the memoryless scheme for working-set based history sensitive computations. Both serial and parallel memoryless schemes are used to provide dataflow (i.e. systolic) solutions to the problems. Solutions are presented for three subproblems known as k-differences, k-mismatches, and exact matching problems. Since the memoryless scheme for the working-set based history sensitive computations (WS) and its parallel approach (PWS) are pure dataflow schemes, the solutions are able to work on

actual dataflow machines and suitable for VLSI implementation. The scheme for the k-differences problem is a parallel algorithm for the dynamic programming method [90] of evaluating minimum edit distances between pattern and any substring of reference string. The scheme uses WS(m) block and the time complexity is O(n + m) to evaluate n * m minimum edit distance table where n and m are the lengths of reference and pattern strings (n >> m). For the k-mismatches and the exact matching problems, three different approaches namely the hierarchical, the linear, and the broadcasting schemes are presented with linear time complexities $O(n + \alpha)$, where $0 \leq \alpha \leq \log m$. The hierarchical and the linear approaches use WS(m) block and, the broadcasting approach uses BC(m) block which is a variation of the WS(m) block. For these two subproblems, further parallelism is gained by using PWS and PBC blocks ,which are parallel versions of WS and BC blocks, based on multiple stream input and output. Time complexities of those parallel schemes are $O((n/d) + \alpha)$, where d represents the controllable degree of parallelism (number of streams used). The variable α is log m for the parallel hierarchical and, m for the parallel linear and the parallel broadcasting schemes. These pure dataflow schemes present methods to design special purpose systolic array hardware for string matching. The designs are linear or hierarchical systolic array of few basic cells. They can process any length reference string and easily extendible for any length pattern. Designs for the parallel schemes including the parallel linear and the parallel broadcasting, which are used for exploiting explicit parallelism on the k-mismatches and the exact matching problems, need d*m PEs, where d is controllable degree of explicit parallelism.

1.2 The Organization of the Thesis

This thesis consists of two major parts. The first part is devoted to the development of efficient schemes for handling history sensitive computation problems in dataflow execution models. The second part applies these methods to solve real life problems such as prefix computation problem and string matching problem. Parallel solutions to those problems are provided in dataflow execution models and are easily converted to hardware solutions to the problems.

The first part consists of Chapter II, Chapter III, Chapter IV, Chapter V, and Chapter VI. In Chapter II, dataflow execution models, which are instruction level parallel architectures, are introduced briefly. Architectural classifications are provided. Since the history sensitive computations belong to stream oriented computations, representations of stream data types in dataflow execution models also are described in this chapter. Chapter III describes the history sensitive computation problems and approaches for solving problems in dataflow execution model. The problems are presented abstractly in two models and both models are defined in this chapter. The proposed scheme, namely "the memoryless scheme", is described in detail in Chapter IV. The scheme is represented in both static and dynamic dataflow execution models. Language constructs of the memoryless scheme are presented in this chapter. Chapter V presents comparison of the memoryless scheme to other approaches using simple example. Then Chapter VI describes the methods of exploiting explicit parallelism in the memoryless scheme. Parallelization mechanisms for both models of history sensitive computation problems are provided with the performance measurements.

The second part consists of Chapter VII, Chapter VIII, Chapter IX, Chapter X, and Chapter XI. In these chapters, the memoryless scheme of handling history sensitive computations are applied to solve problems of prefix computation and string matching. Parallel solutions to these problems are provided in dataflow execution models and these easily are converted to hardware solutions to the problems. In Chapter VII, dataflow and hardware solutions to the parallel prefix computation problems are described. Remaining chapters devote to the string matching problems. Three subproblems (i.e. k-differences, k-mismatches, and exact matching problems) of string matching are discussed in those chapters. Chapter VIII describes the string matching problems and related work in both software and hardware approaches. Since the dataflow solutions are suitable for building special purpose hardware for the string matching tasks, descriptions focus on hardware approaches. In Chapter IX, parallel solution to the k-differences problem which is a version of the approximate string matching problem is described. Chapter X provides parallel solutions to the k-mismatches problem which is the other version of the approximate string matching problem. Finally, Chapter XI describes parallel solutions to the exact string matching problem. The solutions are based on the memoryless scheme for handling history sensitive computations. Performance evaluations and implementation methodologies are provided in each chapter.

Chapter II

PRELIMINARIES

2.1 Dataflow Execution Models: Implicit Parallelism

Most parallel computer architectures are MIMD schemes employing interconnections of conventional von Neumann architectures. Dataflow architectures are completely different from conventional von Neumann computer architectures and are recognized as attractive parallel architectures.

The traditional von Neumann processors have fundamental characteristics that reduce effectiveness of parallel computers. First, their performances suffer from presence of long memory and communication latencies, and these are unavoidable in parallel machines. Second, they do not provide good synchronization mechanisms for frequent task switching between parallel activities, also inevitable in parallel machines. In addition, traditional programming languages are not easily extended to incorporate parallelism. These conventional machines have control-driven organizations. This means that the program has complete control over instruction sequencing. Synchronous computations are performed in control flow (von Neumann) computers using centralized control.

In contrast, dataflow computers have data-driven organizations that are characterized by passive examine stages. An instruction is executable if all its operands are available and arrival of operands activates execution of an instruction. Since this is the only execution sequencing constraint, many instructions can be executed simultaneously and

asynchronously and high degree of implicit parallelism is expected in dataflow computation models. The following are some major advantages of dataflow computation model [47, 95, 96].

- Highly concurrent operations : Parallelism easily can be exposed in a dataflow program graph.
- Matching with VLSI technology : The homogeneity and modularity in cellular structures contribute to the suitability of VLSI implementation of major components in a dataflow computer.
- Programming productivity : A well designed dataflow computer should be able to remove the bottleneck caused by assorted scalar operations in von Neumann machine. In particular, a dataflow language such as Id [10] provides an elegant method for writing concurrent programs.

On the other hand, there exist a number of shortcomings and technical problems to be solved to realize practical dataflow computers.

The typical criticisms of dataflow computers from the architectural view point include :

- A large amount of hardware is needed, and in particular, the matching memory for data synchronization is complex.
- The packet communication network cost is high.
- Not suited to handle structure (array or list) processing.
- Fine-grain parallel processing causes an increase in parallel processing overhead.
- Performance degradation occurs at points of low parallelism within a program.

- The number of instructions executed is relatively more than that of von Neumann computer.

Besides these disadvantages, there are other problems. One of them is that no strategy for full utilization of processing elements has been developed and the other is that resource management is difficult to implement and has much overhead. Since the basic principles were first introduced about 20 years ago, there have been many research projects on dataflow computers under development in the United States of America, Europe, Japan, and Australia [1]. But none of current dataflow machines proves that dataflow computers can surpass conventional von Neumann parallel machines and vector type super computers since the prototypes constructed were small and experimental.

Since the implementation methodologies of dataflow concept are different there are many kinds of prototypes of dataflow machines. We consider these dataflow architectures in two classification terms, namely Static/Dynamic, and Micro/Macro.

2.1.1 Classifications

2.1.1.1 Static/dynamic classification

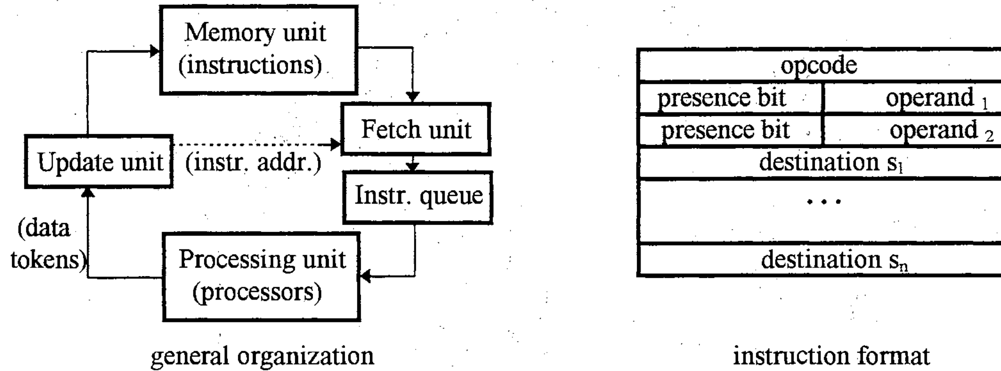
Early dataflow machines traditionally are classified as either static or dynamic. Depending on the way of handling data tokens, dataflow computers are divided into two groups namely static and dynamic models. In the static model, only one token exist on an arc at a time and control tokens are used to acknowledge the proper timing in transferring data tokens from node to node. When an instruction receives all required tokens it is enabled in this model. Jack Dennis and his research group at the MIT has

developed the static machine [29], and McGill university's static "Argument-fetching dataflow architecture" was a coproject with Dennis at MIT [46]. An interesting feature of the "Argument-fetching" dataflow architecture is that only signals flow through the system instead of data. The reason for proposing such an architecture was to answer some criticisms raised against dataflow architectures concerning the unnecessary data movement, an inefficiency in previously proposed architectures [46].

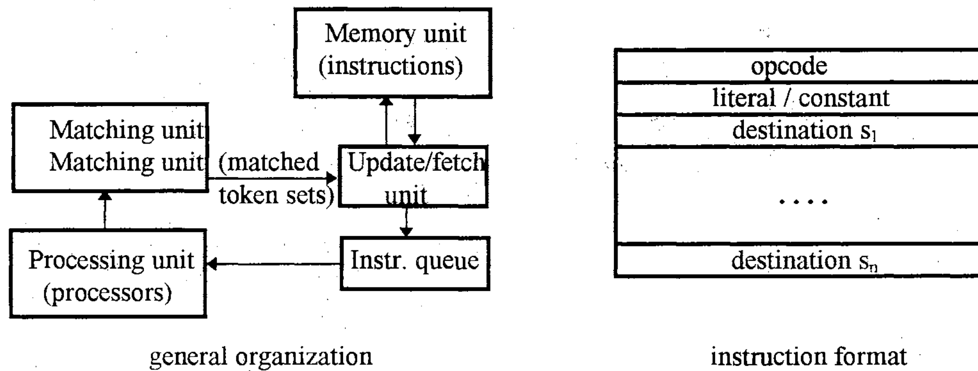
In the dynamic model, more than one token can exist on an arc at a time, and it uses tagged tokens. This dynamically tagged dataflow model suggests that maximum parallelism can be exploited from a program graph. In this model, the system synchronization is based on matching mechanism. Representative dynamic dataflow machines are MIT's "TTDA (Tagged Token Dataflow Architecture)" machine [10, 71, 72, 84], "Manchester" dataflow machine in England [41, 78], and the "EDDY" system in Japan [95].

Both static and dynamic architectures have a pipelined ring structure. Figure-1(a) shows general organization of the static dataflow computer and its instruction format. Figure-1(b) illustrates general organization of the dynamic dataflow computer and its instruction format. There are some advantages and disadvantages of both architectures. In the static machine, hardware required is quite simple, but it is difficult to construct the graph to guarantee one-token per arc restriction. Another problem is that since nodes are permanently assigned to processors, a particular processor may attract an unequal share of the work load. In the dynamic machine, loop unfolding feature can increase the amount of parallelism significantly [1]. Major disadvantages of the dynamic model are

that it has the added complexities of tagging and untagging, the increased network traffic and the resorting of data.



(a). Static dataflow computer



(b). Dynamic dataflow computer

Figure-1. Static / dynamic dataflow computers

There have been approaches to combine these static and dynamic architectures. The "RMIT" dataflow computer combines both architectures by using advantages of both models [1]. Its architecture allows efficient execution of pipelined data sets without the disadvantages of one-token per arc and, the high degree of parallelism obtained in dynamic model without the necessity of always tagging data.

2.1.1.2 Micro/Macro classification

There are two levels of dataflow executions; instruction level dataflow which is also called micro-dataflow (or fine-grain), and procedure level dataflow which is also called macro-dataflow (or large grain). In micro-dataflow, all the instructions are allocated to each PE and each instruction of a procedure is distributed among a set of PEs. In macro-dataflow, PE allocation serves every procedure. Therefore it is possible to construct a macro-dataflow computer by interconnecting several von Neumann type PEs.

Including early dataflow machines such as Dennis' static machine and Arvind's dynamic machine, later pure dataflow machines Monsoon [72] and Epsilon-2 [42] are classified in the micro level dataflow execution model. "Monsoon" uses directly-addressed frames instead of an associative wait-match memory, showing a similarity to von Neumann machines but, it has pure dataflow architecture in the sense that tokens not only schedule instructions but also carry data [72].

On the other hand, there have been research trends on macro (procedure level) dataflow architectures which synthesize dataflow and von Neumann architectures. They are hybrid of von Neumann and dataflow architectures. Nikhil and Arvind suggested "P-RISC" multiprocessor for the hybrid architecture [68]. Closest to "P-RISC" is Iannucci's hybrid architecture [48]. Japanese machines called "TOPSTAR" and "EM-4" are also based on von Neumann/dataflow hybrid architectures [81, 86]. Other hybrid approaches include "LDF 100" [53], *T [69, 70] , and "Threaded Abstract Machine [26].

2.1.2 Prototypes and Implementations

Some proposed dataflow machines such as MIT's Tagged Token machine [10], Monsoon [72], England's Manchester [78] machine, and Japan's Sigma-1 [45] are fully implemented and most of others have been in prototype stage. The Sigma-1, which have been built in Japan's Electro-Technical Laboratory, has been considered as the most complete and impressive dataflow machine to date [45]. This is a representative contemporary dataflow machine.

The practical goal of the Sigma-1 project has been to construct a large-scale, scientific parallel computer consisting of instruction level dataflow PEs which executes practical application programs at an average speed of 100 MFLOPS. The current implementation consists of 128 processors, 128 I-Structure stores, 32 local communication networks, a global communication network, 16 maintenance microprocessors and a host computer. The PEs and structure elements are divided into 32 groups, each of which consists of four PEs, four structure elements and a local communication network. All groups are connected via the global communication network. The whole system is synchronous and operates under a single clock, 100 nano second. This implementation has gone into operation and demonstrated a performance of 170 MFLOPS on a small integration problem [10, 45].

2.2. Representation of Streams in Dataflow Execution Models

A stream can be defined as a sequence of values (tokens), all of the same type, that are passed sequentially according to the order of generation between program modules. The end of stream is indicated by the special marker EOS (end of stream) and the stream is

produced by some producer module with fixed order and consumed by one or more consumer modules in the same order. This order of time property makes stream suitable to manipulate the history sensitive computations since the production order has the meaning of the history. Another role of stream is that it provides the synchronization mechanism among parallel processes by sending and receiving stream of tokens with the action of deferred access (waiting) of non-produced token. Stream is also considered as an alternative for the fixed sized data structure like array in the case of iterative filling and iterative consumption.

In a dataflow environment, streams are represented in either static or dynamic way. In static representations, stream is treated as a data structure and stored in memory space similar to the I-structure [11]. Static representations can be divided into the following three cases:

(1). Dennis and Weng [31] proposed the method of handling the stored stream as a binary tree such as the list in functional programming languages (Figure-2(a)). Stream is built recursively and operations on it are Append, First, and Rest as defined for list manipulation. Thus, stream generation module appends a data to the end of list, and the consumer module accesses the data from the top (root) of the tree. This scheme can be implemented with linked list in memory space and thus, dynamic memory allocation bottleneck should be handled efficiently (i.e. fast enough). The list processing is inherently sequential though pipelining can exploit parallelism.

(2). With the I-structure [11], representations like packed stream method [50] store stream structures in the buffer of consecutive one-dimensional array like memory spaces. In this scheme, the stream must be finite length and the size must be known to allocate the

memory spaces for the buffer. Thus strict implementation is expected but the parallel access is possible with this scheme. For example, in the I-structure a producer dataflow graph writes into an I-structure location while several other consumer dataflow graphs read that location. However the semantics require that consumers should wait until the value becomes available. More detailed description of this deferred access mechanism will be examined in the next chapter.

In the stream buffer representations, the order of stream generation is kept in the index of the structure. Figure-2(b) shows the general conceptual view of these buffered stream representations.

(3). The third case is a hybrid of these two schemes. A paged-memory scheme [21] stores a stream as a linked list of paged memories. When the current page is full, a new page is assigned and starts to fill.

All these static schemes provide deferred access (waiting for unwritten data) mechanisms and the pointer to the stream structure flows as a token to the consumer processes. Stored streams are different from array structures with certain restrictions, such that once written an element cannot be updated and elements must be accessed in sequence. In general, the major advantages of static representations include:

- Entire stream is passed as a token (pointer to stream) to other functions.
- Nested stream structures can be represented.

But there are also disadvantages and the leading ones are:

- Since this approach uses memory references for manipulating the stream, it violates dataflow principles and thus goes one step back to the von Neumann model.

- Complexity of memory allocation and deallocation processes.
- Inefficiency caused by queuing activity of deferred access (read request for unwritten data) mechanism.

In contrast, dynamic representation of stream does not use memory and thus it provides a true dataflow mechanism. In this approach, each stream element is treated as an individual token and we can imagine easily that tokens flow sequentially according to the order of generation along the arcs of dataflow graph (Figure-2(c)). Considering the case of generated stream elements accumulated on certain arcs of the dataflow graph (this is possible when stream generation module iteratively generates elements in parallel or pipelined manner), there should exist the mechanism to keep the order of each stream element. One simple method to solve this problem is using a FIFO queue such that tokens are queued on each arc of the dataflow graph in their arrival order [35]. Of course this possibly brings about the cost of handling large queues. Most popular solution is using tagged token [10, 35, 38, 39]. In this method, each stream element carries its index (or iteration level) as a part of the tag (color) and tag of each stream element recognizes the order of production.

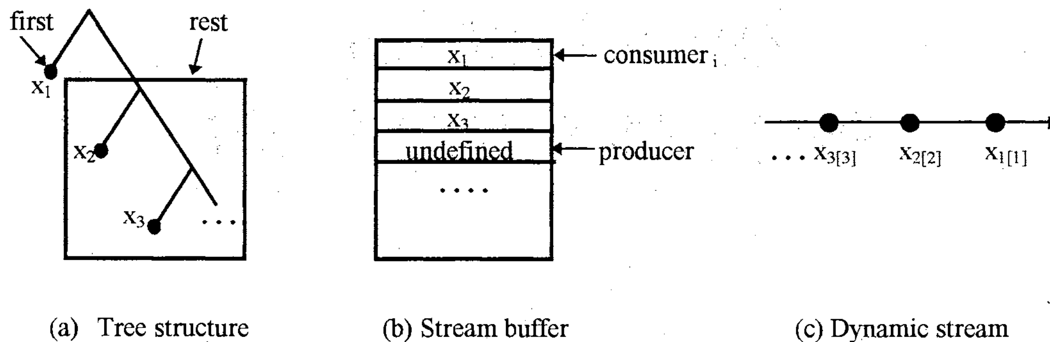


Figure-2. Static and dynamic representations of stream

In the stream generation module, each subsequent element is produced with unique increasing index value and sent to consumer modules. By using token relabeling scheme [38, 39] or actions like tag actor D for the U-interpreter [9], each stream element can proceed.

With the advantages of real dataflow mechanism (no memory references) and its simplicity, there are some drawbacks in the dynamic stream representations:

- Token duplication overhead when stream elements are sent to multiple consumers.
- Difficulty in manipulating nested streams (stream of streams).
- Entire stream can not be passed as a single parameter (token) to functions.
- With the tagged token method, tagging and matching mechanisms need additional hardware costs.

With this basic knowledge of streams, we will discuss the ways of handling history sensitive computations in dataflow execution models in following chapters.

Chapter III

HISTORY SENSITIVE COMPUTATION PROBLEM

With series of input values, the history sensitive computation module generates series of output values in which each (current) output value is dependent on both corresponding (current) input value and history of input values. In this chapter, we define two primitive models of history sensitive computation problems and analyze existing approaches for handling the problems.

3.1 Primitive Models

There are several real time situations which require history sensitivity but most of them can be grouped in two primitive models, namely accumulator based and working-set based models. With little variations or mixtures of these, we can represent various history sensitive problems. We define these two models with the following input and output streams of values.

Input stream : $x_1, x_2, x_3, x_4, x_5, \ldots$

Output stream : $y_1, y_2, y_3, y_4, y_5, \ldots$

In the accumulator based model, each output y_i is defined as

$y_1 = x_1$, and

$y_i = y_{i-1}$ <op> x_i; (for all $i \geq 2$, and <op> represents an operation)

in which x_i is current input and y_{i-1} is result of applying <op> to all previous inputs.

Of course, this definition stands for only history sensitive part of a compound problem which consists of history sensitive and non history sensitive parts. Some examples of this model are cumulative sum, cumulative product, cumulative average, and the difference between cumulative average and current input computation problems. In this paper, the following cumulative sum computation problem is used as a simple example of this model.

[**Example 1**] Cumulative Sum Problem.

With input stream { $x_1, x_2, x_3, x_4, x_5, \ldots$},

the output stream { $y_1, y_2, y_3, y_4, y_5, \ldots$} is defined as:

$y_1 = x_1$, and

$y_i = y_{i-1} + x_i$; (for all $i \geq 2$)

The working-set based model is defined with some function (operations) f on a fixed number (working-set window size) of the most recent inputs. More formally, each output y_i with working-set window size n is defined as:

$y_1 = f(x_1, 0, \ldots\ldots, 0)$,

$y_2 = f(x_2, x_1, 0, \ldots\ldots 0)$,

....

$y_{n-2} = f(x_{n-2}, x_{n-3}, \ldots, x_2, x_1, 0, 0)$,

$y_{n-1} = f(x_{n-1}, x_{n-2}, x_{n-3}, \ldots, x_2, x_1, 0)$, and

$y_i = f(x_i, x_{i-1}, x_{i-2}, \ldots x_{i-(n-1)})$; (for all $i \geq n$)

in which function f has n arguments and each y_i is defined with current input x_i and n-1 most recent inputs.

With stream of inputs, computing the difference between current input and one previous input (thus working-set window size = 2), and computing sum (product, or average) of the most recent n (working-set window size = n) inputs are examples of problems belonging to this model. The problem of computing average of the most recent n inputs is used as an example of this model in this thesis. Since this problem is introduced in [14] under the name of Moving Average Problem, we use the same name in the following example:

[**Example 2**] Moving Average Problem with working-set window size 3.

With input stream $\{ x_1, x_2, x_3, x_4, x_5, \ldots \}$,

the output stream $\{ y_1, y_2, y_3, y_4, y_5, \ldots \}$ is defined as:

$y_1 = x_1 / 3$,

$y_2 = (x_2 + x_1) / 3$, and

$y_i = (x_i + x_{i-1} + x_{i-2}) / 3$;(for all $i \geq 3$)

3.2 Approaches

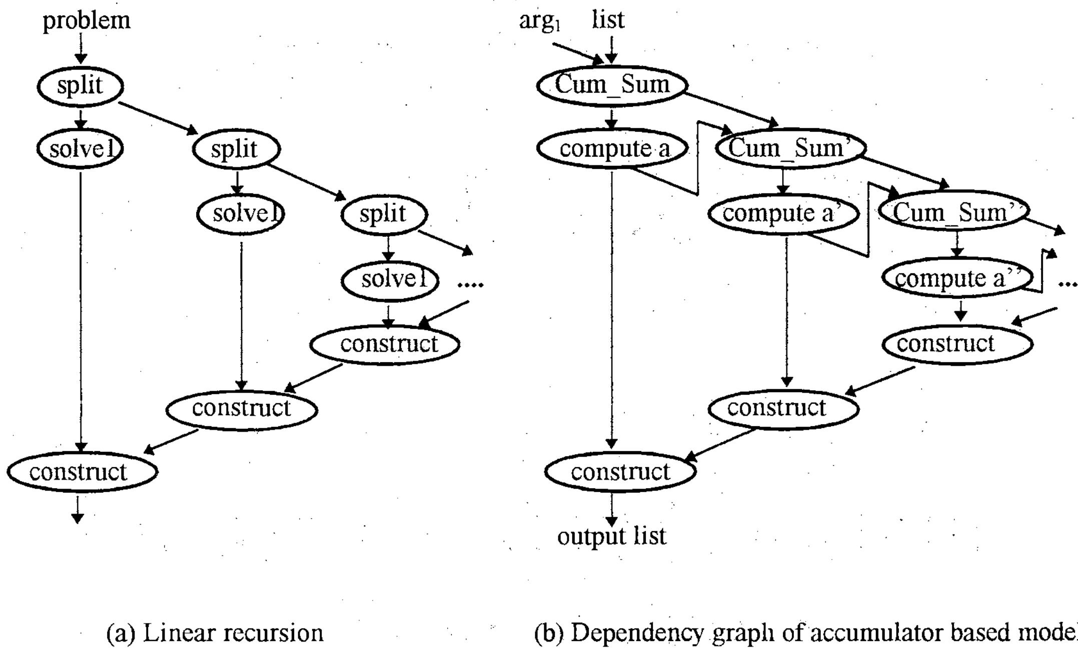
Static schemes use indirect-access (memory reference) stream data structures which are stored in memory locations. With stored stream structure, history sensitive computations are processed recursively or iteratively. We name these the recursive scheme and the iterative scheme respectively in this thesis. In contrast, dynamic schemes use direct-access data elements which flow dynamically as individual tokens.

3.2.1 Recursive Scheme

In the recursive scheme, binary tree structured representation of stored stream [31] is used and treated same as the list in functional programming environment. Since the list's

data structure is defined recursively, functions applied to list are evaluated recursively. The recursive evaluation structures are classified into tree structure (divide and conquer) and linear structure (tail recursion) [4, 5].

For the accumulator based history sensitive computation problems, the divide and conquer scheme cannot be used because the problem can not be split into equal sized sub-problems. The working-set based model also has difficulty for adapting the divide and conquer scheme and thus can be represented using linear structure. In general, the linear recursion scheme illustrated in [5] for solving a problem is shown in Figure-3(a).



(a) Linear recursion

(b) Dependency graph of accumulator based model

Figure-3. Recursive scheme of history sensitive computations

The cumulative sum problem, which is an example of accumulator based model of history sensitive problems, can be represented in functional form as follows:

```
Function Cumulative_sum (arg1, list)  /* initially arg1=0 */
  = construct (a, l)
    where {a = First(list) + arg1,
           l = Cumulative_sum (a, Rest(list))}
```

Figure-3(b) shows the dependency graph of this.

For the working-set based model of history sensitive problems, the moving average problem with working-set window size 3 defined in Example-2 can be represented in functional language form as follows:

```
Function Moving_ave_3 (list)
  = construct (a, l)
    where {a = (First(list) + First(Rest(list) + First(Rest(Rest(list))) ) / 3,
           l = Moving_ave_3 (Rest(list))}
                        /* This code shows only cases where i ≥ 3 and
                           the initial cases are handled appropriately. */
```

For the above two function bodies, computation part for "a" is analogous to "solve-1" node in Figure-3(a).

In general, recursive scheme to handle the history sensitive problems requires high level memory operations (i.e. split and construct) which implies high level pointer operations since variable sized list is implemented with linked list. In the working-set based model, each activation of recursive function accesses a number of list elements equal to the size of the working-set window and these accesses are done in serial pointer traversing. Another factor is that these memory accesses are all redundant. For example, with the

working-set window size n, each list element is accessed n times by adjacent n instances of recursion bodies. If the window size is very big, great performance degradation is anticipated because of excessive number of redundant memory accesses and serial pointer operations in each instance of the recursion. In addition to the memory allocation and deallocation (garbage collection) overheads, the activation frame management problem has been an issue in dataflow execution models [4, 5, 42, 44].

3.2.2 Iterative Scheme

With the static stream representation, another way of implementing history sensitive problems is iterative method. In the iterative scheme, I-structure like stored stream data structures [11, 50] are used and rebinding of iteration variable makes the stream proceed to next element. Since this scheme is based on fixed sized data structure, strict implementation (i.e. stream size should be known at compile time) is expected. In this scheme, the cumulative sum problem (Example-1) can be represented as follows:

for all i with A,

$y_i = y_{i-1} + A_{[i]}$;

here, A is stored stream data structure used as input.

Example-2 can be represented similarly;

for all i with A,

$y_i = (A_{[i]} + A_{[i-1]} + A_{[i-2]}) / 3$;

($i \geq 3$ and initial cases are handled appropriately)

Similar to the recursive scheme, this scheme also suffers from redundant memory access overhead. In the working-set based model, each loop body accesses a number of input

stream elements determined by the window size (3 in our Example-2) and all of them are duplicate accesses by adjacent loop instances. Figure-4 illustrates this.

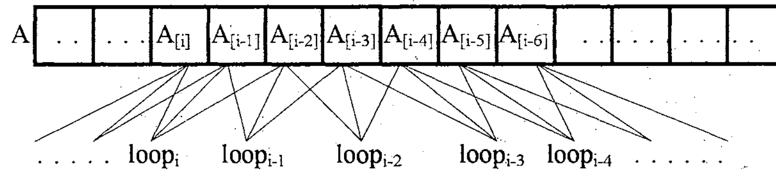


Figure-4. Redundant memory accesses in working-set (window size =3)

When loop unfolding or successive activation of recursion are possible, these redundant memory references bring about memory contention problem which greatly degrades the performance. The iterative scheme does not provide elegant solution when complexity increases with the order of input tokens' dependencies [2, 10, 14, 35]. Further more, these static schemes suffer from the deferred access (waiting for unwritten data) overhead such as maintaining waiting queues [11, 50] if the required input stream elements are not available in the stored stream structure yet.

As mentioned earlier, the deferred access mechanism to handle this situation is complex and costly. The I-structure's deferred access mechanism is shown in Figure-5. In the data storage area, each location has some extra presence bits that specify its state: "present", "absent", or "waiting". When a "read token" arrives, it contains the address of the location to be read and the tag for the instruction that is waiting for the value. If the state is absent (A) or waiting (W), the read is deferred, i.e., the tag is queued at that location. The queue is a linked list of tags in the deferred read request area. In the figure, the location "n+2" and "n+3" are waiting for the data from the producer module

while instructions X, Y, and Z attempt to read. Thus there exist high memory latencies and synchronization costs.

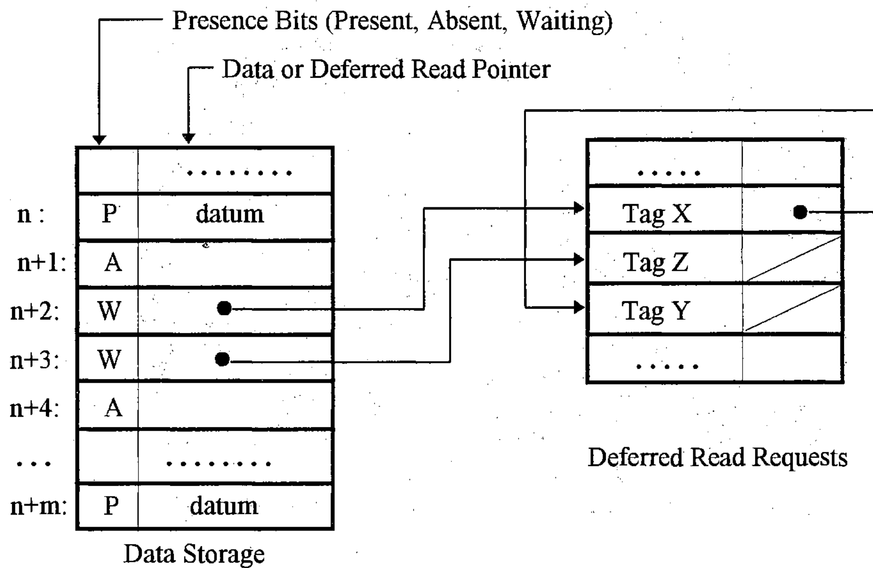


Figure-5. I-Structure memory

In addition to these problems, these static schemes suffer from memory contention. Contention happens when reference counters are needed for garbage collection.

3.2.3 Automata Based Scheme

The dynamic schemes use direct-access data elements that flow dynamically as individual tokens. In the automata based scheme, the history sensitive problems are represented as finite state machines in which states keep the required history of the computation and tokens flow as dynamic stream. Figure-6 illustrates the actions of this scheme. Each state consists of list of values (minimally one) and thus it must be stored in memory as structured data (i.e. array or linked list). New state generation function **g** receives previous state and updates it with current input token. The output generation

function **f** receives required state from **g** and accesses each elements from that state and performs desired computation with input token. Thus the state acts as the history (previous inputs) and the function **g** updates it and the function **f** uses it for the computation.

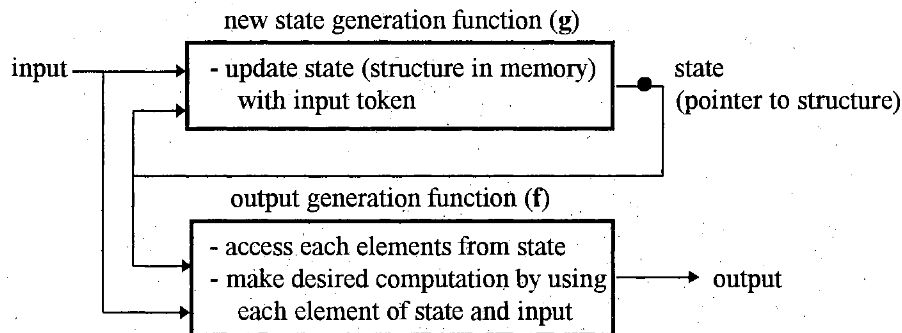


Figure-6. Generic actions of automata based scheme

For our Example-1 (cumulative sum problem), functions **g** and **f** start actions on the initial state <0>. The function **g** outputs the current state and performs binary operation "+" to update the state and the function **f** includes computation "current input + value accessed from state". For the Example-2 (moving average problem with working-set window size 3), the initial state should be <0,0>. The function **g** should perform structure updating action analogous to the action of shift registers of size two. The output generation function **f** includes computation "((add two elements accessed from state) + input) / 3".

Even though this scheme uses dynamic token flow as input and output, it uses memory spaces for storing and updating the state variables to keep the required history. This brings about a more serious problem known as the structured data updating problem in dataflow machines [11, 38, 39, 59]. Functions **f** and **g** must provide state operations

such as accessing and updating memory spaces. The best case is the accumulator based model which uses single element state but the working-set based model needs array size same as the working-set window size. If the working-set window size is very big, this is significant overhead.

Memory access and updating operations degrade the performance of this scheme and thus it does not promise an elegant solution to the history sensitive computation problems.

Chapter IV

PROPOSED APPROACH: THE MEMORYLESS SCHEME

This chapter presents a new scheme for handling history sensitive problems in dataflow execution models. Our proposed scheme named the memoryless scheme does not use stored stream data structure in memory and thus it is a dynamic scheme which uses dynamic stream (token flow) as input and output.

In this chapter, the memoryless scheme will be presented at the dataflow graph level. First it will be described in static dataflow environment and next in dynamic (tagged token) dataflow environment. The two examples from section 3.1 are used to illustrate our design. With appropriately designed language constructs for history sensitive computation parts, compiler can recognize them and generate dataflow graph as we represented in this thesis for the target code.

4.1 Accumulator Based Model

For the accumulator based model of history sensitive problem, we provide an elegant design which receives an input stream token and generates corresponding output stream token dynamically. Our design is motivated by the design of flip-flops. Flip-flops remember the previous information for one clock cycle. There is a feed-back line from the output to the input. In data flow environment, instead of using clock cycle concept, data dependency is used to keep information (history) until next input token arrives. We

combine these two ideas to develop the memoryless scheme. Simplicity is a major characteristic of this scheme which is illustrated in Figure-7.

