WASHINGTON UNIVERSITY

Sever Institute
School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Ron K. Cytron, Chair
Young H. Cho, Co-Chair
John W. Lockwood, Co-Chair
Jeremy D. Buhler
Roger D. Chamberlain
Ronald P. Loui
Robert E. Morley

TECHNIQUES FOR HARDWARE-ACCELERATED PARSING FOR NETWORK AND BIOINFORMATIC APPLICATIONS

by

James M. Moscola

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

May 2008

Saint Louis, Missouri

copyright by

James M. Moscola

2008

Acknowledgments

I would first like to thank those professors who have advised me throughout my graduate studies. First, I would like to thank my initial research advisor, Dr. John Lockwood, for his guidance over the years and for giving me the opportunity to work on a number of interesting, real-world problems. I would also like to extend a special thanks to Dr. Young Cho for taking on the role of a secondary advisor in Dr. Lockwood's absence. During his brief period as a visiting assistant professor at Washington University, Dr. Cho took the time to share a great deal of his knowledge and experiences with me. He also played a key role in formulating the direction of this work. Additionally, I would like to give a special thanks to Dr. Ron Cytron for advising me upon the departure of Dr. Lockwood. Dr. Cytron played an integral part in inspiring the direction of the bioinformatics portion of this dissertation. I am also grateful for all of the time that he dedicated to meeting with me and for the insightful discussions that he provided during that time. In the short period that we worked together, Dr. Cytron has made a strong impact on the way I think about and approach problems.

Next, I would like to thank the members of my dissertation committee, Dr. Jeremy Buhler, Dr. Roger Chamberlain, Dr. Young Cho, Dr. Ron Cytron, Dr. John Lockwood, and Dr. Robert Morley, for providing support and feedback throughout this research.

I would also like to thank Dr. John Lockwood for providing me funding support throughout most of my graduate studies. Without this support I would not have been able to pursue this research. For additional funding support, I would like to thank Dr. Jon Turner and Dr. Jerome Cox. Their support through the last year of my graduate studies allowed me

to not only finish this dissertation, but also to explore new and interesting research topics.

Additional thanks go out to all members, past and present, of the FPX Group (a.k.a. the Reconfigurable Network Group), and to the many other friends and colleagues in the CSE department at Washington University. I would especially like to thank my office mate, Phillip Jones, for many years of valuable feedback and discussion related to this research.

For their support and for always reminding me that I was still in graduate school, I would like to thank my family. Finally, I would like to express the highest level of gratitude to my wife Stephanie for her patience and sacrifice throughout this journey. Her support and encouragement helped to keep me focused on finishing my education. Her smile and laughter helped to remind me that there was life outside of my education.

James M. Moscola

Washington University in Saint Louis
May 2008

Contents

Acknowledgments . . . . **ii**

List of Tables . . . . **viii**

List of Figures . . . . **x**

Abstract . . . . **xv**

1 Introduction **1**

1.1 Parsing in Network Applications 1

1.2 Parsing in Bioinformatic Applications 2

1.3 Reconfigurable Hardware 3

1.4 Contributions 4

1.5 Overview of Dissertation 5

I Hardware Architectures for Pattern Matching and Parsing in Network Applications **7**

2 Hardware-Accelerated Pattern Matching **8**

2.1 Related Work 9

2.2 Pre-Decoded Character Bits 9

2.3 Pipelined Regular Expression Chain 10

2.4 Pipelined Character Grid 11

2.4.1 Regular Expression Extensions to the Pipelined Character Grid . . . 13
2.5 Timed Segment Matching . . . 14
2.5.1 Regular Expression Pattern Detection . . . 17
2.5.2 Scalable Architecture . . . 19
2.5.3 Implementation . . . 21
2.6 Chapter Summary . . . 26

3 Hardware-Accelerated Regular Language Parsing . . . 28
3.1 Related Work . . . 29
3.2 Architecture for High-Speed Regular Language Parsing . . . 29
3.3 Chapter Summary . . . 34

4 Applications of Regular Language Parser . . . 35
4.1 Background . . . 35
4.1.1 Field-Programmable Port Extender . . . 35
4.1.2 Layered Protocol Wrappers . . . 36
4.1.3 TCP Protocol Processor . . . 37
4.2 Content-Based Router . . . 37
4.2.1 Pattern Matcher . . . 39
4.2.2 Parsing Structure . . . 41
4.2.3 Routing Module . . . 43
4.2.4 Implementation and Experimental Setup . . . 45
4.2.5 Area and Performance . . . 47
4.3 Semantic Network-Content Filter . . . 47
4.3.1 Background . . . 48
4.3.2 Application Level Processing System . . . 51
4.3.3 Implementation of Email Parser . . . 52
4.3.4 Data Sets and Results . . . 56
4.4 Chapter Summary . . . 59

II Hardware Architectures for Accelerating RNA Secondary Structure Alignment **60**

5 RNA Secondary Structure Alignment . . . **61**
5.1 Introduction . . . 61
5.2 Related Work . . . 63
5.3 Background . . . 64
5.3.1 Covariance Models . . . 64
5.3.2 Rfam Database . . . 66
5.3.3 Database Search Algorithm . . . 67
5.4 Expressing Covariance Models as Task Graphs . . . 69
5.4.1 Bifurcation States . . . 69
5.5 Covariance Model Numeric Representation in Hardware . . . 71
5.6 Chapter Summary . . . 75

6 The Baseline Architecture . . . **76**
6.1 Overview . . . 76
6.2 Processing Elements . . . 77
6.3 Pipeline . . . 80
6.4 Implementation Results . . . 81
6.5 Expected Speedup for Larger CMs . . . 84
6.6 Chapter Summary . . . 85

7 The Processor Array Architecture . . . **86**
7.1 Overview . . . 87
7.2 Processing Modules . . . 88
7.2.1 Instruction Format . . . 90
7.2.2 Executing Instructions . . . 91
7.3 Shared Memory Structure . . . 91
7.3.1 Writing Results to the Shared Memory Structure . . . 92

7.3.2 Reading Data from the Shared Memory Structure 93
7.4 Reporting Module 95
7.5 Scheduling Computations 96
7.5.1 Optimal Scheduling of Directed Task Graphs 97
7.5.2 Scheduling Task Graphs on Finite Resources with Computational Latency 98
7.6 Architecture Analysis 104
7.6.1 Running Time 105
7.6.2 Scheduling Efficiency 106
7.6.3 Memory Requirements 107
7.6.4 Scalability of Architecture 111
7.6.5 Comparison to Software 117
7.7 Chapter Summary 120

8 Summary and Future Work 121
8.1 Dissertation Summary 121
8.2 Future Work 121

Appendix A Email Grammar in Lex/Yacc Style Format 124

Appendix B Covariance Model Data 129

Appendix C Additional Results for the Baseline Architecture 131

Appendix D Additional Results for the Processor Array Architecture . . 134

Appendix E Additional Results for the Speedup of the Processors Array Architecture Over Infernal 145

References 156

Vita 162

List of Tables

2.1 Device utilization for pipelined regular expression chain and pipelined character grid architectures 23

2.2 Device utilization for the architectures 23

2.3 Device utilization when scanning for regular expressions 26

3.1 *FIRST* and *FOLLOW* sets for symbols in the grammar 31

4.1 *FOLLOW* sets for example grammar 42

4.2 Language identification results for the 300-byte data set 56

4.3 Percentage of correctly classified documents for HAIL alone and ALPS+HAIL 57

4.4 Increase in accuracy when using ALPS+HAIL as opposed to HAIL alone . 58

5.1 Each of the nine different state types and their corresponding SCFG production rules 65

6.1 Performance comparison between Infernal and the baseline architecture . 82

6.2 Estimated speedup for baseline architecture running at 100 MHz 84

7.1 Configuration and resource requirements for different sized processor arrays. 113

7.2 The instruction sizes and bandwidth required for streaming instructions to the processor array architecture. 116

7.3 Estimated runtime and speedup of processor array architecture over Infernal software package. Estimate is based on a processor array running at 250 MHz. Additional results are in Appendix E. 118

B.1 Covariance Model Data 130

C.1 Estimated speedup for baseline architecture running at 100 MHz compared to INFERNAL 133

D.1 Results for processor array architecture using 1 processor 136

D.2 Results for processor array architecture using 2 processors 137

D.3 Results for processor array architecture using 4 processors 138

D.4 Results for processor array architecture using 8 processors 139

D.5 Results for processor array architecture using 16 processors 140

D.6 Results for processor array architecture using 32 processors 141

D.7 Results for processor array architecture using 64 processors 142

D.8 Results for processor array architecture using 128 processors 143

D.9 Results for processor array architecture using 256 processors 144

E.1 Processor array architecture with 1 processor compared to INFERNAL 147

E.2 Processor array architecture with 2 processors compared to INFERNAL . . . 148

E.3 Processor array architecture with 4 processors compared to INFERNAL . . . 149

E.4 Processor array architecture with 8 processors compared to INFERNAL . . . 150

E.5 Processor array architecture with 16 processors compared to INFERNAL . . 151

E.6 Processor array architecture with 32 processors compared to INFERNAL . . 152

E.7 Processor array architecture with 64 processors compared to INFERNAL . . 153

E.8 Processor array architecture with 128 processors compared to INFERNAL . . 154

E.9 Processor array architecture with 256 processors compared to INFERNAL . . 155

List of Figures

2.1 Decoder for 8-bit ASCII character "A" (0x41) 10

2.2 Constructing regular expressions 10

2.3 Simple example of pattern matcher using the pipelined character method . 12

2.4 Scaled pipelined character grid 13

2.5 Patterns with wild card characters 14

2.6 Fast FPGA pattern matchers for "aaaabbbccc" using a pipelined character grid; (a) Z_{AND} uses AND gate matching and (b) Z_{TS} uses TSM. 16

2.7 Pseudocode to generate a string comparator for the TSM architecture . . . 17

2.8 Regular expression operations in the timed segment matching architecture . 18

2.9 TSM example for "aaaa(abc|ba)*ccc" 19

2.10 Wide pipeline grid for 4× scaled architecture 19

2.11 Examples of the *zero-or-more* regular expression operation in the scaled TSM architecture 21

2.12 TSM architecture for ..."(abbacddc)*"... 22

2.13 LUTs vs. Number of Pattern Bytes 24

2.14 LUTs/Byte vs. Number of Pattern Bytes 25

2.15 DFFs vs. Number of Pattern Bytes 25

3.1 Algorithm for finding $FIRST()$ and $FOLLOW()$ sets from a production list [6] 30

3.2 Sample grammar for a regular language 30

3.3 NFA for grammar in Figure 3.2 31

3.4 Hardware parser for grammar shown in Figure 3.2 31

3.5 Sample grammar for a regular language 32
3.6 Hardware parser for grammar in Figure 3.5 33
3.7 Simplified parsing structure for grammar in Figure 3.5 33

4.1 Router in protocol wrappers 36
4.2 Document Type Definition (DTD) for content-based router implementation 38
4.3 Lex/Yacc style grammar 38
4.4 Content-based router architecture 39
4.5 Diagram of pattern matcher pipeline 40
4.6 Diagram of a string detector 41
4.7 Diagram of parsing structure 43
4.8 Diagram of routing module 44
4.9 FPX and GVS-1000 chassis 45
4.10 Test application interface 46
4.11 XML packet contents 46
4.12 Flow of documents through the document classification system 49
4.13 Sample email message 52
4.14 ABNF for date portion of email grammar 53
4.15 BNF for date portion of email grammar 53
4.16 ALPS email parser architecture 54
4.17 Pattern matcher for lexical analysis 55
4.18 Percentage of documents correctly classified by HAIL for each data set . . . 58
4.19 Increase in accuracy when using ALPS+HAIL 59

5.1 A reproduction of an example CM from [23, 53] 65
5.2 The number of covariance models in the Rfam database has continued to increase since its initial release in July of 2002. 66
5.3 An alignment window scans across the genome database. Each window is aligned to the CM via the DP parsing algorithm. 67

5.4 The initialization and recursion equations for the dynamic programming algorithm . . . 68
5.5 Distribution of maximum scores of all CMs in the Rfam 8.0 database 73
5.6 Graph depicting the linear relationship between the number of states in a CM and the maximum score computable for that CM . . . 74

6.1 A small CM, consisting of four nodes and thirteen states, represents a consensus secondary structure of only three residues . . . 78
6.2 A high-level view of a pipeline for the baseline architecture . . . 78
6.3 Each CM state is represented as a two-dimensional matrix, and each matrix cell is represented as a processing element containing adders and comparators. 79
6.4 18 of the 130 PEs required to implement the CM shown in Figure 6.1 using the baseline architecture. The full pipeline structure can be automatically generated directly from a CM. . . . 81
6.5 Percentage of CMs that will fit onto hardware in a given year . . . 83

7.1 A high-level block diagram of processor array architecture with two processing modules. The number of processing modules can be scaled as shown with dashed lines. . . . 87
7.2 Block diagram of a single PM. Dashed lines represent components that are only needed on the first PM. . . . 88
7.3 Write interface configuration for a single PM with six individual memories . 93
7.4 Switched read interface for two PMs, each with six individual memories . . 94
7.5 Flow Diagram of Hu's scheduling algorithm . . . 97
7.6 (a) An example task graph with distance labels; (b) schedule for task graph in (a) using unlimited processors; (c) schedule for task graph in (a) using two processors . . . 98
7.7 Flow Diagram of Modified Hu's scheduling algorithm . . . 100

7.8 (a) An example task graph with distance labels; (b) schedule for task graph in (a) using unlimited processors and accounting for a 10 time unit computational latency; (c) schedule for task graph in (a) using two processors and accounting for a 10 time unit computational latency 101
7.9 (a) An example task graph with distance labels; (b) schedule for task graph as shown in Figure 7.8, but with a memory conflict at time $t = 10$; (c) schedule for task graph with memory conflict resolved. Note that N1 was moved to $t = 11$ and its dependent N0 was moved to $t_{N0} = t_{N1} + l = 21$. 104
7.10 The length of the scheduled computation decreases as the number of processors available for the computation increases (shown on a log-log scale). . . . 105
7.11 The speedup shows the diminishing returns as more processors are added to the processors array. The efficiency decreases as more processors are added to the processors array, indicating that more idle time is inserted into the schedule as the number of processors increases. 107
7.12 The maximum amount of live memory remains fairly consistent regardless of the number of processors. 108
7.13 As the number of processors in the processor array architecture increases, the average memory required per processor decreases. 109
7.14 Memory trace for CM RF00016 110
7.15 Memory trace for CM RF00034 111
7.16 Resource requirements for the processing elements and Banyan switches in different sized processor arrays. 114
7.17 Resource requirements for different sized processor arrays. Note that the Batcher switches account for the majority of the resources. 114
7.18 The estimated time to align a 1 million residue database to four different CMs using varying numbers of processors. The shortest bar represents the time to compute the results using 256 processors. The longest bar represents the time to compute the results using a single processor. 119

7.19 A comparison of the estimated time to align a 1 million residue database using the processor array architecture versus the time required for INFERNAL. The shorter bar in the INFERNAL categories represents the time when using the QDB heuristic. The longer bar represents the time without QDB. 119

ABSTRACT OF THE DISSERTATION

Techniques for Hardware-Accelerated Parsing
for Network and Bioinformatic Applications

by

James M. Moscola

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2008

Ron K. Cytron, Chairperson,

Young H. Cho, Co-Chair, John W. Lockwood, Co-Chair

Since the development of the first parsers, parsing has generally been considered a software problem. Software parsers have been developed for many different uses including compiling software, rendering web pages, and even translating languages. However, as new technologies and discoveries emerge, traditional software techniques for parsing data are either not fast enough to keep up with data rates, or simply take too long to produce results in a reasonable period of time. This dissertation discusses techniques and architectures for accelerating parsing in two different domains. One requires parsing of high-speed streaming data. The other requires parsing of very large data sets with a computationally complex parsing algorithm.

The first part of this dissertation focuses on architectures for accelerated parsing of network data. As network rates continue to increase and the volume of data transferred across networks escalates, it will become progressively more difficult for software packet examination techniques to maintain the required throughput. Couple this with the development of new networking technologies, such as content-based routing and publish/subscribe networks, and it is clear that high-speed architectures for parsing packet payloads are required. New architectures for both pattern-matching and parsing are presented and compared to existing architectures. Additionally, two example applications are presented. The first is a simple content-based router. The second is an email parser capable of delineating and extracting user-specified portions of email messages.

The second part of this dissertation examines another parsing problem where high throughput is desired, but for which many parses are possible for each input, and all such parses must be considered. The difficulty of the problem is amplified by the large volumes of data that must be parsed. More specifically, this work investigates techniques and architectures suitable for accelerating the complex parsing algorithm used for discovering new RNA molecules in genome databases. Two different hardware architectures are presented and evaluated against a well-known software suite.

Chapter 1

Introduction

Throughout the history of computer science the problem of parsing data to derive it's meaning have traditionally been done in software. Parsers are used in a wide variety of applications including, but not limited to, compiler construction, text analysis, language translation, and bioinformatic analysis. A great deal of research, time, and effort has gone into making parsers highly optimized utilities capable of outputting the best results in the shortest time. However, as existing technologies improve and new technologies emerge it is becoming clear that the existing class of software parsers is insufficient for handling all of today's parsing requirements. For some parsing applications, the need for high-speed hardware-accelerated parsers has arisen. This dissertation focuses on two of those applications, network applications and bioinformatic applications, and proposes techniques for accelerating those applications using custom hardware architectures.

1.1 Parsing in Network Applications

One area that accelerated parsing techniques can be beneficial is in network applications. Up until recently, there was not much of a need for high-speed parsers to process network traffic. Most networks merely transported data from one location to another, a task that simply requires the reading of fixed length address fields in fixed offsets of the data. More complex data processing, such as content inspection, was typically relegated to lower bandwidth network links [61], constrained to sampling techniques [41, 54], or handled using clusters of software-based network appliances [72, 70].

However, with new technologies such as content-based routing [4, 14], publish/subscribe networks [5, 60], and semantic networks [31] emerging, faster and more intelligent techniques will soon be required to process network data. The problem is further compounded by the continual increase in network traffic, which is the result of a increasing number of users and increasing connection speeds. As more users begin to use these new technologies the current methods used for content inspection will become insufficient and new, faster techniques will be required.

This work proposes high-speed hardware architectures for content inspection of network data. The work first examines existing high-speed pattern matching architectures and utilizes them as a basis for a new architecture. Simple pattern matching is then augmented with a technique for converting large grammars for regular languages into compact parsers that enable the derivation of semantic meaning for patterns found within network data. The compact parsing structure can automatically be generated given a grammar specification. Finally, two example applications are provided to illustrate the functionality of the combined pattern matcher and parsing architectures.

1.2 Parsing in Bioinformatic Applications

Another area that can benefit from accelerated parsing techniques is the area of bioinformatics, the science of analyzing biological data. Many problems in the field of bioinformatics use common parsing algorithms to compare sequences of biological data. However, the parsing algorithms used typically have $O(n^2)$ or $O(n^3)$ computational complexities. When coupled with the vast amount of biological data that is continually being generated in the field of bioinformatics, these parsing algorithms can be very time-consuming. On some inputs an $O(n^3)$ parsing algorithm may take days, months, or even years to produce results using today's general purpose processors. Such prolonged computations can severely inhibit research in the areas of bioinformatics that require these computationally complex parsing algorithms.

Due to the strong interest in the area of bioinformatics, many researchers have gone to great lengths to achieve faster results. Some researches have developed heuristics that

produce results more rapidly at the expense of accuracy [73, 74, 53]. Others have developed computing clusters in an effort to increase their computing capacity [45], an expensive and unattainable goal for small research labs.

This work proposes a different solution. More specifically, this work proposes the use of high-speed custom hardware architectures to accelerate an $O(n^3)$ parsing algorithm used for detecting homologous RNA sequences. The use of a custom hardware architecture can decrease the time required to parse biological sequences while simultaneously eliminating the need for costly computing clusters. This, in turn, can help to make research in bioinformatics accessible to even the smallest research labs. With so much biological data available for analysis, additional research labs can only help to advance the study of RNA sequences at a pace that exceeds that of today.

1.3 Reconfigurable Hardware

As mentioned in the previous section, there are many ways in which one might go about accelerating the solution to different problems. Possibilities include developing heuristics, using computing clusters, or developing all new algorithms. In addition, since the advent of reconfigurable hardware, custom hardware architectures have quickly become yet another option. Reconfigurable hardware offers advantages over both software solutions and ASIC solutions. A reconfigurable architecture can offer computational power that far exceeds that of software running on a general purpose processor without sacrificing the flexibility required for future modifications. Over an ASIC solution, reconfigurable hardware offers a lower cost for small scale deployments plus the ability to make design modifications when necessary. However, this typically comes at the expense of a lower operating frequency.

One of the most common and flexible types of reconfigurable hardware is the Field-Programmable Gate Array (FPGA). FPGAs consist of a uniform sea of configurable logic blocks (CLBs), each of which can be individually configured and combined with other CLBs to construct the logic necessary for an architecture.

Many of the architectures presented in this work were developed as reconfigurable architectures where the logic required is generated given some set of inputs such as a set

of strings or grammars. That logic can then be mapped onto reconfigurable hardware such as an FPGA. This approach allows for highly-pipelined architectures that are tailored to the exact requirements of the input. However, not all parsing problems require this type of reconfigurable architecture. One such problem, that of parsing bioinformatic data, is discussed in Chapter 7. While the architecture in Chapter 7 does have some aspects which can benefit from the reconfigurability of an FPGA, it is also well-suited for an ASIC implementation.

1.4 Contributions

The contributions of this dissertation are in the area of techniques and architectures for high-speed parsing of network and bioinformatic applications. Specific contributions are:

- **Pattern Matching**
  - proposes regular expression extensions to an existing string matching architecture
  - introduces a compact and scalable regular expression pattern matching architecture
  - provides a comparison of the new pattern matching architecture to existing pattern matching architectures
- **Parsing of Regular Languages**
  - introduces a high-speed architecture for parsing regular languages
  - describes a technique to automatically generate regular language parsers from a grammar specification
  - describes an implementation of a content-based router using the proposed regular language parser
  - describes an implementation of an email processor using the proposed regular language parser

- **Parsing of Stochastic Context-Free Grammars**
  - provides an analysis of the hardware required for processing covariance models in hardware
  - introduces a technique for mapping RNA alignment computations directly into a high-speed pipelined architecture
  - introduces a second architecture for RNA alignment that utilizes an array of custom processors
  - describes a technique for scheduling RNA alignment computations onto a processor array architecture
  - provides an analysis of the scheduling technique used to schedule RNA alignment computations
  - provides an analysis of the scalability of the processor array architecture
  - provides a comparison of the processor array architecture to the INFERNAL software package
  - provides a discussion on future directions for the processor array architecture research

1.5 Overview of Dissertation

The remainder of this dissertation is organized into two parts. Part I focuses on parsing architectures that can be used for network applications. Chapter 2 provides a description of two existing pattern matching architectures followed by a new scalable regular expression pattern matcher. This is followed by an approach for hardware-accelerated parsing of regular languages in Chapter 3. Chapter 4 illustrates two example network applications developed using the techniques from Chapter 2 and Chapter 3, including the implementation of a content-based router and an email message parser.

Part II of this dissertation focuses on parsing architectures for RNA secondary structure alignment. Chapter 5 starts with a background on RNA secondary structure alignment and the probabilistic parsing algorithm utilized for RNA alignment. Chapter 6 provides a

description of a baseline architecture that maps the structure inherent in the parsing algorithm directly onto hardware. This is followed by a more general approach that schedules computations from the parsing algorithm onto an array of processors in Chapter 7 .

Finally, Chapter 8 provides a brief summary of the work presented in this dissertation. Suggestions for future work are also presented in Chapter 8.

Part I

Hardware Architectures for Pattern Matching and Parsing in Network Applications

Chapter 2

Hardware-Accelerated Pattern Matching

Parsing consists of two main steps, lexical analysis and syntactic analysis. In this work, lexical analysis is mapped into hardware using hardware-based pattern matching techniques.

While studying several of the more prominent FPGA-based pattern matching techniques, it was determined that there was an opportunity to combine features from two of them into a single new technique. The new hybrid pattern matching technique, which is called Timed Segment Matching (TSM), has greater regular expression capabilities without sacrificing performance or requiring additional resources. The two contributing pattern matching techniques, the pipelined regular expression chain [56] and the pipelined character grid [8], are described in detail in this chapter.

Section 2.1 starts by discussing related work in the area of hardware-accelerated pattern matching. Section 2.2 then introduces a pre-decoder common to the pattern matchers in this chapter. This is followed by a description of the regular expression chain in Section 2.3. The second contributing architecture, the pipelined character grid, is described in Section 2.4. This section also describes modifications that were made to the pipelined character grid to enable basic regular expression functionality. Finally, Section 2.5 describes the new TSM technique along with a comparison of it to the contributing architectures.

2.1 Related Work

In recent years, a great deal of work has been done in the field of high-speed hardware-accelerated pattern matchers. In 2001, Sidhu and Prasanna presented a method for mapping regular expressions into nondeterministic finite automata (NFA) that could then be mapped onto FPGA hardware [56]. Franklin *et al.* illustrated how this technique can be used to map the patterns found in the Snort [59] database onto an FPGA [28]. An additional pattern matching architecture for Network Intrusion Detection Systems (NIDS) was presented by Cho in [16]. In [50] regular expression patterns were converted into deterministic finite automata (DFA) and mapped onto an FPGA. In [20], Clark and Schimmel introduced the idea of pre-decoding input characters into single bit lines to reduce the number of comparators required for matching patterns. Sourdis presented the idea of using a pipelined comparator for matching patterns in [67]. In [8], Baker presented an architecture where all string comparators are connected to a single pipeline of decoded characters. In other work, Baker combines a small microcontroller with a bit-split architecture to create a high-speed regular expression matcher with the flexibility to modify the pattern set on-the-fly [7]. Work by Bispo expanded on Sidhu's work in [56] by adding support for Perl-compatible regular expressions [10]. Other work by Brodie *et al.* presents a new finite state machine representation capable of making state transition decisions while processing multiple characters per cycle [12].

2.2 Pre-Decoded Character Bits

Common among many of the FPGA-based pattern matchers is the idea of pre-decoding characters. Since the regular expression patterns contain a fixed alphabet of characters, a pre-decoder can be used to reduce the amount of space required by the design [20]. For ASCII, the decoder logic is an 8-bit input AND gate matched to the bits needed to identify each character. An example for the 8-bit ASCII character "A" (hexadecimal 0x41) is shown in Figure 2.1. Each letter used in the pattern is decoded uniquely to assert a single bit. The conversion from 8-bits per character to a single bit per character significantly reduces the

amount of logic and routing resources required by the design [20]. Since there is a relatively small penalty for a large fanout in FPGA, the pre-decoders are used in most of the recent pattern matching architectures.

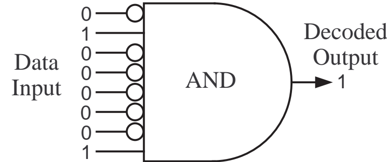


Figure 2.1: Decoder for 8-bit ASCII character “A” (0x41)

2.3 Pipelined Regular Expression Chain

The pipelined regular expression chain is a technique for mapping regular expressions into a hardware representation of an NFA as described by Sidhu and Prasanna in [56]. A regular expression chain is constructed from a regular expression using a method similar to what one might use to construct an NFA from a regular expression. Logic elements are first allocated for sub-expressions of the regular expression. Those logic elements are then

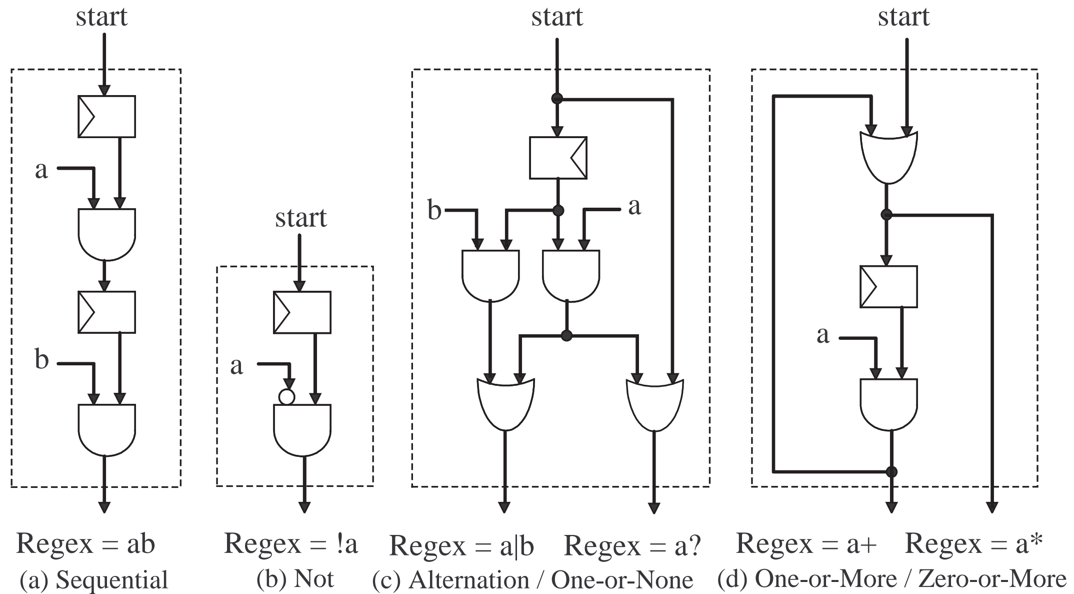


Figure 2.2: Constructing regular expressions

connected together to form a chain of elements capable of matching the complete regular expression.

The logic elements used to build a regular expression chain are shown in Figure 2.2. Figure 2.2a illustrates how two characters in a regular expression, each represented using a single D flip-flop (DFF) and an AND gate, are concatenated to form a chain capable of matching the string "ab". The input values "a" and "b" are single bit line representations of characters that have been decoded from an incoming data stream. On each clock cycle, as the incoming data stream advances by one character, a potential match signal advances through a regular expression chain. For example, a *start* signal is asserted prior to receiving a data stream containing the string "ab". On the first clock cycle, the regular expression chain receives an "a" from the data stream causing the value of the first AND gate to be valid. On the second clock cycle, the second DFF in the chain is valid and the regular expression chain receives a "b" from the data stream causing the value of the second AND gate to be valid. When the second AND gate is valid, the regular expression chain has indicated that a match has been found. Additional DFFs and AND gates can be added to the chain to match longer strings.

The example described above can easily be extended to recognize a richer set of regular expression patterns with functions such as *not*, *or*, *one-or-none*, *one-or-more*, and *zero-or-more*. These functions are represented as logic primitives in Figure 2.2. One can instantiate combinations of these elementary logic templates in a chain-like fashion, connecting the output of one element to the input of the next, to build regular expression detectors in hardware [56].

2.4 Pipelined Character Grid

An architecture by Baker attempts to optimize the use of logic by buffering the pre-decoded characters into a pipelined grid structure [8]. Given such a grid of decoded characters, one can detect patterns by ANDing all of the corresponding decoded bits from different stages of the pipeline as shown in Figure 2.3. Therefore, the pattern length must be less than or equal to the length of the pipeline. Since similar patterns tend to reuse the decoded outputs,

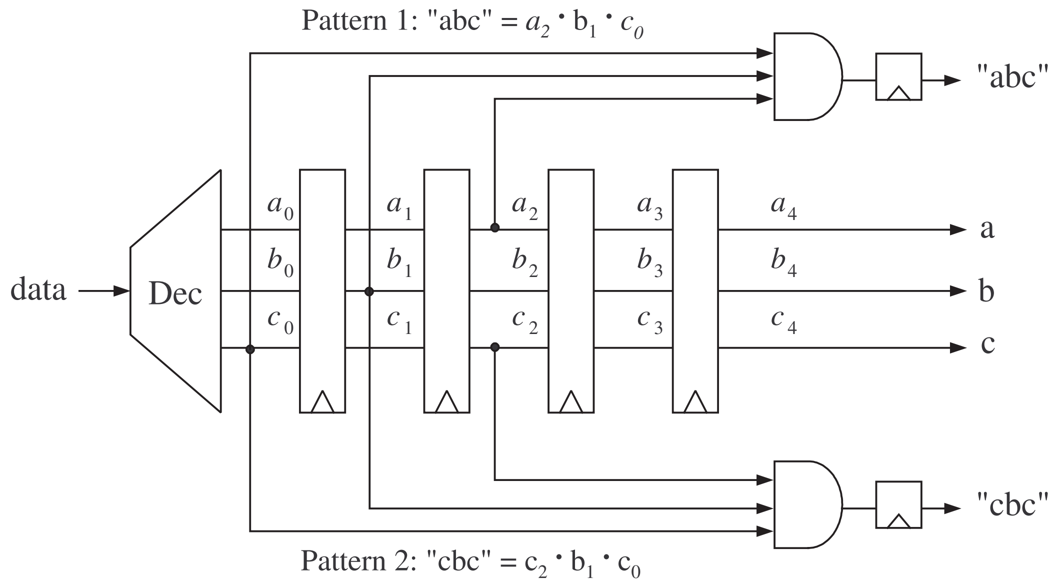


Figure 2.3: Simple example of pattern matcher using the pipelined character method

the size of the grid is relatively constant. Therefore, the DFF resource requirement for this method is proportional to the number of patterns instead of the number of characters. Given four input look-up-tables (LUTs), the number of LUTs for long patterns can be reduced to a quarter of the pipelined regular expression chain method.

The pipelined character grid can also be scaled by widening the input width. Since patterns can begin at any given alignment, a duplicate copy of the character decoder and pipeline registers must be instantiated for each input byte alignment. For each pattern, its corresponding AND gate that detects a pattern must be replicated for each alignment. Then the outputs of the AND gates need to be ORed together to detect a match. An example of how the pipelined character grid can be scaled is shown in Figure 2.4. The example shown has twice the input width as the example shown in Figure 2.3, therefore it must scan for each pattern at two possible starting positions. Due to logic reuse, this architecture tends to yield denser designs as the size of the pattern set increases. However, this logic compression is enabled by sacrificing the regular expression capabilities possible with regular expression chains.

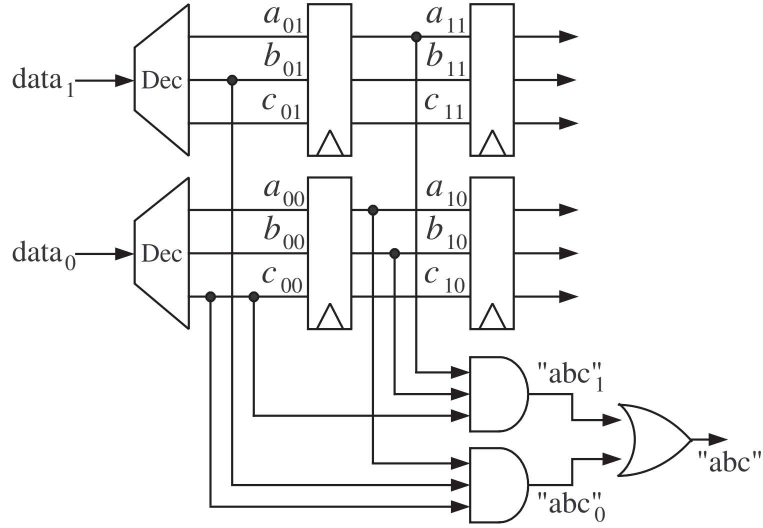
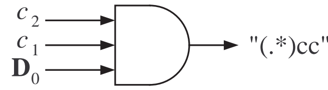


Figure 2.4: Scaled pipelined character grid

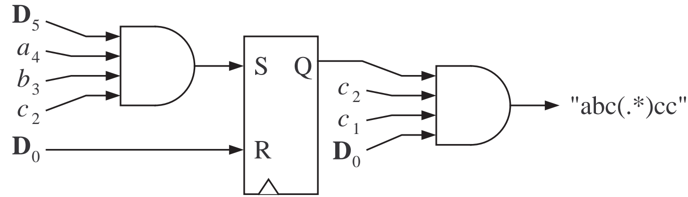
2.4.1 Regular Expression Extensions to the Pipelined Character Grid

While examining the pipelined character grid structure, it was determined that the structure is not flexible enough to allow for the detection of full regular expressions without incurring excessive additional logic. However, simple modifications to the comparator logic can allow detection of certain regular expressions involving wild card characters with little or no additional logic. An inverted delimiter represents all characters that are not a delimiter. Therefore, using this symbol allows the wild card character to be detected.