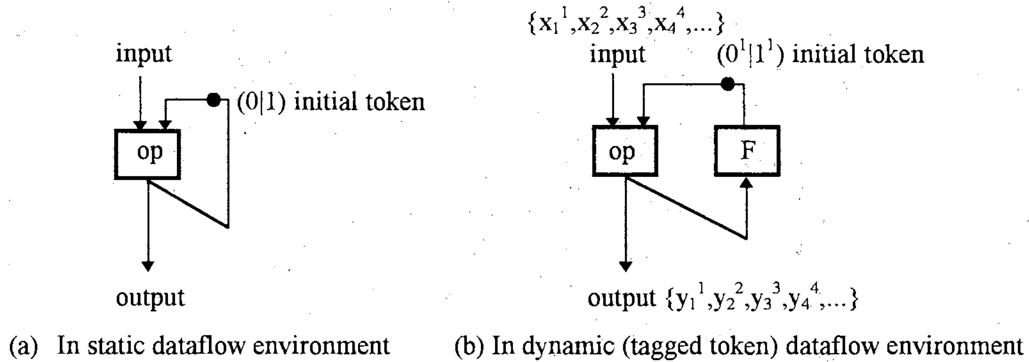


Figure-7. Memoryless scheme for the accumulator based model.

In Figure-7, "op" represents an operation and one single data token feeds back to the operation node. The initial token (0 or 1) depends on the operation; i.e. 0 for "+", and 1 for "*". Unlike the automata based scheme, it does not have state variables stored in memory. This design can be modified appropriately to accommodate more complex problems which consist of both history sensitive and non sensitive computation parts. Figure-7(a) shows the static firing rule which permits only one token on each arc at a time. Thus the concept is simple. Input stream elements arrive one at a time in the generated order and "op" performs the operation with token from feed back arc. Also we can think of this model as a dynamic dataflow model in which multiple tokens can reside on each arc and queued in a FIFO queue in the arrival (stream generation) order. This FIFO queue model is used in RMIT machine [1] which is a hybrid of static and dynamic dataflow architectures. More details on how to keep multiple tokens on arc's can be found in [35]. Figure-7(b) shows the memoryless scheme in dynamic (Tagged Token) dataflow

environment. It uses token relabeling scheme [38, 39]. Token relabeling function F is analogous to the tagging actor D in the MIT's tagged token machine [9, 10]. It increases the tag value of input token by 1 and passes it. In Figure-7(b), input stream element carries its tag value (the superscript) and can arrive in any order. Initially 0^1 and x_1^1 match, and output y_1^1 is generated. The tag value of y_1^1 is increased to y_1^2 by F and sent to "op" node to match x_2^2 from input arc. This process is repeated.

4.2 Working-Set Based Model

For the working-set based model of history sensitive problems, we introduce a synchronization actor which is a unary actor like the D actor. Upon arrival of a token, this synchronization actor simply passes it. Since the data flow model of computation is asynchronous, we have to provide synchronization, which is required in history sensitive problem solving, without storing history in memory. We name this synchronization actor as S and Figure-8 shows how this actor is used in our design.

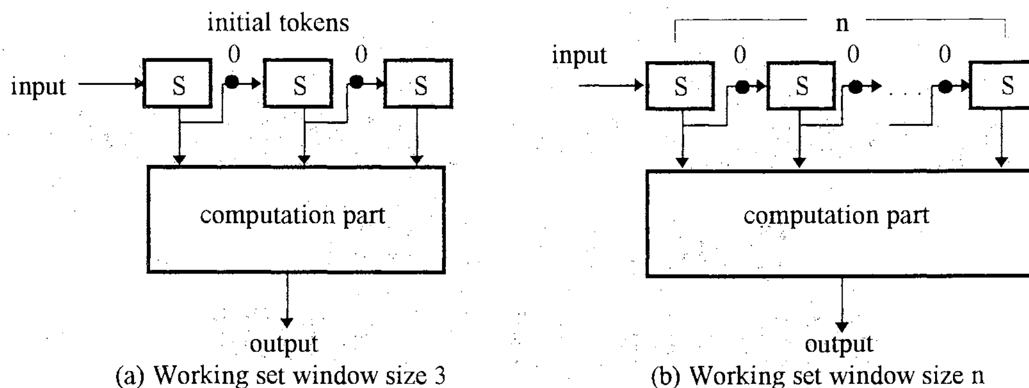


Figure-8. Memoryless scheme for the working-set based model in static dataflow environment

Figure-9 shows the snap shots of history sensitive part as computation progresses with working-set window size 3 based on static firing rule in static dataflow environment. The

computation part component shown in Figure-8 does not contain history sensitivity and so we omit that part in the snap shots shown in Figure-9.

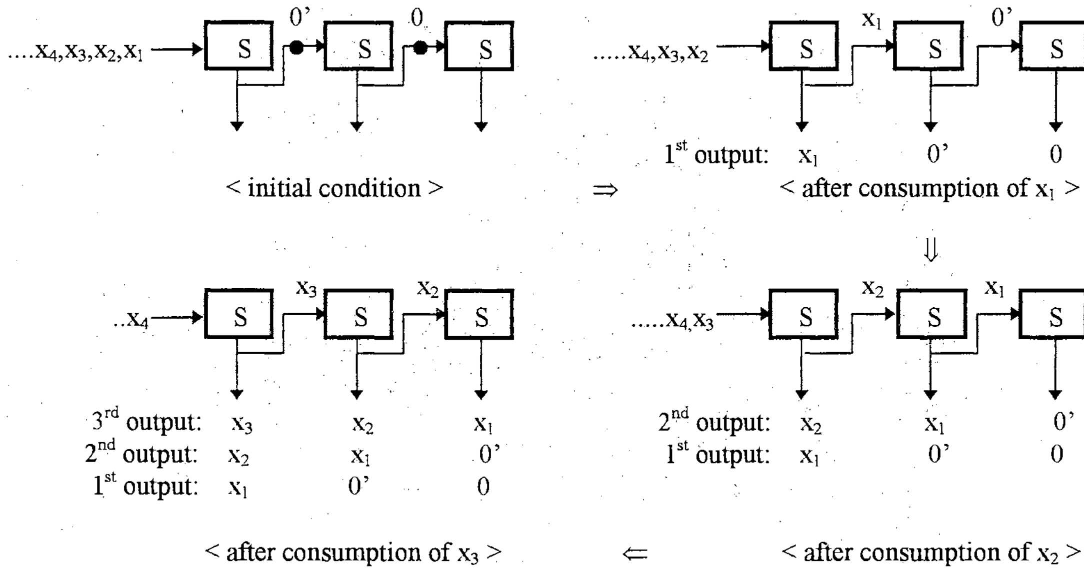


Figure-9. Snap shots of working-set based model with window size 3.

With the static firing rule, input stream token enters into our design one at a time in the stream generation order. An arc between two S actors serves to memorize one previous input token. Similar to the accumulator based model part, this design can also be used in dynamic dataflow environment in which multiple tokens on each arc are queued in a FIFO queue in the generation order. Maintaining such ordered queue is difficult and thus expensive [1, 35].

For the dynamic dataflow machine environment, we slightly modify our model in Figure-8. Instead of using synchronization actor S, the D actor or the token relabeling function F [38, 39] is used. Figure-10 shows our scheme in tagged token architecture. In this scheme we use the D actor. So, replacing the S actor by D actor we get the scheme shown in Figure-10. Note that we removed the 1st D actor for making

optimized design since multiple tokens can reside on each arc in dynamic dataflow environment.

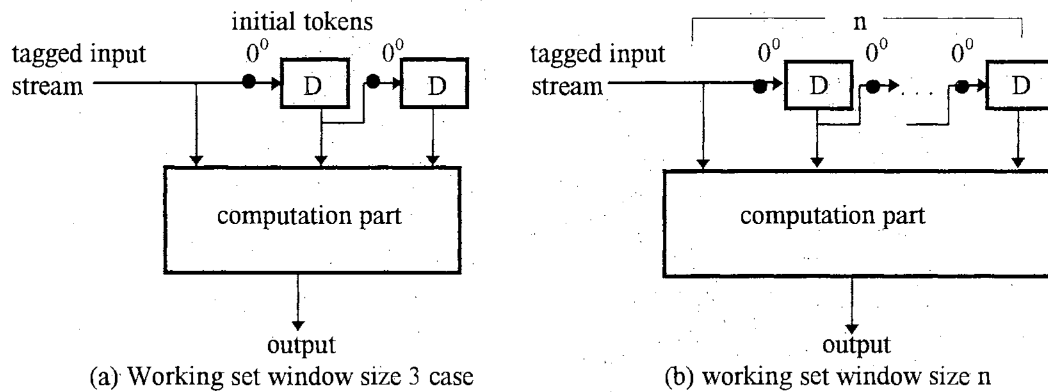


Figure-10. Memoryless scheme for the working-set based model in dynamic (tagged token) dataflow environment

In Figure-11, we will show how this scheme works for the tagged token architecture with the Example-2 (defined in III-1) which is the moving average problem with working-set window size 3. In the figure, a subscript represents index (generation order) of each stream element and the superscript represents the tag of the token. The initial tokens (0's) in front of 1st D actor and between 1st and 2nd D actors have tag values 0's initially. Upon receiving a token the D actor increments the tag value of that token by one and passes it. Input stream elements can reside on input arc in multiple and can be fired in any order. In Figure-11 for example, with the input stream $\{x_1^1, x_2^2, x_3^3, x_4^4, \ldots.\}$ we assume the firing order $\{x_2^2, x_4^4, x_1^1, x_3^3, \ldots..\}$. In the figure, we can observe that the first column from the history sensitive part contains same tokens as in input stream with same tag values. The 2nd column and 3rd column also contain same tokens but tag values are increased by 1 for the 2nd column and 2 for the 3rd column. Thus we made the synchronization for the window size 3 working-set based history sensitive problem.

Each binary operation in the computation part is fired upon matching the two input tokens' tag values. Matching mechanism in tagged token architecture can handle this.

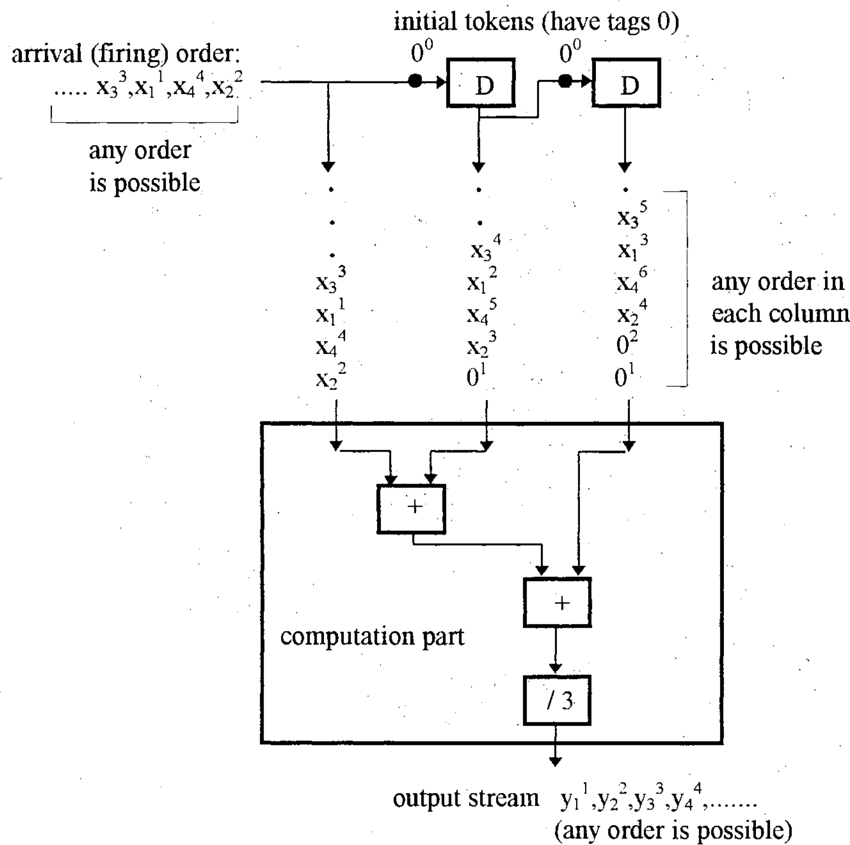


Figure-11. The MAV problem with the Memoryless scheme in dynamic (tagged token) dataflow environment

For generating output y_1^1, for example, the first addition in computation part acts on matching x_1^1 from its left input arc and 0^1 from its right input arc. Then it passes $(x_1^1 + 0^1)^1$ to the second addition's left input arc. Second addition acts on matching this left input and 0^1 from its right input arc because they have same tag values 1. Output from the second addition is $(x_1^1 + 0^1 + 0^1)^1$ and passed to the division function "/3" to generate $((x_1^1 + 0^1 + 0^1) / 3)^1$ as the final output y_1^1. In general, with working-set

window size n, tokens which are used in the computation part for making output y_i^i are x_i^i, x_{i-1}^i, x_{i-2}^i, ... $x_{i-(n-1)}^i$ which have all same tag values i.

4.3 Language Constructs

In this part, we relate the ideas described in the previous parts to a high level language. The language chosen is Backus' FP [12] and the static dataflow environment is assumed for simplicity. Thus high level languages can be used to design solutions to problems.

We design WS(n) for the history sensitive part of the working-set based model with the window size n which is shown in Figure-8 (excluding computation part). Thus WS(3) represents dataflow graph shown in Figure-8(a) which has one input port and 3 output ports (to the computation part).

Let S_1 be the function which produces the 1st (left most) output,

S_2 be the function which produces the 2nd output,

....,

S_n be the function which produces the last (right most) output.

Def **WS(n)** = $[S_1, S_2, ..., S_n]$

where S_1 = id,

$S_2 = \otimes \circ [S_1, \bar{0}]$,

... ,

$S_n = \otimes \circ [S_{n-1}, \bar{0}]$.

In the definition above, $\otimes$ is defined as follows:

Def $\otimes$ = eq_init $\rightarrow$ select 2nd ; select 1st

where eq_init : x = T if x is the initial token

= F otherwise.

Another construct is ACC(op) which represents the history sensitive part of the accumulator based model shown in Figure-7(a). The "op" represents a binary operation and the initial token depends on "op" (i.e. 0 for "+" operation and 1 for "*" operation).

Def **ACC(op)** = op ° [id, c]

where c = $\otimes$ ° [ACC(op), $\bar{0}$ | $\bar{1}$].

With the above definitions, Example-2 which is window size 3 moving average problem (MAV) can be represented as follows:

MAV_3 = div$\bar{3}$ ° + ° WS($\bar{3}$)

where div$\bar{3}$ = / ° [id, $\bar{3}$].

Chapter V

COMPARISON

Since static schemes and the automata based scheme use memory, they all have memory management overheads. In fact, memory size for storing large amounts of data really matters. In the automata based scheme, updating the state (structure in memory) requires excessive amount of memory because entire new structure is created at each update. In addition, static schemes suffer from deferred access overheads. In this section, we compare the Memoryless scheme with these schemes based on processing time regardless of the above disadvantages of other schemes using an example. For the sake of ease and simplicity of comparison, we introduce the following example (Example-3) which is a special case of the working-set based history sensitive problems with the window size 2. The input arrival rate is controlled by the problem itself.

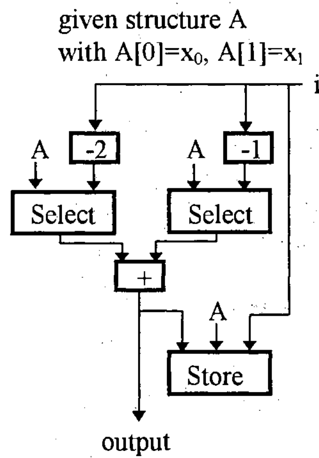
[Example-3] Fibonacci number generation.

given $x_0=1$ and $x_1=1$,

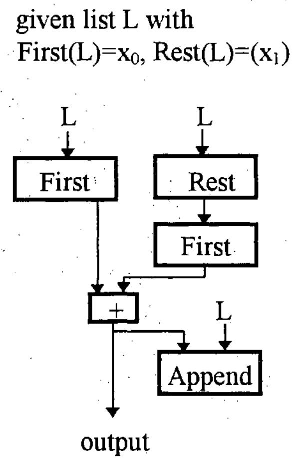
$x_i = x_{i-1} + x_{i-2}$ (for all $i \geq 2$).

Instead of input arriving from the stream generation module, each current output becomes the next input. For example, with current input x_1 computation of current output needs x_0 and x_1 and $(x_0 + x_1)$ is produced as current output. This output becomes the next input x_2 and so on. Thus we expect the outputs {2, 3, 5, 8, 13, 21, 34,....}. We ignore

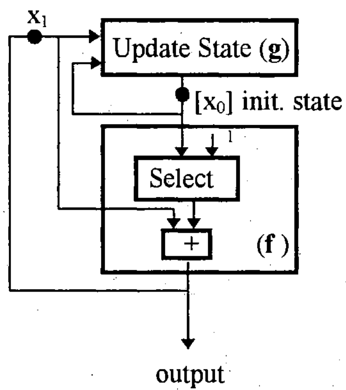
the stopping condition in this example and assume that the mechanism runs continuously. Figure-12 shows the major parts of dataflow graph for this problem in the iterative (a), the recursive (b), the automata based (c), and the memoryless (d) schemes.



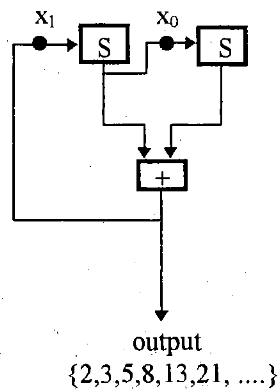
(a) Body of each iteration



(b) Instance of each recursion



(c) The automata based scheme



(d) The memoryless scheme in static environment

Figure-12. Dataflow graphs for Fibonacci number generation

5.1 Iterative Scheme

In Figure-12(a), the 1st Select and the 2nd Select can be done in parallel since deferred access is possible. With loop unfolding, these Select operations in all iteration bodies can conceptually be processed in parallel. But in the Example-3, the 2nd (right) Select of iteration i+1 can not be done until the Store of iteration i completes. Thus in each iteration, we need time for loop control (T_{LOOP}), one Select (T_{READ}), one addition (T_{ADD}), and one Store (T_{WRITE}). Over all, for producing N fibonacci numbers starting from 2, time required is:

$$T_{LOOP} + N * (T_{READ} + T_{ADD} + T_{WRITE})$$

5.2 Recursive Scheme

In Figure-12(b), we observe that the critical path is the right most path which starts from "Rest", followed by "First", followed by "Add", and followed by "Append". Same as the iterative scheme, successive activations of recursive bodies can not exploit full parallelism since there exist dependencies among the instances. Thus time needed to produce N fibonacci numbers is:

$$T_{PROBLEM\text{-}SPLIT} + N * (T_{REST} + T_{FIRST} + T_{ADD} + T_{APPEND})$$
$$= T_{PROBLEM\text{-}SPLIT} + N * (T_{POINT} + T_{REF} + T_{READ} + T_{ADD} + T_{POINT} + T_{ALLOC} + T_{WRITE})$$

In above equation, "T_{POINT}" means time for pointer traversal and "T_{REF}" means time for increasing reference counter because the garbage collection mechanism needs reference counter for multiple access of same node.

5.3 Automata Based Scheme

In Figure-12(c), the new state generation (update state) function **g** needs time for updating the state and the output generation function **f** needs time for selecting the first element in the state and addition. Since **g** and **f** can be done concurrently, time needed for producing N Fibonacci numbers is:

$$N * (\mathrm{Max} (T_g, T_f))$$

where $T_g = T_{UPDATE} = T_{ALLOC} + T_{COPY} + T_{WRITE}$, and $T_f = T_{READ} + T_{ADD}$.

For Example-3, fortunately this scheme uses only one element state but, with large window size (i.e. output depends on current input and number of previous inputs) the state is multi-element structure and the "T_{UPDATE}" will be a serious problem as we mentioned in section 3.2.3.

5.4 The Memoryless Scheme

From the Figure-12(d), we can observe that the 1st S actor and addition constitute the critical path to produce each output, we can easily configure the needed time for producing N fibonacci numbers as follows:

$$N * (T_S + T_{ADD})$$

where T_S means time needed for S actor.

In this expression, T_S is almost negligible because the only action it performs is pass the token along. Therefore, the computation time for the memoryless scheme is approximately $N * T_{ADD}$. This is significantly better than those for the other schemes.

Chapter VI

EXPLOITING MAXIMUM PARALLELISM

In the case of stream generation in an iterative construct, it is possible that stream elements can be produced in parallel or at very high speed by using loop unfolding mechanism. This is possible when each iteration does not incur any data dependencies from previous iterations. For example, each stream element is computed independently or accessed from other structured data or data bases. The functional language VAL has a ***forall*** construct which provides this parallelism [30, 65]. Even when there exist some dependencies between iterations, pipeline mechanism makes it possible to overlap some portions of operations. For this reason, in approaches of handling history sensitive computations in dataflow execution models, mechanisms for exploiting maximum parallelism should be provided to gain high performance.

In the recursive scheme, pipelined parallelism is possible with the linear recursive structure shown in the Figure-3(a). The eager evaluation mechanism with parallel evaluation of arguments (dataflow computation model makes this possible) enables the parallel execution of each function instances. In Figure-3(a), multiple instances of "solve 1" can be activated successively and executed in parallel. In the iterative scheme, loop-unfolding mechanism enables each iteration done in parallel.

Unfortunately, with these static schemes parallel processing is impossible in accumulator based model of history sensitive problems because there exist data

dependencies among instances of iteration (iterative approach) or recursion (recursive approach). Figure-3(b) shows these data dependencies in the recursive scheme. Similarly, iterative scheme has loop dependencies (i.e. sequential loop problem). In working-set based model, these static schemes suffer from memory access latencies. Since each stream element in memory is redundantly accessed working-set window size times by adjacent instances (See Figure-4), parallel processing of these instances causes memory contention problem and this latency degrades the parallelism gained. If the window size is very large, great performance degradation is anticipated.

On the other hand, the automata based scheme which is a dynamic scheme can not provide any parallelism in both accumulator based and working-set based models. It processes one stream element at a time and the critical path includes the state (structure in memory) update part.

With the memoryless scheme, our effort has been to exploit maximum pipelined parallelism in accumulator based model and explicit parallelism comparable to side effect free loop unfolding in working-set based model. In this chapter, mechanisms for exploiting maximum parallelism with the memoryless scheme are presented in both static and dynamic dataflow environments. Since we divide the history sensitive problems into two major groups, namely accumulator based and working-set based models, the two models are handled separately.

6.1 Explicit Parallelism in Accumulator Based Model

With the memoryless scheme, we can exploit high performance pipelined parallelism in accumulator based model by using a forwarding mechanism (PACC). From now on,

"ACC block" represents the memoryless scheme for the accumulator based model of history sensitive computation problems. The name PACC stands for parallel ACC block. Theoretically, any depth of pipeline is possible with this mechanism. The forwarding mechanism is represented in both static and dynamic dataflow environments.

6.1.1 Forwarding Mechanism (PACC)

First we consider the static dataflow environment for the sake of simplicity of presenting the concept. Instead of using single stream input and output, multiple stream input and output are used to exploit explicit parallelism. The forwarding mechanism receives multiple input streams and generates multiple output streams in parallel. We call the forwarding mechanism with parallelism degree d at the ACC block level PACC(d).

For creating multiple input streams from a single stream, we arrange each input stream as illustrated in Figure-13.

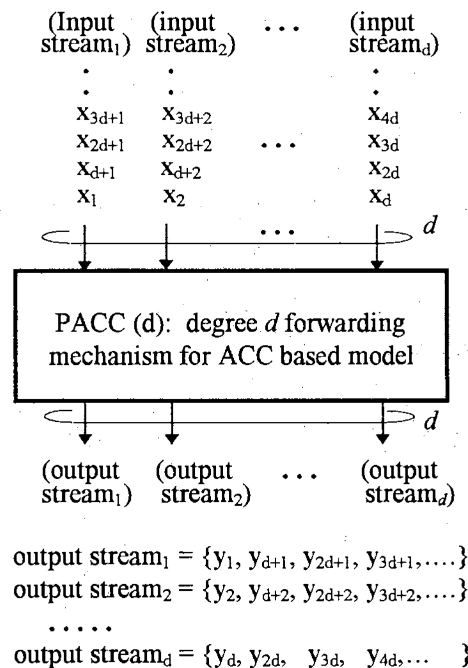
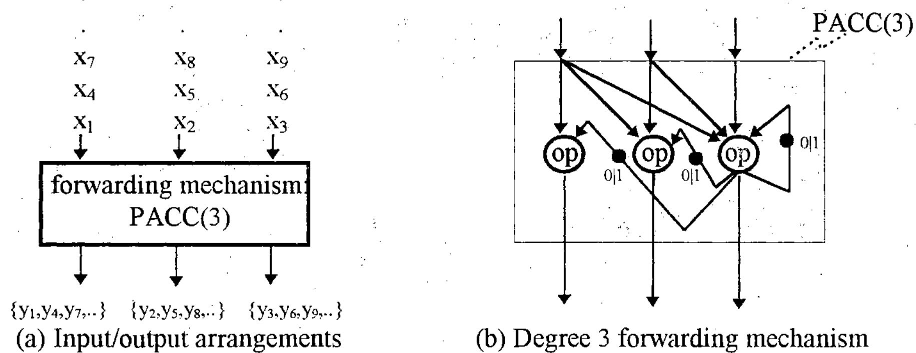


Figure-13. Input/output arrangements of forwarding mechanism (PACC(d))

For parallelism degree d, we need d distinct input streams and d distinct output streams as illustrated in Figure-13. In dataflow environment this arrangement of multiple input streams can be done in the stream generation modules by using counters and MOD functions. Output from PACC(d) block consists of d distinct streams. The ith (from left, and $1 \leq i \leq d$) output stream has output data corresponding to the ith input stream; i.e. they have same order. We will describe this relationship and contents of output streams in detail later. Figure-14 illustrates high level dataflow organization of degree 3 forwarding mechanism (PACC(3)).



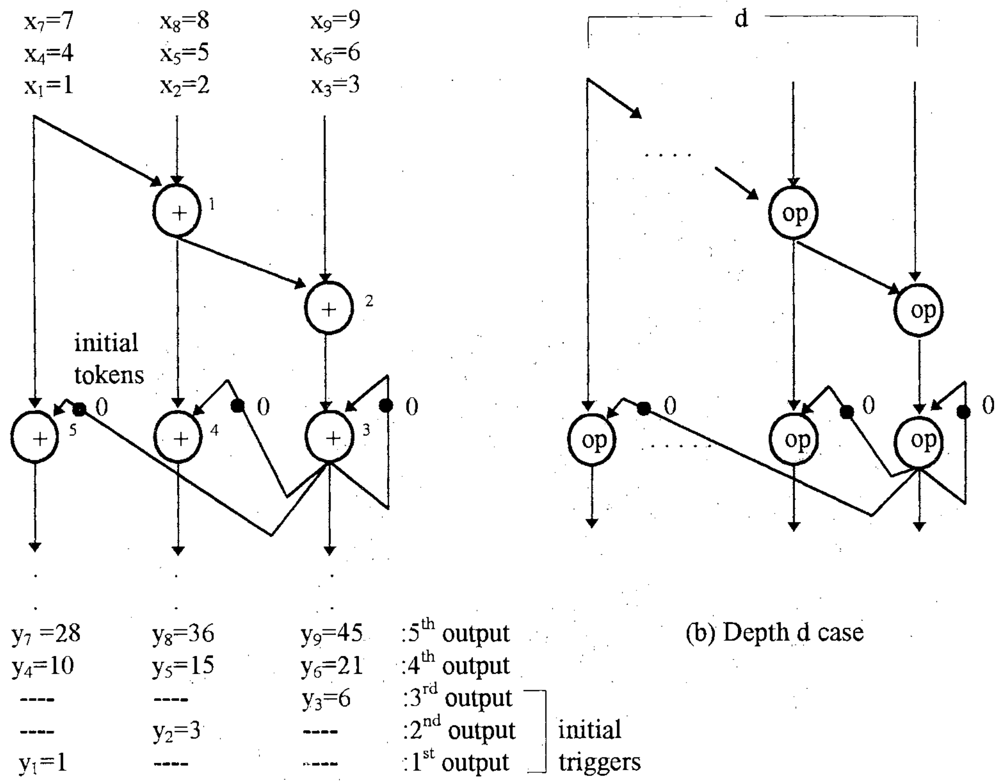
(a) Input/output arrangements (b) Degree 3 forwarding mechanism

Figure-14. Concept of forwarding mechanism with degree d=3 case (PACC(3))

In Figure-14(b), operation nodes "op" are either "+" or "*". In the figure, the left most operation receives two operands, the 2nd (from left to right) operation receives three operands, and the 3rd (the right most) operation receives four operands. The idea is that during op_1 (the 1st (left most) operation) computes the y_1 (the 1st output) by using input token x_1, op_2 computes y_2 by using input tokens x_1 and x_2, and op_3 computes y_3 by using input tokens x_1, x_2, and x_3. These operations are done simultaneously. At next time slice, op_1 can compute y_4, op_2 can compute y_5, and op_3 can compute y6 simultaneously

because they receive y_3 which was computed one time slice ago on op_3. For instance, at second time slice, op_2 receives y_3 from op_3, which is the result of accumulating computations on x_1, x_2, and x_3, and input tokens x_4 and x_5 from the 1st and 2nd input streams. Thus it can generate the y_5 which is the result of accumulating computations on x_1, x_2, x_3, x_4, and x_5. In general, the 1st operation receives 1+1 operands, the 2nd operation receives 2+1 operands, the 3rd operation receives 3+1 operands, and so on. Thus in PACC(d), the dth operation receives d+1 operands with this concept.

Since each operation in the concept (refer Figure-14(b)) uses different numbers of operands it makes the evaluation of performance difficult. So, to examine performance the design of the forwarding mechanism with only binary operations is presented below. Optimized design with identical binary operators is shown in Figure-15. Figure-15(a) shows the operation of degree d=3 forwarding mechanism applied to cumulative sum problem. In fact, the operations are done in a pipelined manner; i.e. it performs like depth d pipeline. In general, degree d (arbitrary d) forwarding mechanism (PACC(d)) can be derived easily and is depicted in Figure-15(b). In Figure-15(a), actual data values are used to make the concept clear. The numbers associated with each operators are identification numbers and will be used in the performance measurement part later. With static firing rule, this forwarding mechanism works like the pipeline in a von Neumann computer. After the d initial trigger time slices, d outputs are available every unit of time (time for one "op") and it makes the total execution time 1/d of the sequential execution in the ideal case with no other overhead. With a depth d pipeline (PACC(d)), d + (d - 1) binary operators are needed in data flow graph, but their executions are done in pipelined manner. It reduces the total execution time by a factor of d.



(a) Depth 3 case with cumulative sum example

Figure-15. Optimized forwarding mechanism (PACC) with binary operators

After the first d initial triggers, we can get d outputs at a time. For the static dataflow environment, we should provide a merging mechanism for the outputs to combine the different streams into one. In applications with multiple input and output streams, spliting and merging of streams is not necessary. In that case there is no overhead associated to the pipeline. In the case of single input stream and single output stream, a spliting node and a merging node are required. However, the overhead associated to these nodes are relatively low compared to the operation nodes. Therefore, their impact on speed up is minimal.

For the dynamic dataflow environment we modify our design slightly and Figure-16 shows our forwarding mechanism (i.e. the depth 3 case) in the dynamic (tagged token) environment. In Figure-16(a), we use D actors for token relabeling. The D^2 and D^3 actors are variations of the D actor and they increase tag value by 2 and 3 respectively. In general, for the depth d forwarding mechanism, we need (d-1) D actors, and d D^i actors (one for each i, $1 \leq i \leq d$) or analogous functions which increase the tag value by i. For example with d=3, which is depicted in Figure-16(a), two D actors, a D1 (same as D), a D2, and a D3 actors are used. Design of the general case (depth d) is straightforward and thus the illustration is skipped.

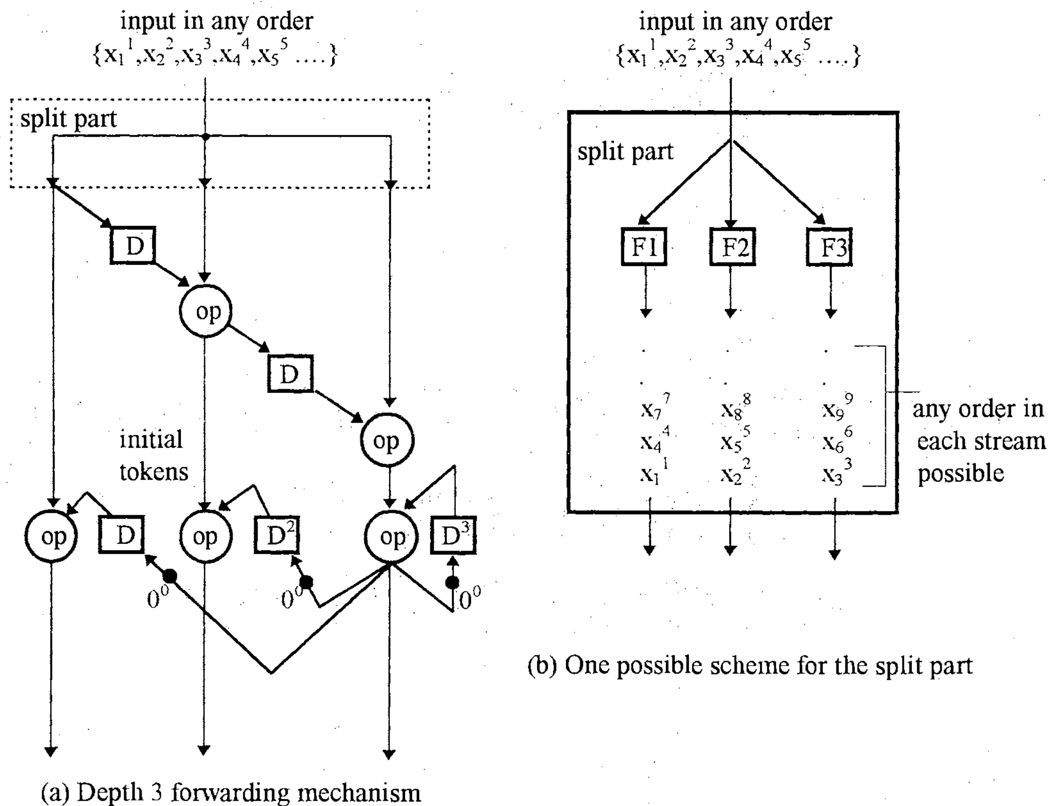


Figure-16. Forwarding mechanism (depth 3 case) in dynamic (tagged token) dataflow environment

The split part copies given input stream (tagged tokens) into d (depth of pipeline) input arcs and by using tag matching mechanism this design works the same as the one in Figure-15(a). Since each output token has unique tag value, the merging part is not necessary.

One possible approach which can save the matching mechanism from useless work is illustrated in Figure-16(b). In the figure, functions F1, F2, and F3 work as follows: Upon receiving a tagged token, F1 checks the condition "tag MOD 3 = 1" and if true, it passes that token otherwise consumes it. F2 and F3 work similarly with checking "tag MOD 3 = 2" and "tag MOD 3 = 0". In general, "tag MOD n" should be used with depth n forwarding mechanism. With this splitting scheme, each input arc has only required input stream tokens in any order. The order of a token is kept in its tag.

6.2 Explicit Parallelism in Working-Set Based Model

With the memoryless scheme, we can exploit explicit parallelism in working-set based model by using a forwarding mechanism (PWS). From now on, "WS block" represents the memoryless scheme for the working-set based model of history sensitive computation problems. The name PWS stands for parallel WS block. Theoretically, any degree of parallelism is possible with this mechanism. The forwarding mechanism is represented in both static and dynamic dataflow environments.

6.2.1 Alternative Representation of WS Block

Before we describe the forwarding mechanism which explicitly speeds up the memoryless scheme for handling working-set based history sensitive problems, an alternative scheme for the WS, which is more profitable for exploiting maximum

parallelism, will be introduced first. In the description of the working-set based model in section 4.2, there exist data dependencies in the history sensitive part (connected S actors or D actors) and it can be an obstacle to seeking explicit parallelism. Thus we need a more elegant design which does not have data dependencies among input arcs to the computation part. Figure-17 shows alternate designs in both static and dynamic (tagged token) dataflow environments.

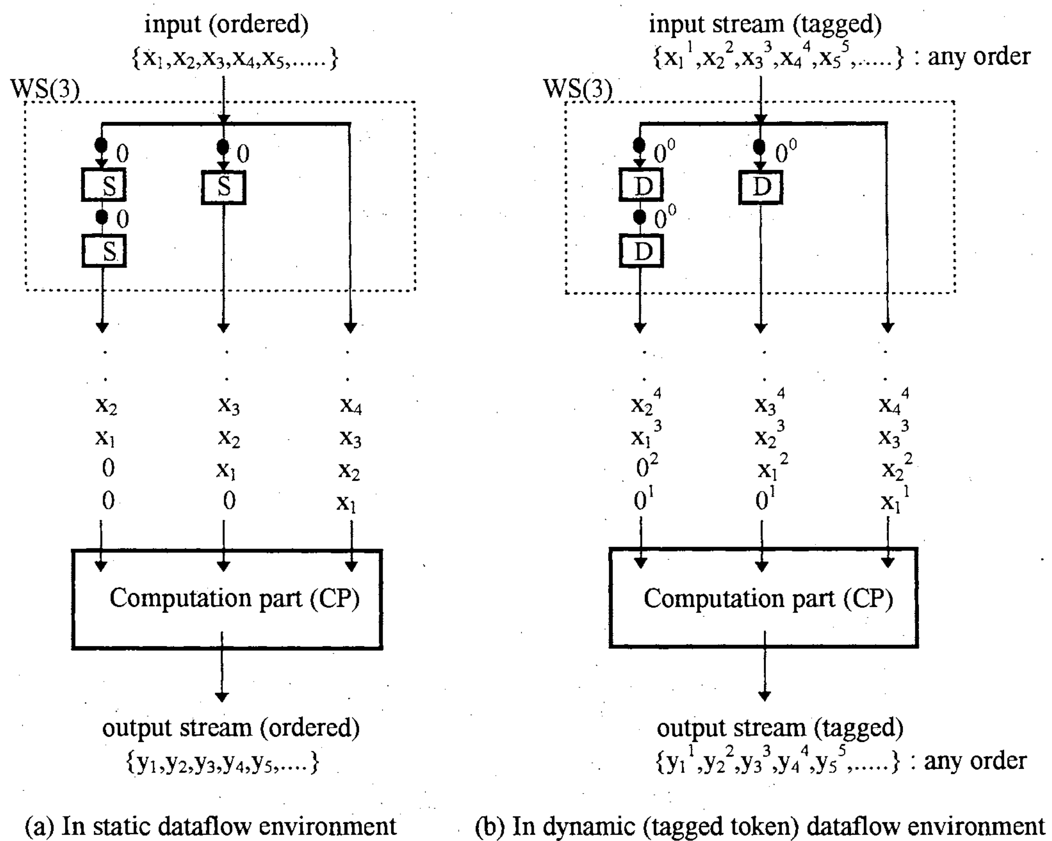


Figure-17. Alternative scheme for the working-set based model (window size 3 case)

In the static dataflow environment, no arc can have more than one token at a time. Thus in Figure-17(a), input arcs to the computation part have synchronization actors "S" which we defined in section 4.2 for controlling initial tokens to meet the static firing rule. In

general, for working-set window size m, m-1 initial tokens (zeros) with m-1 "S" actors are needed on the 1st (left most) input arc to the computation part, m-2 initial tokens with m-2 "S" actors on the 2nd input arc, and so on. With the alternative representation of the WS(m) block, the forwarding mechanism can make the action of the WS(m) block parallel; i.e. it receives multiple input streams instead of single input stream and forwards them to the multiple computation parts so that the multiple computation parts can generate multiple output streams in parallel. In the next section, we will describe how to build the d*dm forwarding mechanism of WS(m) block. We name this PWS(d*dm) block. PWS stands for parallel WS block and, d and m are the parallelism degree and the size of the working-set window respectively. In fact, the alternative representation in Figure-17 is a (1*3) forwarding mechanism (PWS(1*3)) in which the parallelism degree is d=1 (i.e. serial case), and the working-set window size is m=3.

Figure-17(b) shows the alternative scheme in the dynamic (tagged token) dataflow environment. This dynamic design provides more implicit parallelism than static design. In the fine grained analysis, any operation which received same tag valued operands in the computation part can be processed in parallel since input arcs do not have data dependencies among themselves. This implicit parallelism comes from the dynamic nature of the architecture. Figure-17 shows the alternative representation with window size 3. The generalization to the case of window size m is straightforward. Therefore the description is skipped.

6.2.2 Forwarding Mechanism (PWS)

The forwarding mechanism ,which will be described in this section, provides explicit parallelism which speeds up the execution of the memoryless scheme for the working-set based history sensitive computations. The arrangement of multiple input streams is same as one described in forwarding mechanism for accumulator based model (PACC(d)). The forwarding mechanism is described in static dataflow environment first. Also this mechanism can be used in dynamic (tagged token) dataflow environment with little modification by using token relabeling method similar to the forwarding mechanism (dynamic) used in accumulator based model earlier. High level concept and input output stream arrangements of the forwarding mechanism (PWS) are illustrated in Figure-18.

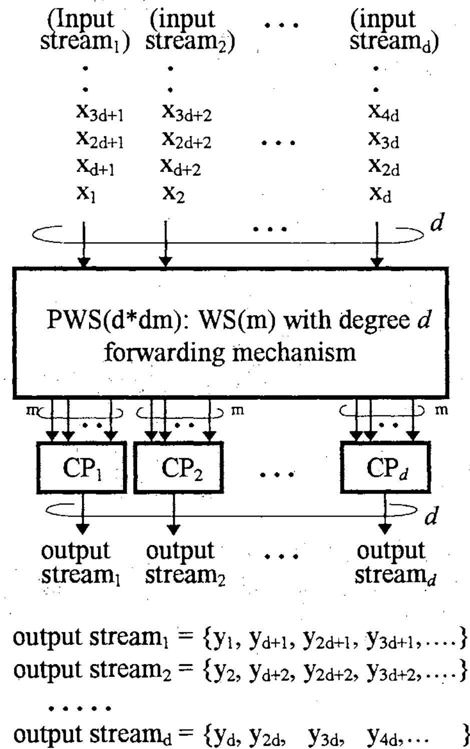


Figure-18. Input/output arrangements of the forwarding mechanism PWS(d*dm)

For parallelism degree d, we need d distinct input streams and d identical computation parts each of which generates one output stream. Parallelism degree d is controllable and is independent from working-set window size m. High level concept of the forwarding mechanism is illustrated in the Figure-19 with example cases.

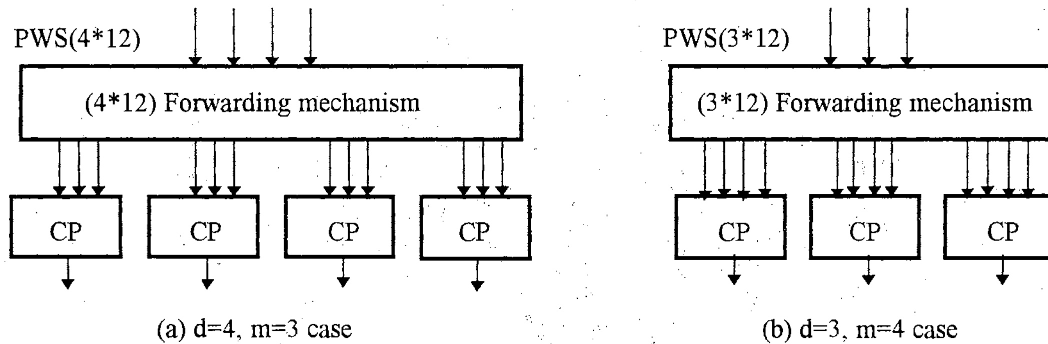


Figure-19. Concept of the forwarding mechanism

Theoretically, we can design this explicit forwarding mechanism in any parallelism degree d with any working-set window size m. For example, for any working-set window size m (arbitrary m) history sensitive computation problem, we can design parallelism degree d (arbitrary d) forwarding mechanism which produces 1/d total execution time. The following algorithm generates the forwarding mechanism for degree d parallelism.

Algorithm-1: Parallel Working-set (PWS(d*dm)) : forwarding mechanism

Assumption: For degree *d* forwarding, inputs are arranged in *d* different input streams;

the 1st (left most) input stream consists of (order MOD d = 1) input elements,

the 2nd input stream consists of (order MOD d = 2) input elements,

the 3rd input stream consists of (order MOD d = 3) input elements,

the dth input stream consists of (order MOD d = 0) input elements.

Let Stream-i represents the ith (from left) input stream.

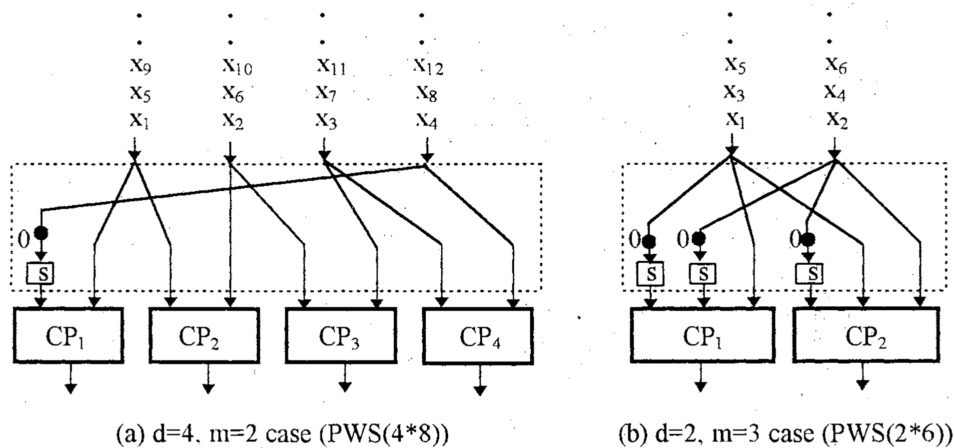
Let CP_i represents the ith (from left) computation part.

(each CP_i has *m* input ports and 1 output port).

Let *m* be the working-set window size and *d* be the degree of the parallelism.

```
INITIALIZE:
  for i := 1 to (m-1)
    {k := i MOD d;
     if k = 0 then k := d;
     Assign (m-1) "S" actors (and initial tokens) on input ports
        of CP_k from left to right;
    }
ASSIGN ARCs from input streams (Stream-i) to Computation Parts (CP):
  for i = 1 to d
    {p := m;
     for j = 0 to (m-1)
       {k := j + (i - 1);
        Assign an Arc from Stream-i to p^th input port (from left) of CP_(k MOD d)+1;
        p := p - 1;
       }
    }
```

In Algorithm-1, each CP (computation part) corresponds to the function *f* we defined with working-set based model in section III(a). Input ports to a CP correspond to parameters of *f*. By using this algorithm, any degree of forwarding mechanism with any working-set window size can be generated. For example, Figure-20(a) shows a degree 4 forwarding mechanism for working-set window size 2 (PWS(4*8)) and, Figure-20(b) shows a degree 2 mechanism for window size 3 (PWS(2*6)). Figure-17 shows the case where d = 1 and n = 3.



(a) d=4, m=2 case (PWS(4*8)) (b) d=2, m=3 case (PWS(2*6))

Figure-20. Examples of forwarding mechanisms generated by Algorithm-1

The forwarding mechanism illustrated in the Figure-21 is PWS(3*9) in which the parallelism degree is d=3 and the window size is m=3 in static dataflow environment. In Figure-21, we observe that three outputs from three computation parts come out in parallel. Output $stream_i$ has corresponding order of input $stream_i$. For example with d=3 case as illustrated in Figure-21, CP_1 (left most CP) generates $\{y_1,y_4,y_7,\ldots\}$ which has corresponding order of the 1st input stream $\{x_1,x_4,x_7,\ldots\}$, CP_2 generates $\{y_2,y_5,y_8,\ldots\}$ which has corresponding order of 2nd input stream $\{x_2,x_5,x_8,\ldots\}$, and CP_3 generates $\{y_2,y_5,y_8,\ldots\}$ which has corresponding order of 3rd input stream $\{x_2,x_5,x_8,\ldots\}$. In static

dataflow environment, when system needs one ordered output stream these outputs should be merged into one. As in the accumulator based model case, this is not a serious matter since the disorder can happen only within the range of parallelism degree d. For example with degree d=3 in Figure-21, outputs y_1, y_2, and y_3 can be generated at the same time and no y_i ($i > 3$) can precede these three. Then y_4, y_5, and y_6 can be generated at the next time period and no y_i ($i > 6$) can precede these three and so on.

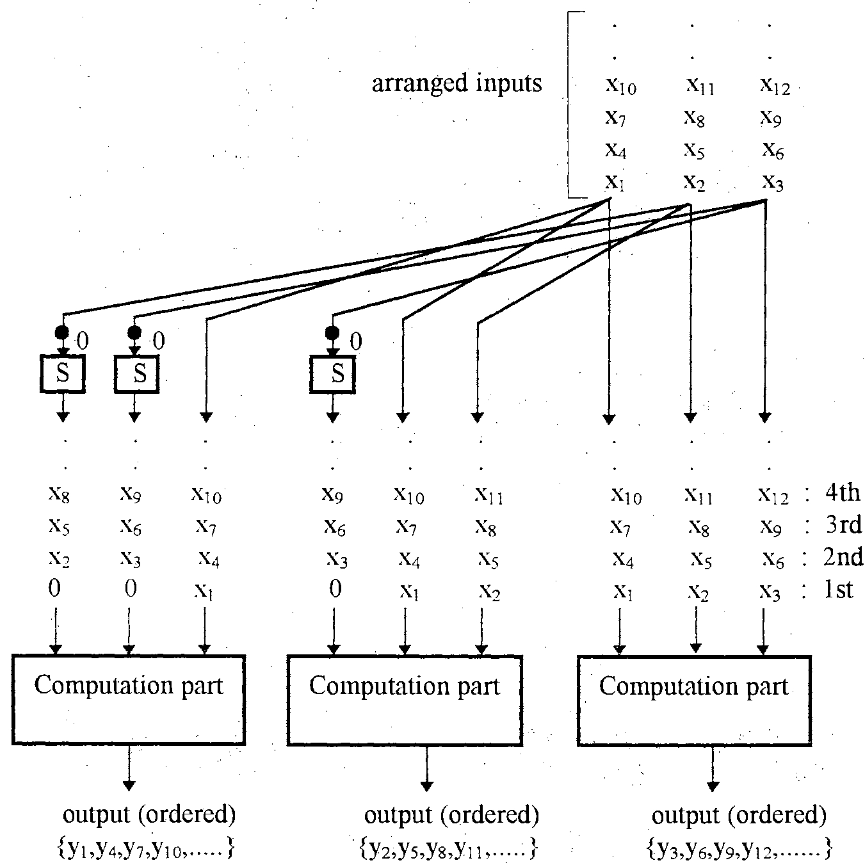


Figure-21. Forwarding mechanism (depth = 3, window size = 3 case) for the working-set based history sensitive problems in static dataflow environment

We can adapt our forwarding mechanism described for the static dataflow environment so far to the dynamic (tagged token) dataflow environment by using token relabeling methods. As shown in Figure-22, the dynamic scheme uses D actors, which are token

relabeling actors, instead of S actors used in static scheme. The forwarding mechanism is the same as one described in Agorithm-1 except assigning D actors and initial tokens. Thus the detailed description is skipped. In dynamic dataflow environment, we do not have to order the output because each output token has its unique tag value.

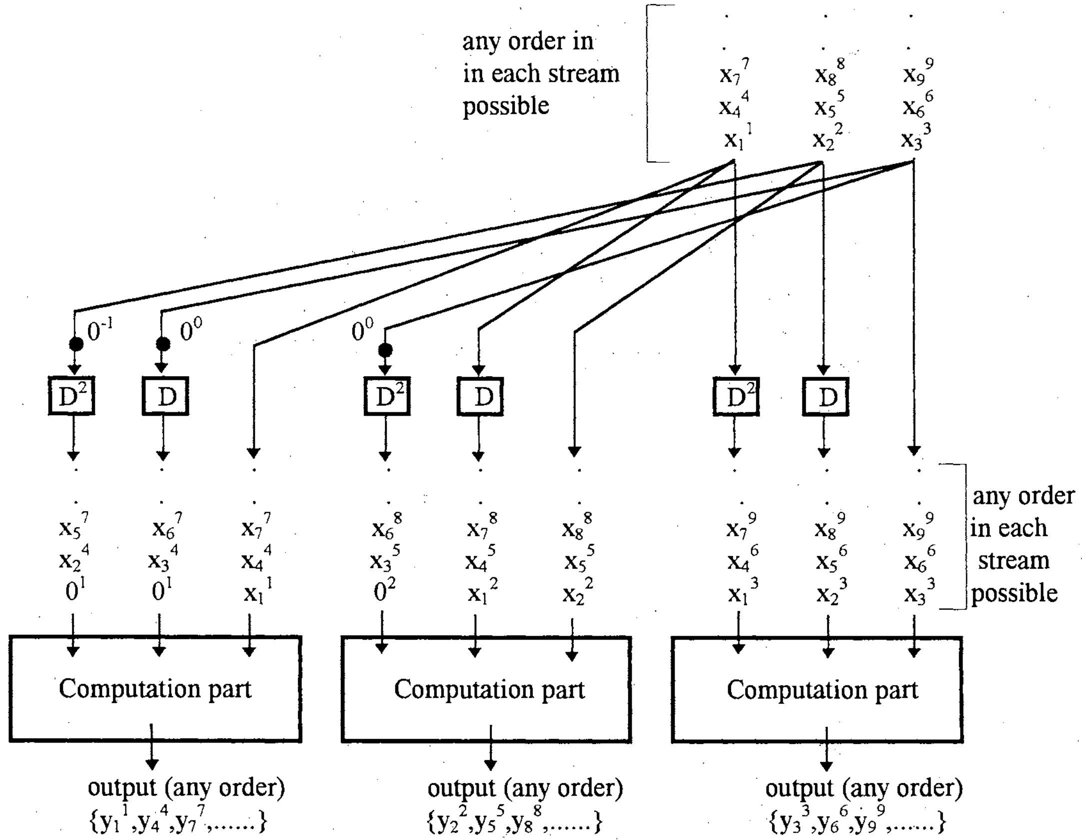


Figure-22. Degree 3 forwarding mechanism (window size m=3 case) in dynamic (tagged token) dataflow environment

6.3 Performance Measurements

In this section, we analyze performances of the forwarding mechanisms proposed in previous sections. We assume that all input stream elements are generated and available for use and there are enough resources. In the accumulator based history sensitive problems, no existing schemes (iterative, recursive, and automata-based approaches)

provide parallelism. The dependency graph shown in Figure-23(a) applies to those methods. The forwarding mechanism with the memoryless scheme presented in section 6.1 of this chapter exploits remarkable pipelined parallelism and the dependency graph with pipelined operations for the depth 3 forwarding mechanism (refer to Figure-15(a)) is illustrated in Figure-23(b). In the figure, the numbers associated with operators are analogous to the operator identification numbers shown in Figure-15(a).

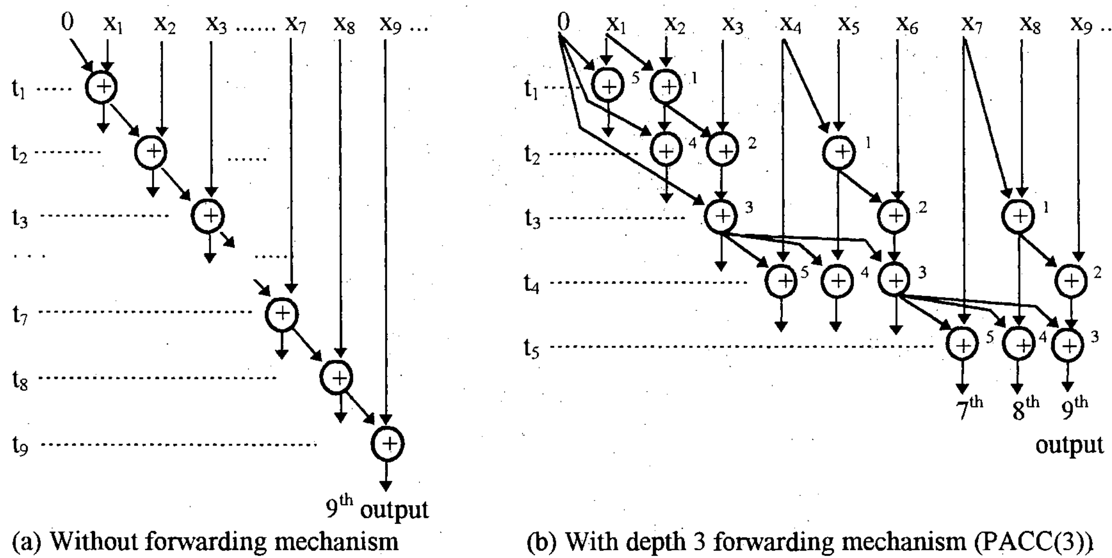
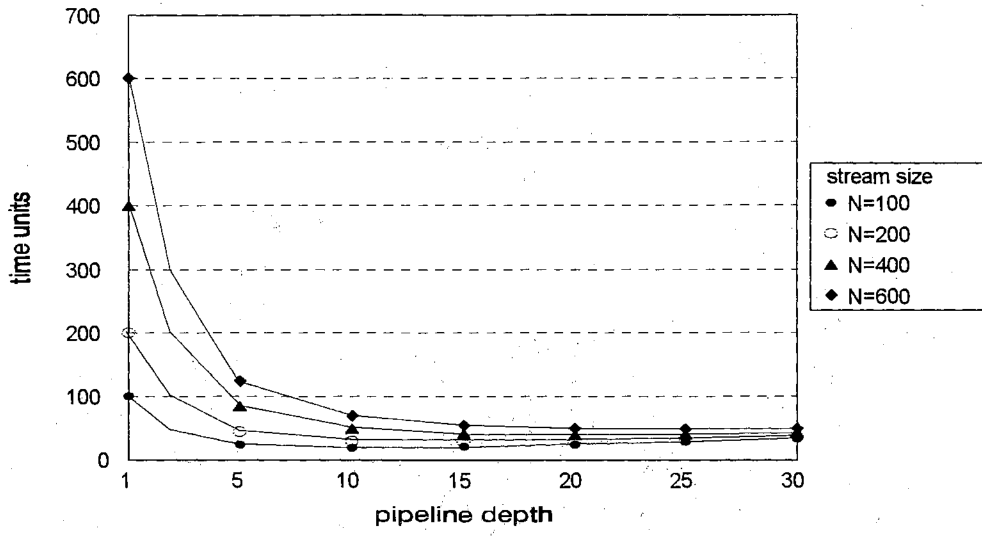
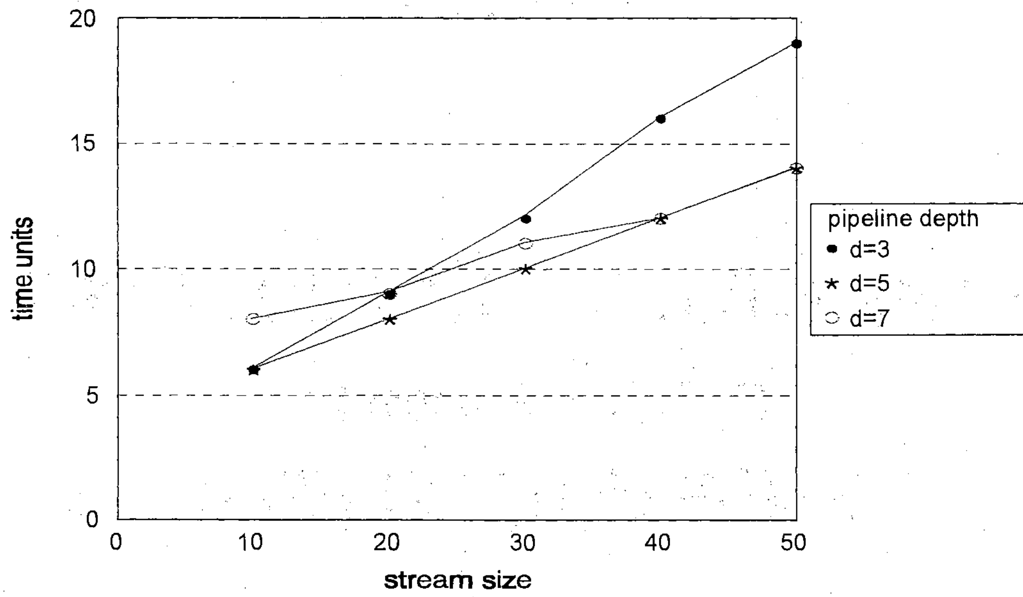


Figure-23. Dependency graphs in accumulator based model

In Figure-23, up to the 9th output, sequential scheme requires 9 units of time (a) but the forwarding mechanism with the memoryless scheme needs only 5 units of time (b). In Figure-23(b), after initial trigger time (t_1, t_2, and t_3), three outputs are produced in each unit of time. After the 3rd output is available at time t_3, the 6th output is available at time t_4 and the 9th output is available at time t_5 and so on. With a large stream size, the initial trigger time (depth of pipeline) can be ignored and the performance gain is 1/depth of total execution time (or speed up = depth of pipeline). More precisely, with stream size N



(a). Depth V.S. performance



(b). Relative weight of N and d V.S. performance

Figure-24. Pipeline performances with stream size (N) and pipeline depth (d)

and pipeline depth d, total execution time needed to perform the entire computation is $\lceil d + \frac{N-d}{d} \rceil$ units of time and it reduces the original execution time by the factor of d (pipeline depth) when stream size N is relatively large compared to d. In fact, the performance depends on the relative weights of N and d. Figure-24(a) shows the performance with various N and d. Each line in the graph shows bitonic curve since it decreases until the optimal d (i.e. gives the best performance) and then increases because d becomes relatively big compare to N. For example with N = 100, the optimal d is 10 which gives 19 time units on y axis. It gives 21 time units with d=15. In general, the bigger the stream size N the bigger the optimal pipeline depth. Figure-24(b) shows performances for different pipeline depths. In Figure-24(b), with stream size 10, 20, and 30 corresponding performances of depth 7 are worse than those of depth 5 because with those stream sizes the depth 5 is closer to the optimal depths. With given stream size N, the optimal depth which gives the best performance can be found around the $\sqrt{N}$. The following lemma gives a formal estimate:

Lemma-1. Let d be pipeline depth and let N be the stream size. Then the minimum execution time is $2\sqrt{N} - 1$ with the optimal pipeline depth *Int.*$(\sqrt{N})$.

Proof: Given formula for the total execution time: $d + \frac{N}{d} - 1$ --------(1)

case 1: $d = \sqrt{N}$

Substitute $d = \sqrt{N}$ into (1) and get the total execution time $2\sqrt{N} - 1$.

Case 2: $d < \sqrt{N}$

Substitute $d = \sqrt{N} - x$ ($1 \le x < \sqrt{N}$) into (1) and get the total execution time

$$\sqrt{N} - x - 1 + \left(\frac{N}{N - x^2}\right)\left(\sqrt{N} + x\right)$$

because ($1 \le x < \sqrt{N}$), the term $\left(\frac{N}{N - x^2}\right)$ is $G.T.$ 1 and we have the total

execution time $\sqrt{N} - x - 1 + \sqrt{N} + x + y = 2\sqrt{N} - 1 + y$ (y is positive number).

Case 3: $d > \sqrt{N}$

Substitute $d = \sqrt{N} + x$ $(1 \le x < \left(N - \sqrt{N}\right))$ into (1) and get the total execution

time $\sqrt{N} + x - 1 + \left(\frac{N}{N - x^2}\right)\left(\sqrt{N} - x\right)$

Sub_case1: $x < \sqrt{N}$

The term $\left(\frac{N}{N - x^2}\right)$ is $G.T.$ 1 and we have the total execution time

$\sqrt{N} + x - 1 + \sqrt{N} - x + y = 2\sqrt{N} - 1 + y$ (y is positive number).

Sub_case2: $x > \sqrt{N}$

The term $\left(\frac{N}{N - x^2}\right)\left(\sqrt{N} - x\right)$ is positive (∵ neg. * neg. = pos.) and the

term $\sqrt{N} + x - 1$ is $G.T.$ $2\sqrt{N} - 1$.

Therefore the total execution time is $2\sqrt{N} - 1 + y$ (y is positive number).

Thus we have the minimal total execution time $2\sqrt{N} - 1$ when we have the

optimal pipeline depth $d = Int.(\sqrt{N})$. □

Chapter VII

APPLICATION TO PARALLEL PREFIX COMPUTATION

The memoryless scheme presented in previous chapters is a pure dataflow scheme which does not use any memory references. Therefore it can be applied to design hardware solutions of some real life problems. In this chapter, a parallel solution to the prefix computation problem is presented based on PACC which is parallel memoryless scheme for accumulator based history sensitive computations. Also, we describe a special purpose hardware (VLSI chip) design scheme for the parallel prefix computation. The scheme is also an efficient solution to the problem in dataflow environment. A design methodology of linear systolic array of simple cells derived from the forwarding scheme of the accumulator based memoryless scheme to address the parallel prefix computation problem is presented in this chapter.

7.1 Prefix Computation Problem

Prefix computation is a basic operation of many important applications including the Grand Challenge problems, circuit design, digital signal processing, and graph optimazations [91]. In section 4.1, we introduced the accumulator based model of the history sensitive problems and it is analogous to the definition of the prefix computation problem; i.e. the cumulative sum (or product) problem is analogous to the prefix sum (or product) problem. The prefix computation problem (i.e. an accumulator based history

sensitive problem) contains the loop carried dependencies (i.e. sequential loop problem) and we developed the pipelined parallelization scheme (PACC) in the dataflow environment. In this section, we apply that scheme to design a special purpose hardware namely parallel prefix computator. For the sake of simplicity, we refer to prefix sums in this section since the results of the prefix sums can be readily applied to prefix computation with other associative operations [91].

7.2 Implementation Methodologies

For the hardware solution, the design should use the clock concept instead of the data dependencies used for the dataflow environment. Thus latches are used in the design to hold the data for synchronization. Figure-25 shows the high level design scheme of the size 4 (pipeline depth=4) parallel prefix sum computator.

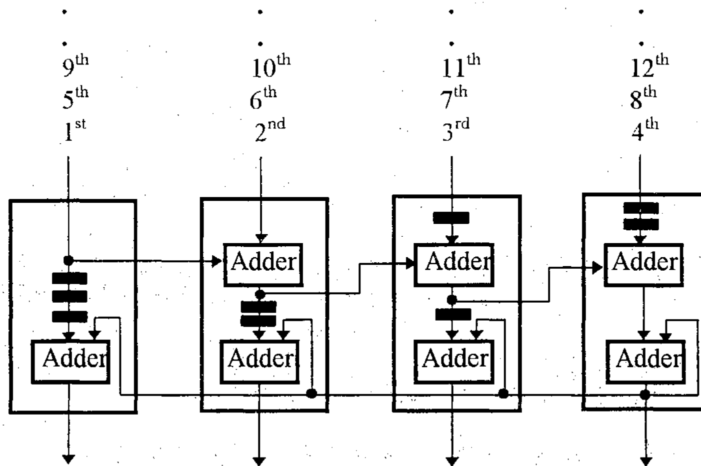


Figure-25. Parallel prefix sum computator of size 4 (pipeline depth=4).

In the figure, black rectangles in each cell are latches which hold the data for a period of one clock cycle. The design in the figure is based on the assumption that each addition

operation takes one clock cycle time. The detailed VLSI implementation is straightforward and thus skipped in this dissertation. In the design, each cell has two adders except the left-most cell and each cell has different arrangements of latches. The method to assign those latches in each cell is provided in Algorithm-2. Figure-26 shows the concept of latch arrangements with an example of size 5 (pipeline depth=5) parallel prefix computation hardware. Except the left-most and the right-most cells, each cell has same structure except the latch arrangements. Thus we can easily design any sized parallel prefix computator by connecting those cells and arranging the latches according to Algoritm-2.

Algorithm-2.

Assumption: For degree d (pipeline depth=d) parallel prefix computation hardware,

we assume d different input streams;

the 1st (left most) input stream consists of (order MOD d = 1) input elements,

the 2nd input stream consists of (order MOD d = 2) input elements,

the 3rd input stream consists of (order MOD d = 3) input elements,

. . . . .

the dth input stream consists of (order MOD d = 0) input elements.

Initial step: The Cell-1 (left-most cell) has only one operator and we arrange (d-1) latches before this operator in Cell-1.

i = 0;

j = d - 2;

```
for k = 2 (from left) to d
  { arrange i latches before 1st (upper) operator in Cell-k;
    arrange j latches before 2nd (lower) operator in Cell-k;
    i = i + 1;
    j = j + 1;
  }
```

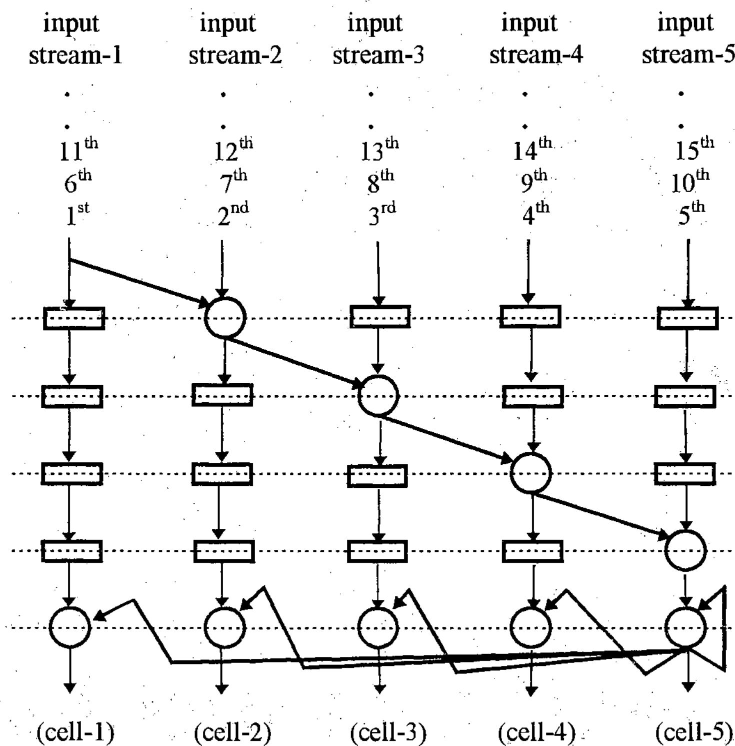


Figure-26. Latch arrangements in parallel prefix computator (pipeline depth = 5 case)
(Each column corresponds to each cell of the design.
The circles are the operators and the rectangles are latches.)