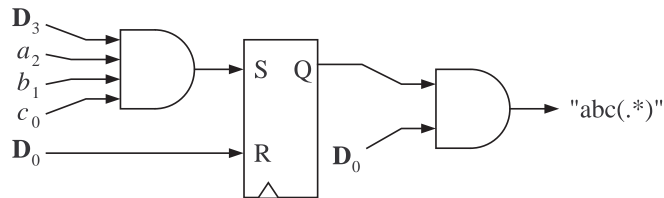
A sequence of wild cards can also be detected by adding a clocked set-reset flip-flop to the comparator logic. Different variations of regular expressions with wild cards are shown in Figure 2.5. As an example, the flip-flop in Figure 2.5b is set when the “abc” substring of the pattern matches. The flip-flop remains active while in the “.*” phase of the match. When the “cc” substring of the pattern followed by a delimiter is detected the pattern has been detected. Any incoming delimiter will reset the flip-flop.



(a) A sequence of wild card characters followed by "cc"



(b) A sequence of wild card characters surrounded by "abc" and "cc"



(c) "abc" followed by a sequence of wild card characters

Figure 2.5: Patterns with wild card characters

2.5 Timed Segment Matching

In studying the above techniques, it was discovered that ideas from both architectures can be merged to create an even more space efficient regular expression pattern matcher [51]. The new architecture consists of a pipeline of decoded characters in conjunction with primitives similar to those used by the regular expression chain. This technique allows efficient use of FPGA resources while still being scalable and capable of matching regular expressions.

A pattern matcher in Baker's pipelined character grid detects an entire pattern at one instance by simply connecting all the corresponding decoded bits to a single AND gate [8]. While examining the design, it became apparent that there was an opportunity to compress the size of the pipeline by matching small segments of a pattern at a time. By matching small segments of a pattern on subsequent clock cycles, some of the pipeline stages are reused allowing the grid structure to be compressed. While formulating the

general effect of this, it was determined that the change was equivalent to combining the pipelined chain with the character grid architecture. Another way to express the approach is to combine four consecutive characters from a pipelined regular expression chain into a single LUT/DFF pair. In order to accomplish this, incoming characters are buffered using a pipelined character grid whose registers are reused for each of the patterns in the pattern set. This matching technique is referred to as Timed Segment Matching (TSM).

In most FPGAs, a basic block consists of a 4-input LUT followed by a DFF and other supporting discrete gates. The design with shortest critical path would be the one where each pipeline stage would consist of only one level of basic blocks. With such a design criterion, the string comparator to match an N character pattern with AND gates, as shown in Figure 2.6a, takes a minimum of $\Sigma_{i=1}^{\lceil log_4 N \rceil} \lceil \frac{N}{4^i} \rceil$ gates. Note that this number does not include the logic required by the decoded character pipeline. By connecting the AND gates in a tree-like structure, the minimum latency is $\lceil log_4 N \rceil$ stages. In addition to pipelined stages of AND gates, one must consider detection latency to indicate where the pattern starts or ends. Since pattern lengths can vary in practice, detection signals of shorter patterns must be delayed by $(S-1)-u$ clock cycles, where S is the number of stages in the longest pattern in the patten set and u is the number of stages in the short pattern of interest. This delay ensures that patterns are detected in the order in which they appear in the data stream and that short patterns cannot indicate a match prior to longer patterns that preceded them in the data stream.

On the other hand, the logic architecture for the TSM method is simpler. The TSM architecture starts with a character pipeline similar to the one used in the pipelined character grid architecture. String comparators, similar to the regular expression chains in Sidhu's architecture, are used to detect each pattern in the pattern set (Figure 2.6b). However, unlike Sidhu's architecture, the TSM architecture utilizes the character pipeline to buffer characters so that multiple characters can be matched by each AND gate. This allows more efficient use of logic resources. String comparators, like the one shown in Figure 2.6b are generated by chaining together enough 4-input AND gates for the pattern of interest. The first AND gate in the chain is used to match the first four characters of

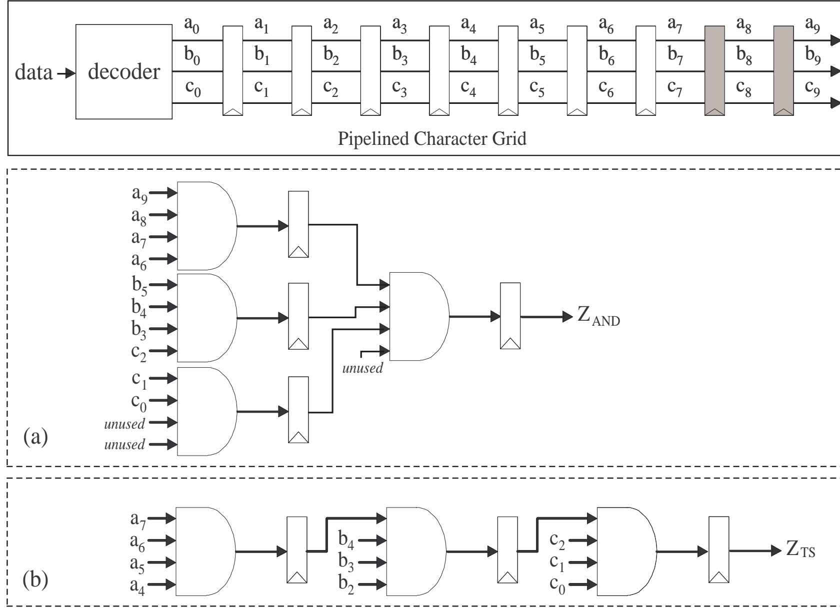


Figure 2.6: Fast FPGA pattern matchers for "aaaabbbccc" using a pipelined character grid; (a) Z_{AND} uses AND gate matching and (b) Z_{TS} uses TSM.

the pattern. Subsequent AND gates in the chain only match 3 characters each since they must also include the result of the previous AND gate. Pseudocode for generating a string comparator for the TSM architecture is shown in Figure 2.7.

Matching an N character pattern using the TSM method yields a minimum of $\lceil \frac{N-1}{3} \rceil$ logic blocks with a latency that is directly proportional to the length of the string. Through simple analytical observations it can be seen that $\Sigma_{i=1}^{\lceil log_4 N \rceil} \lceil \frac{N}{4^i} \rceil \geq \lceil \frac{N-1}{3} \rceil$ for $N > 0$. This indicates that the logic requirements for string comparators in the TSM architecture are less than or equal to logic requirements of string comparators in the pipelined character grid. More interestingly, this is achieved while simultaneously creating an architecture that is capable of matching both strings and regular expression patterns. Another positive consequence of this method is that information about a match is carried through the pipeline allowing the architecture to keep track of partial matches as they occur. This allows the TSM architecture to indicate that a match has occurred immediately after the last character of a pattern enters the pipeline, regardless of the length of the pattern.

```
L = length(input_string);

for (j=1; j<=L; j++) {           // for each character in the input string
  gate_num = ceiling((j-1)/3);   // determine which AND gate to connect character to
  C = input_string[j];           // set C to the jth character of input string

  if (j==1) {                    // if this is the first character of the string
    S = ceiling((L / 3) * 2) + ((L % 3) % 2); // determine what stage of the character
                                             // pipeline to get the character from
    // create first 4-input AND gate in the string comparator
    // connect output of character C from stage S of the character pipeline to AND gate
  }
  else if ((j%3==2) && (gate_num >= 2)) {     // if the previous AND gate is full
    S++;  // account for register delay in the string comparator
    // connect output of previous AND gate to input of new AND gate in string comparator
    // connect output of character C from stage S of the character pipeline to AND gate
  }
  else {
    // connect output of character C from stage S of the character pipeline to AND gate
  }
  S--;  // decrement position of character pipeline to account for consumed character
}
```

Figure 2.7: Pseudocode to generate a string comparator for the TSM architecture

2.5.1 Regular Expression Pattern Detection

Simple AND gate based detection requires a pipelined character grid where the number of pipeline stages is greater than or equal to the longest pattern. Since regular expressions can represent strings that are infinitely long, it is impossible for the simple AND gate matcher and character grid to match all regular expressions. However, TSM modifies the detection method by adding a structure similar to the pipelined regular expression chain. Given such a pattern matching structure, the primitives for the pipelined regular expression chain can be adapted for use in the TSM architecture. The basic regular expression operations are illustrated in the examples shown in Figure 2.8. The regular expression operations (e.g. *zero-or-more*) are shown in the shaded regions. The operations are shown as part of larger patterns to better illustrate how they are used.

Both the *zero-or-more* operation (Figure 2.8a) and the *one-or-more* operation (Figure 2.8b) require feedback paths that allow the chains to iteratively match repeating substrings. Additionally, these two operations require that additional registers be placed in that feedback path. The additional registers ensure that the pipelined character grid has

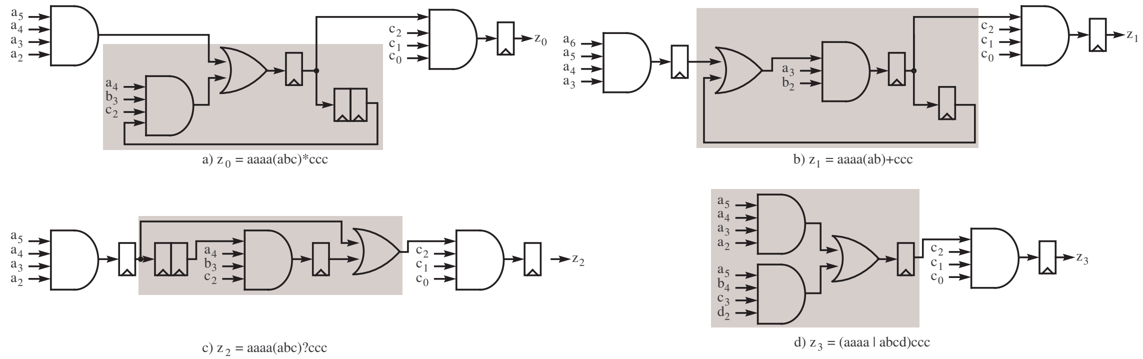


Figure 2.8: Regular expression operations in the timed segment matching architecture

enough time to receive the characters required for the next iteration of the regular expression operation. Note that the total number of registers required in the feedback path is equal to the number of characters involved in the given regular expression operation. For example, in Figure 2.8a the regular expression "aaaa(abc)*ccc" contains the *zero-or-more* operation. The number of characters included in the *zero-or-more* operation is three. This means that the total number of registers required in the feedback path is also three.

The regular expression operation in Figure 2.8c is the *one-or-none* operation. This operation does not include a feedback loop, but it does require additional registers similar to the *zero-or-more* and *one-or-more* operations. Again, the number of additional registers required is equal to the number of characters included in the regular expression operation. The final operation shown in Figure 2.8d, the *alternation* operation, is the simplest of the regular expression operations. It requires the addition of only a single OR gate to the chain. No additional delay registers are required.

Accordingly, it is easy to see how more complex regular expressions can be detected using the basic components for each of the regular expression operations. An example using both the *zero-or-more* operation and the *alternation* operation is shown in Figure 2.9. Notice that the different size substrings in the *alternation* operation require a different number of delay registers. Thus an extra DFF is inserted at the input of the larger substring.

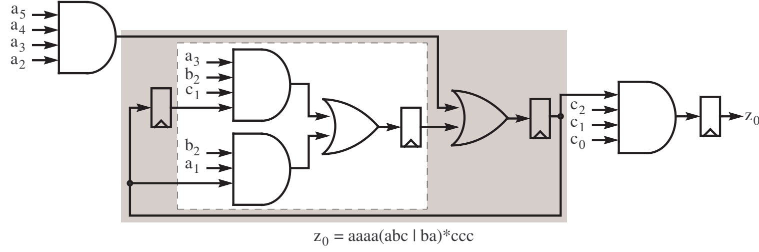


Figure 2.9: TSM example for “aaaa(abc|ba)*ccc”

2.5.2 Scalable Architecture

The TSM architecture can also be scaled while still maintaining its small size and the ability to scan for regular expression patterns. First, the character grid pipeline must be duplicated for each character of the input width to provide decoded bits for every alignment, as shown in Figure 2.10. Then the pipelined regular expression chains are constructed for each alignment. Although there are specific techniques to handle different regular expressions, this section focuses on the architecture for *zero-or-more*. Once its concept is understood, one can easily elaborate to build circuits for other operations (e.g. “a+” is equivalent to “aa*”).

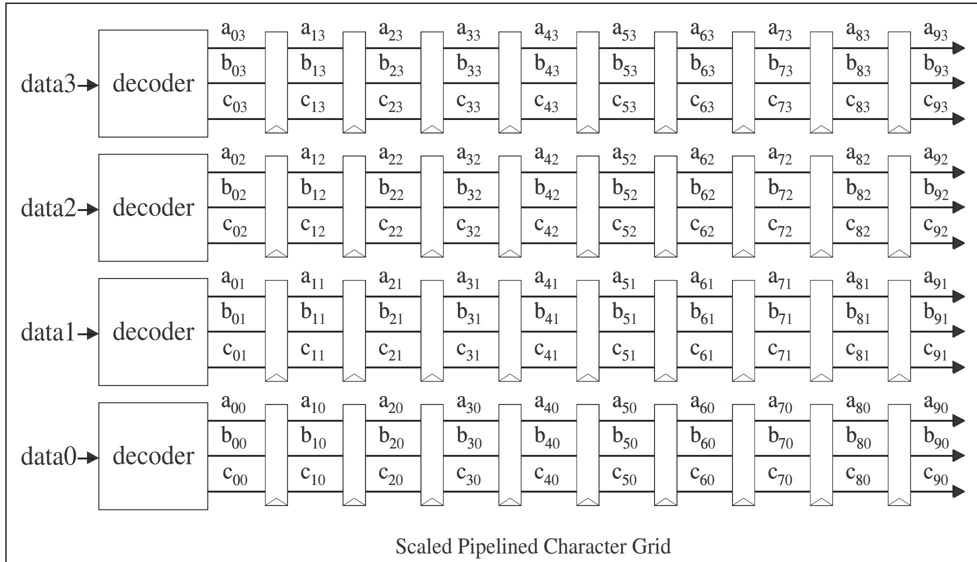


Figure 2.10: Wide pipeline grid for 4× scaled architecture

For a fixed string, the matching circuitry at each byte offset can be treated as an independent engine until they are ORed together at the end. However, the logic design complexity can grow rapidly when one attempts to scale regular expressions. This is because expressions that immediately follow *alternation*, *one-or-none*, *one-or-more*, or *zero-or-more* operations may start at multiple alignments. When the matchers are instantiated for all possible alignments, the entire circuitry can become exponentially large. Fortunately, most of the instantiations turn out to be duplicated logic which can be combined and reused.

To clearly understand the design process, two examples of the *zero-or-more* regular expression operation on a 4-byte input datapath are presented. Four copies of the character grid are instantiated as shown in Figure 2.10. The pre-decoded characters bits from these grids are then used by the pattern matching pipelines. Construction of a scaled chain works in a similar fashion to that of the single character wide version, by concatenating AND gates together (or regular expression primitives as shown in previous sections). Determining the location of the character grid from which to retrieve a character is done in a similar fashion to the single character wide TSM architecture. However, in the scaled version, four characters advance through the character grid on each clock cycle. This means that if the scaled chain is currently examining the *data2* position of stage 0 in the scaled pipeline grid, on the next clock cycle it must examine the *data1* position of stage 1. This ensures that all characters get examined while processing a data stream.

The simplest example of a scaled regular expression pattern matcher in the TSM architecture occurs when the regular expression operation includes the same number of characters as the width of the input. In the example in Figure 2.11a, notice that the *zero-or-more* regular expression operation ("*") is operating on the four character string "bbbb". In this case, the substring in the "*" operation is the same length as the width of the data input. Since any number of iterations of the "*" operation does not change the alignment of the immediately following substring "ccc", the pattern matchers do not cross over to the other parallel matchers. In the example in Figure 2.11b, the "*" is operating on the single character string "b". In this case, the substring in the "*" operation is a single character.

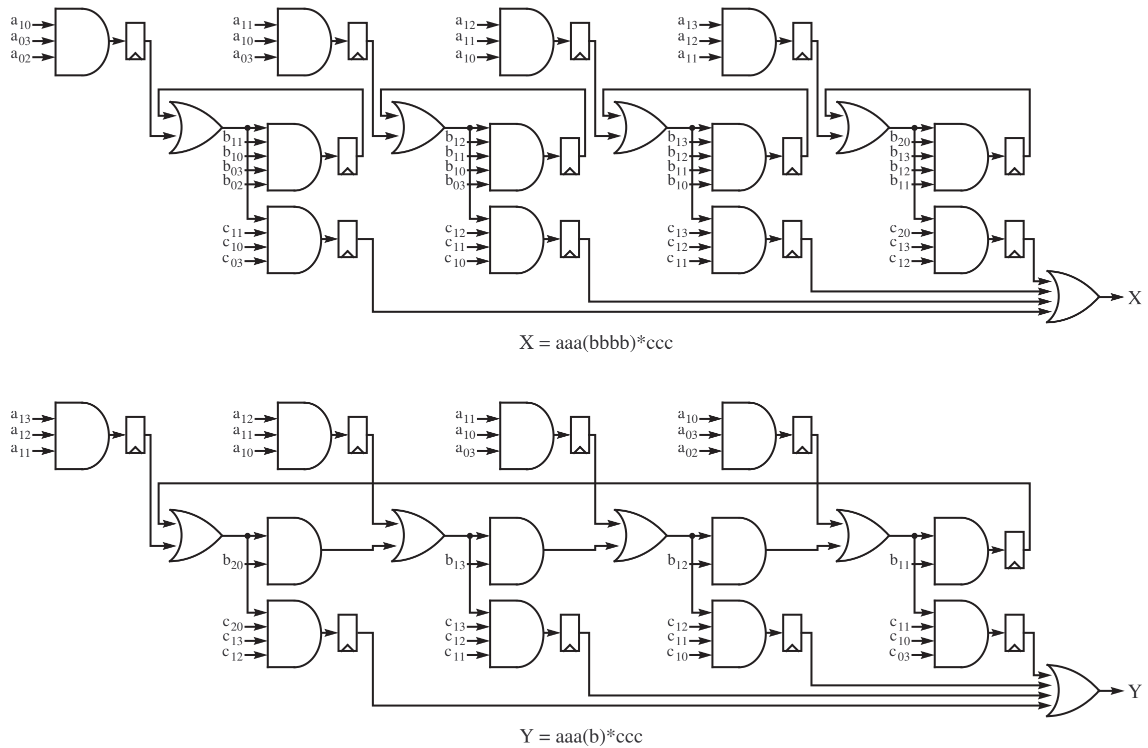


Figure 2.11: Examples of the *zero-or-more* regular expression operation in the scaled TSM architecture

Therefore, every iteration of “b” changes the expected alignment for the “ccc” substring. As a result, every matcher is connected to its adjacent matcher.