For the degree d (pipeline depth=d) parallel prefix computator, each cell has two operators and (d-2) latches except the first (left-most) cell which has one operator and (d-1) latches. Data dependencies used in the dataflow environment are handled by using

these latch arrangements. The initial tokens which input to the 2nd (lower) operators in the dataflow environment (refer to the Figure-15) are handled as follows: at clock d (d = pipeline depth), load the initial data value (0 for the prefix sum and, 1 for the prefix production) to each cell's 2nd (lower) operators' right input port. This can be done by using the clock counter. Of course the first (left-most) cell has only one operator and it receives the initial data value at clock d. Another method is using registers. At initialization time, initial values are preloaded in registers. For a stream size N, we can design the optimal parallel prefix computator with size = Int($\sqrt{N}$) which is the optimal pipeline depth as we verified in the previous chapter. As we see in the Figure-26, after d (= 5) initial trigger time, all cells generate one output at each clock cycle time.

Chapter VIII

APPLICATION TO PARALLEL STRING MATCHING

8.1 String Matching Problem

String matching is the problem of finding all occurrences of a pattern string in a reference string with or without errors. It has been one of the most extensively studied problems in computer science during the past two decades. It performs important tasks in many applications including information retrieval, library systems, artificial intelligence, pattern recognition, molecular biology, and text search and edit systems. Survey and comparison of well known algorithms can be found in [3, 52, 57]. The string matching problem is subdivided into two major categories, namely "exact string matching problem" and "approximate string matching problem". Algorithms have been developed separately for both problems. There exist two variations of the approximate string matching problem namely k-differences and k-mismatches problems. Remaining chapters deal with all three subproblems; i.e. the k-differences problem, k-mismatches problem and the exact matching problem.

Since VLSI technology has developed rapidly and building special purpose hardware is not a difficult problem, hardware approaches also have been proposed [22, 34, 49, 66, 67, 80]. Since high performance software algorithms are mostly multiphase (mark and scan) algorithms which use preprocessings with table look up methods, they cannot be applied to design special purpose hardware string matcher. Thus, we need high performance

algorithms for building special purpose hardware for string matching tasks. This is the motivation for developing efficient dataflow solutions to the problems.

Following chapters provide elegant parallel schemes for handling the k-differences, the k-mismatches, and the exact matching problems based on dataflow which are suitable for VLSI implementation. Then we apply these dataflow algorithms to design special purpose hardware (VLSI chip) string matcher which can function as a component device of a general-purpose computer. The designs are based on linear systolic array of basic cells and our pure dataflow algorithms are well suited for these. Thus our goal includes the design of efficient dataflow algorithms and parallelization schemes which provide parallel solutions to the string matching problems and, applying those algorithms to build special purpose parallel string matching hardwares. Since dataflow graph is the machine language of dataflow architectures, we will present the algorithms at the dataflow graph level. Thus, our schemes are also the efficient parallel solutions to the string matching problems in the dataflow machines which are attractive instruction level parallel architectures.

In the next section, problems are defined and related works in both software and hardware approaches are briefly reviewed. General design strategy of proposed dataflow schemes is described in section 8.3. In chapters IX, X, and XI, the k-differences, the k-mismatches, and the exact matching problems are discussed separately. Dataflow scheme and implementation methodologies of each subproblem is described in each chapter. The edit distance computation, which is essential task in approximate string matching, based on dynamic programming method is represented differently for the k-differences and the k-mismatches problems in corresponding chapters (chapters IX and

X). Since we specify the exact matching problem reside in the scope of the k-mismatches problem, it uses same schemes and the implementation methodologies with the k-mismatches problem with little variation. Chapter XI is devoted to exact string matching.

8.2. Problem Definitions and Related Work

String matching algorithms are used in many applications including genetic database search, speech recognition, text search and editing, and error-correcting compilers. In such applications, finding substrings both with and without errors are frequently needed. In this section we define three variations of the problem and software and hardware approaches to the problems are briefly reviewed. Before providing detailed descriptions, we categorize the subproblems according to their characteristics. Table-1 shows characteristics of three subproblems. As we see in the table, the super set is the k-differences problem. It contains the k-mismatches problem which contains the exact matching problem. Thus, the relationship among subproblems is as following:

k-differences problem $\supseteq$ k-mismatches problem $\supseteq$ exact matching problem

Subproblem	k-differences	k-mismatches	exact matching
Lengths of substrings	$\|pattern\| \pm k$	$\|pattern\|$	$\|pattern\|$
Edit cost	k	k	0
Edit operations	insertion deletion substitution	substitution	$\varnothing$

Table-1. Characteristics of string matching subproblems

8.2.1 K-differences Problem

With given reference string T ($|T| = n$) and pattern P ($|P| = m$), $n >> m$, k-differences problem consists of finding all occurrences (ending positions) of substrings of T which need at most k editing operations to convert to P. Editing operations include insertion ($\varepsilon \rightarrow c$), deletion ($c \rightarrow \varepsilon$), and substitution ($c_1 \rightarrow c_2$). For the sake of simplicity, in this work we assign edit cost of one to theses edit operations. The solution to this problem is very useful in fields including molecular biology.

There are several algorithms proposed for this problem. The problem can be solved in O(mn) time by dynamic programming [90]. For finding the minimum edit distance between two strings, dynamic programming technique uses table look up method. The main idea in dynamic programming method is that computed information is kept in memory space (table) and used for later computations instead of recomputing that. All existing algorithms use this technique with variations. Thus they need to keep and look up a table of size (n+1)*(m+1) [52]. Some optimized algorithms use preprocessing method with extra storage [52, 77, 88, 94]. They can reduce the worst case time and currently O(kn) is the best worst-case bound known if the preprocessing time is allowed to be at most $O(m^2)$. But these algorithms are two-phase algorithms which preprocess the pattern and keep the useful information in extra memory space in addition to the size (n+1)*(m+1) table for the scan phase. The best time bound O(kn) stands for only the scan phase. A parallel algorithm based on PRAM model which can be simulated on a bounded degree network is proposed in [16]. It provides parallel scheme which has complexity O(mn) for the product of time and number of processors used, but the implementation is vague.

8.2.2 K-mismatches Problem

K-mismatches problem is a subset of the k-differences problem in which the only editing operation permitted is substitution ($c_1 \rightarrow c_2$). Thus, with given reference string T (|T| = n) and pattern P (|P| = m), n << m, k-mismatches problem consists of finding all occurrences (ending positions) of substrings of T which are of the same length as the pattern and contains at most k mismatches.

Naive dynamic programming method [90] can solve this problem in O(mn) time and O(mn) space similar to the k-differences problem. Landau and Vishkin [56] presented O(k(n + m log m)) time and O(k(n + m)) space algorithm. Galil and Giancarlo [36] improved this algorithm with O(kn + m log m) time and O(m) space although it performs worse in practice. Generalized Boyer-Moore algorithm [88] was developed to solve this problem in O(kn(1/(m-k)+(k/c))) expected time where "c" denotes the size of the alphabet. Automata based method with linear time complexity can be found in [13]. Except the naive dynamic programming method, all the algorithms require extra time and space for preprocessings that are not included in the complexities shown above. The preprocessing part consists either of gathering useful information about pattern and reference strings or of building the finite state machine.

8.2.3 Exact Matching Problem

With given reference string T (|T| = n) and pattern P (|P| = m), n << m, exact matching problem consists of finding all occurrences (starting positions) of substrings in T which are exactly same as pattern P. The naive algorithm to solve this problem has a quadratic O(nm) worst case time complexity [25]. Each attempt takes m comparisons and there

exist n-m+1 attempts. Several linear time algorithms have been developed in the last twenty years [13, 24, 25, 57]. They could reduce the expected time to O(n+m) in practice but, the worst case time could not be changed. Most of the linear string matching algorithms preprocess the pattern before the scan phase. Others impose certain restrictions. The work done during the preprocessing phase is kept in extra space which is linear in the length of the pattern. Then this space is referenced in each attempt to minimize the comparison time. Automata-based algorithms also need preprocessing for building finite state machine, which corresponds to each pattern, in memory space [13, 55, 57].

With the conceptual CRCW-PRAM model, O(1) algorithm can exist theoretically but, it requires m*n processors. Optimal algorithms presented in [19, 37] have constant and O(loglog m) time complexities with n and (n / loglog m) processors respectively. But, these parallel algorithms also need extra preprocessing time and additional space.

8.2.4 Need for Special Purpose Architecture

Most software algorithms for string matching need to store entire reference string in main memory to manipulate the algorithm. For speed up, they use preprocessing time and extra memory space for storing useful information. Automata based algorithms use complex control and spaces for the finite state machine simulation. Parallel algorithms work on conceptual CRCW-PRAM model theoretically but how to assign processors to the operations is vague.

Because speed is the essential factor of those applications that need string matching tasks, special purpose hardware should be attached to the host computer as a peripheral

device like sorter or FFT device. Figure-27 shows the organization of general purpose computer with such special purpose devices attached. In addition, the high level language code needs several levels of translations before execution, the software control required to execute the basic instructions results in longer execution times. Using special purpose VLSI chips, most of the basic operations can be completed in a single clock.

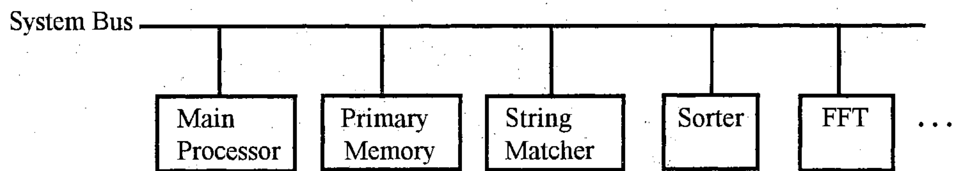


Figure-27. System block diagram of general purpose computer

For example the Splash-1 [32], which is special purpose hardware attached to a Sun workstation for DNA sequence searching (approximate string matching), outperformed a Cray-2 super computer by a factor of 325. The Fast Data Finder (FDF) [51], which is a hardware accelerator coupled with the Pracel's Biology Tool Kit (BTK) for DNA sequencing or the 3-dimensional visualization of complex biomolecules, performed 1700 times faster than a Sparc-5 Unix workstation without FDF. The FDF accelerator performed the sequence match in less than a minute, compared with 21 hours on a high performance workstation [51].

8.2.5 Hardware Approaches

8.2.5.1 K-differences problem

Since the solution to the k-differences string matching problem, which is mostly represented as the approximate string matching problem, is very useful in the molecular

biology area (i.e. DNA sequence checking), there have been many research projects exploring hardware solution. Prominent hardware approaches found in the literature that are related to the k-differences problem are briefly reviewed in this section. Cheng and Fu [22] proposed a VLSI architecture for computing the edit distance and sequence between two strings. In this approach, two dimensional arrangement of n*m processing elements were used for processing strings of lengths n and m which leads the cost problem. In addition, in each clock cycle time it needs n + m inputs to be provided to the architecture. Proposed 2-dimensional array of cells is illustrated in Figure-28.

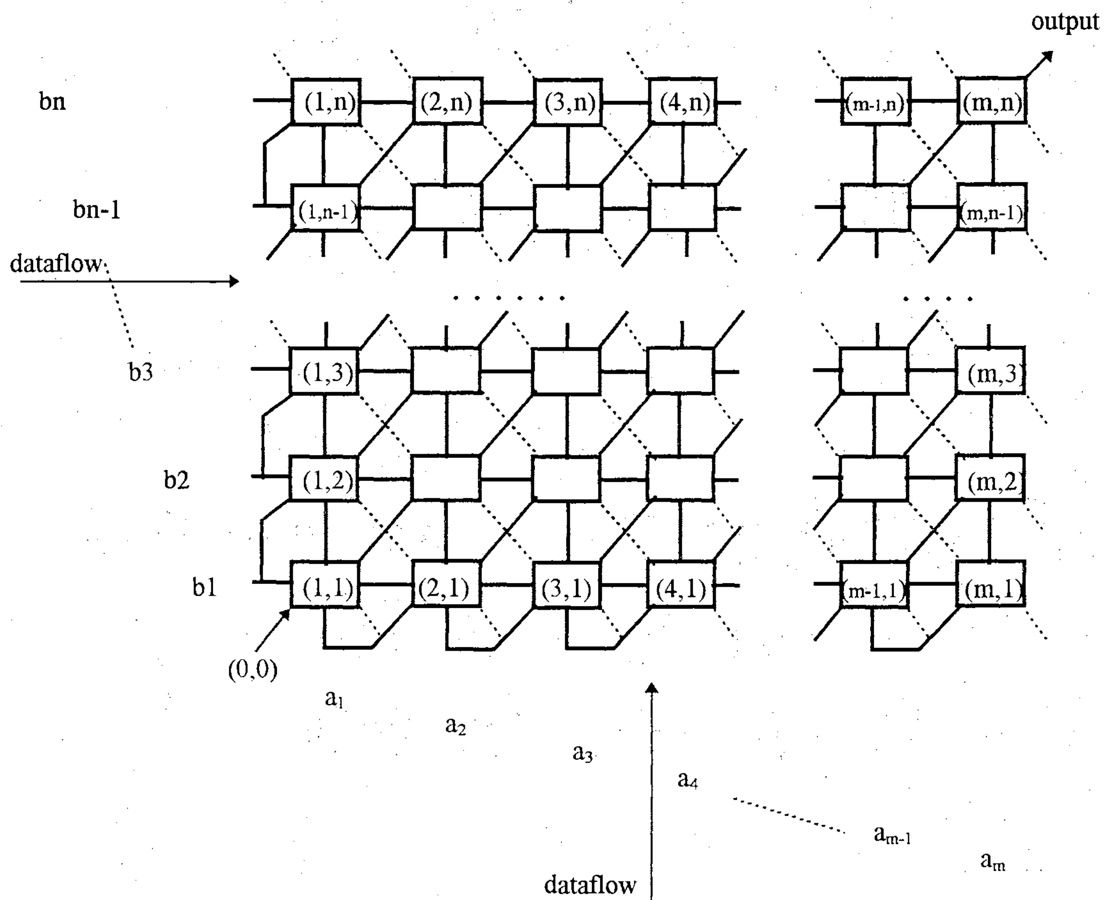


Figure-28. 2-dimensional structure for string distance computation by Cheng and Fu

P-NAC (Princeton Nucleic Acid Comparator) [61] was built using linear systolic array architecture for comparing DNA sequences. Similarly, Sastry et al [80] presented a VLSI architecture for computing similarity between two strings based on linear systolic array of cells. In these linear approaches, two strings to be compared are entered from the opposite sides of the array of cells. These approaches require m+n-1 processing elements to process stings of lengths n and m. It is better than n*m but, when string lengths are very long it still has problems of space and cost. In [79], partitioning idea is suggested when strings are too long to be compared by given array of processing elements. But it requires multiple passes of processing and it degrades the performance.

The above approaches suffer from the extreme hardware space and cost when dealing with very long reference strings which are commonly used in many applications. Also they do not work on infinite length strings. Another draw back of these hardware approaches is that they only compute similarity between two complete strings; i.e. they can not check similarities between pattern and all substrings of reference string (text).

In real world, implemented products have been used in molecular biology area. Two generations of the Splash processors (Splash-1 and Splash-2) which are based on systolic arrays of FPGAs (field-programmable gate arrays) have been designed at the Super computing Research Center (SRC) [32]. The Splash-1 includes a 32-stage linear logic array with a VME interface to a Sun workstation. Each stage consists of an XC3090 FPGA and a 128-kbyte static memory buffer. The Splash-2 is an attached processor with an Sbus interface to a Sun Sparcstation. Each card holds 16 XC4010 FPGA devices coupled with 256 * 16-bit RAMs; up to 16 cards can be used in a system.

The Fast Data Finder (FDF) [51] is a hardware accelerator coupled with the Pracel's Biology Tool Kit (BTK) for DNA sequencing or the 3-dimensional visualization of complex biomolecules. The FDF was developed in partnership with Perkin-Elmer Corp.'s Applied Biosystems Division as a SCSI peripheral to a Unix host. Its speed comes from its parallel internal architecture, which uses multiple VLSI processor chips to divide and conquer tough searching problems (from biological sequencing to ordinary text database searches [51]).

8.2.5.2 Exact matching problem

Some serial and automata based hardware algorithms and implementations for the exact matching problem are found in the literature. Mukhopadhyay [67] proposed a primitive nonnumeric processor scheme in which the characters of pattern string are preloaded in each processing element and input characters of the reference string are applied (broadcast) to all processing elements. Foster and Kung [34] proposed design of VLSI chip for which linear systolic array of cells (processing elements) are used. This scheme uses alternative input of pattern and reference string characters one at a time into the array of cells from both directions. During each pair of consecutive time slices the chip inputs two characters and returns one output. Thus the required time slices to process entire reference string of length n is 2n slices which is twice its length. To minimize the number of cells to the number of characters in the pattern, the pattern recirculation method (i.e. the last character of the pattern followed by the first character of the pattern again) can be used in this approach. Architectural design scheme and the simulation of the automata based approach are found in [49]. Since this scheme needs to construct the finite state

machine, complex control communications and memory spaces to manage the automata action are required.

8.3 Design Strategy

As we mentioned in earlier chapters, dataflow machines have been proposed and developed steadily as an attractive instruction level parallel computation model [8, 29, 45]. In the dataflow environment, an operation (instruction) is executable if all its required operands are available based on data driven mechanism and any set of enabled operations can be executed in parallel. To be a good parallel architecture, it should also provide good performance on non-computational tasks such as string matching problems. In dataflow environment, array or table handling is a critical problem [38, 59] and we should not use multi-phase (i.e. mark and scan phases) table look up methods that most high performance string matching algorithms use. They are also not proper for the VLSI implementation which needs elegant systolic algorithm. Thus we need efficient single pass dataflow scheme using only data flow without preprocessing and table look up methods. This pure dataflow scheme can be used to build VLSI chip since it does not use any memory references and can be converted to systolic array of cells easily.

In hardware approaches which use linear systolic array of processing elements, two ways of matching each pattern character with each reference character have been proposed in literature. One method used in [34, 61, 80] is processing both pattern and reference strings moving through the array of cells from opposite directions. This method has the disadvantage of using many cells. Indeed in [80], m+n-1 cells (processing elements) are needed where m and n are the lengths of pattern and reference

strings respectively. The other method used in [66, 67] stores pattern string in the array of cells and processes the reference string from one direction into the array of cells. Our approach uses the later method which stores the pattern string in the structure.

As Foster and Kung [34] mentioned, the good algorithms for VLSI implementation are not necessarily those requiring minimal computation. Computation is cheap in VLSI and the communication determines the performance. This matter is also applicable to the dataflow environment. As used in high performance software string matching algorithms, trial of skipping operations will degrade the over all performance of the pipelining in the dataflow algorithm. In VLSI special purpose chip design, the most important thing is choosing good algorithm since it determines the cost and performance of the design. The good algorithm for this purpose is called systolic algorithm and has the following properties [34]:

- The algorithm can be implemented by only a few different types of simple cells.
- Its data and control flow is simple and regular; i.e. cells are connected by local and regular interconnections.
- The algorithm uses extensive pipelining and multi-processing. Multiple data streams move at constant speed over fixed paths in the structure. Thus, large number of cells are active at one time so that the computation speed can keep up with the data rate.

Systolic algorithms have several advantages which help reduce the VLSI implementation cost:

- Since most cells are copies of few basic cells, one can design and test only few cells.
- Regular interconnection implies that the design can be modular and extendible.

- Pipelining and multi-processing by including many identical cells provide high performance.

Proposed dataflow schemes in the following chapters meet above characteristics of systolic algorithm very well. The next chapter will show how the implicit parallelism in the proposed dataflow scheme provide time complexity $O(n + m)$ for the k-differences problem. The k-mismatches problem and the exact matching problem, for which we can exploit explicit parallelism in addition to the implicit parallelism of dataflow method, will be considered separately in the following chapters. Time complexities of these explicit parallel schemes are $O((n/d) + \alpha)$ where d represents the number of streams used (controllable parallelism degree) and, $\alpha = 0$, log m, or m.

The schemes are presented at the dataflow graph level. For simplicity, the static dataflow environment is assumed because it is easy to implement. In static dataflow environment, only one token can reside on an arc at a time. Upon arrival of all required operands, the operation fires according to data driven mechanism.

Chapter IX

K-DIFFERENCES PROBLEM

As defined in previous chapter, the k-differences problem consists of finding all occurrences (ending positions) of substrings of a reference string (T), which need at most k editing operations to convert to pattern (P). Editing operations include insertion, deletion, and substitution.

9.1 Edit Distance Computation

In approximate string matching, the essential task is computing edit distance between two strings; i.e. the pattern string and any substring of the reference string (text). In this section, we introduce the dynamic programming strategy [90] for finding minimum edit distance between pattern string and any substring of reference string. For computing the minimum edit distance, we need a table of size (m+1)*(n+1) where m and n are the lengths of pattern and reference strings. Let D be the table in which each entry $D_{i,j}$ represents the minimum edit distance between $p_1..p_i$ and any substring of the reference string T ending at t_j. Then solutions to the k-differences problem can be found in m^{th} (last) row of the table D; i.e. if $D_{m,j} \leq k$ where $1 \leq j \leq n$, then there is an approximate occurrence of pattern ending at position j of the reference string with edit distance less than or equal to k. The minimum edit distance table D for the k-differences problem is defined as following:

$$D_{0,j} = 0, \quad 0 \le j \le n$$

$$D_{i,j} = \min \begin{cases} D_{i-1,j} + 1 & \text{/* for deletion */} \\ D_{i-1,j-1} \begin{cases} +0, & \text{if } p_i = t_j \\ +1, & \text{else.} \end{cases} & \text{/* for substitution */} \\ D_{i,j-1} + 1 & \text{/* for insertion */} \end{cases}$$

For the k-differences problem, allowed editing operations are:

insertion ($\varepsilon \rightarrow c$) : a character c is inserted into the empty position.

deletion ($c \rightarrow \varepsilon$) : a character c is deleted.

substitution ($c_1 \rightarrow c_2$) : a character c_2 is substituted by new character c_1.

Each edit operation has corresponding cost and we assume cost 1 for all operations for the sake of simplicity. In the table D, edit sequence from left to right implies insertion, from top to bottom implies deletion, and from left_upper to right_down (diagonal) implies the substitution operation. Following example illustrates the idea:

Example-1. Pattern(P) = "cacd", Text(T) = "bcbacddc"

Minimum edit distance table D looks like:

		b	c	b	a	c	d	d	c
	0	0	0	0	0	0	0	0	0
c	1	1	0	1	1	0	1	1	0
a	2	2	1	1	1	1	1	2	1
c	3	3	2	2	2	1	2	2	2
d	4	4	3	3	3	2	1	2	3

$D_{4,5}$

Initial values are kept in 0th row and 0th column. For k = 2, for instance, there are approximate occurrences of P ending at t_5, t_6, and t_7 because $D_{4,5}$, $D_{4,6}$, and $D_{4,7}$ are 2, 1, and 2 respectively which are less than or equal to k=2. For example with $D_{4,5}$ which represents the approximate occurrence of pattern with edit distance 2, one possible edit sequence is depicted (dotted lines) in the table of Example-1. Clearly the pattern "cacd" can be converted into "bac" which is a substring of T ending at t5 with edit cost 2. The edit sequence consists of substitution (c→b), substitution_0 (a→a) , substitution_0 (c→c), and deletion (d→ε). Substitution_0 represents the self substitution and has cost 0. Of course there are other possible edit sequenses which leads the pattern to be converted into same substring or other substrings of T ending at same position t_5 with same edit cost 2. For example, pattern "cacd" can be converted into other substrings of T "cbac", and "ac" which end at position t_5 with edit cost 2. In fact, "cbac" needs an insertion (ε→b) and a deletion (d→ε), and "ac" needs two deletions (c→ε, d→ε). When the case of tie, the edit sequence depends on the algorithm used.

Table D can be evaluated column by column in time O(mn) by naive dynamic programming method. Our dataflow scheme, which is suitable for the VLSI implementation, parallelizes this table computation by the order of m (pattern length) without using any memory space and preprocessing. It calculates m entries of table D at once. In fact, the calculation is done in pipelined manner. Our parallel scheme is represented in the next section.

9.2 Dataflow Scheme: Implicit Parallelism

We start from the edit distance table represented in previous section. Our design implements and parallelizes this table computation by the order of m (pattern size) without using any memory space to keep information needed. It calculates m entries of table D at once. In fact, the calculation is done in pipelined manner and one diagonal (m entries) of the table can be evaluated at a time.

Figure-29 illustrates the very high level dataflow used in the scheme. To arrange the reference characters for our need in static dataflow environment, we use the WS block which is the memoryless scheme for handling working-set based history sensitive computation presented in Chapter 4.

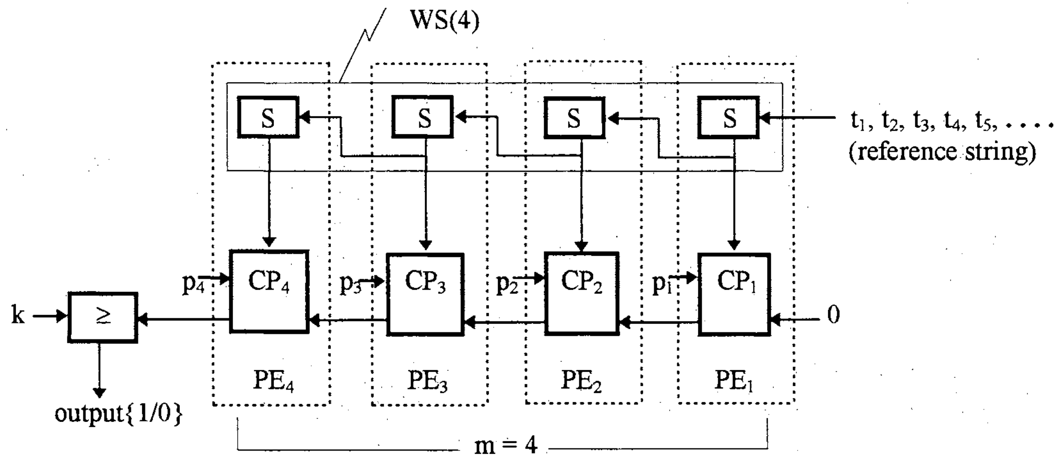


Figure-29. High level dataflow in parallel scheme (m=4 case).

In Figure-29, WS(4) is the window size 4 working-set based model of history sensitive block. Each unary actor "S" is a synchronization actor and simply passes one data token upon receiving it; i.e. it acts like a latch in hardware. In general, the block WS(m) is used for pattern size m and acts as a size m shift register. The behavior of the

WS(m) block used in the scheme in static dataflow environment is shown in Figure-30. The definition and language construct of WS(m) in dataflow environment are described in detail in Chapter 4.

Figure-30. Behavior snap shots of WS(m) block in the scheme (m=3 case).

Our intention with the design in Figure-29 is that, for instance, CP_4 computes $D_{4,1}$ while CP_3 computes $D_{3,2}$, CP_2 computes $D_{2,3}$, and CP_1 computes $D_{1,4}$. Thus one diagonal of the table can be evaluated at the same time and we can achieve 1/m total processing time by using m processing elements. The output of the scheme is a boolean string of length n (same length as the reference string). If any substring of the reference string ending at position t_j matches the pattern with edit distance at most k, the jth output is 1 otherwise it is 0. Thus, with the example of the previous section in which pattern(P) = "cacd", text(T) = "bcbacddc", and k=2 corresponding output string generated is {00001110} because $D_{4,5}$, $D_{4,6}$, and $D_{4,7}$ of the minimum edit distance table D are less than or equal to k=2. The history sensitivity required to compute each entry of the table D resides in our

design. Each computation of $D_{i,j}$ needs previous entries $D_{i-1,j-1}$, $D_{i-1,j}$, and $D_{i,j-1}$ in addition to current inputs (p_i and t_j). Thus computation parts (CP_1..CP_4) are connected from right to left to preserve the histories of $D_{i-1,j-1}$ and $D_{i-1,j}$. The history of $D_{i,j-1}$ is kept by using feed back arc in each CP_i. In our scheme, all required histories are kept in dataflow graph itself during run time without using any memory space and preprocessing times. Each CP_i evaluates table entries $D_{i,j}$ for all j ($1 \leq j \leq n$). A refined scheme is depicted in Figure-31.

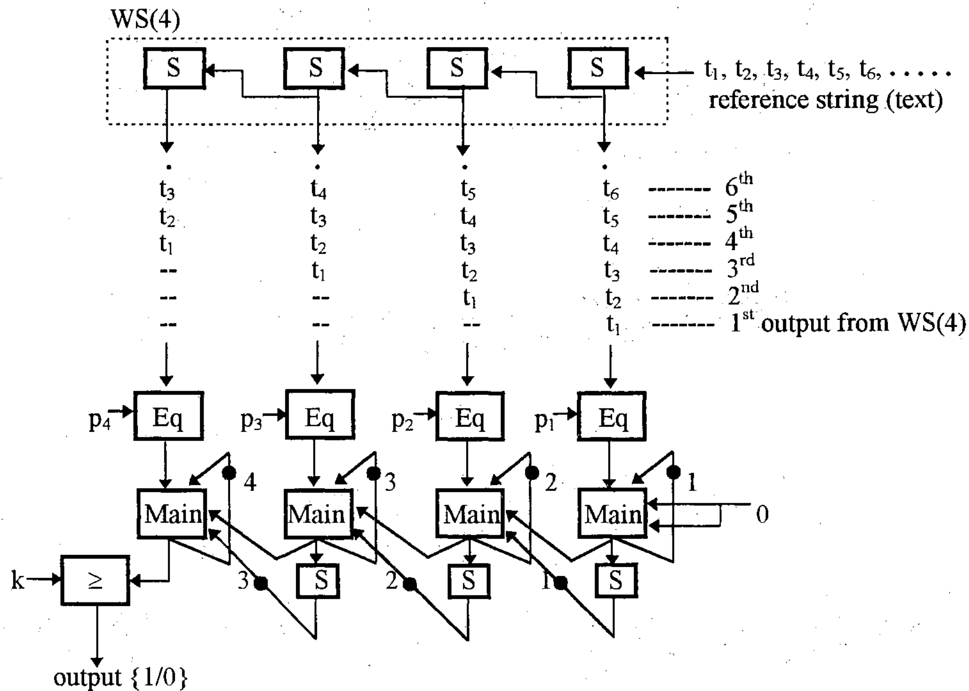


Figure-31. Refined scheme for approximate string matching (m=4 case)

In Figure-31, initial tokens are used to assign values of the 0th column in the edit distance table D. Initial values of 0th row of the table D are embedded in the operation block "Main" of the 1st (right most) PE. The operation block "Eq" compares two operands (t_j and p_i) and if they are same, it returns 0 otherwise it returns 1. This data is needed for

the edit operation substitution. The major operation block "Main" finds optimal edit pass, which has the minimum edit cost, from the left (for insertion $\varepsilon \rightarrow c$), upper (for deletion $c \rightarrow \varepsilon$), and diagonal (for substitution $c_1 \rightarrow c_2$) entries of the table D. To Compute the table entry $D_{i,j}$, its feed back arc provides left ($D_{i,j-1}$) entry and two input arcs from the right side provide upper ($D_{i-1,j}$) and diagonal ($D_{i-1,j-1}$) entries of the table D. Value for the diagonal entry is passed through a synchronization actor "S" to create one delay so that the $D_{i-1,j-1}$ can meet the equality information of p_i and t_j for evaluating $D_{i,j}$. Details of the block "Main" is illustrated in Figure-32.

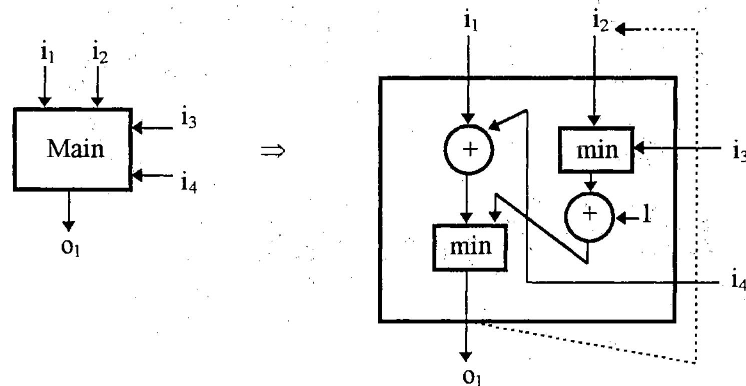


Figure-32. Refined operation block "Main"

In Figure-32, the operation block "min" does the comparison work and outputs the smaller operand from two input operands. By using formal dataflow graph notation [47], it can be depicted in detail as shown in Figure-33. Figure-33(a) shows the typical sort of merging actor in dataflow environment and Figure-33(b) illustrates the "min" block by using it. In the figure, synchronization actor "S" is used for the static dataflow environment. The block "min" can be implemented by using a comparator and multiplexer. "S" actors can be implemented with latches.

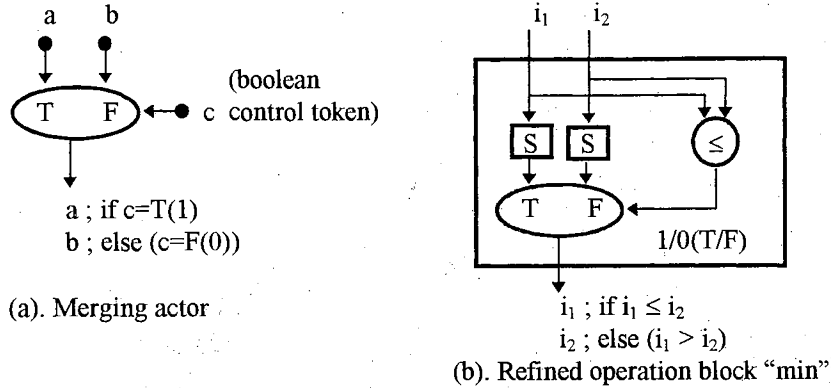
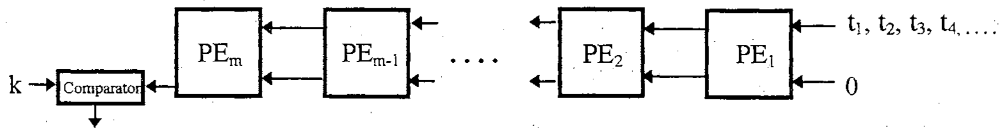


Figure-33. Dataflow graph (static) for the operation block "min"

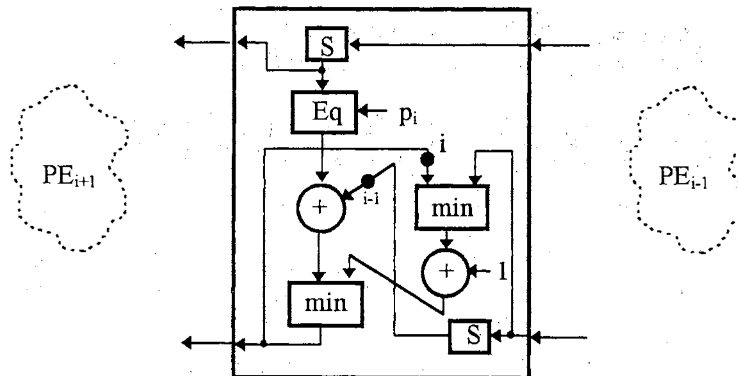
We will explain our scheme with the "Main" block depicted in Figure-32. Let's assume the "$Main_i$" block in i^{th} (from right to left) PE (refer to Figure-29) which evaluates $D_{i,j}$ ($1 \leq j \leq n$). Input i_1 which is a binary value resulting from equality checking (i.e. 0 if equal, or 1 else) of p_i and t_j is added to i_4 which is diagonal entry $D_{i-1,j-1}$ computed in "$Main_{i-1}$" block of PE_{i-1} at two time periods before. Since i_4 is supplied by a synchronization actor "S", it can provide the value of the diagonal entry (refer to Figure-31). Thus we have information for the edit operation substitution which we defined in section 9.1; i.e. $D_{i-1,j-1}$ + 0 if $p_i = t_j$, else + 1. Input i_2 is feed back data which corresponds to the left entry $D_{i,j-1}$ and computed in same "$Main_i$" block (in PE_i) at the previous time period. It provides the data for the edit operation insertion. Input i_3 corresponds to the upper entry $D_{i-1,j}$ and is produced by "$Main_{i-1}$" in PE_{i-1} at one previous time. I_3 provides the data for the edit operation deletion. Then the lesser of i_2 and i_3 is chosen by block "min" and added with 1. This provides the information $min(D_{i-1,j} + 1, D_{i,j-1} + 1)$. This data and the other data which we mentioned earlier for the substitution operation are compared in the other "min" block and the smaller one is output (o_1) as the value of $D_{i,j}$. This output is then sent to itself (feed back) for evaluating $D_{i,j+1}$ during the next time period. It is sent to "$Main_{i+1}$"

in PE_{i+1} for evaluating $D_{i+1,j}$. It is sent to "$Main_{i+1}$" in PE_{i+1} for evaluating $D_{i+1,j+1}$ two time periods later. Thus we achieved the goal of computing the minimum edit distance between $p_1..p_i$ and any substring of T which ends at position t_j; i.e. $min(D_{i-1,j} + 1, D_{i,j-1} + 1,$ and $(D_{i-1,j-1} + 0$ if $p_i = t_j$, else $+ 1))$. The final output of the scheme is generated through "$Main_m$" of PE_m which computes the entries of mth row of the minimum edit distance table D; i.e. D_{mj} for $1 \le j \le n$. The output from PE_m is compared with error bound k and if it is less than or equal to k, 1 is generated, otherwise 0 is generated, at a time as the final output. For example the jth output 1 means that approximate occurrence of the pattern $(p_1..p_m)$ with edit distance less than or equal to k is found at ending position j of the reference string (text).

Our proposed scheme is a linear systolic array of processing elements (PEs) which is suitable for VLSI implementation. The framework of the array and optimized dataflow graph for each PE_i is depicted in Figure-34. In Figure-34, operation blocks "Eq" and "min" are same as we defined and illustrated previously.



(a). Linear systolic array of m (pattern size) processing elements

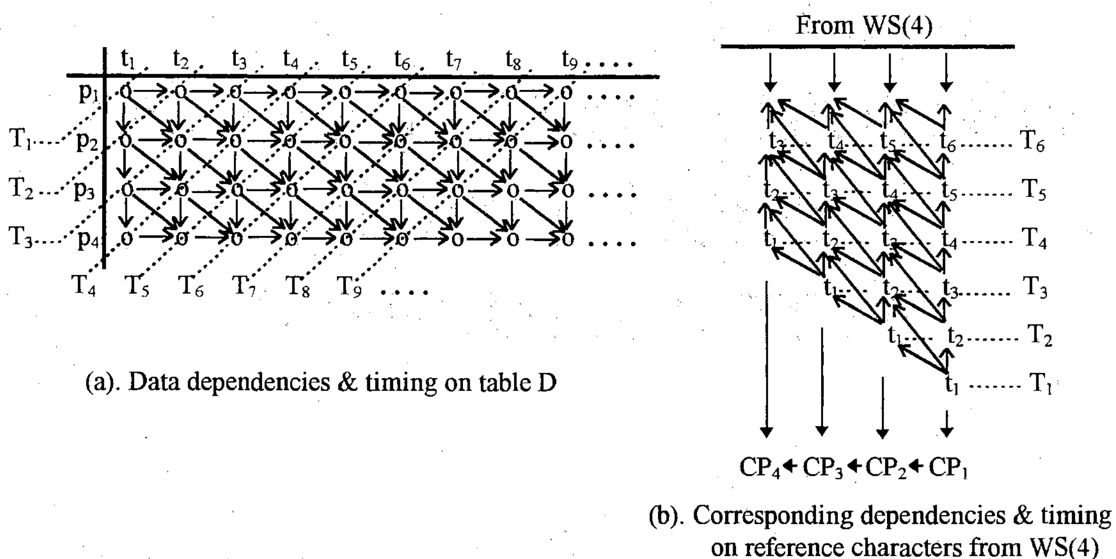


(b). Dataflow graph for each PE_i

Figure-34. Linear systolic array for the k-differences problem

9.3 Performance

In this section, we analyze the performance of the parallel scheme. With serial dynamic programming method, quadratic time complexity (O(mn)) is needed to evaluate the minimum edit distance table D. Since data dependencies among entries in table D allow us to evaluate one diagonal (dotted lines in Figure-35(a)) of entries at same time, our proposed parallel scheme reduces the total processing time by a factor of m (pattern size). Figure-35(a) illustrates the data dependencies and parallel timing on the table D. As we see in the figure, we cannot process more than one diagonal at a time because each diagonal is dependent on its previous diagonal. For example, $D_{3,4}$ which belongs to the 6th diagonal (marked with virtual time T_6) is dependent on $D_{2,4}$ and $D_{3,3}$ which belong to the 5th diagonal (marked with virtual time T_5), and so on. For parallel processing of m entries on each diagonal at a time, we used the mechanism WS(m) to arrange the reference characters and connected each CPs horizontally (see Figure-29). Figure-35(b) shows the corresponding data dependencies and timing on the reference characters arranged.



(a). Data dependencies & timing on table D

(b). Corresponding dependencies & timing on reference characters from WS(4)

Figure-35. Data dependencies and paralleism on table D (m=4 case)

At initial time (T_1), CP_1 processes t_1 and computes $D_{1,1}$. At time T_2, CP_1 and CP_2 process t_2 and t_1, and compute $D_{1,2}$ and $D_{2,1}$ accordingly and so on. Thus, beginning the time period T_m, all m CPs work and compute one diagonal entries (m) of the table D in parallel; i.e. at time T_4, CP_1 processes t_4, CP_2 processes t_3, CP_3 processes t_2, and CP_4 processes t_1 simultaneously and they generate $D_{1,4}$, $D_{2,3}$, $D_{3,2}$, and $D_{4,1}$ respectively in parallel. Entries of edit distance table D evaluated from our scheme, which is a linear systolic array of m processing elements, at each time slice is illustrated in Figure-36.

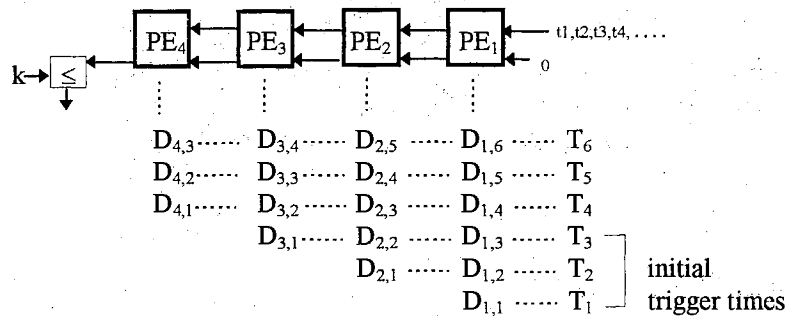


Figure-36. Parallel evaluation of the table D (m=4 case)

As we see in the figure, each PE_i is responsible for evaluating one row ($D_{i,j}$ for all $1 \leq j \leq n$) of the table D. PE_m generates the last row of the table which is the solution to the problem. After m-1 initial trigger time slices, one output is generated from PE_m at each time slice. Since there is n columns in the table D, required time to evaluate the entire table is (m - 1) + n time slices. The actual clock cycle time in each time slice is decided by the critical path in the PE. With considerably long pattern string, the initial trigger time m - 1 cannot be ignored and the time complexity of the scheme is O(n + m).

9.4 Implementation Methodologies

Since the proposed dataflow scheme is simple and easy to be modularized, we can build VLSI chip for the approximate string matching problem in a straightforward way. Our scheme can be easily converted into the systolic array architecture. The linear systolic array architecture of our scheme is depicted in Figure-34(a). The structure of each cell (PE) is identical and thus we need only one type of basic cell to build any size special purpose hardware. The architecture can be extended by copying and connecting the PE cells to the array. For checking error bound k for the final output of the scheme, a comparator which checks the condition *L.E.* should be connected to the final PE (PE_m) as we see in Figure-34(a). To convert our dataflow scheme into hardware design, we should use the clock concept instead of the data dependency. Thus latches are used for synchronization. We present very high level design methodologies to develop special purpose VLSI chip namely parallel approximate string matcher (PASM) by using high level block diagram. Dataflow graph representation of each PE_i is depicted in Figure-34(b) and the corresponding implementation scheme is illustrated in Figure-37. In Figure-37, latches are used for synchronization between clock phases. The right most latch is used for holding data for one clock cycle to reserve the diagonal (substitution) information. Register R_{i1} is preloaded with the pattern character p_i at initialization time. Another register R_{i2} is preloaded with the value 1 for the addition operations. For the optimal design, we used a 3-way minimum comparator for finding the minimum value from three input values. This design can work on two-phase nonoverlaping clocking scheme. A reference character is entered and the character equality checking (dotted box) is done in each odd (even) numbered PE during clock phase ϕ_1 (ϕ_2). Minimum

computation in each odd (even) numbered PE is done during the clock phase ϕ_2 (ϕ_1). Thus, each PE is active in both phases. The initial tokens used in the dataflow graph representation can be preloaded appropriately by using registers or latches if needed.

Figure-37. Structure of each PE_i for the k-differences problem

Chapter X

K-MISMATCHES PROBLEM

As defined in Chapter VIII, k-mismatches problem consists of finding all occurrences (ending positions) of substrings of T which are same length of the pattern and contains at most k mismatches. The k-mismatches problem is a subset of the k-differences problem with substitution as the only permitted editing operation.

Since the exact matching problem is a subset of the k-mismatches problem, the schemes described in this chapter are also applicable to the exact matching problem.

10.1 Edit Distance Computation with K-mismatches

Since k-mismatches problem is a subset of the k-differences problem, we start from the edit distance computation described in previous chapter. The only allowed edit operation in the k-mismatches problem is substitution ($c_1 \rightarrow c_2$). Since all substrings of reference string (text) searched are same length of the pattern, edit operations insertion and deletion are not needed and thus not allowed.

Let D' be the (m+1)*(n+1) table in which each entry $D'_{i,j}$ represents the edit distance between $p_1..p_i$ and a substring of the reference string T ending at t_j which has length i. Then solutions to the k-mismatches problem can be found in m^{th} (last) row of the table D'; i.e. if $D_{m,j} \leq k$ where $1 \leq j \leq n$, then there is an approximate occurrence of pattern ending

at position j of the reference string with number of mismatches less than or equal to k. The edit distance table D' for the k-mismatches problem is defined as following:

$D'_{0,j} = 0, \quad 0 \le j \le n$ /* initial values */

$D'_{i,0} = m, \quad 0 \le i \le m$ /* initial values */

$$D'_{i,j} = D_{i-1,j-1} \begin{cases} +0, & \text{if } p_i = t_j \\ +1, & \text{else.} \end{cases}$$ /* for substitution */

In the table D', edit sequence from left_upper to right_down (diagonal) implies the substitution operation. Following example illustrates the idea:

Example-2. Pattern(P) = "cacd", Text(T) = "bcbacddc"

Edit distance table D' looks like:

		b	c	b	a	c	d	d	c
	0	0	0	0	0	0	0	0	0
c	4	1	0	1	1	0	1	1	0
a	4	5	2	1	1	2	1	2	2
c	4	5	5	3	2	1	3	2	2
d	4	5	6	6	4	3	1	3	3

dummy data (upper right), dummy data (lower left), $D_{4,6}$

Initial values are kept in 0th row and 0th column. Since all substrings searched have same lengthes (m) with pattern string, first (m-1) solutions are dummy. Thus we assigned initial values m in the 0th column. That makes solutions $D'_{m,1}$ .. $D'_{m,m-1}$ have values outside the bound of the k. For the k=2 with Example-2 for instance, there are approximate occurrences of pattern (P) ending at t_6 because $D_{4,6}$ is 1 which is less than or equal to k=2. Edit sequence ,which includes all substitutions, is depicted (dotted lines) in

the table of Example-2. Clearly the substring “bacd” is converted to the pattern (“cacd”) by substitution (b→c), substitution_0 (a→a) , substitution_0 (c→c), and substitution_0 (d→d). As mentioned earlier, substitution_0 does not cost any.

Three dataflow schemes, which are suitable for the VLSI implementation, are presented in the next section. Those schemes evaluate entries of edit distance table D’ one diagonal or one column (i.e. m entries) at once in pipelined manner. Thus the gain is reduced time complexity by a factor of m. Furthermore parallel schemes based on multiple input and output streams will reduce the time complexity by a factor of d, where d stands for the number of input/output streams used (parallelism degree). These schemes are described in following sections.

10.2 Dataflow Scheme: Implicit Parallelism

Different from k-differences problem, there are no vertical (from top to bottom) and horizontal (from left to right) data dependencies among entries of table D’. That leads our dataflow schemes evaluate entries of table D’ one diagonal or one column (m entries, where m stands for the length of pattern) at a time and accommodate explicit parallelism (i.e. multiple diagonals or columns are computed at a time). In this section, three different dataflow schemes such as the hierarchical, the linear, and the broadcasting schemes are presented. Based on these serial schemes, parallel schemes are developed and represented in the next section. The hierarchical and the linear schemes are based on WS block which is the Memoryless scheme for working-set based history sensitive computations defined in chapter IV. The broadcasting scheme is based on BC block which is a variation of the WS block and does not use any synchronization actors. The

name BC stands for Broadcasting and the BC block is represented in section 10.2.3. With these schemes, m entries of the table D' are evaluated at a time and one element of the last row, which contains the solution to the problem, is generated at a time slice. In fact, the calculations are done in pipelined manner. With the linear and the broadcasting schemes, one column (m entries) of the table D' is evaluated at a time. With these three dataflow schemes (serial schemes), the gain is $O(n + \alpha)$ time complexity of evaluating all entries of table D' where n stands for the length of reference string and α is log m for the hierarchical, m for the linear, and 0 for the broadcasting scheme. When the reference string is very long ($n >> m$) and the pattern string length is considered as a constant, the α (initial trigger time) can be ignored in the hierarchical and the linear schemes. Based on these serial schemes, parallel schemes are developed by using multiple stream input and output; i.e. parallel hierarchical, parallel linear, and parallel broadcasting schemes. The parallel hierarchical and the parallel linear schemes use PWS block which is parallel WS block and defined in Chapter VI. The parallel broadcasting scheme uses PBC block which is parallel broadcasting (BC) block. The PBC block is a variation of the PWS block and it is represented in section 10.3.2. These three parallel schemes are presented in section 10.3.

10.2.1 Hierarchical Scheme

In previous chapter, the dataflow scheme for the k-differences problem, which evaluates one diagonal (from left-bottom to right top) of the edit distance table D at a time by using WS block, was represented. That method with simplification can be used for the k-mismatches problem but, it does not accommodate more parallelism because there exist

dependencies among such diagonals. Based on that mechanism, parallel scheme can not be exploited. Since entries of the edit distance table D' for the k-mismatches problem still have diagonal data dependencies (for edit operation substitution), evaluation of more than one diagonal with that mechanism is impossible. Thus we design the dataflow scheme for the k-mismatches problem to evaluate one diagonal of the table D' which has opposite sequence from the diagonal considered in k-differences problem; i.e. diagonal entries from left top to right bottom. Figure-38 illustrates the data dependencies and the conceptual timing of the scheme on the edit distance table D' for the k-mismatches problem. Since there exist no dependencies among such diagonals, parallel approach, which evaluates multiple diagonals at a time, can be derived from this concept.

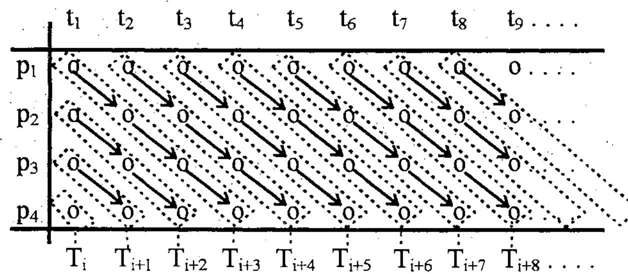


Figure-38. Data dependencies & conceptual timing on table D' (m=4 case)

In this section, a serial dataflow scheme named the hierarchical scheme which evaluates one diagonal (from left_top to right_bottom direction) of the table D' at a time is presented by using the WS block. In fact the evaluation is done in a pipelined manner. The parallel scheme, which evaluates multiple diagonals at a time, is presented in the next section. As shown in Figure-38, entries in each diagonal have data dependencies among them; i.e. dependencies from each $D'_{i,j}$ to each $D'_{i+1,j+1}$. For evaluating the $D'_{m,j}$, diagonal entries $D'_{1,j-(m-1)}$ .. $D'_{m-1,j-1}$ should be evaluated serially. But, since the solution

to the k-mismatches problem needs only the last low of the table D', we do not have to evaluate entries in each diagonal serially. The intention of the hierarchical scheme is that in each diagonal, matching information (0 if p_i and t_j are matched, else 1) of m entries are all added to produce the last entry ($D'_{m,j}$) of the jth diagonal. The addition is done hierarchically by using binary addition operations in a pipelined manner. After log m initial trigger times (pipeline depth), one output (each entry of the last row of the table D') is generated from the scheme at a time. We name this dataflow scheme the hierarchical scheme since its computation part consists of hierarchically (binary tree structure) connected operation blocks.

In the static dataflow environment, to arrange the characters of the reference string for our need, we use WS(m) block as shown in Figure-39; i.e. t_1 meets p_1, t_2 meets p_2, t_3 meets p_3, and so on. Figure-39 illustrates a high level concept of the scheme. In the figure, WS(4) is the window size 4 working-set based model of history sensitive block as we used in the scheme for the k-differences problem in previous chapter. After the initial trigger time of m-1 (= 3 in the Figure-39), "t_1,t_2,t_3,t_4" comes out from the WS(4) the first time, "t_2,t_3,t_4,t_5" comes out next, "t_3,t_4,t_5,t_6" next, and so on. The special dummy character "ε" is used to handle the initial cases and shown in the WS(4) block as initial tokens in the figure. By the nature of dataflow environment, instruction level parallelism is gained and the time complexity for evaluating the table D' is reduced by a factor of m.

The scheme illustrated in Figure-39 is a pipelined execution scheme. The operation block "Eq" is same as one used in the scheme for the k-differences problem in the previous chapter. It compares two characters (t_i and p_i) and if they are same, returns 0 otherwise it returns 1.

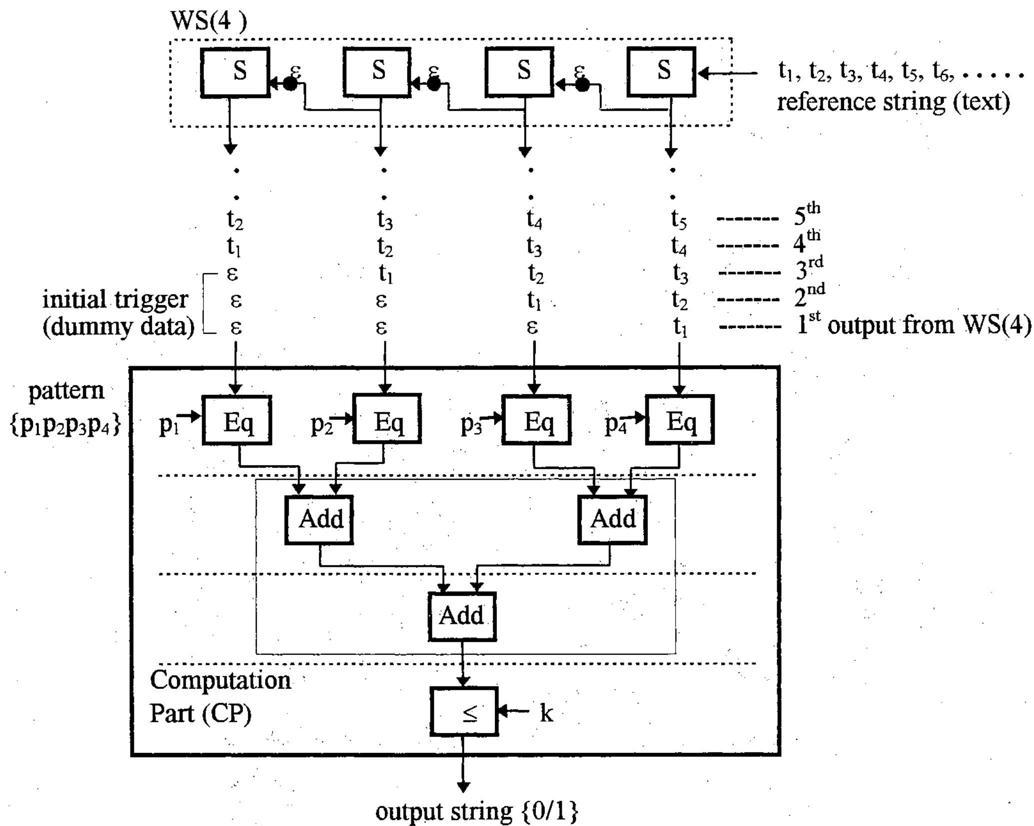


Figure-39. Dataflow scheme for the k-mismatches problem (pattern length m=4 case)

The operation block "Add" performs the addition operation upon receiving two operands and returns the sum. The final block "≤" checks whether the number of mismatches are less than or equal to k and if so, it returns 1 otherwise returns 0 as the final output of the scheme. All blocks are assumed to have constant execution time and all operations in a level (separated by dotted lines in the figure) execute simultaneously. With length m pattern, levels needed for "Add" blocks are $\lceil \log m \rceil$. Thus $\lceil \log m \rceil$ - 1 time slices after the first (upper most) level "Add" operations were activated, all levels of "Add" blocks are activated in a pipelined manner. We can replace these "Add" blocks with one global block with constant execution time and thus achieving time complexity O(n). But, with

considerably long pattern string if the initial trigger time of $\lceil \log m \rceil$ can not be bounded by a constant, then the time complexity of this hierarchical scheme is O(n + log m).

As same as the scheme for the k-differences problem, output of the scheme is a length n boolean string. The scheme receives the reference string one character at a time and returns the output stream which consists of 0's and 1's only. If a substring of length m ending at position t_j matches the pattern with at most k substitutions, the jth output is 1 otherwise it is 0. With the Example-2 in which pattern(P) = "cacd", text(T) = "bcbacddc", and k=2 output string generated is {00000100} because only $D'_{4,6}$ meets the condition; i.e. substring of length m=4 whose ending position is 6 ("bacd") can be converted to pattern with edit operations less than or equal to k=2. As illustrated in Figure-39, each "Eq" operation receives m-1 dummy data initially and outputs (ending positions) having value 1 come out after m-1 dummy outputs (0's). The pipelined execution is discussed in more detail in the performance section (10.2.2.4).

10.2.2 Linear Scheme: Linear representation of the computation part

So far, we represented a scheme which evaluate m entries in each diagonal hierarchically. There is the alternative way of evaluating entries in each diagonal linearly. Since this scheme uses linear evaluation, the initial trigger time expected in the computation part is m which is greater than that ($\lceil \log m \rceil$) of the hierarchical scheme. The reason for developing the linear scheme is that it can be converted to the linear systolic array of identical cells which has advantages in hardware implementation (special purpose VLSI chip for the string matching). The design will be easily extended by connecting same type of cells linearly.

Different from the hierarchical scheme, entries in each diagonal (from left_top to right_bottom direction) of the table D' are evaluated serially. Thus it needs m time slices to evaluate one diagonal. But the computations are done in a pipelined manner and, after m-1 initial trigger time slices (pipeline depth), one output (one entry of the last low) is generated at a time. Figure-40 illustrates the timing concept of the scheme. In fact, all (m) entries in each column are computed at a time with the linear scheme. In the figure, after 3 (m-1) initial trigger time, the first column entries ($D'_{1,1}$, $D'_{2,1}$, $D'_{3,1}$, $D'_{4,1}$) are evaluated and one output is available. At the next time slice, entries in the second column are computed and the next output is available, and so on.

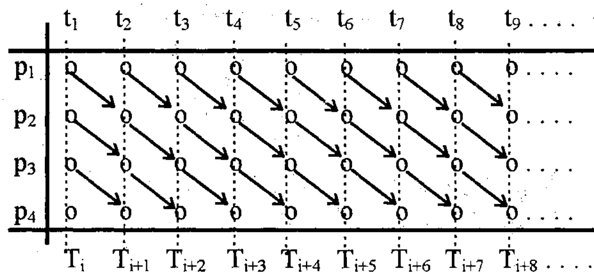


Figure-40. Evaluation timing of the linear scheme on table D'

The high level dataflow concept of the linear scheme for the k-mismatches problem is illustrated in Figure-41. PEs are identical and play the same role. In the figure "GB"

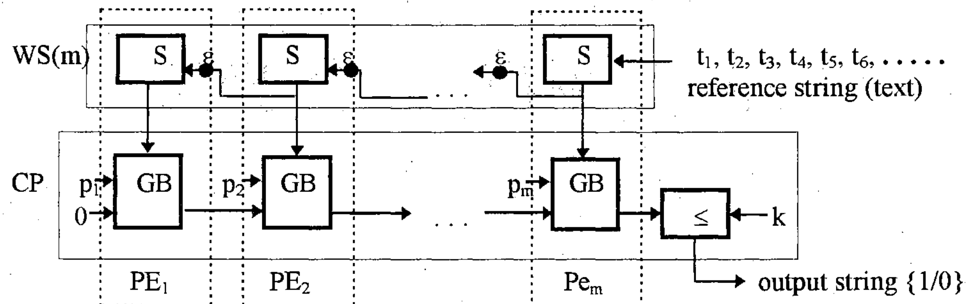


Figure-41. The linear scheme for the k-mismatches problem

represents the global block which consists of some refined operation blocks. Same as the hierarchical scheme, the linear scheme uses WS block. Entries of one column of the edit distance table D' are evaluated at a time. Computations are done in a pipelined manner. Behavior of each global block "GB" in the computation part (CP) in Figure-41 is illustrated in Figure-42 and the refined computation part is shown in Figure-43. In the refined scheme in Figure-43, the left most operation block "Add" is useless and skipped for the optimal design.

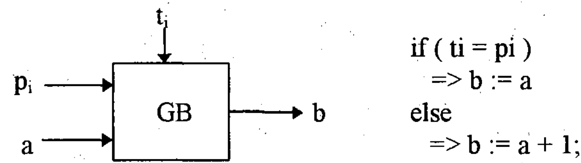


Figure-42. Behavior of each global block (GB) in the computation part

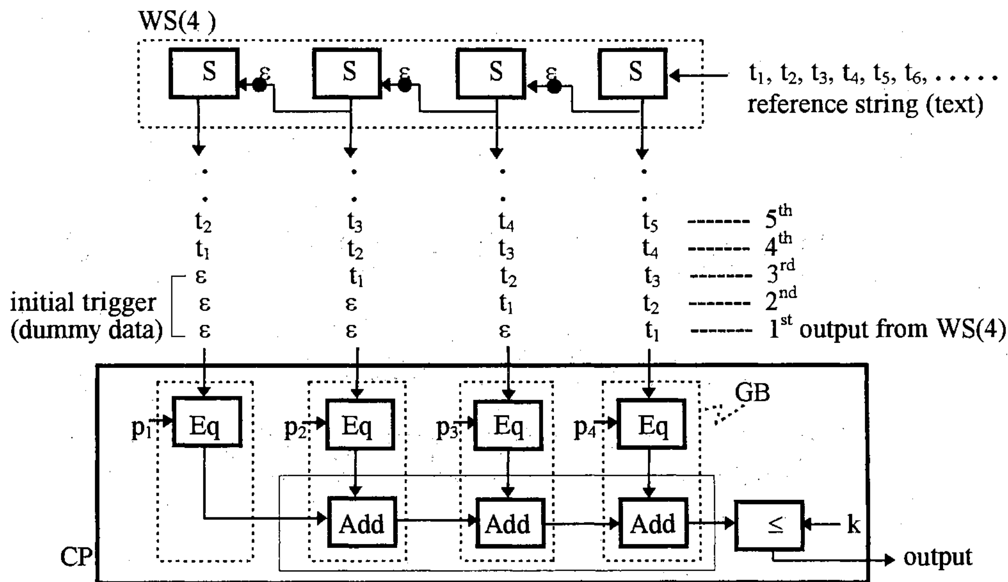


Figure-43. Refined linear scheme for the k-mismatches problem (m=4 case)

"Add" blocks are serially connected. M-2 time slices after the first (left most) block is activated, all "Add" operations are active at each time slice in a pipelined manner. The linear scheme has same output as the hierarchical scheme; i.e. after m-1 dummy outputs (0's), the first outputs (ending positions) having values 1 can come out. The initial trigger times required to generate the first output data are little different depending on the method used to represent the computation part. With both the hierarchical and the linear schemes, reference characters enter the scheme one at a time and all "Eq" blocks receive data simultaneously. From "Eq" blocks, time required to generate the first binary output is $\lceil \log m \rceil$ for the hierarchical scheme and m (m-1 with optimized design) for the linear scheme. But, once the first output comes out, both methods generate one output data at every time unit in a pipelined manner. As discussed in the hierarchical scheme, with considerably long pattern string, the initial trigger time (m - 1) can not be ignored and the time complexity of the linear scheme is O(n + m).

One obstacle of the linear scheme is that the synchronization problem. As seen in Figure-43, reference characters from the WS(m) block (say t_j .. t_{j+m-1}) enter "Eq" blocks at same time and matching information from "Eq" blocks (say $x_{1,j}$ .. $x_{m,j+m-1}$) are produced at same time. Since the adder blocks are connected linearly, $x_{3,j+2}$ should wait on the 3rd adder block one time slice, $x_{4,j+3}$ should wait on 4th adder block 2 time slices, and so on. These data dependencies bring about multiple tokens stacked on an arc. In dynamic dataflow environment, this does not cause any problem but, in static dataflow environment synchronization should be managed so that only one token can reside on an arc at a time. For the hardware implementation, schemes are described in the static dataflow

environment and thus the static synchronization should be provided. In the static dataflow environment, the synchronization actor "S" which is defined in Chapter IV is used to handle the problem. In the hardware design, latches are used to hold the data to enforce synchronization in the place of "S" actors in the dataflow scheme. Thus the computation part depicted in Figure-43 should be changed by using "S" actors as illustrated in Figure-44. Thus in the PE_k (k^{th} from left), we should assign k-2 "S" actors (latches in hardware) between "Eq" and "And" blocks.

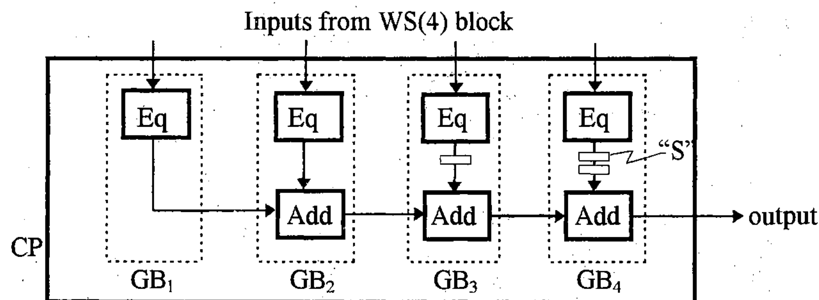


Figure-44. Linear computation part for the static dataflow environment (m=4 case)

For the hardware implementation, since each PE of the design has different number of latches, we need many different types of basic cells with the linear scheme. If the pattern size m is very small, this matter can be ignored. Otherwise it can cause the cost problem. Thus more efficient dataflow scheme which does not have such synchronization problem should be designed. In the next section (10.2.3) the broadcasting scheme which does not cause such synchronization problem is represented. Only one basic cell type is needed to design the entire string matcher with the broadcasting scheme.

10.2.3 Broadcasting scheme

Similar to the linear scheme, the broadcasting scheme evaluates all entries (m) in each column at once. Figure-40 in the previous section also illustrates the timing concept of the broadcasting scheme. The difference is the initial trigger time. In the broadcasting scheme, initial trigger time is not needed. Thus the first column entries ($D'_{1,1}$, $D'_{2,1}$, $D'_{3,1}$, $D'_{4,1}$) are computed at the first time slice and the 1st output is available at time T_1. At the 2nd time slice (T_2), entries in the second column are computed and the 2nd output is available, and so on.

In order to evaluate each column of the table D' simultaneously without the initial trigger time, we use the broadcasting method with which each reference character tj ($1 \leq j \leq n$) is compared with all pattern characters (p_1 .. p_m) at the same time. Thus instead of using the working-set based history sensitive block WS(m) which we used in schemes so far, we broadcast each input reference character to all processing elements (PEs). Figure-45 illustrates the high level concept and dataflow used in the broadcasting scheme.

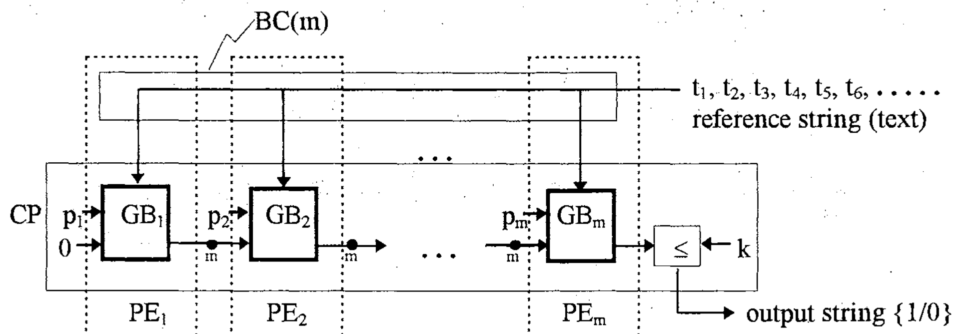


Figure-45. Concept and dataflow of the broadcasting scheme

In Figure-45, BC(m) stands for the size m (m-way) broadcasting and GB stands for global operation block. This dataflow scheme can be implemented with linear systolic array of simple identical cells. In the figure, processing elements (PEs) are identical and play the same roles. GBs are also identical components.