The above examples are regular expressions that are less than or equal to the width of the pipeline. However, the TSM architecture is not restrictive. For substrings that are greater than the width of the pipeline, the substring can simply be broken down into smaller substrings while still applying the rules described in the previous sections. The example in Figure 2.12 shows the iterative loop required for a regular expression operation that is larger than the pipeline width.

2.5.3 Implementation

Using an automatic VHDL generator, hardware was generated for the five different pattern matching architectures described in this chapter. The architectures implemented in this

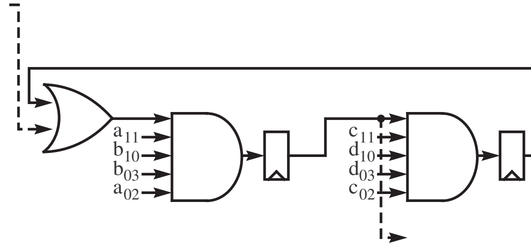


Figure 2.12: TSM architecture for …"(abbacddc)*"…

section include the 8-bit regular expression chain[1], the 8-bit and 32-bit pipelined character grid[2], and the 8-bit and 32-bit version of timed segment matcher architectures. To evaluate the five architectures, each was generated with five different pattern sets. The pattern sets range in size from 300 bytes to 3000 bytes.

Additionally, architectures were generated that scan for full regular expressions using the 8-bit regular expression chain, and the 8-bit and 32-bit TSM architectures. The regular expressions were randomly generated in the form of "abcd(efgh)*ij". Because the 8-bit and 32-bit pipelined character grid architectures do not support regular expressions, no regular expression hardware was generated for those architectures.

String Patterns

The hardware for each pattern set was synthesized, placed, and routed on Xilinx Virtex 4 LX200 -11 chips. Synplicity's Synplify Pro v8.1 was employed for synthesis. Placing and routing was completed using version 7.1 of the Xilinx back-end tools. Tables 2.1 and 2.2 show the complete results for all the different architectures for each of the pattern sets that were implemented.

As with the regular expression chain and the pipelined character grid, the TSM architecture presented here allows for very high clock frequencies. Both the 8-bit and 32-bit TSM architectures are capable of running at similar clock frequencies to those of the equivalent size chain or pipelined grid architectures. These high clock frequencies translate to throughputs of up to 3.46 Gbps for the 8-bit architecture and 12.90 Gbps for the 32-bit

[1]Implemented as described by Sidhu [56]

[2]Implemented as described by Baker [8]

8-bit Input (1 Character Wide)															
	Regex Chain					Pipelined Grid					TSM				
# of Chars	Freq MHz	TP Gbps	LUTs	LUTs /Byte	DFFs	Freq MHz	TP Gbps	LUTs	LUTs /Byte	DFFs	Freq MHz	TP Gbps	LUTs	LUTs /Byte	DFFs
300	436	3.49	319	1.06	471	438	3.50	172	0.57	730	432	3.46	168	0.56	600
600	433	3.46	572	0.95	747	399	3.19	266	0.44	1032	429	3.43	290	0.48	864
1200	441	3.53	1016	0.85	1212	436	3.49	422	0.35	1508	430	3.44	504	0.42	1289
2100	426	3.41	1672	0.80	1886	403	3.22	684	0.33	2197	435	3.48	856	0.41	1921
3000	411	3.29	2287	0.76	2503	420	3.36	931	0.31	2818	432	3.46	1202	0.40	2497

Table 2.1: Device utilization for pipelined regular expression chain and pipelined character grid architectures

32-bit Input (4 Characters Wide)										
	Pipelined Grid					TSM				
# of Chars	Freq MHz	TP Gbps	LUTs	LUTs /Byte	DFFs	Freq MHz	TP Gbps	LUTs	LUTs /Byte	DFFs
300	398	12.74	586	1.95	1609	403	12.90	542	1.81	1153
600	382	12.21	913	1.52	2292	391	12.52	942	1.57	1731
1200	368	11.76	1511	1.26	3426	395	12.64	1676	1.40	2766
2100	377	12.05	2433	1.16	5049	374	11.98	2773	1.32	4315
3000	359	11.48	3372	1.12	6576	384	12.29	3832	1.28	5831

Table 2.2: Device utilization for the architectures

architecture. It is worth noting that as the size of pattern sets increase, the clock frequency for most of the different architectures decreases slightly. This decrease in frequency is attributed to the increasing fanout of the decoded character bits as the size of the pattern set increases.

Including all of the decoder logic, the 8-bit TSM architecture utilizes only 0.40 LUTs/byte of the pattern for the 3,000 byte pattern set. This is almost half the size of the regular expression chain architecture which requires 0.76 LUTs/byte for the same pattern set. The 8-bit pipelined character grid is slightly smaller than the TSM architecture, requiring only 0.31 LUTs/byte. The 32-bit TSM architecture and the 32-bit pipelined character grid are also similar in size, requiring 1.28 LUTs/byte and 1.12 LUTs/byte respectively. The graph in Figure 2.13 shows the number of LUTs required by each of the different architectures. The graph shows that the LUT resource requirement for the 8-bit regular expression chain is significantly larger than either of the other two 8-bit architectures. The 8-bit TSM

architecture, while slightly larger than the 8-bit pipelined character grid, manages to stay close in size even with the added ability to do regular expression pattern matching.

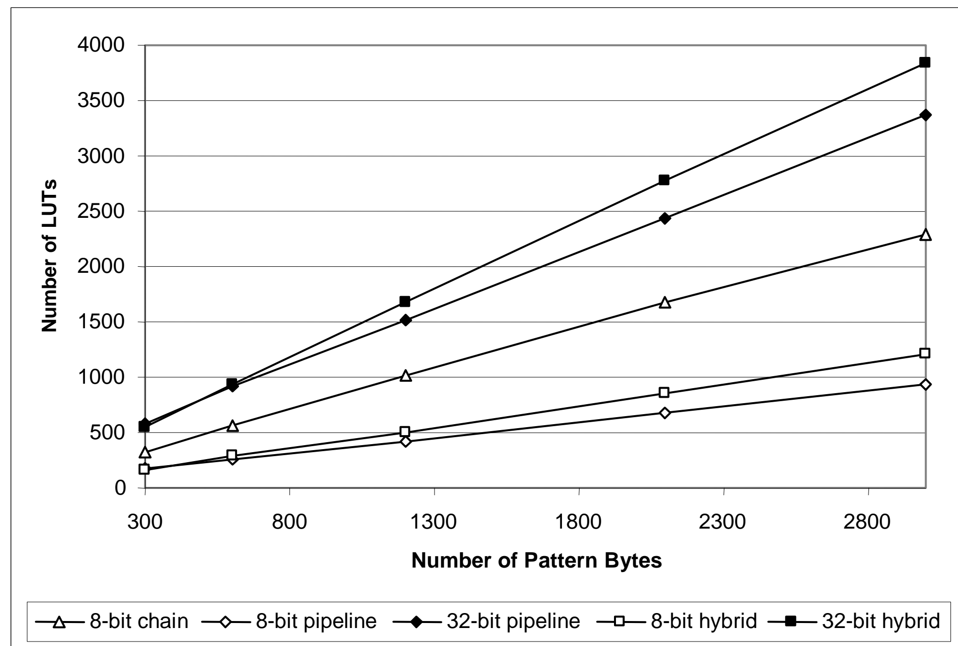


Figure 2.13: LUTs vs. Number of Pattern Bytes

The number of LUTs/byte achieved by all of the architectures decreases as the size of the pattern set increases. This is because as the size of the pattern set increases, all of the decoder logic required by the architectures becomes a smaller and smaller percentage of the overall logic required. This means that the number of LUTs/byte for all of the architectures will asymptotically approach some value representing the minimum space requirements achievable by the architecture as shown in Figure 2.14. The graph also shows that the lowest LUT resource utilization for the 8-bit and 32-bit TSM architectures is comparable to that of the 8-bit and 32-bit pipelined character grid architectures.

The timed segment matching designs are the smallest architecture in terms of DFF utilization. For the largest pattern set, the 8-bit TSM architecture requires fewer DFFs than either of the other two 8-bit architectures. The 32-bit TSM architecture also requires fewer DFFs than the 32-bit pipelined character grid as shown in Figure 2.15.

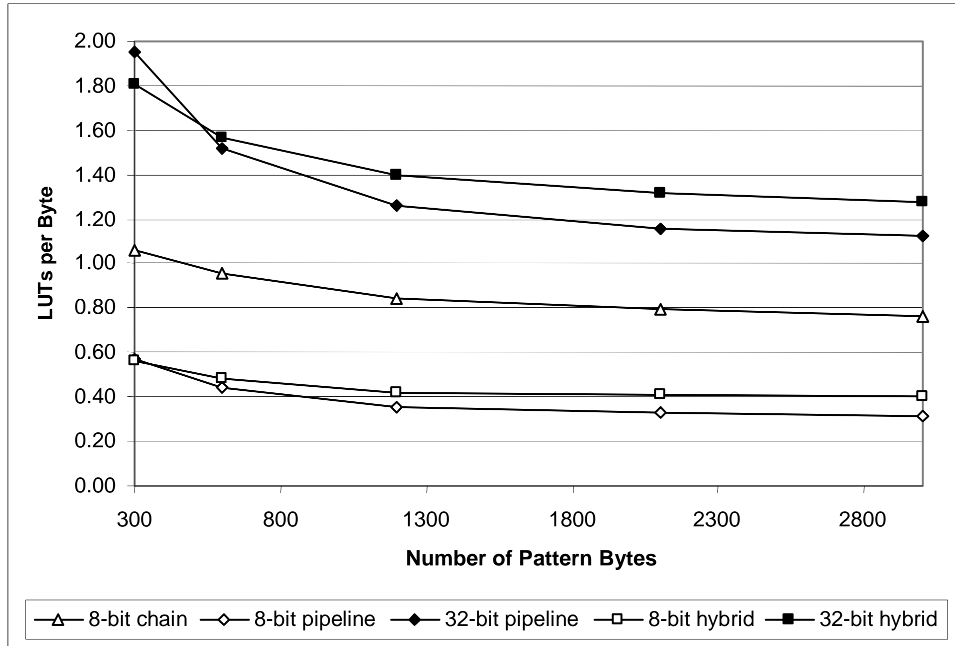


Figure 2.14: LUTs/Byte vs. Number of Pattern Bytes

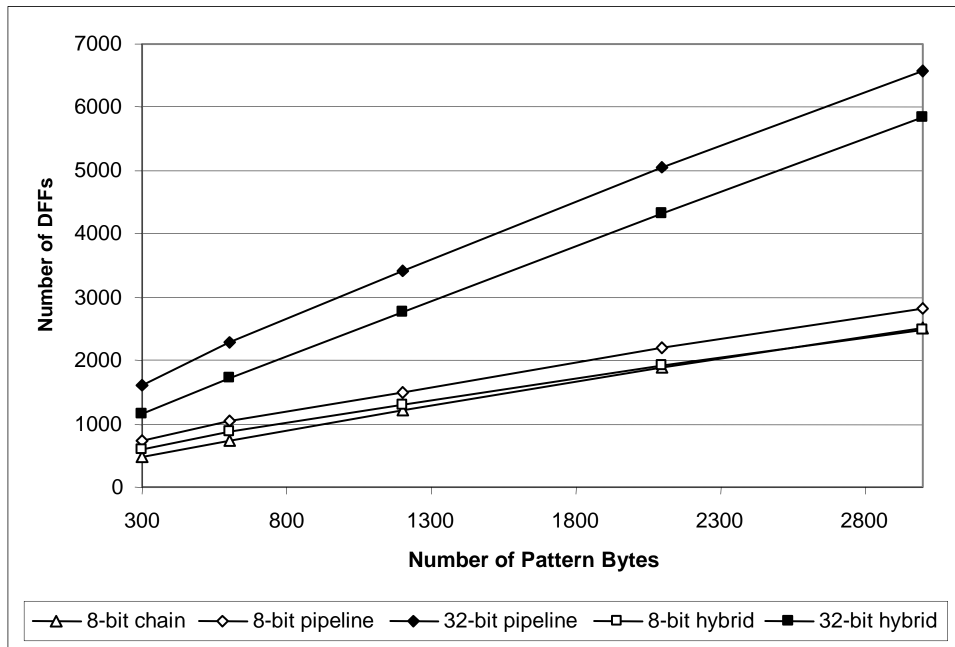


Figure 2.15: DFFs vs. Number of Pattern Bytes

By calculating trend lines for both the number of LUTs and DFFs required for each architecture, it becomes clear that the DFFs are the limiting factor in the number of pattern bytes that can be programmed into the FPGA. Therefore, by having a smaller DFF resource requirement, the TSM architecture will be able to scan for larger pattern sets than both the regular expression chain and the pipelined character grid architecture. Using the trend line, the maximum number of strings that the Xilinx Virtex 4 LX200 FPGA can accommodate using the TSM architecture can be calculated to be approximately 31K patterns, given that the patterns have an average length of 8 bytes.

Regular Expression Patterns

In addition to fixed strings, regular expression pattern matchers were generated using the 8-bit regular expression chain, and the 8-bit and 32-bit TSM architectures. Each architecture was generated to scan for 300 regular expression patterns in the form "abcd(efgh)*ij". The results for the architectures are shown in Table 2.3.

The results clearly show that 8-bit TSM architecture is much smaller than the 8-bit chain architecture in both LUT and DFF resource utilization when scanning for regular expressions.

	8-bit Input (1 Character Wide)			32-bit Input (4 Characters Wide)	
	Regex Chain	**Pipelined Grid**	**TSM**	**Pipelined Grid**	**TSM**
LUTs	2913	N/A	1409	N/A	6676
DFFs	3087	N/A	1748	N/A	5395

Table 2.3: Device utilization when scanning for regular expressions

2.6 Chapter Summary

This chapter described two well-known pattern matching architectures, the regular expression chain and the pipelined character grid. Regular expression extensions were proposed to enhance the functionality of the pipelined character grid architecture. Additionally, a hybrid architecture which combines attributes of the regular expression chain and the

pipelined character grid was presented. The hybrid TSM architecture maintains the ability to search for regular expression patterns and requires fewer hardware resources than either of the two contributing architectures. Additionally, the TSM architecture maintains the high performance of the two contributing architectures with a maximum throughput of about 12 Gbps.

Chapter 3

Hardware-Accelerated Regular Language Parsing

While pattern matching may be sufficient for network applications such as intrusion detection systems and spam filters, it may not be powerful or expressive enough for many other applications. One such application is content-based routing. When a spam filter detects a false positive the end result may be that a valid email is incorrectly labeled as spam. However, false positives in a content-based router may have many undesirable affects, including improperly routed packets, excessively high latencies, packets that never reach their destinations, and networks that are bogged down by incorrectly multicasted packets. By incorporating a parser into the network a higher level of understanding of streaming data can be achieved. This chapter describes a fast regular language parser that is designed to augment the functionality of the pattern matchers described in Chapter 2 with minimal additional logic. The regular language parser can add semantic meaning to patterns that are found within streaming data, thereby helping to alleviate the aforementioned problems and other problems that may arise from misinterpreting data. Chapter 4 shows how these techniques can be used in the implementation of various network applications.

3.1 Related Work

Hardware-accelerated parsing has not been an extensively studied problem. However, there does exist a small amount of previous work. Hardware-based parsers have been implemented using the Cocke-Younger-Kasami (CYK) algorithm [18, 19]. While these implementations do manage to decrease the $O(n^3)$ time complexity of the CYK algorithm to $O(n^2)$, the space required by the algorithm remains unchanged at $O(n^2)$, where n is the length of the input string. Such a large space requirement makes the CYK algorithm unsuitable for network applications that need to maintain parsing information for millions of network flows simultaneously.

Other previous work includes a hardware-based implementation of an Earley parser [42]. Again, the space requirements for this table driven parsing algorithm make it unsuitable for network applications.

More recently, work by Cho presents two architectures, one for an LL(1) parser and another for an LR(1) parser [15]. Both architectures take an approach similar to their software counterparts using table lookups in conjunction with a stack to parse the input.

3.2 Architecture for High-Speed Regular Language Parsing

Unlike other parsers which use a table to look up the next state of the parser, the regular language parser presented here converts the rules of a large grammar for a regular language into a nondeterministic finite automaton (NFA) [52]. Representing the grammar as an NFA has two main benefits. First, it allows the grammar to be converted into a highly pipelined logic structure that can be mapped directly onto an FPGA. An NFA representation also allows the parser to exploit the parallelism of the FPGA by exploring all possible parsing paths in parallel. Additionally, an NFA generally has fewer states than its deterministic counterpart, thus allowing for a more compact representation.