Our intention with the design in Figure-45 is that, for instance with pattern size m=4, GB_1 computes $D'_{1,1}$ while GB_2 computes $D'_{2,1}$, GB_3 computes $D'_{3,1}$, and GB_4 computes $D'_{4,1}$. Thus one column of the table D' can be evaluated at the same time and we can achieve O(n) worst case time by using m processing elements. Each computation of $D'_{i,j}$ needs previous entry $D'_{i-1,j-1}$ (diagonal entry) in addition to current inputs p_i and t_j. Thus GBs are connected linearly (from left to right) to reserve the history of the diagonal (from left upper to right down in table D') entries. All GBs are active at each time slice with new inputs (reference character and diagonal entry). They work in pipelined manner. Output of the scheme is also a length n boolean string. If a substring of length m ending at position t_j matches the pattern with edit distance at most k, the jth output is 1 otherwise it is 0.

Same as the linear scheme, required histories (information from diagonal entries) are kept in dataflow graph itself during run time without using any memory space and preprocessing times. Each GB_i evaluates table entries $D_{i,j}$ for all j ($1 \leq j \leq n$). Refined illustration of the broadcasting scheme is depicted in Figure-46. In the refined scheme in Figure-46, the left most (PE_1's) operation node "+" can be omitted for the optimal design. But it should be kept in VLSI implementation because one type of basic cell (PE) is used to build linear systolic array of cells and it will reduce the cost of implementation.

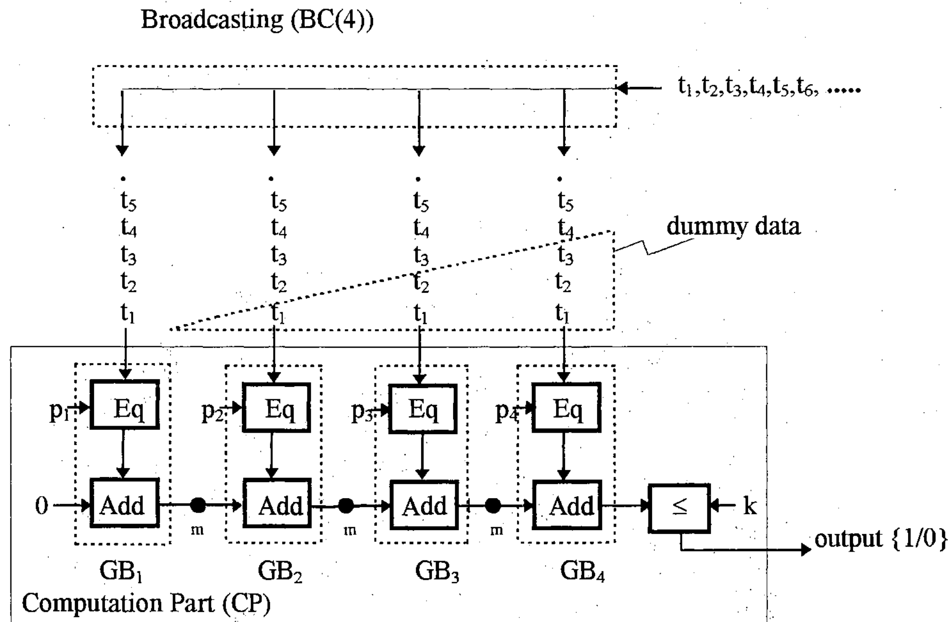


Figure-46. The broadcasting scheme (m=4 case)

Initial tokens of values m are assigned on each connection between adjacent GBs to reserve the values of the 0th column in edit distance table D'. In fact, we can assign all "0" initial tokens instead of "m" because the first m-1 outputs are dummy and we do not care what their values are. The operation block "Eq" is same as one we used in previous schemes. It compares two operands (p_i and t_j) and if they are same, it returns 0 otherwise it returns 1. This data is then added to data passed from diagonal entry ($D'_{i-1,j-1}$) to generate $D'_{i,j}$; i.e. for example with i^{th} GB, operation "+" receives one operand from "Eq" and the other operand from operation "+" in GB_{i-1} which is generated one time unit before. In fact, GB_m generates $D'_{m,j}$ at virtual time T_j by using $D'_{m-1,j-1}$ which was generated at time T_{j-1} by using $D'_{m-2,j-2}$ which was generated at time T_{j-2} by using $D'_{m-3,j-3}$ and so on. In this manner values of jth diagonal entries are accumulated on $D'_{m,j}$ which is the jth output in the design (broadcasting scheme). In pipelined manner, all components

are active at each time slice and this scheme generates one output data at each time unit. Since this scheme does not suffer from the initial trigger time, the worst case time bound is O(n). Thus it is the most efficient scheme among three schemes.

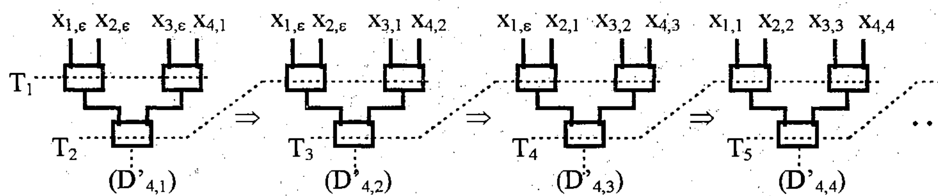
For the hardware implementation, the structures of PEs are same and one basic cell can be used for hardware extension for making linear systolic array.

10.2.4 Performance

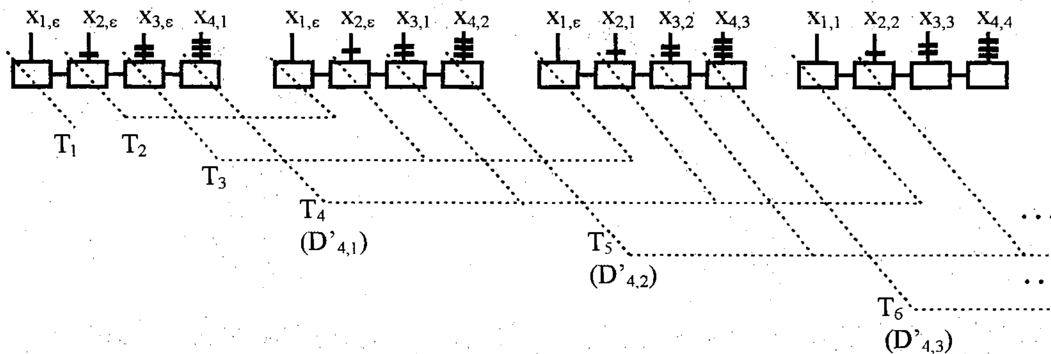
Timing concepts of three schemes were described in corresponding sections with figures. Figure-38 in section 10.2.1 illustrates virtual timing of the hierarchical scheme on the edit distance table D'. Figure-40 in section 10.2.2 illustrates entry evaluation timing on the table D' for both the linear and the broadcasting schemes. In this section, pipelined operation in each scheme is illustrated with the initial trigger time analysis. Illustrations use the pattern length as 4 an example. Figure-47 shows the pipelining in three schemes. Analysis are done on adder blocks since their performances are dependent on the pattern length m. Each input $x_{i,j}$ represents datum from "Eq" block; i.e. result of matching p_i and t_j. Since the hierarchical and the linear scheme use the WS(m) blocks with initial dummy tokens "ε", illustrations in Figure-47(a),(b) use input $x_{i,\varepsilon}$. Four instances (snap shots) in each scheme are depicted. Each instance has different input data. Figures are simplified and each empty block in figures represents an adder block. In the Figure-47(b), black rectangles between input data and add blocks represent the "S" actors used for synchronization in static dataflow environment.

As illustrated in Figure-47(a), in the hierarchical scheme, after $\lceil \log m \rceil - 1$ initial trigger time slices, one output ($D'_{m,j}$; $1 \leq j \leq n$) is produced during every time period. Thus the

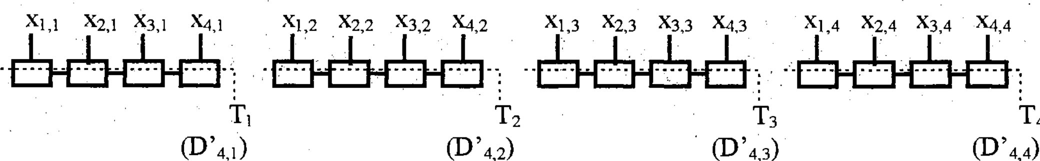
time complexity of the hierarchical scheme is O(n + log m). The last entry in each diagonal (from left_top to right_bottom direction) of the edit distance table D' for the k-mismatches problem is evaluated by adding all entries in that diagonal hierarchically. Pipelined computations among diagonals make the scheme generate one output at each time slice after the initial triggers (pipeline depth). In fact, the hierarchical scheme computes only the last row of the table D' which contains the solution to the problem.



(a). Hierarchical scheme (m=4 case)



(b). Linear scheme (m=4 case)



(c). Broadcasting scheme (m=4 case)

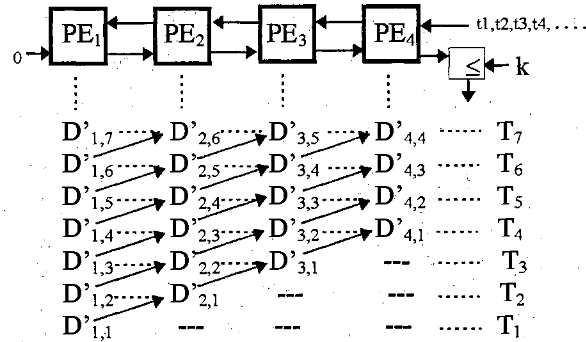
Figure-47. Computation time analysis of dataflow schemes for the k-mismatches problem

In the linear scheme illustrated in Figure-47(b), after m - 1 initial trigger time slices, one output ($D'_{m,j}$; $1 \le j \le n$) is produced during every time slice. Thus the time complexity of the linear scheme is O(n + m). The last entry in each diagonal (from left_top to right_bottom direction) of the edit distance table D' is available one at each time slice after the initial trigger time (pipeline depth). Computations among diagonals are done in a pipelined manner. In fact, all entries in each column of the table D' are evaluated at a time slice after the initial triggers (m -1).

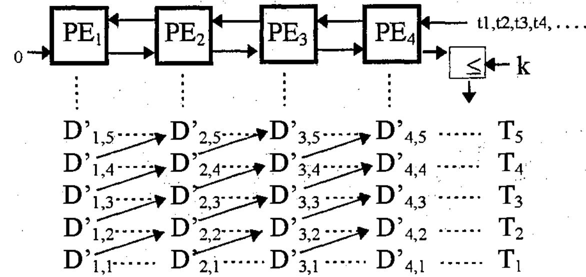
On the other hand, the broadcasting scheme shown in Figure-47(c) does not need the initial trigger time since it uses broadcasted input and initial tokens between adder blocks. Thus the time complexity of the broadcasting scheme is O(n). Among those three dataflow schemes, this scheme has the best time complexity. By the way, this does not guarantee the best time in the parallel designs which will be described in the next section. Same as the linear scheme, the last entry in each diagonal (from left_top to right_bottom direction) of the edit distance table D' is available one at each time slice but, there is no initial trigger time. Also in this scheme, computations among diagonals are done in a pipelined manner and, in fact all (m) entries in each column of the table D' are evaluated at a time slice. Compared to the linear scheme which also has a linear structure, the broadcasting scheme does not need the initial trigger time and provides better implementation scheme which is explained in sections 10.2.2 and 10.2.3.

Figure-48 illustrates the evaluations of table D' by the linear scheme (a) and the broadcasting scheme (b). The pattern size m=4 is used for the simplicity. Entries of the table evaluated in each time slice are depicted for both schemes. In the figure, arrows imply data movements of the pipelined operations. With the linear scheme, the first

useful solution $D'_{m,m}$ is generated at time $T_{m+(m-1)}$ and the final solution $D'_{m,n}$ is generated at time $T_{n+(m-1)}$. On the other hand, with the broadcasting scheme the first useful solution $D'_{m,m}$ is generated at time T_m and the final solution $D'_{m,n}$ is generated at time T_n.



(a). The linear scheme



(b). The broadcasting scheme

Figure-48. Evaluation of table D' by the linear and the broadcasting schemes (m=4 case)

10.3 Parallelization with Multiple Streams: Explicit Parallelism

So far we described serial schemes for the k-mismatches problem which are basic designs for building parallel schemes. In this section, mechanism for exploiting maximum parallelism on the hierarchical, the linear, and the broadcasting schemes are represented in the static dataflow environment. Instead of using single stream input and output, multiple stream input and output are used to exploit explicit parallelism. We can exploit any degree of parallelism by using the forwarding mechanism developed for the WS and

the BC blocks. In the illustrations of the serial schemes, two parts (CP and WS (or BC)) have been separately depicted because the separate concepts are needed to describe parallelization schemes. Since the hierarchical and the linear schemes use the WS block, parallelized WS block named PWS block is used for those schemes. The PWS block, which is the parallelized memoryless scheme for working-set based history sensitive computations, is defined in Chaper VI. On the other hand, since the broadcasting scheme uses the BC block, which is a variation of the WS block, parallelized BC block named PBC is used for this scheme. The PBC block is described in this section. The contents of the computation parts remain the same and multiple computation parts are connected to the parallelized WS block (PWS) or parallelized BC block (PBC). For the sake of convenience, the names HCP, LCP, and BCP are used to represent the computation parts of the hierarchical scheme, the linear scheme, and the broadcasting scheme respectively.

Forwarding mechanisms PWS and PBC make the actions of the WS and the BC blocks parallel; i.e. they receive multiple input streams instead of single input stream and forward them to the multiple computation parts (HCPs, LCPs, and BCPs) so that multiple computation parts can generate multiple output streams in parallel. From the point of view of the special purpose hardware string matcher based on these parallel schemes, the string matching component receives multiple input streams from the host computer and returns multiple output streams to the host. Multiprocessor or pipelined host computer can manipulate these multiple streams. Since the hierarchical and the linear schemes use the PWS block, their parallel schemes are described together in section 10.3.1. The parallel broadcasting scheme is described separately with its forwarding mechanism PBC.

10.3.1 Parallel Approaches for Hierarchical and Linear Schemes

By using the PWS block, very high level conceptual view of the parallel hierarchical and the parallel linear schemes are illustrated in Figure-49. PWS(d*dm) block which is degree d parallel WS(m) block is defined in Chapter VI. In the figure, CPs can be HCPs for the hierarchical scheme and LCPs for the linear scheme. Each HCP (or LCP) is exactly same as the computation part (CP) which is described in the serial hierarchical (or serial linear) scheme. Each input token t_j is an element of the reference string and, all CPs used are identical components.

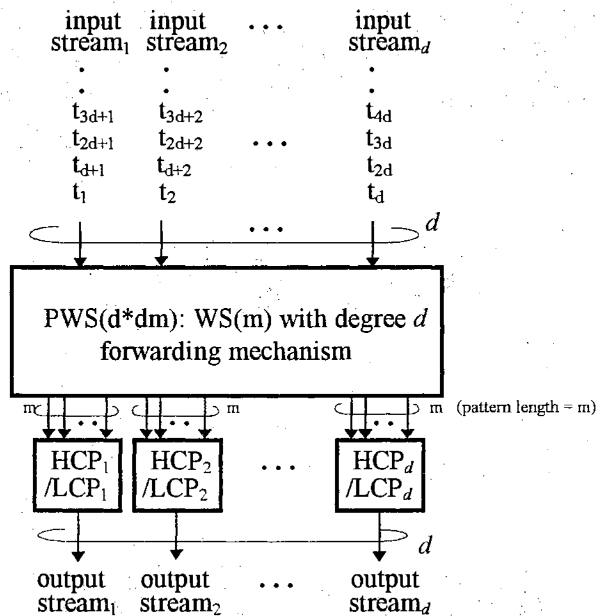


Figure-49. Concept of the parallel hierarchical / linear schemes

As described in Chapter VI, for the parallelism degree d, the parallel scheme needs d distinct input streams and d identical computation parts. Parallelism degree d is independent of the length of the pattern (m). For the k-mismatches problem with any pattern length, we can design any degree parallel hierarchical (and linear) scheme which produces 1/d total execution time for any degree. Algorithm-1 in Chapter VI is used to

generate the forwarding mechanism (PWS block) at compile time in the static dataflow environment. The working-set window size (m) in the algorithm is applied to the pattern length (m) in the parallel schemes (hierarchical and linear) for the k-mismatches problem. By using the PWS block and computation parts used in the serial hierarchical (HCP) and the serial linear schemes (LCP), any degree (arbitrary d) of parallel schemes can be represented for any length (arbitrary m) pattern. Figure-50 shows two examples of the parallel scheme.

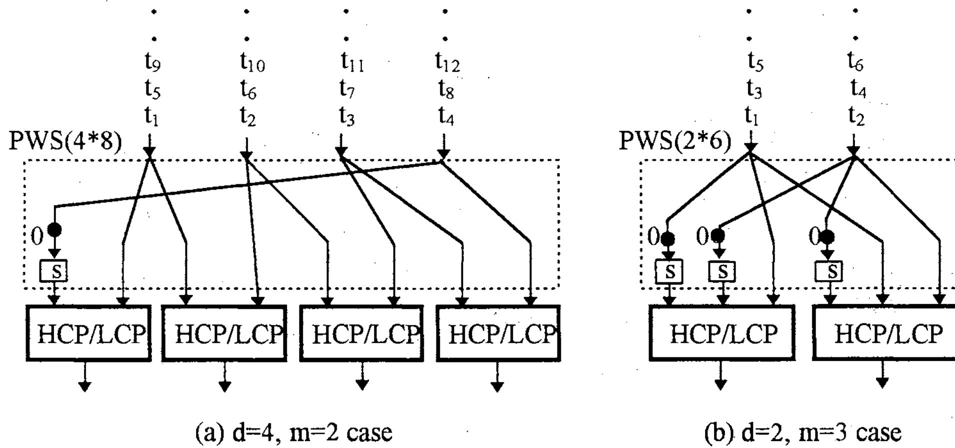


Figure-50. Examples of parallel schemes (hierarchical / linear)

Figure-51 shows the parallel dataflow scheme for the k-mismatches string matching problem with parallelism degree d=3 and the pattern length m=4 case.

With the parallel schemes (hierarchical and linear) for the k-mismatches string matching problem, the total processing times are reduced by factors of d which is the controllable degree of the parallelism. The worst case time complexities become $O((n/d) + \alpha)$, where α is the initial trigger time bound in the computation part; i.e. $\alpha = \log m$ for the parallel hierarchical scheme and, $\alpha = m$ for the parallel linear scheme. Since each computation

part has identical initial trigger time, the initial trigger times for these two parallel schemes are same as those in the serial schemes. They can not be reduced by a factor of the parallelism degree d.

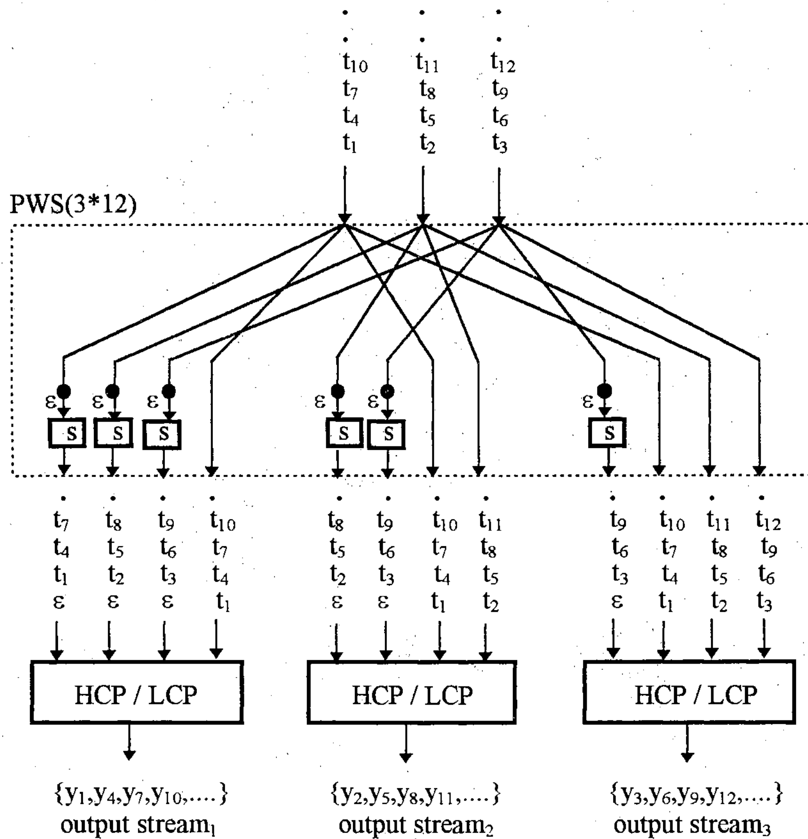


Figure-51. Parallel string matching for the k-mismatches problem (d=3, pattern size m=4 case). HCP is used for the hierarchical scheme and, LCP is used for the linear scheme

Output manipulation of the parallel scheme:

Outputs from the parallel schemes are different from those of the serial schemes since multiple computation parts work in parallel and generate d different output streams simultaneously. So far, outputs from schemes are assumed the ending positions of the substrings of the reference string. For example with an output $y_j = 1$, it means that a substring which ends at position j of the reference string matches the pattern (with error

bound k). In parallel schemes, outputs (ending positions) from CP_i (HCP_i for the hierarchical and LCP_i for the linear scheme) have corresponding orders on the elements of the input $stream_i$. In Figure-51 which illustrates the d=3 and m=4 case, output $stream_1$ from the CP_1 is $\{y_1,y_4,y_7,y_{10},\ldots\}$ which has same order of the input $stream_i$ which is $\{t_1,t_4,t_7,t_{10},\ldots\}$. In general with the parallelism degree d, output $stream_i$ from CP_i is $\{y_i,y_{i+d},y_{i+2d},y_{i+3d},\ldots\}$ which has same order of the input $stream_i$ which is $\{t_i,t_{i+d},t_{i+2d},t_{i+3d},\ldots\}$.

With parallel schemes using the PWS block (hierarchical and linear schemes) for the k-mismatches problem (and the exact matching problem), output manipulation problem occurs when the consideration is the starting positions of the substrings instead of the ending positions; i.e. an output y_j means that a substring which starts at position j of the reference string matches the pattern. When starting positions are considered for the output, following algorithm is used to manipulate outputs from the parallel scheme; i.e. it tells what ordered outputs are generated from each computation part.

Algorithm-3: Output (starting positions) manipulation of the parallel hierarchical and the parallel linear schemes

Let m be the pattern length and, d be the parallelism degree.

i := ((m-1) MOD d) + 1;

for j := 1 to d

{ q:= ((m-1) DIV d);

if ((m-1) MOD d) ≥ i then q := q + 1; /* # of dummy outputs */

CP_i has corresponding outputs of the jth (from left) input stream to the PWS block

```
        after q dummy outputs;
    i := i + 1;
    if  (i > d) then  i :=  i MOD d;
}
```

When considering the starting positions of the substrings as outputs from the parallel schemes (the hierarchical and the linear) we can manage the outputs from each CP in the parallel scheme by using Algorithm-3. For example with d=3 and m=4, which Figure-51 illustrates, corresponding outputs of the 1st (from left) input stream $\{t_1,t_4, t_7, t_{10},....\}$ to the PWS block is generated by CP_1 after 1 dummy output; i.e. $\{0,y_1,y_4,y_7,y_{10},.....\}$. The first one (0) is dummy output and if each y_j is 1, it means that the pattern matches at the starting position j of the reference string. Otherwise (y_j is 0), mismatch occurs at that starting position (j). The corresponding output stream of the 2nd input stream $\{t_2,t_5,t_8,t_{11},....\}$ to the PWS block is generated by CP_2 after 1 dummy output $\{0,y_2,y_5,y_8,y_{11},....\}$, and so on. For another example with d=2 and m=4, CP_1 generates output $\{0,0,y_2,y_4,y_6,...\}$ and CP_2 generates output $\{0,y_1,y_3,y_5,...\}$. In this example, CP_1 produces output stream which has same order of the input $stream_2$ after 2 dummy outputs and, CP_2 produces output stream which has same order of the input $stream_1$ after 1 dummy output. Figure-52 illustrates this.

In the parallel hierarchical and the parallel linear schemes, when outputs represent the ending positions, CP_i (i^{th} from left, and $1 \leq i \leq d$) produces outputs having corresponding orders of the input $stream_i$. When outputs represent the starting positions, each CP produces outputs according to the Algorithm-3. The methods of considering outputs

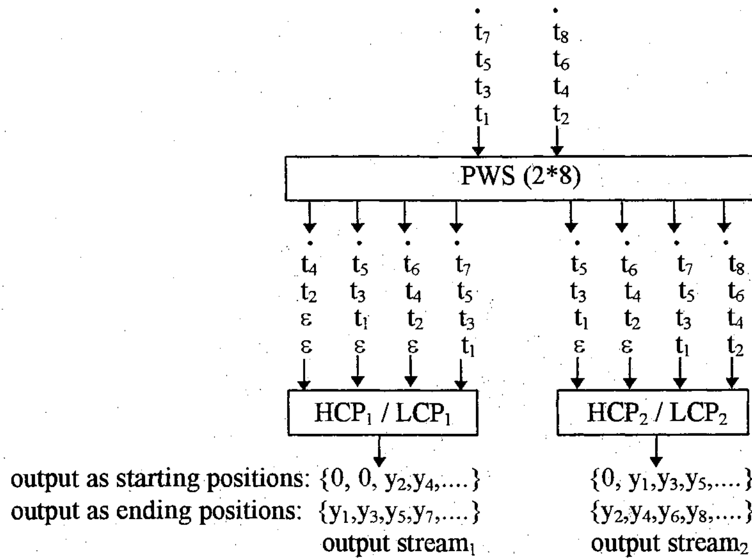


Figure-52. Outputs from the parallel hierarchical / linear schemes

(i.e. ending positions or starting positions) do not affect the number of outputs generated and thus the time complexities of the parallel schemes are not affected by those methods. The initial trigger times in the computation parts ($\lceil \log m \rceil -1$ for the hierarchical and, $m -1$ for the linear) are also not affected.

10.3.2 Parallel Broadcasting Scheme

Since the broadcasting scheme uses the BC block which is a variation of the WS block, parallel broadcasting scheme can not use the PWS block which are used for the paralell hierarchical and the parallel linear schemes. In this part, the parallel broadcasting scheme which uses parallelized broadcasting block is presented. Description of how to build the parallel broadcasting mechanism is provided. We name this PBC(d*dm) block. PBC stands for "parallel broadcasting" and d is controllable parallelism degree as used in PWS block. In fact in the serial broadcasting scheme representation in section 10.2.3, the

broadcasting block BC(m) is (1*m) parallel broadcasting mechanism (PBC(1*m)) in which the parallelism degree (d) is 1 and broadcasting size is m.

Very high level conceptual view of the parallel broadcasting scheme is depicted in Figure-53. In the figure, each BCP (broadcasting computation part) is exactly same as the CP described in serial broadcasting scheme. All BCPs used are identical components. Arrangement of the multiple input streams is same as that in the PWS block. For parallelism degree d, we need d distinct input streams and d identical computation parts (BCPs). Parallelism degree d is independent of the size of the broadcasting block (BC) m (i.e. pattern length).

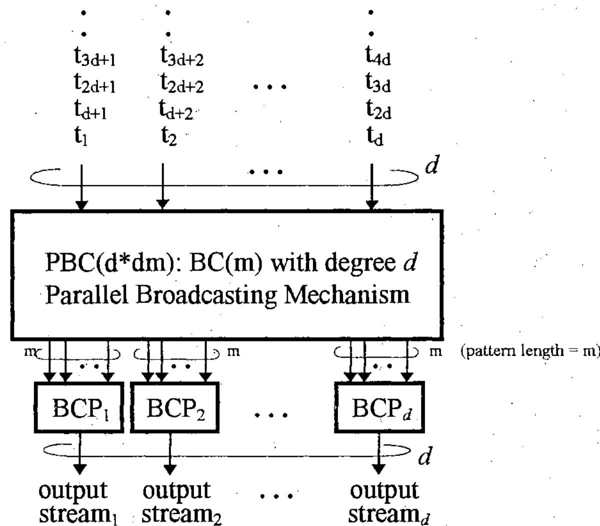
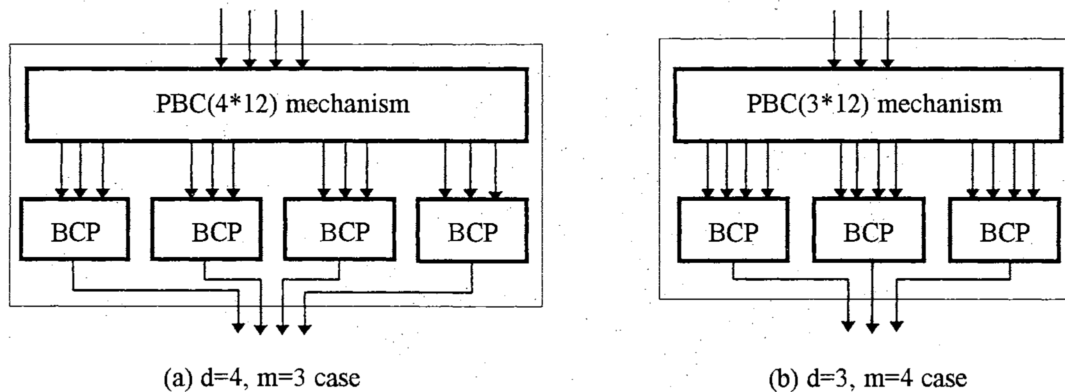


Figure-53. Concept of the parallel broadcasting scheme and input arrangement

Output from parallel broadcasting scheme consists of d distinct streams since d computation parts (BCPs) work in parallel. Relationships and contents of output streams are described in detail later. Figure-54 shows some examples of the scheme in conceptual view. The parallel broadcasting mechanism provides explicit parallelism

which speeds up the execution of the serial broadcasting scheme for the k-mismatches problem by a factor of d (number of streams used).



(a) d=4, m=3 case (b) d=3, m=4 case

Figure-54. Conceptual view of the parallel broadcasting scheme

Parallel broadcasting mechanism: PBC(d*dm)

We can generate the parallel broadcasting mechanism (PBC) with any parallelism degree d and any broadcasting size (pattern length) m. The PBC(d*dm) block provides 1/d total execution time of the BC(m) block. The following algorithm is used to generate the parallel broadcasting mechanism at compile time in the static dataflow environment. This algorithm is also used for building PBC block component in hardware implementation by substituting latches for the synchronization actors "S".

Algorithm-4: Parallel Broadcasting mechanism (PBC(d*dm))

Assumption: For degree *d* forwarding, we assume inputs are arranged in *d* different input streams;

the 1st (left most) input stream consists of (order MOD d = 1) input elements,

the 2nd input stream consists of (order MOD d = 2) input elements,

the 3rd input stream consists of (order MOD d = 3) input elements,

. . . . .

the dth input stream consists of (order MOD d = 0) input elements.

Let Stream-i represents the ith (from left) input stream.

Let BCP_i represents the ith (from left) broadcasting computation part.

(each BCP_i has *m* input ports and 1 output port).

Let *m* be the broadcasting size and *d* be the degree of the parallelism.

INITIALIZE:

```
1)   for i := 1 to d        /*number of input streams (degree of parallelism)*/
2)     {k := 1;
3)       for j := 2 to m
4)         {Assign k "S" actors (and initial tokens) on j^th input port of BCP_i;
5)          k := k + 1;
         }}
```

ASSIGN ARCs from input streams (Stream-i) to Computation Parts (BCPs):

```
6)   for i = 1 to d         /*number of input streams (degree of parallelism)*/
7)     {k := i;             /*stream_i is assigned to BCP_i initially (on the 1^st port)*/
8)       for p = 1 to m     /*number of ports on each BCP*/
9)         {q = (k + (p - 1) - i) DIV d;             ┐
10)         On the p^th input port (from left) of BCP_k, ┘   /*input alignment*/
              Deassign q "S" actors with initial tokens;
```

11) Assign an Arc from Stream-i to p^{th} input port (from left) of BCP_k;

12) k := k - 1;

13) if k = 0 then k = d;

}

}

Each BCP_i ($1 \leq i \leq d$) receives m consecutive reference characters on a diagonal (left bottom to right top direction); i.e. "t_i, t_{i+1},..., $t_{i+(m-1)}$" on the 1st diagonal, "t_{i+d}, t_{i+d+1}, ..., $t_{i+d+(m-1)}$" on the 2nd diagonal, and so on. These consecutive m reference characters are used to generate one output $D'_{m,i+(m-1)}$ (i.e. $y_{i+(m-1)}$) from BCP_i. Figure-55 illustrates the diagonal entries to each BCP.

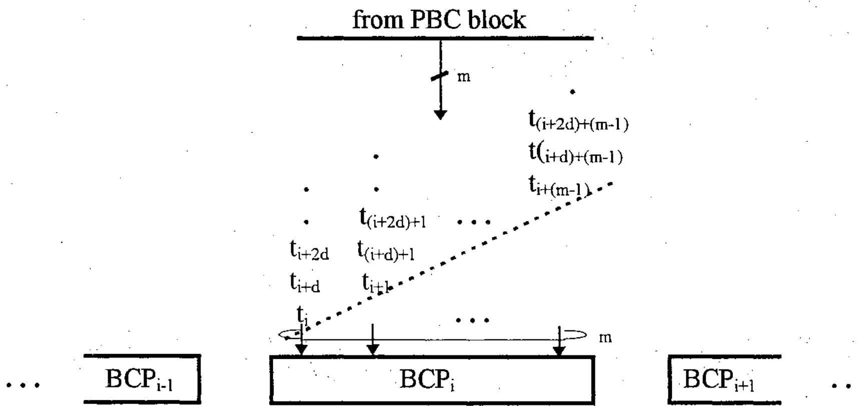


Figure-55. Diagonal entries to each computation part (BCP)

By using Algorithm-4, any degree of parallel broadcasting mechanism with any broadcasting size (pattern length) can be generated. For example, Figure-56(a) shows degree 4 parallel broadcasting mechanism for the broadcasting size m = 2 and, Figure-56(b) shows degree 2 mechanism for the broadcasting size m = 3. In the figures,

computation parts (BCPs) are identical. The broadcasting block (BC(4)) depicted in Figure-45 (and Figure-46) is a PBC with parallelism degree d=1 and m=4 (i.e. PBC(1*4)). Figure-57 shows an example in which d=3 and m=4.

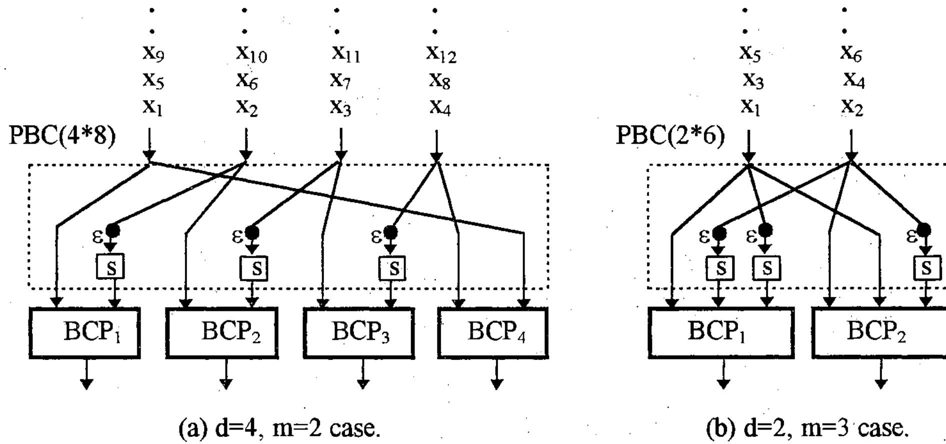


Figure-56. Examples of parallel broadcasting blocks

Input alignment mechanism (line 9 and 10) in the Algorithm-4 is explained with the illustration in Figure-57. As seen in Figure-57, for the 3rd (from left to right) computation part (BCP_3) for instance, consecutive 4 (m=4) reference characters t_3, t_4, t_5, and t_6 must be aligned on a diagonal (left-bottom to right-top) over the dummy data (dotted triangle). Since positions of the dummy data are fixed in the broadcasting scheme (i.e. right bottom (m-1)*(m-1) triangle), we assigned that many "S" actors and initial tokens (ε) onto input arcs to BCP_3 at the initialization part of the algorithm (line 1-5). Without doing the alignment (line 9 and 10), effect of this initialization after assigning the input streams to BCP_3 is illustrated in Figure-58(a). Thus the alignment is needed and the result of applying it is illustrated in Figure-58(b); i.e. "t_3,t_4,t_5,t_6" are aligned on a diagonal (from left bottom to right top direction).

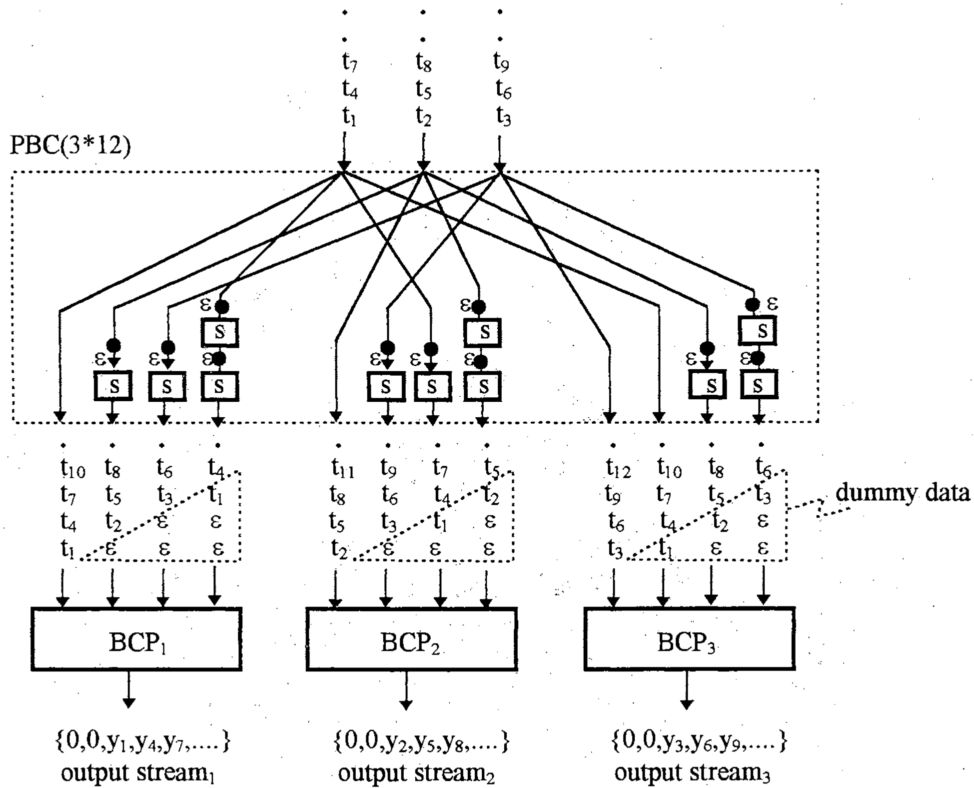
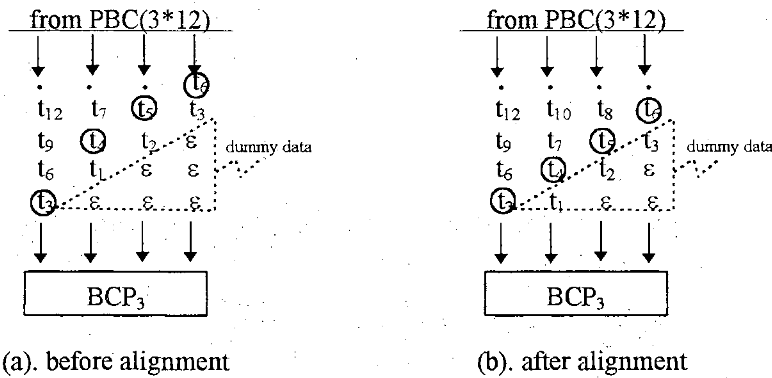


Figure-57. Parallel broadcasting scheme for k-mismatches problem (d=3, m=4 case)



(a). before alignment

(b). after alignment

Figure-58. Inputs to BCP before and after the alignment mechanism

With the parallel broadcasting scheme, the total processing time of the serial broadcasting scheme is reduced by a factor of d which is the controllable degree of the parallelism. In fact, the gain is not exactly 1/d total execution time. Since each BCP in

the parallel scheme needs aligned inputs (i.e. consecutive m input reference characters are arranged on a diagonal as illustrated in Figure-55 and Figure-57), each BCP receives dummy data (dotted triangle in Figure-57). Thus required time to generate the output y_m is same as the serial broadcasting scheme (i.e. m time slices). Parallel scheme can not reduce that time by a factor of the parallelism degree d. After m-1 time slices (when m - 1 outputs are generated), d outputs are generated from the parallel broadcasting scheme at a time. Therefore the time complexity of the parallel scheme is $O(m + ((n - m) / d))$. Time complexity is discussed more in detail in the performance part.

Output manipulation of the parallel broadcasting scheme:

Output manipulation of the parallel broadcasting scheme is different from that of the parallel hierarchical (and linear) scheme. The reason is using the PBC block instead of the PWS block used in other parallel schemes. Outputs shown in Figure-57 is based on the ending positions of substrings. The fact is opposite from other parallel schemes. When considering the starting positions as the outputs, output manipulation of the parallel broadcasting scheme is simple because it has a fixed rule. Each BCP_i generates an output stream which has the same order as the input $stream_i$ after m -1 dummy outputs (0's). Thus each output $stream_i$, which is from BCP_i, consists of first m-1 0's, y_i, y_{i+d}, y_{i+2d}, y_{i+3d}, and so on; i.e. output $stream_i = \{0^1, 0^2, \ldots, 0^{m-1}, y_i, y_{i+d}, y_{i+2d}, y_{i+3d}, \ldots\}$. If y_j $(1 \le j \le n\text{-}m)$ is 1, it means that length m substring, whose starting position is j, matches the pattern of length m with error bound k (less than or equal to k substitutions). Otherwise ($y_j = 0$), it implies that a match did not occur at starting position j on the reference string. For example with d=3 and m=4 which Figure-57 illustrates, BCP_1 produces

$\{0,0,0,y_1,y_4,y_7,y_{10},\ldots\}$, BCP_2 produces $\{0,0,0, y_2,y_5,y_8,y_{11},\ldots\}$, and BCP_3 produces $\{0,0,0,y_3,y_6,y_9,y_{12},\ldots\}$.

With parallel scheme using the PBC block (parallel broadcasting scheme) for the k-mismatches problem (and the exact matching problem), output manipulation problem occurs when the consideration is the ending positions of the substrings instead of the starting positions. This is different from other parallel schemes (i.e. hierarchical and linear). When ending positions are considered for the output, following algorithm is used to manipulate outputs from the parallel broadcasting scheme; i.e. it tells what ordered outputs are generated from each computation part.

Algorithm-5: Output (ending positions) manipulation of the parallel broadcasting scheme

Let m be the pattern length and, d be the parallelism degree.

```
1)  k = 1;
2)  for i = 1 to (m - 1)
3)    { k := k - 1;                 /*finds a BCP which generates output corresponding to
4)      if k = 0 then k := d;          the 1st input stream*/
    }
5)  for i := 1 to d                 /*number of input streams*/
6)    {q := m -1;                   /*initially the last port of each BCP has m-1 dummies*/
7)      r := (k + (m - 1) - i) DIV d; /*number of deassigned dummis*/
8)      q := q - r;                 /*number of dummy outputs*/
9)      BCPk has corresponding outputs of the ith (from left) input stream
          after q dummy outputs;
```

10) $k := k + 1;$

if $(k > d)$ then $k := k \text{ MOD } d;$

}

Using Algorithm-5, outputs from the parallel broadcasting scheme can be managed when the consideration is ending positions of the substrings. For example with d=3 and m=4, which Figure-57 illustrates, corresponding outputs (ending positions) of the 1st (from left) input stream $\{t_1, t_4, t_7, t_{10}, \ldots\}$ to the PBC block is generated by BCP_1 after 2 dummy outputs; i.e. $\{0,0,y_1,y_4,y_7,y_{10},\ldots\}$. The corresponding output stream of the 2nd input stream $\{t_2,t_5,t_8,t_{11},\ldots\}$ to the PBC block is generated by CP_2 after 2 dummy output $\{0,0,y_2,y_5,y_8,y_{11},\ldots\}$, and so on. If each y_j is 1, it means that the pattern matches at the ending position j of the reference string. Otherwise (y_j is 0), mismatch occurs at that ending position (j). For another example with d=2 and m=4, BCP_1 generates output $\{0,0,y_2,y_4,y_6,\ldots\}$ and BCP_2 generates output $\{0,y_1,y_3,y_5,y_7,\ldots\}$. In this example, BCP_1 produces output stream which has same order of the input $stream_2$ after 2 dummy outputs and, CP_2 produces output stream which has same order of the input $stream_1$ after 1 dummy output. Figure-59 illustrates this.

In the parallel broadcasting scheme, when outputs represent the starting positions, BCP_i (i^{th} from left, and $1 \leq i \leq d$) produces outputs having corresponding orders of the input $stream_i$ after m -1 dummy outputs (0's). When outputs represent the ending positions, each BCP produces outputs according to the Algorithm-5. The methods of considering outputs (i.e. ending positions or starting positions) do not affect the number of outputs generated and thus the time complexity is not affected by those methods.

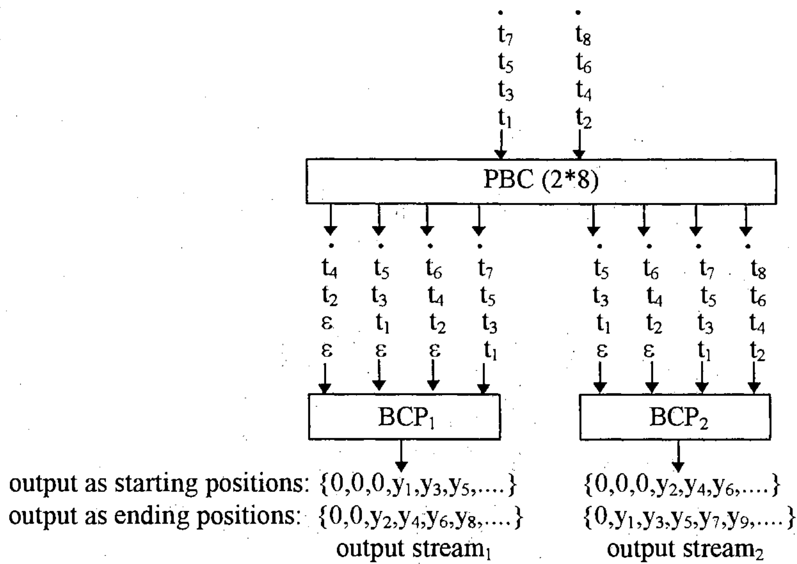


Figure-59. Outputs from the parallel broadcasting scheme

10.3.3 Performances

For the parallel hierarchical scheme, it is difficult to show all entries of the edit distance table D' (for the k-mismatches problem) evaluated in each time slice. Instead of computing all entries, the hierarchical scheme computes only the last row of the table (i.e. $D'_{m,j}$ $(1 \leq j \leq n)$) since the last row of the table D' is the solution to the problem. Figure-60 shows entries of the last row of the table D' produced from the parallel hierarchical scheme at each time slice. Example case of d=3 and m=4 which Figure-51 illustrates is used. The figure is simplified. Figure-61 shows data dependencies of the edit distance table D' and time analysis of the parallel hierarchical scheme. Available entries of the last row are marked for each time slice. Same example case of d=3 and m=4 is used for the purpose of illustration. In Figure-61, CP represents the hierarchical computation part (HCP). After $\lceil \log m \rceil$ - 1 initial trigger times (T_1 not shown in the figure), d solutions (entries of the last row) are available at a time slice.

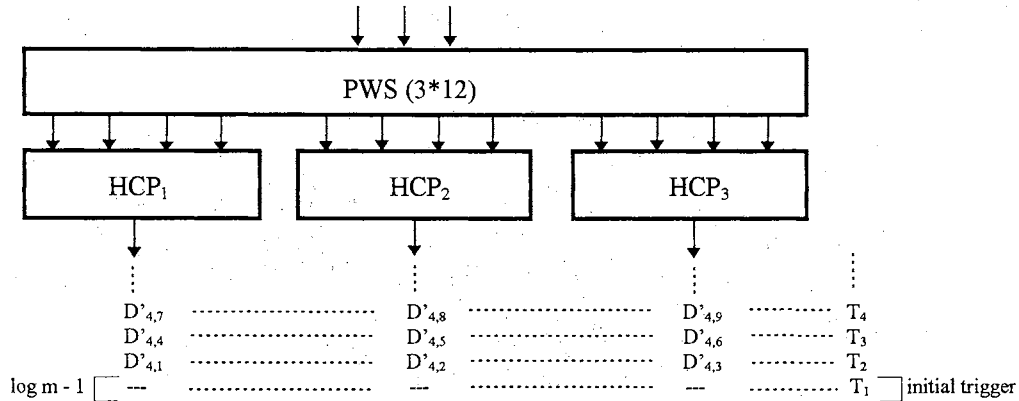


Figure-60. Evaluation of table D' by parallel hierarchical scheme (d=3, m=4 case)

Since there are n entries in the last low of the table D', time slices required to evaluate entire table is ($\lceil \log m \rceil$ - 1) + n/d. This yields the time complexity of the parallel hierarchical scheme O((n/d) + log m).

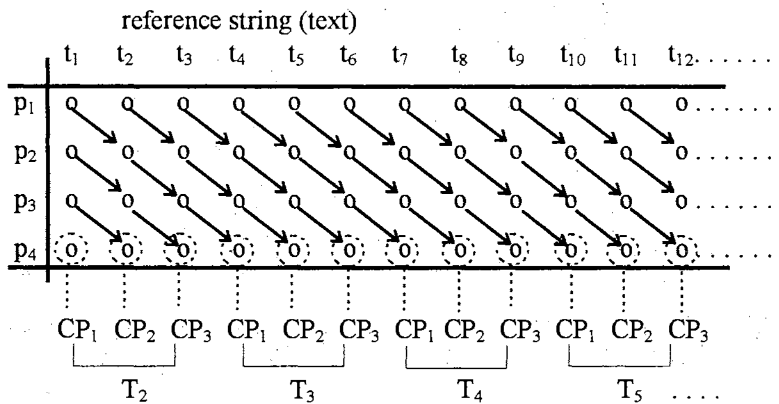


Figure-61. Entries of table D' available on time slices of the parallel hierarchical scheme (d=3, m=4 case)

In Figure-61, CP represents the hierarchical computation part (HCP). After $\lceil \log m \rceil$ - 1 initial trigger times (T_1 not shown in the figure), d solutions (entries of the last low) are available at a time slice. Since there are n entries in the last row of the table D', time

slices required to evaluate entire table is ($\lceil \log m \rceil$ - 1) + n/d. This yields the time complexity of the parallel hierarchical scheme O((n/d) + log m).

For the parallel linear scheme, entries of table D' evaluated in each time slice are illustrated in Figure-62. Corresponding reference characters processed in each GB in each time slice are also marked (diagonal dotted lines in the figure). "GB" represents the global block which are depicted in Figure-41..44 Each computation part (LCP) is depicted in simplified form. Actually each LCP is exactly same as we illustrated in Figure-44. In Figure-62, arrows imply the data movements among processing elements (GBs) for the pipelined operation. The last processing element (GB_m) in each computation part (LCP) generates the solutions to the k-mismatches problem. Each LCP has identical initial trigger time (m - 1) and the parallel scheme does not reduce this. Thus, in general, after m-1 initial trigger times, d solutions are generated in each time slice from d identical LCPs. Therefore the total execution time of evaluating the table D' is (m - 1) + n/d which yields the time complexity O((n/d) + m).

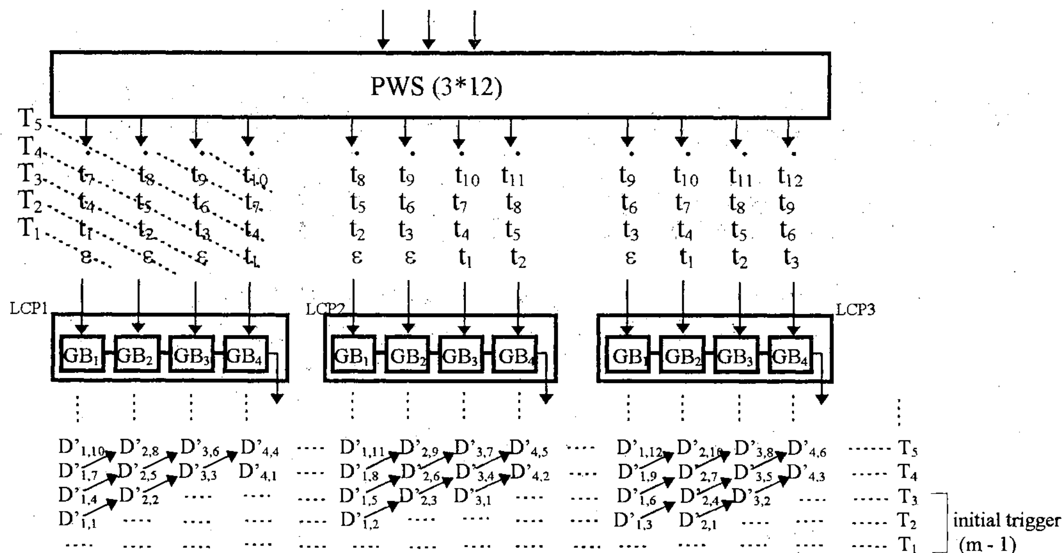


Figure-62. Evaluation of table D' by parallel linear scheme (d=3, m=4 case)

Figure-63 shows data dependencies of the edit distance table D' and time analysis of the parallel linear scheme. Same example case of d=3 and m=4 is used for the illustration.

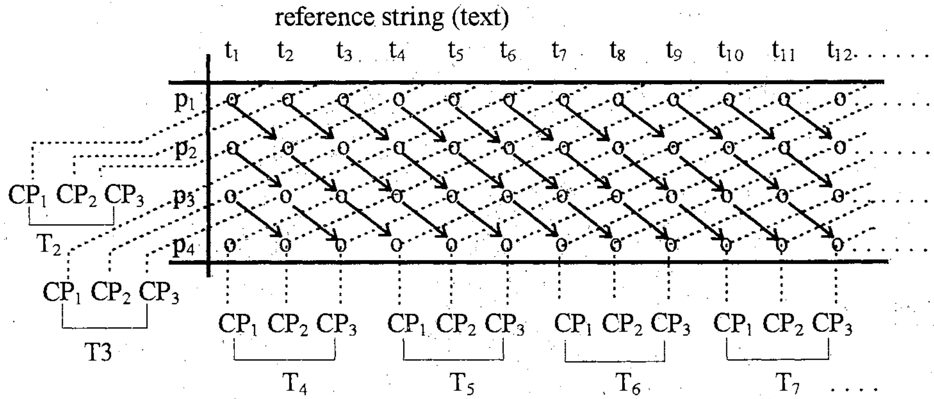


Figure-63. Entries of table D' available on time slices of the parallel linear scheme (d=3, m=4 case)

In Figure-63, arrows represent data dependencies of the table D' and dotted line represents all entries evaluated in one LCP at a time slice. "CP" stands for the LCP and T_i represents the time unit. As we see in Figure-63, one diagonal (from left-up to right-down arrows) entries can be processed in one specific LCP during m consecutive time units; i.e. for example, $D'_{1,1}$ is processed in LCP_1 at time T_2, $D'_{2,2}$ is processed in LCP_1 at time T_3, $D'_{3,3}$ is processed in LCP_1 at time T_4, and $D'_{4,4}$ is processed in LCP_1 at time T_5 and so on. All d LCPs are active at each time unit with pipelined operations and d solutions are generated at each unit of time after the initial trigger time of m - 1 slices. In figure-63, after T_3 (m-1 = 3) d LCPs generate d solutions (entries of the last low of the table D') at a time. Thus total execution time of evaluating entire table is (m - 1) + n/d time slices which yields the time complexity O((n/d) + m).

For the parallel broadcasting scheme, entries of table D' evaluated in each time slice are illustrated in Figure-64.

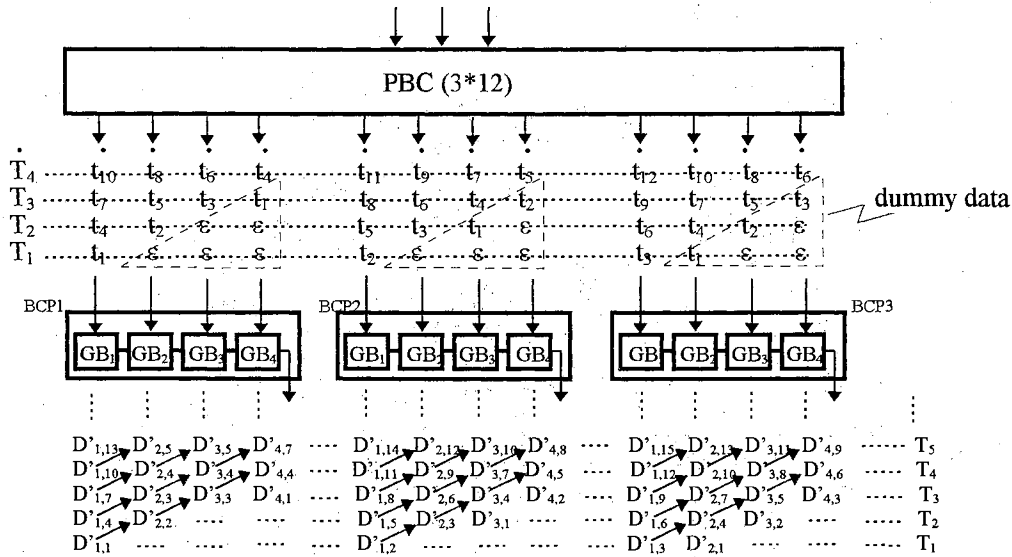


Figure-64. Evaluation of table D' by parallel broadcasting scheme (d=3, m=4 case)

Each computation part (BCP) is depicted in simplified form. In fact, each BCP is exactly same as illustrated in Figure-46. In Figure-64, arrows imply the data movements among GBs (global blocks) for the pipelined operation. The last global block (GB_m) in each computation part (BCP) generates the solutions to the k-mismatches problem.

Lemma-2. In parallel broadcasting scheme, time slices to produce the output y_m ($D'_{m,m}$) does not exceed m.

proof:

Since the parallel broadcasting mechanism (Algorithm-4) initially assigns dummy characters (ε) to each BCP (line 1..5 of the Algorithm-4), the last global block (GB_m) receives m - 1 dummy characters initially. The input alignment mechanism (line 9..10 of the Alogrithm-4) does not increase the number of dummy characters and thus the maximum number of dummy characters in the mth GB in each LCP is m - 1.

The mth solution $D'_{m,m}$ is generated from GB_m of a BCP.

Therefore the $D'_{m,m}$ is generated within m time slices. □

In general, after m-1 time slices, d solutions are generated in each time slice. Figure-65 illustrates the concept of the total execution time using serial and parallel broadcasting scheme.

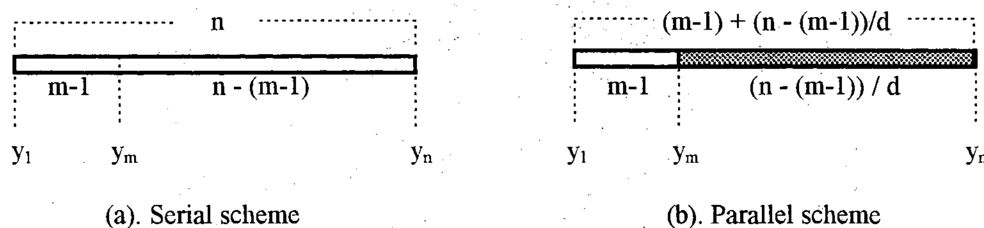


Figure-65. Total execution times on serial and parallel broadcasting schemes

Time to generate the output y_m (i.e. $D'_{m,m}$) is same as the serial broadcasting scheme. Thus, total execution time for evaluating the entire table is (m-1) + (n - (m-1)) / d time slices. This yields the time complexity of the parallel broadcasting scheme $O(((n - m)/d) + m)$. Note that the initial trigger time (which other parallel schemes have) in the computation part does not exist in both serial and parallel broadcasting schemes.

Figure-66 shows data dependencies of the edit distance table D' and time analysis of the parallel broadcasting scheme. Same example case of d=3 and m=4 is used for the illustration. In Figure-66, arrows represent data dependencies of the table D' and dotted line represents all entries evaluated in one BCP at a time slice. CP in the figure stands for the BCP and T_i represents the time unit. As shown in Figure-66, one diagonal (from left-up to right-down arrows) entries can be processed in one specific BCP during m

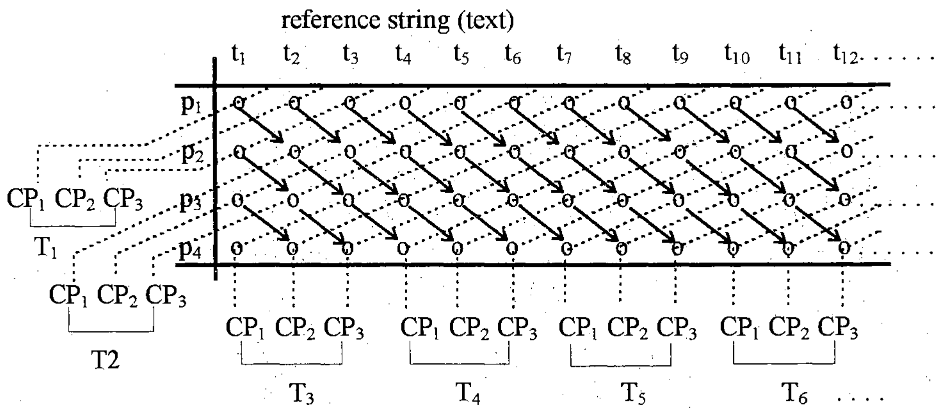


Figure-66. Entries of table D' available on time slices of the parallel broadcasting scheme (d=3, m=4 case)

consecutive time units; i.e. for example, $D'_{1,1}$ is processed in BCP_1 at time T_1, $D'_{2,2}$ is processed in BCP_1 at time T_2, $D'_{3,3}$ is processed in BCP_1 at time T_3, and $D'_{4,4}$ is processed in BCP_1 at time T_4 and so on. All d BCPs are active at each time unit with pipelined operations and d solutions are generated at each unit of time from the mth time slice. Thus time complexity of evaluating entire table D' becomes $O(((n - m) / d) + m)$.

Table-2 shows time complexities of three schemes (i.e. hierarchical, linear, and broadcasting) in both serial and parallel cases.

Scheme	Serial	Parallel
Hierarchical	O(n + lon m)	O((n / d) + log m)
Linear	O(n + m)	O((n / d) + m)
Broadcasting	O(n)	O(((n - m) / d) + m)

Table-2. Time complexities of schemes for k-mismatches (and exact matching) problem

10.4 Implementation Methodologies

Since our dataflow schemes are simple and easy to be modularized, we can build VLSI chip for the k-mismatches problem in a straightforward way. Using dataflow schemes described so far, we can use a few simple basic cells and most of the cells are copies of these basic cells. For the hierarchical scheme, cells are connected hierarchically. The linear scheme and the broadcasting scheme can be implemented as linear systolic array of cells. In our design, interconnection of these cells are regular which implies that our design can be modular and extensible to build large chips. This section presents very high level design methodologies to develop special purpose VLSI chip for the k-mismatches problem. With consideration of the clock synchronization, the dataflow schemes can be pipelined hardware solutions to the k-mismatches problem. By using the forwarding mechanisms PWS and PBC blocks, the schemes provide high performance parallel hardware solutions.

10.4.1 Hierarchical Scheme

For the hierarchical scheme illustrated in Figure-39, two basic cells are used. We combine the "S" actor and the "Eq" block vertically and make the "S-Eq" cell. The second basic cell is the "Adder" cell. "Adder" cells are hierarchically connected. Figure-67 illustrates the design. The "S-Eq" cell can be implemented with 8-bit comparator, which compares two 8-bit characters, a register in which the pattern character is preloaded at initialization time. An 8-bit latch can be used for the action of the "S" actor. The initial tokens "ε" which are special dummy characters are loaded into these 8-bit latches (except the right most one) at the initialization time.

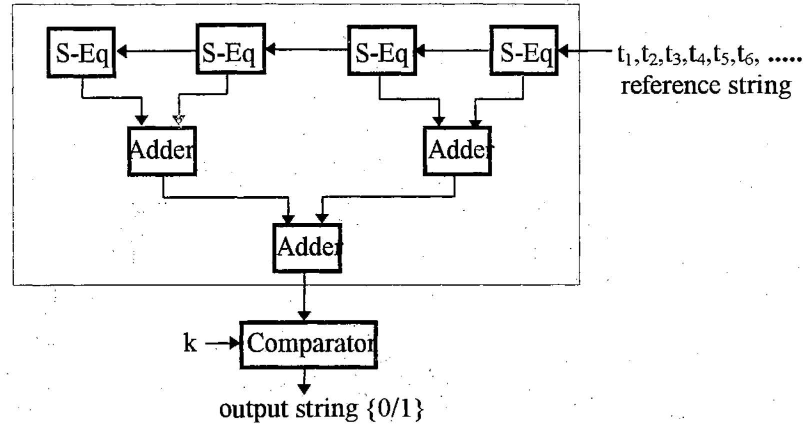


Figure-67. Hierarchical array of cells for the k-mismatches problem (serial scheme with m=4 case)

Figure-68(a) shows the structure of the kth (from left to right) "S-Eq" cell. In the figure, R_k is the register in which the kth pattern character is preloaded. For the "Adder" cell, we can use an adder. When the pattern size is m ($\neq 2^k$), we should use latches in the hierarchical connections of the "Adder" cells for synchronization of the data.

For the parallel design, we should not use the "S-Eq" cell. Instead, we can simply connect multiple number of the computation parts (HCP) which consist of "Eq" and "Adder" cells to the parallel WS block (PSW). The computation part shown in the Figure-39 is used and the implementation of the "Eq" cell is shown in the Figure-68(b). The PSW(d*dm) block illustrated in Figure-50 and Figure-51 can be implemented with wires and the 8-bit latches for the "S" actors. Initial tokens "ε" are loaded into the latches at the initialization time. Connection of the PWS block and HCPs are depicted in Figure-49. The implementation is straightforward and thus the illustration of the parallel hierarchical design is skipped.

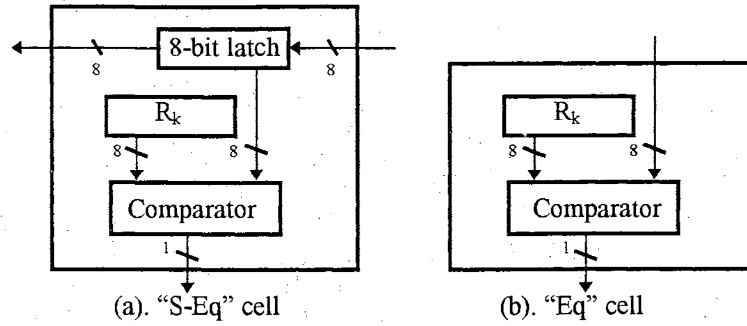


Figure-68. The structures of basic cells "S-Eq" and "Eq"

10.4.2 Linear Scheme

The linear scheme can be implemented as a linear systolic array of cells. For the serial design, each PE is connected linearly as illustrated in Figure-41. We can implement each PE as a basic cell and simply copy and connect multiple PEs linearly. For a pattern length m, we need m PE cells for the serial design. Thus the array looks like the one in Figure-69. The structure of each PE is illustrated in Figure-70.

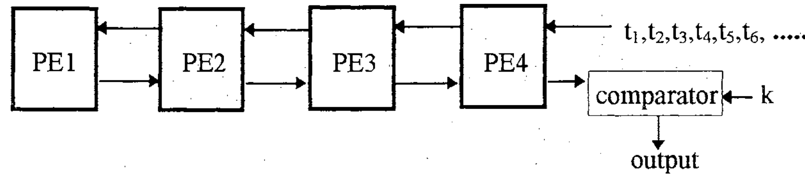


Figure-69. Systolic array of PEs for serial linear scheme (m=4 case)

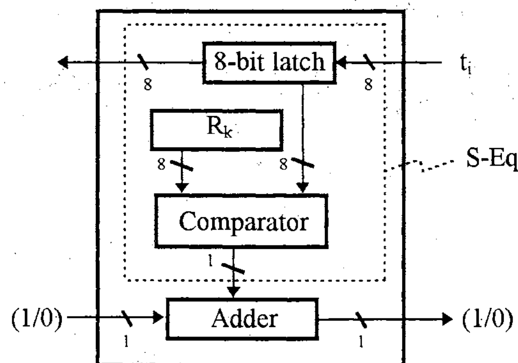


Figure-70. Structure of each PE of the linear scheme

But, with this design scheme we encounter the synchronization problem. As described in section 10.2.2, the linear scheme should use the synchronization actor "S" in the static dataflow environment. This brings about the usage of latches for the synchronization in hardware implementation. Thus in the PE_k (k^{th} from left), we should assign k-2 latches between "S-Eq" and "Adder" blocks. Positions of these 1-bit latches in each PE depicted in Figure-70 are between the components "Comparator" and "Adder". Figure-71 illustrates the array using these latches.

Since each PE of the design has different number of latches, m different types of basic cells are needed with this design. If the pattern size m is very small and the parallelism degree d is big, this overhead can be ignored. Otherwise it can cause the design cost problem.