To model the grammar as an NFA, the well-known $FIRST$ and $FOLLOW$ set algorithms for predictive parsers [1] are used. These algorithms accept a grammar as input and output all the information necessary to construct an NFA. As defined in [6], $FIRST(\alpha)$

```
For each terminal symbol Z
FIRST[Z] ← {Z}
repeat
  For each production X → Y_1 ... Y_k
    if Y_1 ... Y_k are all nullable (or if k=0)
      then nullable[X] ← true
    For each i from 1 to k, each j from i+1 to k
      if Y_1 ... Y_{i-1} are all nullable (or if i=1)
        then FIRST[X] ← FIRST[X] ∪ FIRST[Y_i]
      if Y_{i+1} ... Y_k are all nullable (or if i=k)
        then FOLLOW[Y_i] ← FOLLOW[Y_i] ∪ FOLLOW[X]
      if Y_{i+1} ... Y_{j-1} are all nullable (or if i+1=j)
        then FOLLOW[Y_i] ← FOLLOW[Y_i] ∪ FIRST[Y_j]
until FIRST, FOLLOW and nullable no longer change
```

Figure 3.1: Algorithm for finding $FIRST()$ and $FOLLOW()$ sets from a production list [6]

is the set of all terminal symbols that can begin any string derived from α. $FOLLOW(\alpha)$ is defined as the set of terminal symbols that can immediately follow α. The $FIRST$ and $FOLLOW$ set algorithms are shown in Figure 3.1. The sample grammar shown in Figure 3.2 is used to illustrate these algorithms. The $FIRST$ and $FOLLOW$ sets for this grammar are displayed in Table 3.1.

No.	Production	
1	S →	L done
2	L →	received from L | received by

Figure 3.2: Sample grammar for a regular language

Using the $FIRST$ and $FOLLOW$ sets, an NFA for the grammar can be constructed as follows. First, a state is created for each of the terminal symbols in the grammar, including the end-of-input symbol (`$`) which represents the accepting state. In the sample grammar, the terminal symbols are `received`, `from`, `by`, and `done`. Then, for each terminal symbol, a transition is created to each of the terminal symbols in its $FOLLOW$ set. Finally,

Symbol	*FIRST* Set	*FOLLOW* Set
S	{ received }	{ $ }
L	{ received }	{ done }
received	{ received }	{ from , by }
from	{ from }	{ received }
by	{ by }	{ done }
done	{ done }	{ $ }

Table 3.1: *FIRST* and *FOLLOW* sets for symbols in the grammar

the start states (multiple start states are possible) are identified. The *FIRST* set of the starting symbol identifies all of the terminal symbols that can start the grammar. The corresponding state for each starting symbols is identified as a starting state. Figure 3.3 shows the NFA for the sample grammar in Figure 3.2.

A similar approach can be used to map the grammar directly into hardware. To build the hardware logic for the regular language parser, each terminal symbol in the grammar is represented using a simple primitive. The primitive, which can be seen in Figure 3.4,

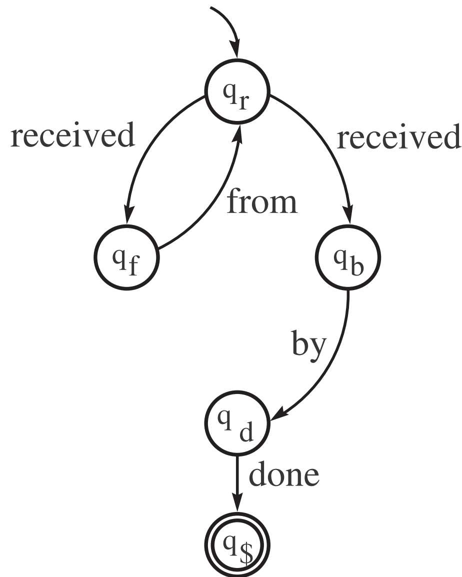


Figure 3.3: NFA for grammar in Figure 3.2

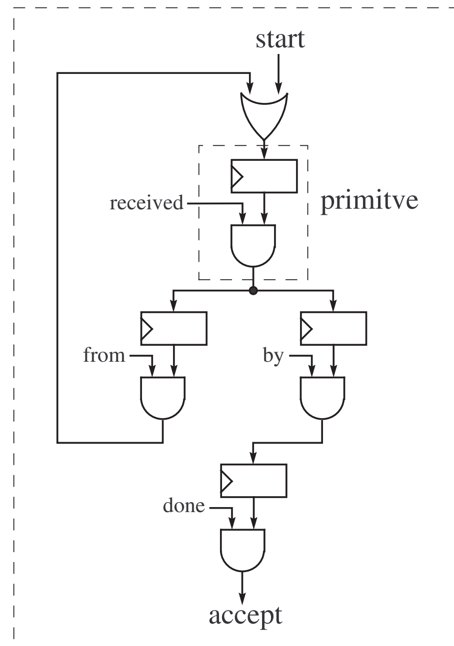


Figure 3.4: Hardware parser for grammar shown in Figure 3.2

consists of a single register and an AND gate. The inputs to each of the AND gates include the current state of the primitive (i.e. the output of the register) and a detection signal from a pattern matcher. When the pattern matcher detects a string which is representative of a terminal symbol in the grammar, it asserts a signal which is connected to the AND gate of the primitive for that symbol. By separating the pattern matcher from the regular language parser, redundant hardware can be eliminated for grammars that have multiple instances of patterns. The output of each AND gate represents a transition in the state of the grammar. Transitions are again determined from the *FOLLOW* sets of each of the terminal symbols in the grammar. The output of an AND gate for a terminal symbol is routed to the input of each of the primitives in its *FOLLOW* set. If the register of a primitive requires multiple inputs, an OR gate is used to combine the inputs into a single input for the primitive. As with the NFA model, the *FIRST* set of the starting symbol of the grammar provides the initial input to the parsing structure. This input can be asserted at the beginning of a data stream to initialize the parsing structure.

While this technique does work for the grammar shown in Figure 3.2, it may not work for all grammars. For example, adding the rule `S` $\rightarrow$ `not received done` to the grammar results in the addition of `done` into the *FOLLOW* set of `received`. This, in turn, results in the addition of a connection between the output of the `received` primitive and the input of the `done` primitive in Figure 3.4. The addition of this connection results in a parsing structure that accepts invalid inputs. For example, the new parsing structure accepts the input `received done`, which is not defined by the grammar. To alleviate this problem, the original grammar can be modified by making each occurrence of a terminal symbol a unique symbol that is assigned its own primitive in the parsing structure. To illustrate, Figure 3.5 shows the original grammar after adding the rule `S` $\rightarrow$ `not received done` and making each terminal symbol unique.

No.	Production	
1	S $\rightarrow$	L done_1 | not_1 received_1 done_2
2	L $\rightarrow$	received_2 from_1 L | received_3 by_1

Figure 3.5: Sample grammar for a regular language

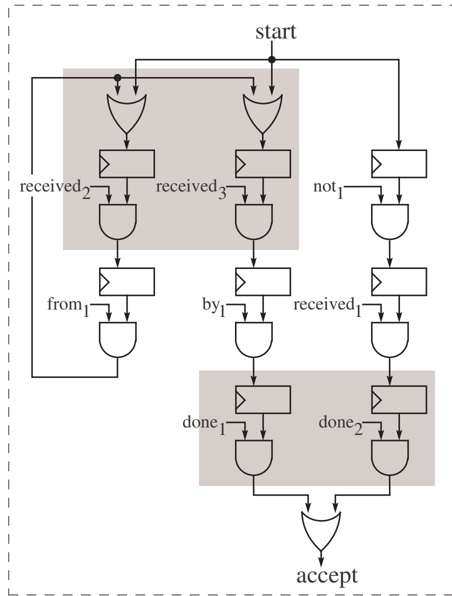


Figure 3.6: Hardware parser for grammar in Figure 3.5

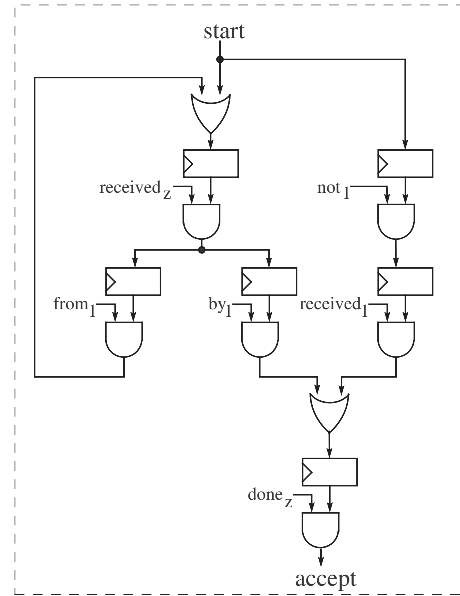


Figure 3.7: Simplified parsing structure for grammar in Figure 3.5

The parsing structure that is generated for this new grammar using the *FIRST* and *FOLLOW* set technique described earlier is shown in Figure 3.6. Note that there is now a primitive for each occurrence of each terminal symbol in the grammar and the structure only accepts the intended language. This structure can subsequently be minimized to decrease the number of primitives required for the parsing structure. For each terminal symbol X, if the input sets to X_i and X_j are equivalent, the primitives for X_i and X_j can be merged into a single primitive. This can be seen in Figures 3.6 and 3.7 where $\texttt{received}_2$ and $\texttt{received}_3$ are merged into $\texttt{received}_z$. Primitives for terminal symbols X_i and X_j can also be merged if the *FOLLOW* sets of X_i and X_j are equivalent. This is shown in Figures 3.6 and 3.7 where $\texttt{done}_1$ and $\texttt{done}_2$ are merged into $\texttt{done}_z$. Figure 3.7 shows the parsing structure for the grammar in Figure 3.5 after being minimized.

When combined with a pattern matcher, the hardware parser described in this chapter is capable of maintaining the state of a data stream. While doing so, the parser can also forward pattern information along with the state of the parsing structure for each pattern to a back-end for further processing. The back-end can utilize this information to take

appropriate actions for the desired application. The following chapter describes how this technique can be used to develop intelligent network applications.

3.3 Chapter Summary

This chapter described a high-speed architecture for parsing regular languages. The architecture can be automatically generated from a Lex/Yacc style grammar. A method for generating the architecture using the *FIRST* and *FOLLOW* set algorithms was also presented. The regular language parser can be combined with a pattern matcher to provide syntactic information to external modules.

Chapter 4

Applications of Regular Language Parser

To help illustrate the hardware-based parsing techniques described in Chapters 2 and 3 and how they can be used in intelligent network applications, this chapter describes the implementation of two high-speed network applications. The first application is a content-based router that uses the fast regular language parser to parse packet payloads and makes intelligent routing decisions based on packets' contents. The second application is a data filter that is used to extract the message contents of in-flight email messages while discarding header information and attachments.

4.1 Background

All of the applications described in this chapter were implemented on the Field-Programmable Port Extender (FPX) platform and utilize a set of layered protocol wrappers. This section provides a brief background on the implementation platform.

4.1.1 Field-Programmable Port Extender

The FPX is a general purpose, reprogrammable platform that performs data processing in FPGA hardware [46]. As data packets pass through the device, they can be processed in the

hardware by user-defined, reprogrammable modules. Hardware-accelerated data processing enables the FPX to process data at multi-gigabit per second rates, even when performing deep processing of packet payloads.

The FPX contains two FPGAs. A Xilinx Virtex XCV600E FPGA routes packets into and out of the FPX. It also controls the routing of packets to and from the application FPGA. The application FPGA, which executes the user-defined hardware modules, is a Xilinx Virtex XCV2000E (upgraded from an XCV1000E that was used on the first version of the platform). The FPX also contains two banks of 36-bit wide Zero-Bus-Turnaround Static RAM (ZBT SRAM) and two banks of 64-bit PC-100 Synchronous Dynamic RAM (SDRAM). Fully configured, the FPX can access four parallel memories with a combined capacity of 1 gigabyte.

4.1.2 Layered Protocol Wrappers

To provide a higher level of abstraction for packet processing, a library of layered protocol wrappers was implemented for the FPX [11]. They use a layered design and consist of different processing circuits within each layer. At the lowest level, a cell processor processes raw cells between network interfaces. At the higher levels, a frame processor reassembles and processes variable length frames while an IP processor processes IP packets. Figure 4.1 shows the configuration of an application module in the protocol wrappers.

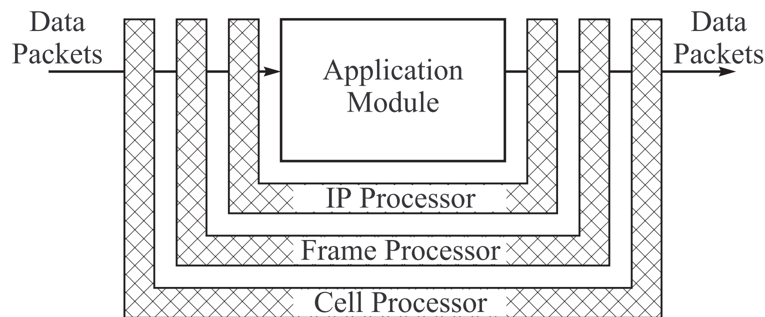


Figure 4.1: Router in protocol wrappers

4.1.3 TCP Protocol Processor

The TCP-Processor is an FPGA based TCP protocol processor [63]. It was designed and implemented to support the processing of up to 8 million simultaneous TCP flows in high-speed networks. The TCP-Processor provides stateful flow tracking and TCP stream reassembly for network applications which process TCP data streams. Additionally, the TCP-Processor includes encoder and decoder circuits which enable it to serialize a TCP flow along with corresponding flow information. This serialized information can then be transported off-chip to multiple other FPGAs for packet processing.

4.2 Content-Based Router

As the Internet continues to expand, researchers are starting to look at content-based routing as a mechanism to improve upon and/or to add new services for managing the distribution of data. Content-based routing improves upon the existing Internet model by giving users the freedom to describe routing schemes in the application layer of network packets. Content-based routers then inspect and interpret packet payloads and route packets according to the content of the packet.

One example of this type of interaction can be seen in publish/subscribe networks [5, 60]. Users can subscribe to information that is interesting to them by sending high level descriptions to routers using the application layer (layer 7) of the packet. Content-based routers then interpret the subscription packet content and route all messages with matching contents to the subscriber. Some examples for publish/subscribe networks include the routing of stock quotes, distribution of weather reports, and streaming video broadcasts. Content-based routing can also be used for applications such as load balancing in web server clusters [17], or routing of online transactions to the appropriate shipping warehouse. It is this class of content-based routing applications that is the focus of this implementation.

To route packets based on values that appear in the payload, efficient methods for packet payload processing are needed. Carzaniga, Rutherford, and Wolf have presented a software based routing algorithm [14]. However, due to the processing power required by deep content inspection, software approaches are unlikely to maintain the throughput

of multi-gigabit networks. This can potentially limit the adoption of content-based networks. As such, a reconfigurable hardware architecture capable of fast, intelligent content inspection was developed.

The remainder of this section describes how the fast regular language parser described in Chapter 3 was used to implement a high-speed content-based router. The message format for the content-based router implementation is the XML format defined in Figure 4.2. A Lex/Yacc style version of this XML specification is shown in Figure 4.3. This grammar is passed into a custom compiler that automatically generates the VHDL required for the architecture. The layout of the main components of the architecture, including a pattern matcher, a parsing structure, and a routing module, is shown in Figure 4.4. The

```
<!ELEMENT card      (routekey, name, title?, phone?)>
<!ELEMENT routekey  (#PCDATA)>
<!ELEMENT name      ((first, last) | (last, first))>
<!ELEMENT first     (#PCDATA)>
<!ELEMENT last      (#PCDATA)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT phone     (#PCDATA)>
```

Figure 4.2: Document Type Definition (DTD) for content-based router implementation

```
STRING [a-zA-Z0-9-]+
%%
card:       "<card>" routekey name title phone "</card>"
routekey:   "<routekey>" route "</routekey>"
route:       routefirst | routelast
routefirst: "first"
routelast:  "last"
name:       "<name>" nameN "</name>"
nameN:       nameFL | nameLF
nameFL:      firstFL lastFL
nameLF:      lastLF firstLF
firstFL:    "<first>" STRING "</first>"
lastFL:     "<last>" STRING "</last>"
lastLF:     "<last>" STRING "</last>"
firstLF:    "<first>" STRING "</first>"
title:      "<title>" STRING "</title>" | ε
phone:      "<phone>" STRING "</phone>" | ε
%%
```

Figure 4.3: Lex/Yacc style grammar

architecture has been fully implemented on the FPX platform which allows for rapid deployment and testing in gigabit-rate networks.

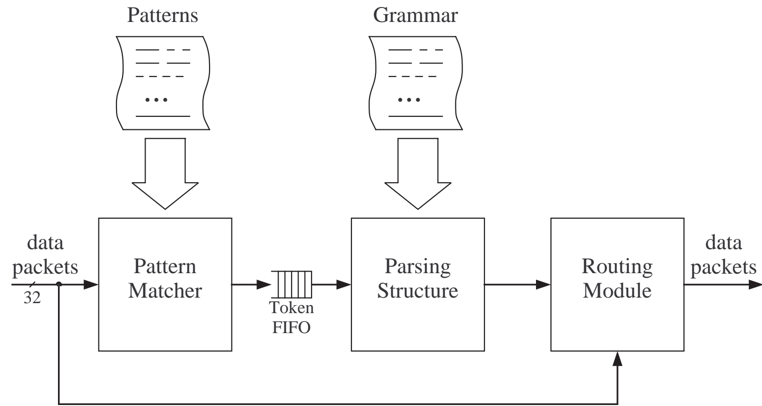


Figure 4.4: Content-based router architecture

4.2.1 Pattern Matcher

Data packets enter the content-based router after being processed by the layered protocol wrappers. The first stage in processing each packet for routing is pattern matching. The modular design allows a variety of techniques to be used for pattern matching. For this implementation, a modified pipelined character grid [8] was employed, which has been scaled to accept a four character wide (32-bit) input. Scaling is achieved by replicating the pipeline until there is one pipeline for each character in the input width. A detailed block diagram of the decoded character pipeline is shown in Figure 4.5.