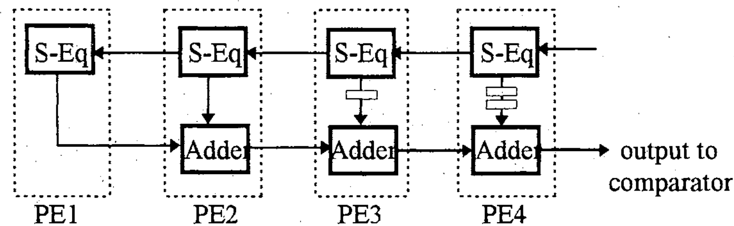


Figure-71. Latches used in implementation of the linear scheme (m=4 case)

For the parallel design, we can simply connect multiple number of computation part (refer Figure-44) to the PWS block. Connection of the PWS block and LCPs are depicted in Figure-49. The implementation is straightforward and thus the illustration of the parallel linear design is skipped.

10.4.3 Broadcasting Scheme

Since the broadcasting scheme is simple and easy to be modularized, we can build VLSI chip for the k-mismatches problem in a straightforward way. As same as the design for the linear scheme, broadcasting scheme can also be implemented as a linear systolic array of basic cells. Only one basic cell is needed and it is extended to build the linear systolic array architecture. This is the advantage over the linear scheme. As used in design for other schemes, we connect one comparator to the end of the systolic array to check the error bound k. In this part, very high level design methodologies to develop special purpose VLSI chip for the k-mismatches problem by using the broadcasting scheme is presented. For the serial broadcasting scheme, we use the design depicted in Figure-45 in which the broadcasting part is included in each processing element (PE). Organization of the linear systolic array is depicted in Figure-72. Components and dataflow of each processing element is illustrated in Figure-73. As described earlier, the first (left most) PE does not have to have the adder. For making identical basic cell, all PEs should have same structure. Initial tokens shown in dataflow environment can be preloaded into the last (right-bottom) latch at initialization time.

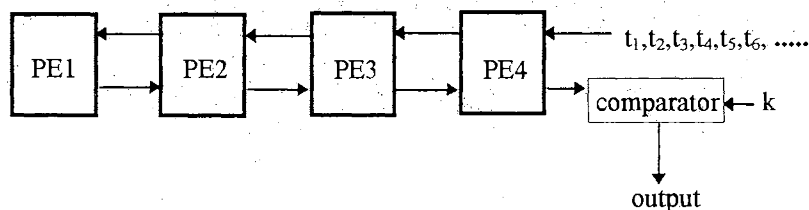


Figure-72. Systolic array of PEs for serial broadcasting schem (m=4 case)

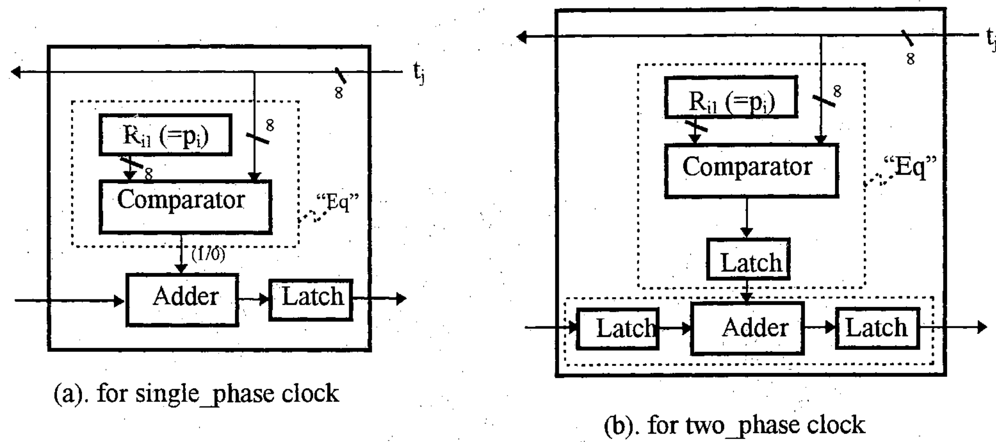


Figure-73. Structure of each PE (PE_i) of the broadcasting scheme

For implementation of the parallel broadcasting scheme the broadcasting part, which is included in each PE in serial design, is excluded and d identical computation parts (BCPs) are connected to parallel broadcasting block (PBC) as illustrated in Figure-53. Thus structure of each PE is same as the serial case except the input reference characters come from PBC block. Each computation part (BCP) has m (pattern length) identical such PEs and d identical BCPs are connected to the PBC(d*dm) block. Dataflow version of the BCP block is illustrated in Figure-46 and each BCP block can be implemented same as the serial implementation we described so far with the exception that the input reference characters come from the PBC block. The PBC block can be implemented simply with wires and latches (for the "S" actors) as shown in Figure-56 and Figure-57. The implementation of the parallel broadcasting scheme is straightforward and the detailed illustration is skipped.

Chapter XI

EXACT MATCHING PROBLEM

For the exact matching problem, solution scheme is exactly same as the one we designed for the k-mismatches problem since the problem can be considered as a special case of the k-mismatches problem in which the only difference is that the error bound k = 0. On the schemes which we described for the k-mismatches problem in previous chapter, we simply set the value of the error bound k = 0 for the exact matching problem. They can find all substrings of the text (reference string) which have same length (m) with the pattern and need 0 substitutions to convert to the pattern. The hierarchical, the linear, and the broadcasting schemes for the k-mismatches problem also work for the exact matching problem. Both serial and parallel versions of those schemes are used.

Implementations are also the same as those described in the chapter of the k-mismatches problem. Proposed dataflow schemes (both serial and parallel) in previous chapter accommodate both k-mismatches and exact matching problems. The only difference is setting of the error bound k. This leads the cost efficiency of the special purpose VLSI chip design for both subproblems.

11.1 Alternative Schemes

There are alternative schemes for the exact matching problem which do not need adders and the comparator for checking error bound k. Since the value of the k is fixed to 0, we

do not need to add matching results of pattern characters and reference characters to compute $D'_{m,j}$ ($1 \leq j \leq n$) in the edit distance table of the k-mismatches problem. Instead, AND gates are used.

Instead of using the dynamic programming method, which is used for the approximate string matching problems and requires edit distance table computation, the naive algorithm for the exact matching problem is used since it provides good systolic mechanism for implementation. Thus, the alternative scheme starts from the naive algorithm in which the pattern string P is compared to all same sized substrings of the reference string T:

```
Naive-String-Matching (T, P)
   n := length [T];   /*reference string*/
   m := length [P];   /*pattern string*/
   for s := 0 to (n-m)
      if P[1..m] = T[s+1 .. s+m] then
         output "pattern occurs with shift s ";
```

The naive algorithm requires n-m+1 attempts and each attempt takes m comparisons. Comparison results are passed through binary two-input AND gates to produce the solutions (i.e. binary stream). AND gates are simply implemented and this design eliminates the need of adders and a comparator in the computation part (CP). Alternative methods for the hierarchical, the linear, and the broadcasting schemes are illustrated in Figure-74, Figure-75, and Figure-76 respectively. From the schemes presented for the k-mismatches problem, only computation parts (HCP, LCP, and BCP) are changed. Number of AND gates required in the alternative scheme is same as that of

adders used in original schemes. In figures, AHCP represents the alternative HCP, ALCP represents the alternative LCP, and ABCP stands for the alternative BCP.

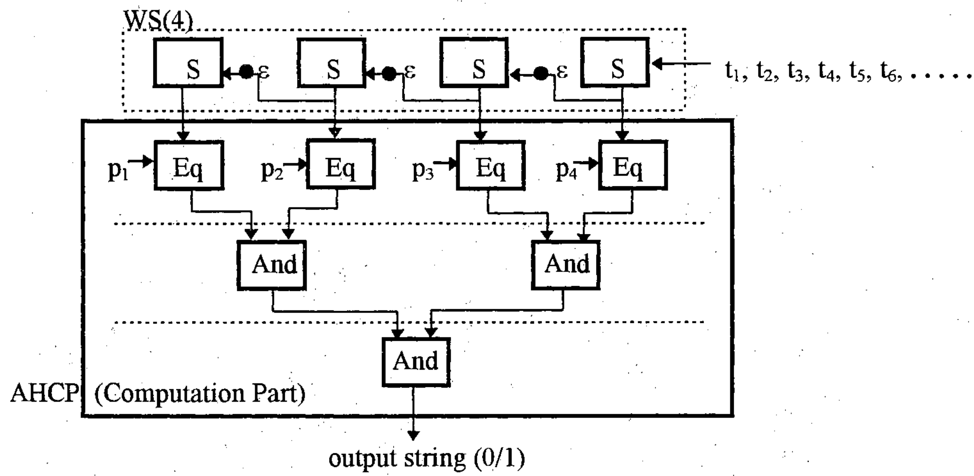


Figure-74. Alternative hierarchical scheme for the exact matching problem (serial, m=4 case)

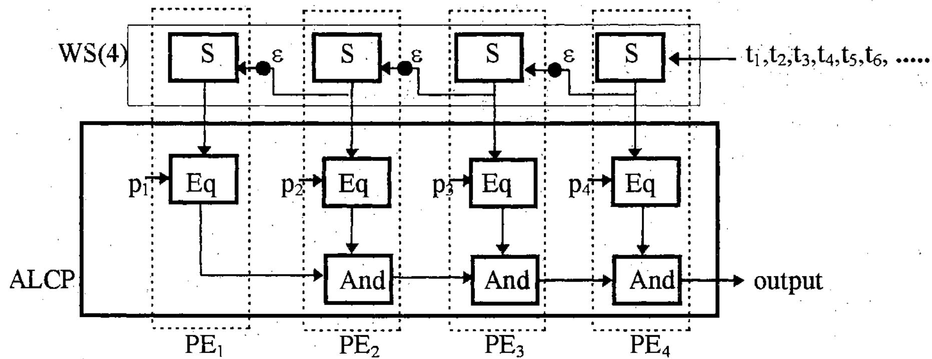


Figure-75. Alternative linear scheme for the exact matching problem (serial, m=4 case)

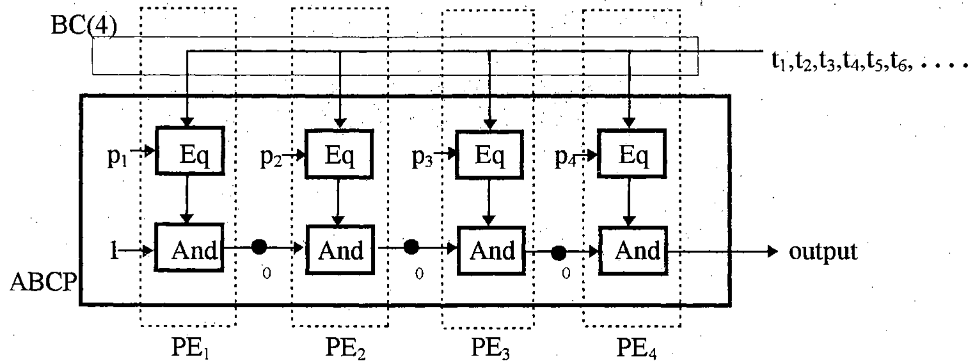


Figure-76. Alternative broadcasting scheme for the exact matching problem (serial, m=4 case)

Refined dataflow graph representation of each "And" block is depicted in Figure-77. Merging actor is used and the definition of that actor is illustrated in Figure-33(a).

For the implementation, a simple binary two input AND gate is used for the operation "And". Since the implementations are same as those described for the k-mismatches problem except the AND gates are used in place of adders and the comparators are eliminated, the description is skipped.

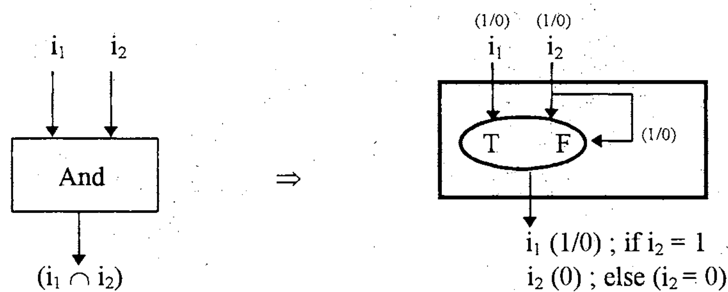


Figure-77: Refined dataflow graph (static) for the operation block "And"

For the parallel design, we can simply connect multiple number of the alternative way of computation parts (AHCPs, ALCPs, or ABCPs) to the PWS block or the PBC block as we described in the chapter for the k-mismatches problem.

Time analysis with alternative computation part is same as the original schemes. Time complexities of all schemes (i.e. the hierarchical, the linear, the broadcasting) are not affected by using alternative computation parts for the exact matching problem.

Chapter XII

CONCLUSION

In this dissertation, a new scheme namely the memoryless scheme of handling the history sensitive computations in dataflow execution model is proposed. This scheme is a real dataflow scheme which does not use any memory references to store the stream or the history required to solve the given history sensitive problems. Thus it fulfills the principles of dataflow execution model. At run time, the required history is preserved in dataflow graph itself. As a dynamic scheme, this memoryless scheme can support infinite length stream manipulation. Designs for both static and dynamic dataflow environment were provided. The memoryless scheme is described on two primitive models of the history sensitive problems such as the accumulator based model and the working-set based model. Without incurring overhead caused by memory references, the scheme provides an elegant solution to the history sensitive computation problems in dataflow execution model. Language constructs are defined and performance advantage of the memoryless scheme over the methods found in the literatures were analyzed.

To gain high performance, forwarding mechanisms which exploit maximum parallelism in both accumulator based and working-set based models are designed and adapted in the memoryless scheme. They provide high performance pipelined parallelism in accumulator based model and explicit parallelism which is comparable to loop unfolding in working-set based model. With these forwarding mechanisms namely PACC and PWS,

degree of parallelism is controlled explicitly. This is a great advantage over other schemes for handling history sensitive computations.

Developing elegant memoryless scheme for handling the history sensitive problems in dataflow execution model has important role in the digital signal processing. The memoryless scheme provides efficient systolic algorithms for the prefix computation problem and the string matching problem. These systolic algorithms are parallel hardware solutions to the problems since they can be used to design special purpose VLSI chip for those problems. Chapters VIII .. XI are devoted to the application of the memoryless scheme to the string matching problem. Based on the memoryless scheme, efficient dataflow schemes for three subproblems of the string matching were presented. Proposed schemes provide elegant parallel solutions to those three subproblems (i.e. k-differences, k-mismatches, and exact matching) of string matching in dataflow environment. Since these schemes are one-pass pure dataflow (i.e. systolic) algorithms, they do not need any preprocessing or extra memory space. Thus, they are suitable for VLSI implementation. By using few simple basic cells, those schemes can be implemented as linear (or hierarchical) systolic arrays of cells and easily extendible.

For the k-differences problem, implicit parallelism of the dataflow scheme reduces the total processing time of evaluating $m*n$ edit distance table by the factor of m. The result is O(n + m) time complexity which includes the initial trigger time (m - 1), where n and m are the lengths of reference and pattern strings. The design is a linear systolic array of m identical processing elements (PEs). Advantage over other hardware approaches in literature includes that the proposed scheme checks similarities between pattern and all

substrings, which have lengths $m \pm k$, of the text and reports all approximate occurrences of the pattern in the text.

For the k-mismatches and the exact matching problems, three versions of dataflow schemes are presented. They are the hierarchical, the linear, and the broadcasting schemes. For a reference string length n and a pattern length m, implicit parallelism of the dataflow provides linear time complexities $O(n + \alpha)$ to those three schemes. The α in the time complexity is log m for the hierarchical, m for the linear, and 0 for the broadcasting scheme. To gain high performances parallel schemes are developed by using the forwarding mechanisms PWS and PBC; i.e. parallel hierarchical scheme, parallel linear scheme, and parallel broadcasting scheme. The PWS is the forwarding mechanism of the working-set based memoryless scheme for the history sensitive problems and, the PBC is a variation of that. They exploit explicit parallelism based on multiple input (and out) streams. They can provide any degree of explicit parallelism which other hardware approaches do not provide. With these parallel schemes, degree of the parallelism is controlled explicitly. This is an important factor for resource management in dataflow machine environment and special purpose hardware implementation. In dataflow environment, the language constructs which invoke the string matching scheme may include a parameter for controlling the degree of the parallelism. Otherwise the compiler should do this job automatically. For the special purpose hardware design, there should be the limit of system bus bandwidth and number of processors for the multi-stream manipulation. Building the special purpose hardware should include the careful consideration of the system resources to choose the number of multiple streams (degree of parallelism) to gain the maximum benefit of the parallelism. Those three parallel schemes

solve the k-mismatches and the exact matching problems with time complexities reduced by a factor of d where d represents the controllable degree of the parallelism. The parallel hierarchical scheme has time complexity $O((n / d) + \log m)$, the parallel linear scheme has time complexity $O((n / d) + m)$, and the parallel broadcasting scheme has time complexity $O(((n - m)/d) + m)$. The hierarchical scheme can be implemented by hierarchical systolic array of few basic cells. The design can be extended easily. With linear systolic array architectures, design of serial and parallel broadcasting schemes need m and d*m identical processing elements respectively.

In general, proposed dataflow schemes for the string matching problems can work on both on-line and off-line inputs; i.e. they can work on unknown sized reference strings. They check similarities between pattern and all possible substrings of the text for given problem and, report all occurrences (approximate or exact) of the pattern in the text.

In this dissertation, we applied the memoryless scheme for handling history sensitive computation to one-dimensional pattern matching (i.e. string matching problems). Future work should focus on two-dimensional pattern matching by developing efficient mechanism based on methods developed for string matching problems.

REFERENCES

[1] D. Abramson, and G. Egan, " The RMIT Data Flow Computer : A Hybrid Architecture," The Computer Journal, Vol. 33, No. 3, pp. 230-240, March 1990.

[2] William B. Ackerman, "Data Flow Languages," Computer, Vol. 15, pp. 15-25, Feb. 1982.

[3] A. V. Aho, "Algorithms for Finding Patterns in Strings," Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., Elsevier Science Publishers B. V., New York, 1990, pp. 257-297.

[4] Makoto Amamiya, Masaru Takesue, Ryuzo Hasegawa, and Hirohide Mikami, "Implementation and Evaluation of A List-Processing-Oriented Data Flow Machine," Proc. 13th Annual Symposium on Computer Architecture, pp. 10-19, 1986.

[5] Makoto Amamiya, Masaru Takesue, Ryuzo Hasegawa, and Hirohide Mikami, "A Data Flow Machine Architecture for Highly Parallel Symbol Manipulations," Journal of Information Processing, Vol. 10, No. 4, pp. 227-236, 1987.

[6] Amihood Amir, Gary Benson, and Martin Farach, "An Alphabet Independent Approach to Two-Dimensional Pattern Matching," Siam J. Comput., Vol. 23, pp. 313-323, April 1994.

[7] Alberto Apostolico and Dany Breslauer, "An Optimal O(log log N)-Time Parallel Algorithm for Detecting All Squares in a String," Siam J. Comput., Vol. 25, pp. 1318-1331, Dec. 1996.

[8] Arvind, and David E. Culler, "Dataflow Architectures," Annual Review in Computer Science, Vol. 1, pp. 225-253, 1986.

[9] Arvind, and Kim P. Gostelow, "The U-Interpreter," Computer Vol. 15, pp. 42-49, Feb. 1982.

[10] Arvind, and Rishiyur S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," IEEE Trans. on Computers, Vol. 39, No. 3, pp. 300-318, March 1990.

[11] Arvind, and Robert E. Thomas, "I-Structures: An Efficient Data Type for Functional Languages," MIT Lab. for Computer Science Technical Report TM-178, Cambridge, MA, Sept. 1980.

[12] J. Backus, "Can Programming Be Liberated From The Von Neumann Style? A Functional Style and Its Algebra Of Programs," Comm. ACM, 21, No. 8, Aug. 1978.

[13] Ricardo Baeza and G. H. Gonnet, "A New Approach to Text Searching," Comm. ACM, Vol. 35, pp. 74-81, Oct. 1992.

[14] S. Bandyopadhyay, S. Ghosh, C. Mazumdar, and S. Bhattacharya, "An Alternative Approach to History Sensitive Computation in Dataflow Model," Journal of Information Processing, Vol. 12, No. 1, pp. 16-19, 1988.

[15] J. Backus, "Can Programming Be Liberated From The Von Neumann Style? A Functional Style and Its Algebra Of Programs," Comm. ACM, 21, No. 8, Aug. 1978.

[16] A. A. Bertossi, F. Luccio, L. Pagli, and E. Lodi, "A Parallel Solution to the Approximate String Matching Problem," The Computer Journal, Vol. 35, pp. 524-526, 1992.

[17] R. Boyer and S. Moore, "A Fast String Searching Algorithm," Comm. ACM, Vol. 20, pp. 762-772, 1977.

[18] David F. Brailsford, and R. James Duckworth, "The MUSE Machine-an Architecture for Structured Data Flow Computation," New Generation Computing, pp. 181-195, March 1985.

[19] Dany Breslauer and Zvi Galil, "An Optimal O(log log n) Time Parallel String Matching Algorithm," Siam J. Comput., Vol. 19, pp. 1051-1058, Dec. 1990.

[20] Dany Breslauer and Zvi Galil, "A Lower Bound for Parallel String Matching," Siam J. Comput., Vol. 21, pp. 856-862, Oct. 1992.

[21] L. J. Caluwaerts, J. Debacker, and J. A. Peperstraete, "Implementing Streams on a Data Flow Computer System with Paged Memory," Proc. 10th Annual Symposium on Computer Architecture, pp. 76-83, 1983.

[22] H. D. Cheng and K. S. Fu, "VLSI Architecture for String Matching and Pattern Matching," Pattern Recognition, Vol. 20, pp. 125-141, 1987.

[23] Richard Cole, "Tighter Bounds on the Complexity of the Boyer-Moore String Matching Algorithm," Siam J. Comput., Vol 23, pp. 1075-1091, Oct. 1994.

[24] Richard Cole, Ramesh Hariharan, Mike Paterson, and Uri Zwick, "Tighter Lower Bounds on the Exact Complexity of String Matching," Siam J. Comput., Vol. 24, pp. 30-45, Feb. 1995.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms, McGraw Hill, 1990.

[26] D. E. Culler et al., "Fine-Grain parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," Proc. Fourth International Conf. on Architectural Support for Programming Languages and Operating Systems, 1991.

[27] David E. Culler and Arvind, "Resource Requirements of Dataflow Programs," Proc. 15th Annual Symposium on Computer Architecture, pp. 141-150, 1988.

[28] Alan L. Davis, and Robert M. Keller, "Data Flow Program Graphs," Computer, Vol. 15, pp. 26-41, Feb. 1982.

[29] Jack B. Dennis, "Data Flow Supercomputers," Computer, Vol. 13, pp. 48-56, Nov. 1980.

[30] Jack B. Dennis and Gao Guang Rong, "Maximum Pipelining of Array Operations on Static Data Flow Machine," Proc. 1983 International Conference on Parallel Processing, pp. 331-334, 1983.

[31] Jack B. Dennis, and Ken K. S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," Proc. 1979 Int. Conference on Parallel Processing, pp. 35-45, Aug. 1979.

[32] Bradly Fawcett, "Reconfiguring a Computing Pattern," Electronic Engineering Times, Manhasset, pp. 64- , April 1995.

[33] Hose A. B. Fortes, and Benjamin W. Wah, "Systolic Arrays - A Survey of Seven Projects," Computer, July 1987, pp.91-103.

[34] M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," Computer, pp. 26-38, Jan. 1980.

[35] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," Computer, Vol. 15, pp. 58-69, Feb. 1982.

[36] Z. Galil and R. Giancarlo, "Improved String Matching with K Mismatches," SIGACT News 17, pp. 52-54, 1986.

[37] Zvi Galil, "A Constant-Time Optimal Parallel String-Matching Algorithm," Journal of the ACM, Vol. 42, pp. 908-918, July 1995.

[38] Jean-Luc Gaudiot, "Structure Handling in Data-Flow Systems," IEEE Trans. on Computers, Vol. C-35, No. 6, pp. 489-502, June 1986.

[39] Jean-Luc Gaudiot, and Yi-Hsiu Wei, "Token Relabeling in a Tagged Token Data-Flow Architecture," IEEE Trans. on Computers, Vol. 38, No. 9, pp. 1225-1239, Sept. 1989.

[40] Narain Gehani, and Andrew D. McGettrick, Concurrent Programming, Addison-Wesley, 1988.

[41] Dipak Ghosal, and Laxmi N. Bhuyan, "Performance Evaluation of a Dataflow Architecture," IEEE Trans. on Computers, vol. 39, no.5, pp. 615-627, May 1990.

[42] V. G. Grafe, and J. E. Hoch, "The Epsilon-2 Hybrid Dataflow Architecture," COMPCON 90, 35th IEEE Computer Society International Conference, pp. 88-93, 1990.

[43] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," Communications of the ACM, Vol. 28, No. 1, pp. 34-52, Jan. 1985.

[44] James Hicks, Derek Chiou, et al., "Performance Studies of Id on the Monsoon Dataflow System," Journal of Parallel and Distributed Computing, 18, pp. 273-300, 1993.

[45] Kei Hiraki et al., "The Sigma-1 Dataflow Supercomputer: A Challenge for New Generation Super Computing Systems," Journal of Information Processing, Vol. 10, No. 4, pp. 219-226, 1987.

[46] Herbert H. J. Hum and Guang R. Gao, "Summary of the Second Workshop on Frontier in Functional Programming and Dataflow Architecture," ACM SIGARCH Computer Architecture News, vol. 16, no.5, pp. 12-19, Dec. 1988.

[47] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, New York: McGraw-Hill, 1984.

[48] Robert A. Iannucci, "A Dataflow/von Neumann Hybrid Architecture," Technical Report, MIT/LCS/TR-418, May 1988.

[49] Merrill E. Isenman and Dennis E. Shasha, "Performance and Architectural Issues for String Matching," IEEE Trans. on Computers, Vol. 39, No. 2, pp. 238-250, Feb. 1990.

[50] Noriyoshi Ito, Eiji Kuno, and Teruhiko Oohara, "Efficient Stream Processing in GHC and Its Evaluation on a Parallel Inference Machine," Journal of Information Processing, Vol. 10, No. 4, pp. 237-243, 1987.

[51] R. Colin Johnson, "Desktop CAD comes to bioengineering," Electronic Engineering Times, Manhasset, Issue 925, pp. 40- , Oct. 1996.

[52] Pettery Jokinen, Jorma Tarhio and Esko Ukkonen, "A Comparison of Approximate String Matching Algorithms," Software-Practice and Experience, Vol. 26, pp. 1439-1457, Dec. 1996.

[53] Ian Kaplan, "The LDF 100: A Large Grain Dataflow Parallel Processor," ACM SIGARCH Computer Architecture News, vol. 15, no.3, pp. 5-12, June 1987.

[54] K. Kawakami, and J. R. Gurd, "A Scalable Dataflow Structure Store," Proc. 13th Annual Symposium on Computer Architecture, pp. 243-250, 1986.

[55] D. E. Knuth, J. Morris, and V. Pratt, " Fast Patern Matching in Strings," SIAM J. of Comput., Vol. 6, pp. 323-350, 1977.

[56] G. Landau and U. Vishkin, "Efficient String Matching with K Mismatches," Theoretical Comput. Sci., 43, pp. 239-249, 1986.

[57] Thierry Lecroq, "Experimental Results on String Matching Algorithms," Software-Practice and Experience, Vol. 25, pp. 727-765, Jul. 1995.

[58] Ben Lee and A. R. Hurson, "A Dataflow Architectures and Multithreading," COMPUTER, pp. 27-39, August 1994.

[59] Ben Lee, A. R. Hurson, and Behrooz Shirazi, "A Hybrid Scheme for Processing Data Structures in a Dataflow Environment," IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 1, pp. 83-96, Jan. 1992.

[60] David R. Lester, "Stacklessness: Compiling Recursion for a Distributed Architecture," Proc. 4th Int. Conference on Functional Programming Languages and Computer Architecture (FPCA 89), pp. 116-127, 1989.

[61] Daniel P. Lopresti, "P-NAC : A Systolic Array for Comparing Nucleic Acid Sequences," Computer, pp. 98-99, July 1987.

[62] Maurice Maes, "Polygonal Shape Recognition Using String Matching Techniques," Pattern Recognition, Vol. 24, pp. 433-440, 1991.

[63] G. M. Magson, "Efficient Systolic String Matching," Electronics Letters, Vol. 26, No. 24, pp. 2039-2041, Nov. 1990.

[64] Udi Manber and Gene Myers, "Suffix Arrays: A New Method for On-Line String Searches," Siam J. Comput., Vol. 22, pp. 935-948, 1993.

[65] James R. McGraw, and Stephen K. Skedzielewski, "Streams and Iteration in VAL, Additions to A Data Flow Language," 3rd International Conference on Distributed Computing Systems, pp. 730-739, Oct. 1982.

[66] Amar Mukherjee, "Hardware Algorithms for Determining Similarity Between Two Strings," IEEE Trans. on Computers, Vol. 38, No. 4, pp. 600-603, April 1989.

[67] Amar Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," IEEE Trans. on Computers, Vol. C-28, No. 6, pp. 384-394, June 1979.

[68] Rishiyur S. Nikhil, and Arvind, "Can Dataflow subsume von Neumann computing?," Proc. 16th Annual Symposium on Computer Architecture, pp. 262-272, 1989.

[69] Rishiyur S. Nikhil, Gregory M. Papadopoulos and Arvind, "*T: a Killer Micro for a Brave New World," Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1991.

[70] Rishiyur S. Nikhil, Gregory M. Papadopoulos and Arvind, "*T: A Multithreaded Massively Parallel Architecture," Proc. 19th Annual Symposium on Computer Arch., pp. 156-169, 1992.

[71] Gregory M. Papadopoulos, "Implementation of a Dataflow Multiprocessor," Technical Report, MIT/LCS/TR-432, Aug. 1988.

[72] Gregory M. Papadopoulos, and David E. Culler, "Monsoon: an Explicit Token-Store Architecture," Proc. 17th Annual Symposium on Computer Architecture, pp. 82-91, 1990.

[73] Jin H. Park, and K. M. George, "A Memoryless Scheme for History Sensitive Problems in Dataflow Machines," Proceedings of the Second International Conference on Massively Parallel Computing Systems, pp. 64-71, May 1996.

[74] Jin H. Park, and K. M. George, "Parallel History Sensitive Computations in Dataflow Architecture," Proceedings of the IEEE Second International Conference on Algorithms & Architectures for Parallel Processing, pp. 522-529, June 1996.

[75] L. M. Patnaik, R. Govindarajan, and N. S. Ramadoss, "Design and Performance Evaluation of EXMAN: An EXtended MANchester Data Flow Computer," IEEE Trans. on Computers, Vol. C-35, No. 3, pp. 229-243, March 1986.

[76] John Peterson, "Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time," Proc. 4th Int. Conference on Functional Programming Languages and Computer Architecture (FPCA 89), pp. 89-99, 1989.

[77] P. A. Pevzner and M. S. Waterman, "Multiple Filtration and Approximate Pattern Matching," Algorithmica, 1995, pp. 135-154.

[78] J. Sargeant, and C. C. Kirkham, "Stored Data Structures on the Manchester Dataflow Machine," Proc. 13th Annual Symposium on Computer Architecture, pp. 235-242, 1986.

[79] Raghu Sastry, N. Ranganathan, "A Systolic Array for Approximate String Matching," Proc. Int'l Conf. Computer Design, pp. 402-405, Cambridge, Mass., 1993.

[80] Raghu Sastry, N. Ranganathan, and Klinton Remedios, "CASM: A VLSI Chip for Approximate String Matching," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 17, pp. 824-830, Aug. 1995.

[81] M. Sato, Y. Kodama et al., "Thread-based Programming for the EM-4 Hybrid Dataflow Machine," Proc. 19th Annual Symposium on Computer Architecture, pp. 146-155, 1992.

[82] Carlo H. Sequin, Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference, The MIT Press, Cambridge, MA, pp. 290-306, 1991.

[83] Toshio Shimada, Kei Hiraki, et al., "Evaluation of A Prototype Data Flow Processor of The SIGMA-1 for Scientific Computations," Proc. 13th Annual Symposium on Computer Architecture, pp. 226-234, 1986.

[84] Gregory B. Shippen, and James K. Archibald, "A Tagged Token Dataflow Machine for Computing small, Iterative Algorithms," ACM SIGARCH Computer Architecture News, Vol. 15, No. 6, pp. 9-18, Dec. 1987.

[85] M. Sowa, "A Method for Speeding up Serial Processing in Dataflow Computers by Means of a Program Counter," The Computer Journal, Vol. 30, No. 4, pp. 289-294, 1987.

[86] Tatsuo Suzuki et al., "Procedure Level Data Flow Processing on Dynamic Structure Multimicroprocessors," Journal of Information Processing, vol. 5, no.1, pp. 12-16, 1982.

[87] Masura Takesue, "A Unified Resource Management and Execution Control Mechanism for Data Flow Machines," Proc. 14th Annual Symposium on Computer Architecture, pp. 90-97, 1987.

[88] Jorma Tarhio and Esko Ukkonen, "Approximate Boyer-Moore String Matching," Siam J. Comput., Vol. 22, pp. 243-260, April 1993.

[89] Mario Tokoro, J. R. Jagannathan, and Hideki Sunahara, "On The Working Set Concept for Data-Flow Machines," Proc. 10th Annual Symposium on Computer Architecture, pp. 90-97, 1983.

[90] Robert A. Wagner, "The String-to-String Correction Problem," Journal of the ACM, Vol. 21, pp. 168-173, Jan. 1974.

[91] Haigeng Wang, Alexandru Nicolau, and Kai-Yeng S. Siu, "The Strict Time Lower Bound and Optimal Schedules for Parallel Prefix with Resource Constraints," IEEE Trans. on Computers, Vol. 45, No. 11, pp. 1257-1271, Nov. 1996.

[92] Ian Watson, and John Gurd, "A Practical Data Flow Computer," Computer, Vol. 15, pp. 51-57, Feb. 1982.

[93] Alden H. Wright, "Approximate String Matching using Within-word Parallelism," Software-Practice and Experience, Vol. 24, pp. 337-362, April 1994.

[94] Sun Wu and Udi Manber, "Fast Text Searching Allowing Errors," Comm. ACM, Vol. 35, pp. 83-91, Oct. 1992.

[95] Toshitsugu Yuba, "Research and Development Efforts on Dataflow Computer Architecture in Japan," Journal of Information Processing, vol. 9, no.2, pp. 52-60, 1986.

[96] Toshitsugu Yuba, Toshino Shimada, et al., "Dataflow Computer Development in Japan," ACM SIGARCH Computer Architecture News, vol. 18, no.3, (1990 International Conference on Supercomputing) pp. 140-147, Sept.1990.

VITA

JIN HWAN PARK

Candidate for the Degree of

Doctor of Philosophy

Thesis: **STREAM ORIENTED COMPUTATION IN DATAFLOW EXECUTION MODEL AND APPLICATION TO STRING MATCHING PROBLEMS**

Major Field: Computer Science

Biographical:

Personal Data: Born in Pusan, Korea, on August 07, 1958, the son of Sang-Ho Park and Ki-Nam Jeong.

Education: Received Bachelor of Science degree in Physics in 1985. Received Master of Science degree in Computer Science from The Ohio University in 1987. Completed the requirements for the Doctor of Philosophy degree with a major in Computer Science at Oklahoma State University in December 1998.

Professional Memberships: ACM SigArch and SigDA.