The scaled pipeline receives four characters (32-bits) per clock cycle from the layered protocol wrappers. Characters 1, 2, 3, and 4 are passed into pipeline alignments 3, 2, 1, and 0 respectively. Before entering the pipeline registers, characters are passed into an 8-to-256-bit decoder. The 256-bit output represents a single bit line for each of the 256 possible extended-ASCII characters. This decreases the routing resource required for string detectors. The decoded character lines are passed into the pipeline registers as illustrated in Figure 4.5. The pipelined character grid can detect patterns that are less than or equal

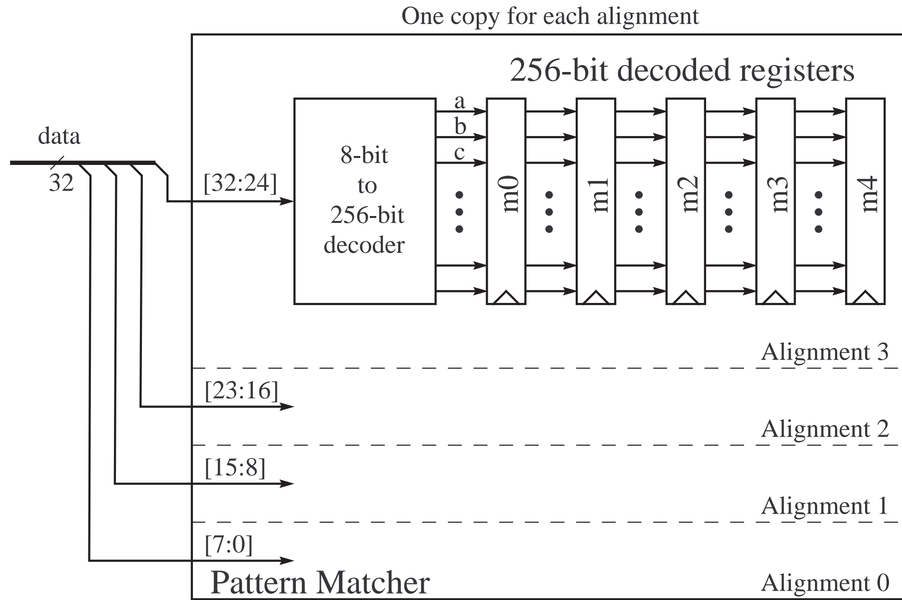


Figure 4.5: Diagram of pattern matcher pipeline

to the length of the pipeline. Additionally, the pipeline only needs to be as long as the longest pattern in the grammar.

The actual pattern matching is executed by a series of string detectors. A string detector is generated for each of the patterns in the input grammar. For the example grammar in Figure 4.3 there are 17 unique patterns: `<card>`, `</card>`, `<routekey>`, `</routkey>`, `first`, `last`, `<name>`, `</name>`, `<first>`, `</first>`, `<last>`, `</last>`, `<title>`, `</title>`, `<phone>`, `</phone>`, and `STRING`. Each of these patterns can be detected by ANDing together the appropriate bits from the decoded character pipeline. Since this design uses a scaled pipeline, the presence of a pattern needs to be checked at each possible starting alignment. A pattern is detected if it is found at any one of the four possible starting alignments. Figure 4.6 illustrates the logic required to match the patten `<card>`. The notation shown in Figure 4.6 is *Register[Alignment][Character]*. For example, *m1[0][c]* represents the 'c' character bit of register *m1* in alignment 0. A single bit line is output from the pattern matcher to the parser structure for each of the string detectors.

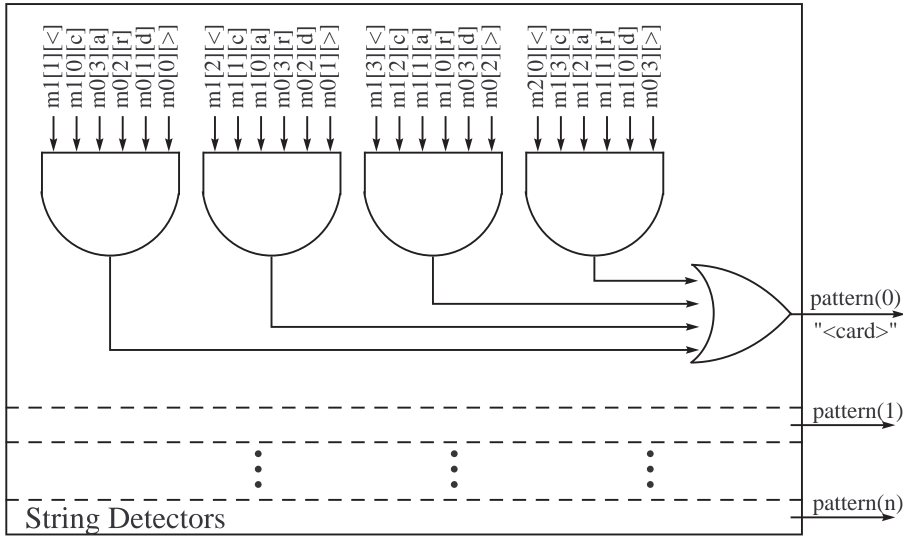


Figure 4.6: Diagram of a string detector

4.2.2 Parsing Structure

The parsing structure gives the content-based router a higher level of understanding than simple pattern matching. It defines the semantics of patterns as they are detected by the pattern matcher. The hardware logic for the parsing structure is determined from the input grammar (or grammars). The production list of a grammar defines all of the possible transitions for a grammar. While processing data, the parser maintains the state of the grammar allowing it to determine which patterns can occur next.

As described in Section 3.2 the structure for the parser is determined using the *FIRST* and *FOLLOW* set algorithms for predictive parsers. The *FOLLOW* sets for the content-based router architecture are shown in Table 4.1. The corresponding parsing structure is illustrated in Figure 4.7.

The generated parsing structure processes packets one pattern at a time. At the start of a packet, the starting register (register P0 in Figure 4.7) is set. As packets are processed, the parsing structure receives a signal from the pattern matcher for each pattern that is found. These signals allow the parsing structure to traverse through the grammar and maintain the semantics of the data stream. During processing, all signals from the pattern matcher are sent downstream to the routing module accompanied by the state of the parsing

Pattern	*FOLLOW* Set
<card>	<routekey>
<routekey>	first, last
first, last	</routekey>
</routekey>	<name>
<name>	<first>$_1$, <last>$_1$
<first>$_1$, <last>$_1$, <first>$_2$, <last>$_2$, <title>, <phone>	STRING
STRING	</first>$_1$, </last>$_1$, </first>$_2$, </last>$_2$, </title>, </phone>
</first>$_1$	<last>$_1$
</last>$_2$	<first>$_2$
</last>$_1$, </first>$_2$	</name>
</name>	<title>, <phone>, </card>
</title>	<phone>, </card>
</phone>	</card>

Table 4.1: *FOLLOW* sets for example grammar

structure. The state of the parsing structure indicates where in the grammar each pattern is found. Knowing where in the grammar a pattern is found allows the routing module to make more intelligent decisions regarding packets' destinations. To better understand this, consider that the parsing structure in Figure 4.7 is searching for several instances of `STRING`. However, if the router is configured to route on a first name, only `STRING` values that occur at register `P7` or `P16` should affect the action that the router takes. Other occurrences of `STRING`, those at `P10`, `P13`, `P20`, and `P23`, should not affect the action taken by the content based router. Without parsing the entire data stream and maintaining the context of where each `STRING` value occurs, this behavior is not possible.

Validating XML Input

To avoid routing invalid or malformed XML messages, the content-based router validates all XML messages prior to routing them. As shown in Figure 4.7, an `XML valid` signal is asserted when the parsing structure successfully traverses through the entire grammar. The `XML valid` signal is forwarded to the routing module which can subsequently take the appropriate routing action on the XML message.

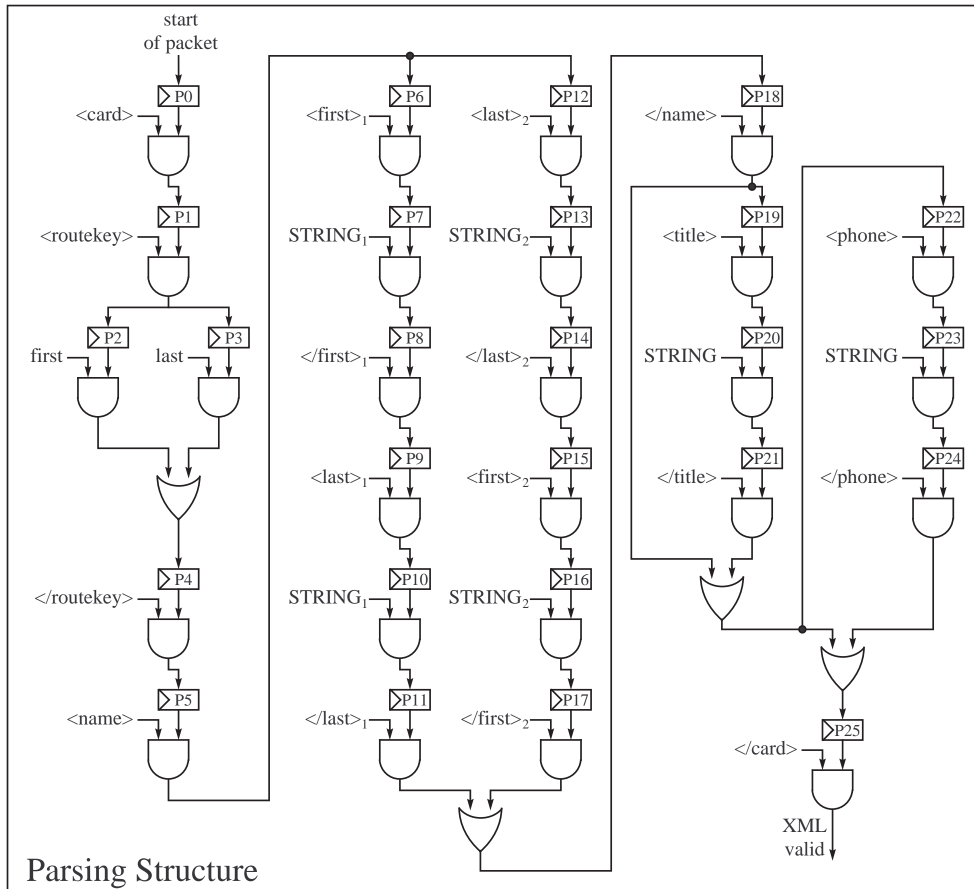


Figure 4.7: Diagram of parsing structure

4.2.3 Routing Module

The routing module (Figure 4.8) is responsible for modifying the IP header of each packet to route the packet to the appropriate destination. As packets enter the content-based router they are buffered in the routing module until the packet has been completely processed. Prior to routing any packet, the routing module verifies that the packet is the correct format. Most importantly, this entails validating the XML message. XML messages that do not strictly adhere to the grammar provided will not be rerouted by the module. Optionally, the module can also check for specific IP address and port ranges prior to routing.

For the example implementation, packets are routed based on the first character of either the first name or the last name specified in the XML message. The `routekey`

value specifies which name to use for routing. A series of control signals received from the pattern matcher and the parsing module allow the routing module to route packets accordingly. These control signals are described below and can be seen in Figure 4.8.

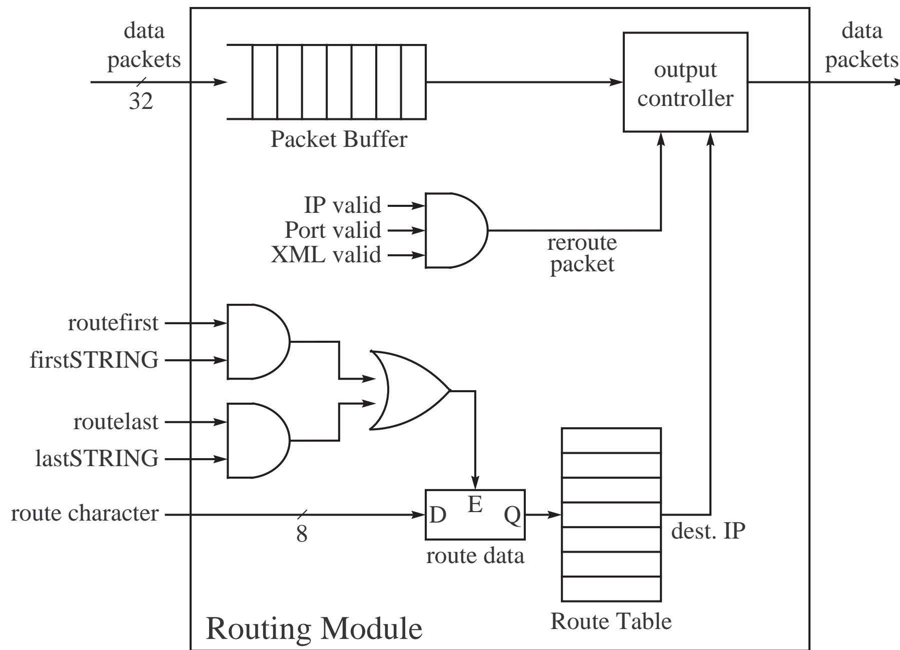


Figure 4.8: Diagram of routing module

The value `routefirst` is enabled by the parsing structure when register `P2` is set and the pattern `first` is detected by the pattern matcher. This value indicates that the packet should be routed according to the first name in the XML message. Similarly, the value `routelast` is enabled when register `P3` is set and the pattern `last` is detected. It indicates that the packet should be routed according to the last name in the XML message. These values stay enabled for the duration of the packet.

The `firstSTRING` value is enabled by the parsing structure when either register `P7` or register `P16` are set and a `STRING` pattern is detected by the pattern matcher. Similarly, the `lastSTRING` value is enabled when either register `P10` or `P13` are set and a `STRING` pattern is detected. The `firstSTRING` and `lastSTRING` values are only valid for a single clock cycle. During this clock cycle, the first character of the `STRING` pattern (the `route character`) is

forwarded to the routing module and stored. This value is then used to address a routing table which determines the next destination of the packet being processed.

Once a packet has been fully processed, the output controller reads the packet from the packet buffer for output. If the packet contains a valid XML message (and optionally, IP address and port ranges), then the IP header is rewritten with the new destination address as it is output.

4.2.4 Implementation and Experimental Setup

The content-based router described in this section was fully implemented and tested on the Xilinx Virtex XCV2000E FPGA on the FPX platform. The FPX was integrated into a Global Velocity GVS-1000 chassis. A photograph of an FPX and the GVS-1000 chassis is shown in Figure 4.9.

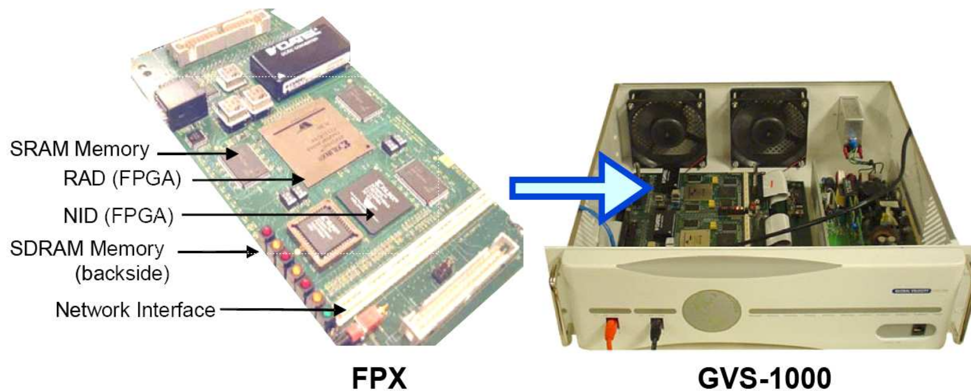


Figure 4.9: FPX and GVS-1000 chassis

The GVS-1000 has two bidirectional gigabit interfaces for passing traffic into the FPX. To test the content-based router architecture, each of the gigabit interfaces on the GVS-1000 was connected to a different host machine. One machine was used to generate and send XML messages into the content-based router. The second machine was used as a receiver for routed messages. Since only two machines were used for these experiments, XML messages were routed to different ports on the receiving machine based on the message

content. Both Ethereal and a small counter application were used to verify that XML messages arrived at the correct destination port on the receiving machine.

XML data messages were generated on the sending machine via the small test application shown in Figure 4.10. The test application creates XML messages using the values specified in the text fields and sends them as UDP packets into the content-based router. Additionally, the test application can randomly generate and send a specified number of XML messages into the content-based router. An example XML message is shown in Figure 4.11.

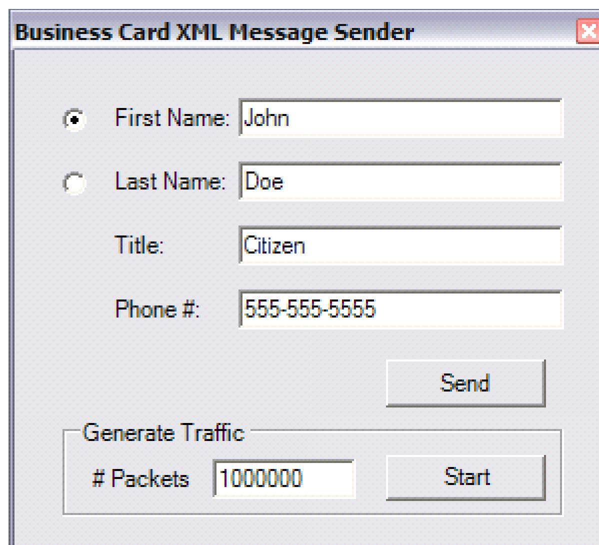


Figure 4.10: Test application interface

```
<card>
  <routekey>first</routekey>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
  <title>Citizen</title>
  <phone>555-555-5555</phone>
</card>
```

Figure 4.11: XML packet contents

4.2.5 Area and Performance

For this application the maximum clock frequency is limited to 100 MHz by the layered protocol wrappers. At this speed the content-based router can achieve a maximum throughput of 3.2 Gbps. Without the protocol wrappers, the core of the content-based router architecture can achieve clock frequencies over 200 MHz. At this speed the content-based router can route XML data messages at about 6.4 Gbps.

The content-based router requires 3751 slice flip-flops, approximately 9% of the available flip-flop resources. The architecture requires 3058 4-input LUTs, approximately 7% of the available LUT resources.

The layered protocol wrappers alone require 2623 flip-flops and 2196 4-input LUTs. This is approximately 6% and 5% of the available flip-flop and LUT resources respectively.

The core of the content-based router (without the protocol wrappers) requires approximately 1128 slice flip-flops and 862 4-input LUTs. This is approximately 2.9% and 2.2% of the available flip-flop and LUT resources respectively. Such a small space requirement for the core of the routing architecture means much larger and/or many more grammars can fit onto the FPGA.

4.3 Semantic Network-Content Filter

In previous work [26, 47], a hardware-accelerated semantic processing system was developed for analyzing computer network traffic. This system uses latent semantic indexing techniques to analyze and classify the topic of streaming documents at multi-gigabit per second data rates [65].

In order to assist the system to efficiently process the data, an additional module was developed that identifies language and character encoding used in network data. This module, called HAIL (Hardware-Accelerated Identification of Languages), uses N-grams discovered from training documents to determine the language and encoding of documents that pass through the system. Experimental results with multilingual datasets showed the accuracy of HAIL to be very high. In many cases, the identification accuracy exceeded 99% [40].

However, most Internet documents such as XML, HTML, and email contain a large amount of markup and header data. This markup data can have a negative impact on the language detection algorithms used by HAIL. To improve the accuracy of HAIL, and hence the accuracy of the whole document classification system, there is a need to identify and extract document content from the markup data.

The remainder of this section describes how the regular language parser described in Chapter 3 was used to implement a high-speed filter capable of extracting text from email messages as they traverse a network. The filter removes headers and attachments from email messages, leaving only the message content to be processed by the language identification algorithm and document classifier. By removing markup and header data, the filter can improve the accuracy of language identification algorithms. The filter described in this section can process documents at over 600 Mbps on the FPX platform.

4.3.1 Background

This section provides a brief background on the semantic classification system and the HAIL module.

Semantic Classification System

As described in [26, 47, 65], a hardware-accelerated document classification system has been developed that uses latent semantic indexing techniques to classify streaming documents. The system is designed to ingest and classify large volumes of network data in real time. Classification is accomplished through a series of mathematical transformation algorithms as shown in Figure 4.12. Each step of the transformation is implemented in reconfigurable hardware on a series of stackable FPX devices.

As data packets enter the system, they are first processed by the TCP processor. The TCP processor reconstructs packets into consistent TCP flows. Each flow is augmented with control signals that indicate where the IP header, the TCP header, and data segment of each packet starts. Additionally, a flow identification number is assigned to each TCP

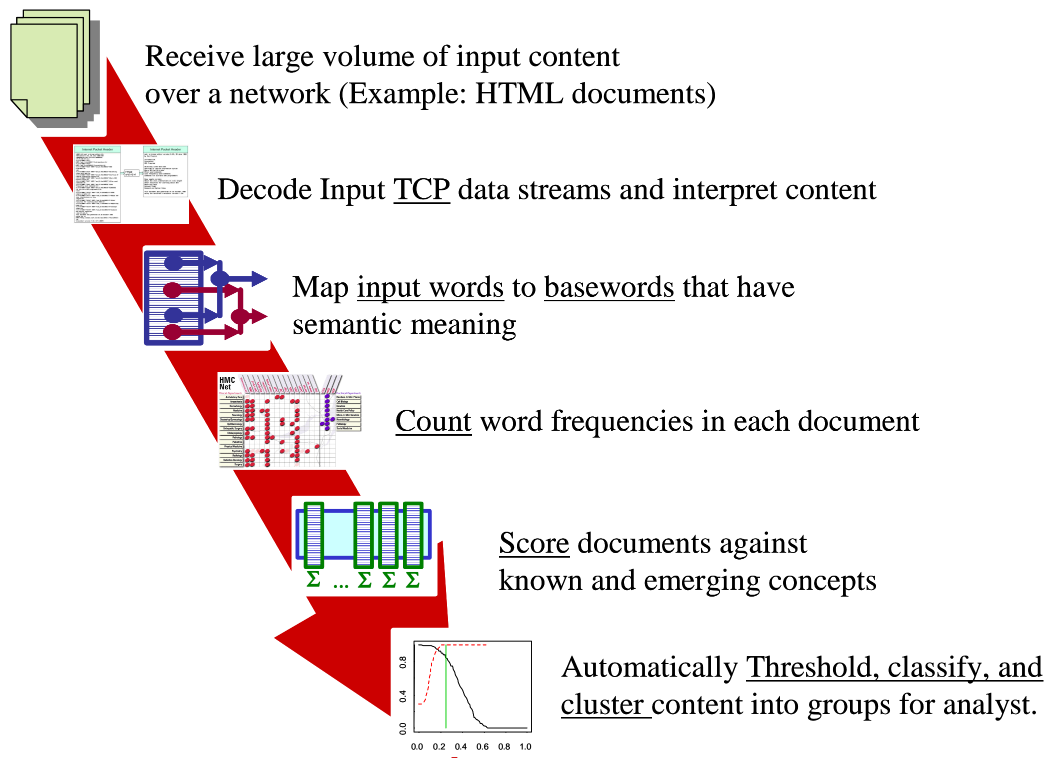


Figure 4.12: Flow of documents through the document classification system

flow so downstream components can manage per-flow context. Each TCP flow is considered to be a single document.

Reconstructed TCP flows are subsequently processed by a word mapping circuit. The word mapping circuit tokenizes each TCP flow to find words in the flow. For each word that is found, a hash is computed that is used as an address into a word mapping table (WMT). The WMT is a 1 million entry table that maps 1 million input words down to 4000 *basewords*. A *baseword* is a numerical representation of the semantic meaning of the input word. For example, the input words "sleep", "sleeping", "nap", and "siesta" all have similar semantics. As such, the WMT would map all of these words to the same baseword value. For details on the different techniques used to create the WMT, refer to [26].

Once a flow has been tokenized, the list of basewords is sent downstream to a module that maintains a count of the basewords for each active flow. The count is maintained in

a 4000-dimension document vector, where each dimension represents one of the possible baseword outputs of the WMT.

At the conclusion of each flow, the 4000-dimension document vector is sent to the scoring module. The scoring module computes a dot product of the document vector against up to 30 previously defined 4000-dimension concept vectors. The flow is subsequently classified according to the highest scoring concept.

Language Identification Hardware

In an effort to further enhance the capabilities of the semantic document classification system, the HAIL module was developed. This module integrates into the document classification system and identifies the language and character encoding of each TCP flow prior to classification [40, 39].

Identifying the language and character encoding allows the semantic classification system to employ encoding specific tokenizers as well as language specific WMTs. Using an encoding specific tokenizer allows the system to reduce noise by only accepting characters known to be a member of a given character set in a given language. Language specific WMTs allow the system to differentiate between words that occur in multiple languages but have different meanings in each language. Separating WMTs by language also has the benefit of breaking one large WMT into many smaller WMTs, thereby reducing the number of entries in each table. Fewer table entries can help to alleviate any hash collisions that may occur during the baseword translation.

HAIL utilizes N-grams to identify both the language and character encoding of a data stream. An N-gram consists of N sequential characters that have been extracted from the data stream. As HAIL processes a data stream, a series of five sequential characters (tetra-grams) are extracted to represent the document. Each unique tetra-gram is associated with a single language/encoding pair through offline training. As tetra-grams are extracted from a document, a counter for the associated language/encoding pair is incremented. At the end of a document, HAIL identifies the language and encoding based on the counter with the highest value.

4.3.2 Application Level Processing System

When properly trained, HAIL can accurately identify the language of a streaming document up to 99.95% of the time. However, this result assumes that HAIL is processing clean documents composed of mostly (if not entirely) text in a single language. This is not likely to be the case when processing many types of Internet traffic. For example, both HTML and email documents contain headers and tags that are likely to be identified as English whereas the body of the document may actually be Spanish, German, or Arabic. Given a document with enough header information, HAIL may incorrectly identify the language of the document due to the N-grams found in the header.

Consider the sample email shown in Figure 4.13. The email header consists of over 1500 bytes of data, whereas the email body consists of only 65 bytes of data. It is desirable that HAIL identify the language found in the email body and not in the email header. Additionally, it is desirable that only the body of the email message is processed by the document classification system. If the email is processed by both HAIL and the document classification system with the headers intact, it is unlikely that either HAIL or the document classification system will exhibit the desired behavior.

To alleviate the problems described above, an additional module, the Application Level Processing System (ALPS), was built using the fast regular language parser. ALPS has been implemented on the FPX platform to serve as a preprocessor to the HAIL module. The ALPS circuits can be automatically generated via a custom compiler when provided with a grammar in a Lex/Yacc style format. Additionally, ALPS allows specific fields of a grammar to be either forwarded or removed based on a users preference. Referring back to the sample email in Figure 4.13, ALPS can be configured to identify and parse the entire email message and forward only the email body downstream to HAIL and the document classification system. This will allow HAIL to more accurately identify the language and allow more accurate classification by the document classification system.

```
Return-Path: <sender@smtp.server.com>
X-Original-To: sender@smtp.server.com
Delivered-To: receiver@smtp.server.com
Received: from Lappy (24-107-16-209.dhcp.stls.mo.charter.com [24.107.16.209])
         by smtp.server.com (Postfix) with ESMTP id BA74636BF2
         for <receiver@smtp.server.com>; Fri, 27 Oct 2006 03:35:29 -0500 (CDT)
Message-ID: <039801c6f9a2$c06f36d0$eaa0fc80@Lappy>
From: "Sender" <sender@smtp.server.com>
To: "Receiver" <receiver@smtp.server.com>
Subject: Example Email
Date: Fri, 27 Oct 2006 03:34:42 -0500
MIME-Version: 1.0
Content-Type: multipart/alternative;
            boundary="----=_NextPart_000_0395_01C6F978.D7451B60"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2900.2869
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2900.2962
X-Virus-Status: No
X-Virus-Checker-Version: clamassassin 1.2.3 with clamdscan /
                         ClamAV 0.88.2/2105/Thu Oct 26 03:14:55 2006
X-Spam-Checker-Version: SpamAssassin 3.0.5 (2005-11-28) on smtp.server.com
X-Spam-Level:
X-Spam-Status: No, score=-0.3 required=6.0 tests=AWL,BAYES_00,HTML_80_90,
            HTML_MESSAGE,RCVD_IN_NJABL_DUL,RCVD_IN_SORBS_DUL autolearn=no
            version=3.0.5
Status: O
X-UID: 39213
Content-Length: 1082
X-Keywords:

This is a multi-part message in MIME format.

------=_NextPart_000_0395_01C6F978.D7451B60
Content-Type: text/plain;
            charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

Hola mama. La universidad es grande. Enviar por favor el dinero.

------=_NextPart_000_0395_01C6F978.D7451B60--
```

Figure 4.13: Sample email message

4.3.3 Implementation of Email Parser

Implementing an email parser using ALPS first required defining a grammar for email. For this, the standard email grammar as defined in RFC 2822 [57] was utilized. Additionally, grammars defined in RFC 2045 [29] and RFC 2046 [30] were incorporated to properly parse both MIME header extensions and multipart email messages. A small excerpt representing the date portion from RFC 2822 is shown in Augmented Backus Naur Form (ABNF) in Figure 4.14. Figure 4.15 shows the same portion of the grammar after it has been converted

```
date-time = [ day-of-week "," ] date FWS time [CFWS]
day-of-week = [FWS] day-name
day-name = "Mon" / "Tue" / "Wed" / "Thu" / "Fri" / "Sat" / "Sun"
date = day month year
year = 4*DIGIT
month = FWS month-name FWS
month-name = "Jan" / "Feb" / "Mar" / "Apr" / "May" / "Jun" /
             "Jul" / "Aug" / "Sep" / "Oct" / "Nov" / "Dec"
day = [FWS] 1*2DIGIT
time = time-of-day FWS zone
time-of-day = hour ":" minute [ ":" second ]
minute = 2DIGIT
second = 2DIGIT
hour = 2DIGIT
zone = ( "+" / "-" ) 4DIGIT
```

Figure 4.14: ABNF for date portion of email grammar

```
date-time:       date-time-opt date FWS time CFWS-opt;
date-time-opt:   day-of-week "," | ;
day-of-week:     FWS-opt day-name;
day-name:        "Mon" | "Tue" | "Wed" | "Thu" | "Fri" | "Sat" | "Sun";
date:            day month year;
year:            DIGIT DIGIT DIGIT DIGIT year-opt;
year-opt:        DIGIT year-opt | ;
month:           FWS month-name FWS;
month-name:      "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun“ |
                 "Jul" | "Aug" | "Sep" | "Oct" | "Nov" | "Dec";
day:             FWS-opt DIGIT day-2dig;
day-2dig:        DIGIT | ;
time:            time-of-day FWS zone;
time-of-day:     hour ":" minute time-of-day-opt;
time-of-day-opt: ":" second | ;
minute:          DIGIT DIGIT;
second:          DIGIT DIGIT;
hour:            DIGIT DIGIT;
zone:            zone-a zone-b;
zone-a:          "+" | "-";
Zone-b:          DIGIT DIGIT DIGIT DIGIT;
```

Figure 4.15: BNF for date portion of email grammar

from ABNF to the format accepted by the ALPS hardware generator. The complete grammar for the email parser consists of over 160 tokens and 200 production rules. Appendix A contains the complete Lex/Yacc style grammar used to generate the hardware.

As shown in Figure 4.16, the email parser consists of several main components: the lexical analyzer, the parsing structure, and the filtering module. Both the lexical analyzer and the parsing structure are automatically generated from the grammar tokens and the grammar production rules respectively. The filtering module receives information from the parsing structure and can be configured to either keep or discard data given the state of

the parser. The following sections discuss the architecture and generation of the lexical analyzer and the parsing structure.

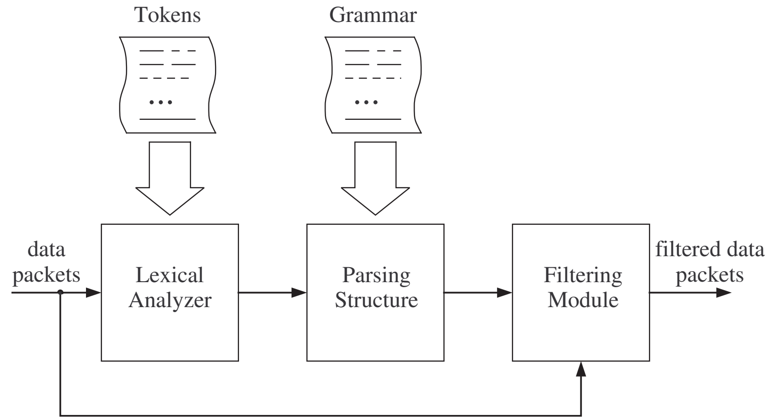


Figure 4.16: ALPS email parser architecture

Lexical Analyzer

As data enters the email parser it is first processed by the lexical analyzer. The lexical analyzer is a pattern matcher that scans the input data for strings that match tokens in the input grammar. The pattern matcher architecture uses an 8-bit pipelined character grid as described by Baker [8]. A detailed block diagram of the decoded character pipeline is shown in Figure 4.17. The top half of the figure shows the decoded character pipeline, whereas the bottom half shows a sample string detector required to match the `From:` token in the email grammar. The string detectors required for each of the tokens in the grammar are automatically generated by custom tools.

The pipeline receives one character per clock cycle from the input data stream. Before entering the pipeline registers, characters are passed into a decoder which outputs a single bit line for each of the characters needed by the string detectors. This decreases the hardware routing resources required for matching the tokens in the grammar. The decoded character lines are passed into the pipeline registers as illustrated in Figure 4.17. Additionally, the decoder is generated along with the string detectors and contains only

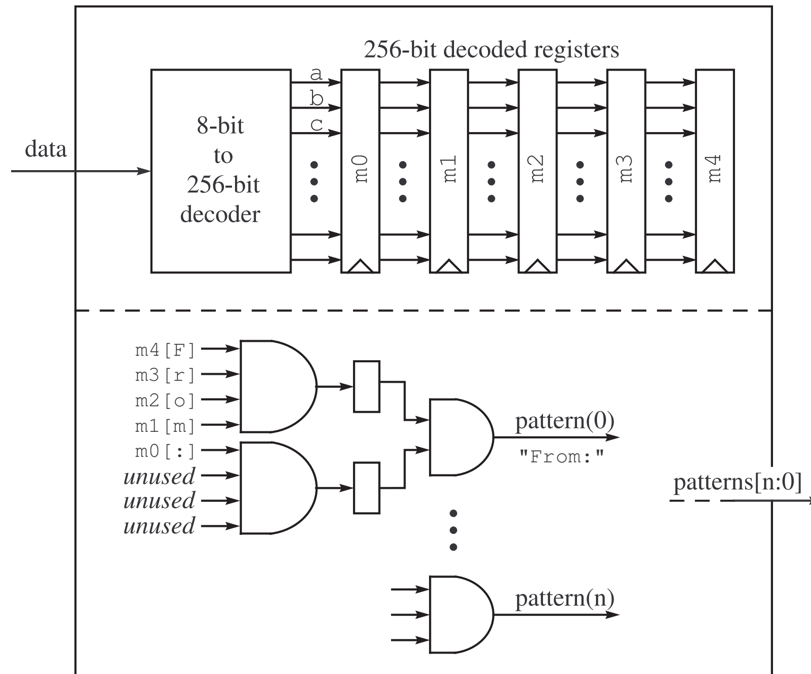


Figure 4.17: Pattern matcher for lexical analysis

the logic required to decode the characters needed by those string detectors (up to 256 characters).

The actual pattern matching is executed by a series of string detectors which are automatically generated. A match is found by ANDing together the appropriate bits from the decoded character pipeline as characters traverse through the pipeline. A single bit line is output from the lexical analyzer to the parsing structure for each of the string detectors. These signals indicate to the parser when a match is found.

Parsing Stucture

The parsing structure gives the email parser the ability to understand input data stream at the application level. It defines the semantics of tokens as they are detected by the lexical analyzer and maintains the contextual state of the data stream. The hardware logic required for the parsing structure is determined from the input grammar using the *FIRST* and *FOLLOW* set algorithms as described in Chapter 3. The production list of the email grammar defines all of the possible tokens and transitions for the grammar.

4.3.4 Data Sets and Results

To test the effectiveness of the ALPS email parsing circuit, five different data sets were created. Each data set consisted of 10,816 email messages in 14 different languages. The email headers for each of the emails in all five data sets were similar in both size and content to the email header shown in Figure 4.13. The size of the email body was different for the five different data sets, with the smallest email body being 75 bytes and the largest being 1200 bytes. The source text for the email bodies was the same for all five data sets. Therefore, the first 75 bytes of all 1200 byte emails is identical to the first 75 bytes of the corresponding email in the 75 byte data set (and all other data sets). Table 4.3 shows the email body sizes for all five different data sets.

Each of the five data sets was first created as 10,816 text files that consisted of the email headers and body. A tool was built to convert these text files into 10,816 TCP flows that could be replayed into the FPX hardware for live testing.

The results for processing the 300 byte emails are shown in Table 4.2. The *Lang ID* column is a unique number that HAIL uses to represent each language. The *Language* column shows the 14 different languages that were part of the experimental data set. The *TRUTH* column represents the number of documents in the data set that were actually the given language. The *HAIL* column represents the number of documents HAIL reported as

Lang ID	Language	TRUTH	HAIL	# Correct	# Incorrect	ALPS+HAIL	# Correct	# Incorrect
1	Albanian	1	0	0	0	6	1	5
2	Arabic_trans	1447	390	390	0	1448	1447	1
4	Czech	29	1	1	0	77	21	56
5	English	2691	0	0	0	2535	2321	214
6	Estonian	1	10280	1	10279	28	1	27
7	French	916	0	0	0	875	835	40
8	German	700	0	0	0	604	585	19
13	Italian	789	0	0	0	960	739	221
20	Norwegian	43	144	39	105	75	40	35
24	Portuguese	1634	0	0	0	1146	1103	43
28	Spanish	2514	0	0	0	2346	2130	216
29	Swedish	20	0	0	0	1	0	1
32	Turkish	20	0	0	0	4	3	1
34	Uzbek	11	0	0	0	3	3	0
	TOTALS	**10816**	**10815**	**431**	**10384**	**10108**	**9229**	**879**
	% Correct			**3.98%**			**85.33%**	

Table 4.2: Language identification results for the 300-byte data set

the given language when operating alone. The *ALPS+HAIL* column represents the number of documents HAIL reported as the given language when each flow was preprocessed using the ALPS email parser. Note that the total number of documents in the *HAIL* and the *ALPS+HAIL* columns are not the same as the total number of documents in the *TRUTH* column. This is the result of some documents being reported as a language that was not represented in the data set (i.e. some documents may have been reported with a language ID of 35 which is not shown in the table).

The data in Table 4.2 indicates that the email headers used in the data set have a strong negative affect on the language identification results for the 300 byte emails. From the results, it appears that the email headers have a significant number of Estonian tetra-grams. When using HAIL alone, 10,280 out of 10,816 documents are identified as Estonian; only one of those documents is actually Estonian. This is because the email headers in the data set are 1500 bytes, whereas the email body is only 300 bytes. This means HAIL extracted five times more tetra-grams from the email headers than it extracted from the email body. This large disparity gave the email headers a greater significance than the email body when counting the number of tetra-grams to identify the language.

Overall, for the 300 byte data set, it is shown that HAIL alone only correctly identified the language of 3.98% (431 out of 10,816) of the documents, whereas ALPS+HAIL correctly identified the language of 85.33% (9,229 out of 10,816) of the documents.

The percentage of correctly identified documents for all five of the data sets is shown in Table 4.3. The same data is represented graphically in Figure 4.18. The data for HAIL alone has the expected behavior. When the size of the email header is significantly larger than the email body, the output of HAIL alone is skewed towards the language identified

Body Size	# Correct	% Correct HAIL Alone	# Correct	% Correct ALPS+HAIL
75-bytes	24	0.22%	5006	46.28%
150-bytes	38	0.35%	7532	69.64%
300-bytes	431	3.98%	9229	85.33%
600-bytes	4925	45.53%	9699	89.67%
1200-bytes	7290	67.40%	9837	90.95%

Table 4.3: Percentage of correctly classified documents for HAIL alone and ALPS+HAIL

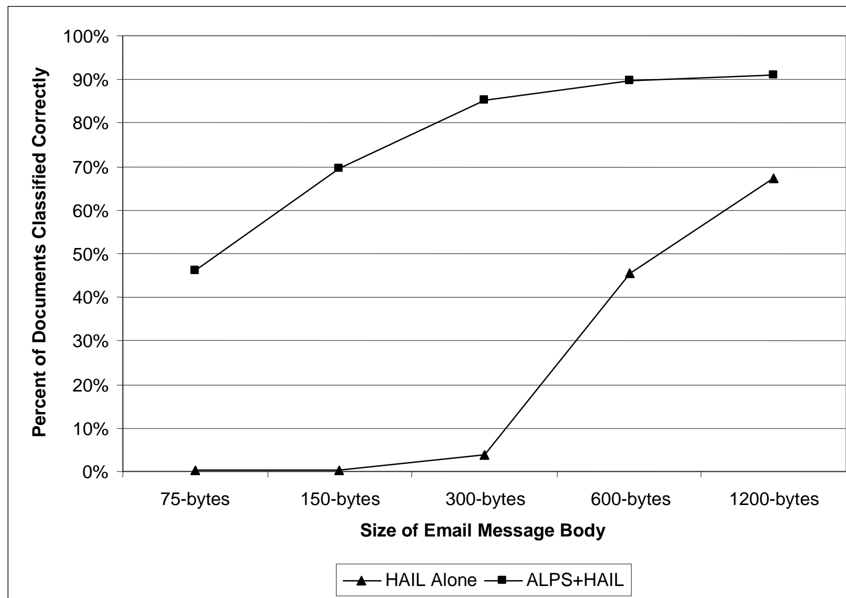


Figure 4.18: Percentage of documents correctly classified by HAIL for each data set

in the email header. However, as the size of the email body increases and becomes a larger percentage of the complete email message, the results of HAIL alone become more accurate. As illustrated by the trend in Figure 4.18, if the email body is large enough in comparison to the email header, it will counterbalance the affects of the email header.

The data for ALPS+HAIL in Figure 4.18 illustrates that the accuracy of HAIL increases as the size of a clean document increases. HAIL is more likely to correctly identify the language of a larger documents due to the larger number of available tetra-grams.

Table 4.4 and Figure 4.19 show the percent increase in accuracy that is achieved when using ALPS+HAIL as opposed to HAIL alone. For large emails where the size of the email body is close to the size of the email headers using ALPS provides a 34.94% improvement

Data Set	% Increase
75-bytes	20758.33%
150-bytes	19721.05%
300-bytes	2041.30%
600-bytes	96.93%
1200-bytes	34.94%

Table 4.4: Increase in accuracy when using ALPS+HAIL as opposed to HAIL alone

on the experimental data set. For smaller emails, using ALPS as a preprocessor to HAIL provides over 200× improvement.

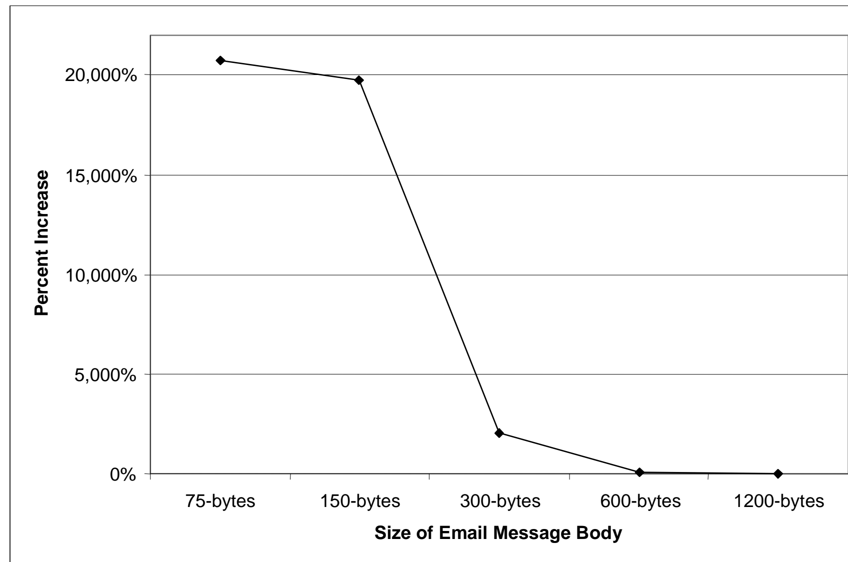


Figure 4.19: Increase in accuracy when using ALPS+HAIL

4.4 Chapter Summary

This chapter described two example applications of the regular language parser from Chapter 3. The first application was a content-based router that routes packets based on the string values extracted from the packet. The content-based router was generated from an XML grammar and is capable of presenting specified values embedded in the XML message to a routing module. The extracted value can be used to route packets accordingly.

The second example application was an email processor. The email processor was automatically generated from the email specification. The utility of the email processor was illustrated through its use as a preprocessor for a language identification module. The email processor was used to remove email headers from email messages and present clean text to the language identification module. Using the email processor improved the results of the language identification module by approximately 35% for large email messages. A 200× improvement was seen for small email messages.

Part II

Hardware Architectures for Accelerating RNA Secondary Structure Alignment

Chapter 5

RNA Secondary Structure Alignment

The hardware-accelerated parsing techniques described in Part I have been developed for streaming networking applications. However, there are other areas that can also benefit from the ability to parse large volumes of data at very high speeds. Specifically, the area of bioinformatics utilizes common parsing algorithms to compare DNA, RNA, and protein sequences. Part II of this work focuses on a high-speed hardware architectures designed to parse large genome databases in an effort to find homologous RNA molecules.

5.1 Introduction

In the field of bioinformatics, sequence alignment plays a major role in classifying and determining relationships among related DNA, RNA, and protein sequences. Sequence alignment is very well studied in the areas of DNA and protein analysis, and many tools have been developed to aid research in these areas [2, 55, 35]. However, only recently has the importance of non-coding RNAs (ncRNAs) been discovered [69].

An ncRNA is an RNA molecule that is not translated into a protein, but instead performs some other cellular process. Examples of ncRNA molecules include transfer RNAs (tRNAs) and ribosomal RNAs (rRNAs). These molecules are typically single-stranded

nucleic acid chains that fold upon themselves to create intramolecular base-paired hydrogen bonds [23]. The result of these bonds is a complex three-dimensional configuration (shape) that partially defines the function of an RNA molecule. This configuration is known as the *secondary structure* of the RNA molecule. Two molecules are said to be *homologous* if they have similar to identical secondary structures.

Homologous RNA molecules are likely to have similar biological functions, even though they may have dissimilar primary sequences. For example, the primary sequence `AAGACUUCGGAUC` creates the secondary structure shown in the left portion of Figure 5.1(b). A homologous molecule with a highly dissimilar primary sequence could result from exchanging the sequence's `G` and `C` nucleotides, or by substituting `U` for `C` and `A` for `G`.

The DNA sequences from which ncRNA molecules are transcribed are known as RNA genes. However, unlike protein-coding genes, RNA genes cannot be easily detected in a genome using statistical signals [58]. Moreover, because related RNA genes tend to exhibit poor primary sequence conservation, techniques developed for comparing DNA sequences are ill-suited for detecting homologous RNA sequences [23]. Instead, techniques have been developed that utilize the consensus secondary structure of a known RNA family to detect new members of that family in a genome database [25]. To date, these techniques have proven to be very computationally complex and can be quite demanding even for the fastest computers [24].

The consensus secondary structure of an RNA family can be represented as a stochastic context-free grammar (SCFG) that yields high-probability parses for, and only for, the primary sequences of molecules belonging to the family [64]. Several tools have been developed that employ SCFGs as a means of detecting homologous RNA molecules in a genome database. At the forefront of these tools is the Infernal software package [37] and the Rfam database [33] which currently includes SCFGs for over 600 RNA families.

Infernal has shown great success in its ability to detect homologous RNA sequences in genome databases. Additionally, since it was first released in 2002, Infernal has been frequently updated and extended to incorporate different heuristics to decrease the computational costs of finding homologous RNA sequences [74, 53]. However, even with these

improvements, the time to scan a large genome database can still take many hours, days, or even years, which greatly limits the usefulness of tools such as INFERNAL.

This work examines architectures for finding RNA homologs using custom hardware. The remainder of this chapter presents related work and a brief background on the techniques used to model RNA secondary structures and the algorithms used for finding homologous sequences in a genome database. Chapter 6 presents a highly-pipelined initial architecture that is capable of scanning genome databases at very high speeds. Although extremely fast, the resource requirements for the baseline architecture prohibit the architecture from practical use for even average size RNA models. In Chapter 7, a more practical architecture for scanning genome databases is presented that involves scheduling computations onto a set of parallel processing elements.

5.2 Related Work

While heuristics [13, 44, 48, 73, 74, 53] can dramatically accelerate the computationally complex process of detecting homologous RNA sequences in genome databases, the completeness and quality of the results are often sacrificed. For example, some heuristics pre-filter sequences based on their *primary* sequence similarity, applying the more complex secondary structure alignment algorithms only on the sequences that pass the filter [73, 74]. However, because these filters are based on the consensus primary sequence of an RNA family, they do not work well on families that have limited primary sequence conservation [53]. Furthermore, as an RNA family grows, and more variation is introduced into the consensus primary sequence, these filtering techniques may become ineffective.

Concurrency can also be exploited to improve performance. Liu and Schmidt [45] utilized coarse-grained parallelism and a PC cluster to achieve a 36× speedup. While not directly related RNA secondary structure alignment, Aluru [3] and Schmidt [62] present approaches for taking advantage of finer grained parallelism in primary sequence comparisons.

5.3 Background

Transformational grammars were first described as a means for modeling sequences of nucleic acids by Searls [64]. Grammars provide an efficient means of modeling the long range base-pair interactions of RNA secondary structures. More specifically, stochastic context-free grammars (SCFGs) provide the framework required to represent probabilistic models of both the primary sequence and the base-paired secondary structure of an RNA molecule. Given a multiple sequence alignment of an RNA family, one can construct a profile-SCFG, also referred to as a covariance model (CM), that can subsequently be used to detect homologs in a genome database [25]. The remainder of this section provides background on CMs and how they are used for scanning genome databases.

5.3.1 Covariance Models

A covariance model (CM) is a specialized SCFG developed specifically for modeling both the primary sequence and secondary structure of RNA families. Figure 5.1 shows the development of a CM as follows. In Figure 5.1(a), three RNA sequences are aligned, with connected boxes showing the consensus base pairs that bind to form the secondary structure; the top sequence's secondary structure is called out in Figure 5.1(b). The consensus secondary structure of all three sequences is represented by the directed binary tree in Figure 5.1(c), whose nodes indicate the binding pattern of the sequences' nucleotides.

While the three sequences of Figure 5.1(a) fit identically onto the binary tree, other sequences may fit the model only if appropriate insertions and deletions are applied. Such edits are accommodated by state transitions within a node. For three nodes of the binary tree in Figure 5.1(c), Figure 5.1(d) shows those nodes' internal states and possible transitions.

For the purposes of assessing the fitness of an RNA string for membership in an RNA family, the RNA string is parsed using grammar rules that model state transitions such as those in Figure 5.1(d). Each state can be represented as a SCFG production rule that has the form of one of the nine non-terminal (or state) types shown in Table 5.1. Each state has its own set of *emission probabilities* for each of the single or base-paired residues

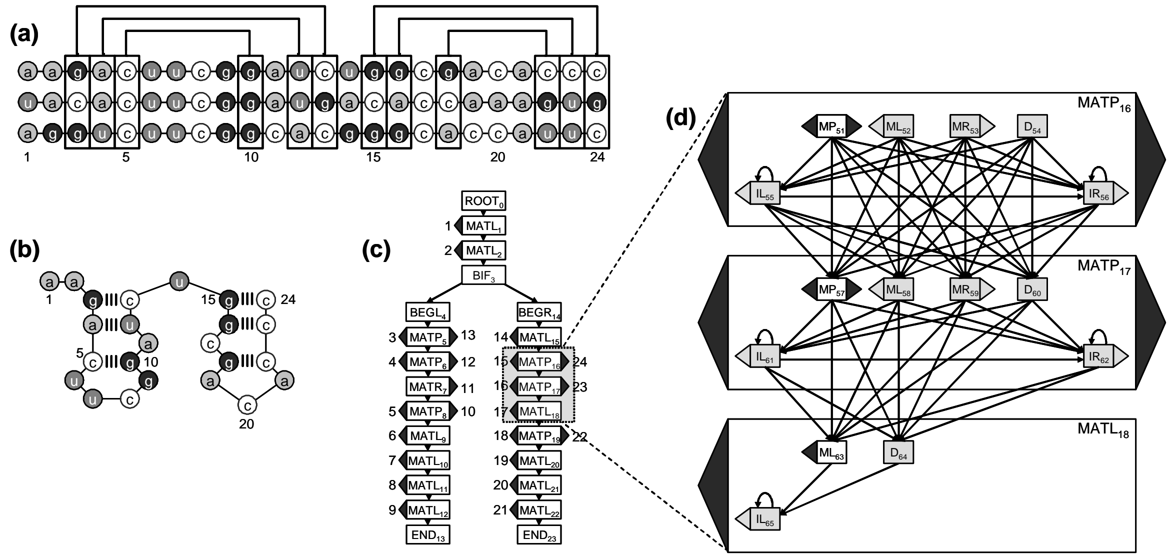


Figure 5.1: A reproduction of an example CM from [23, 53]

State Type	Description	Production	Emission	Transition
MP	Pair Emitting	$P \rightarrow x_i Y x_j$	$e_v(x_i, x_j)$	$t_v(Y)$
ML / IL	Left Emitting	$L \rightarrow x_i Y$	$e_v(x_i)$	$t_v(Y)$
MR / IR	Right Emitting	$R \rightarrow Y x_j$	$e_v(x_j)$	$t_v(Y)$
B	Bifurcation	$B \rightarrow SS$	1	1
D	Delete	$D \rightarrow Y$	1	$t_v(Y)$
S	Start	$S \rightarrow Y$	1	$t_v(Y)$
E	End	$E \rightarrow \epsilon$	1	1

Table 5.1: Each of the nine different state types and their corresponding SCFG production rules

that can be emitted from that state. Additionally, each state has its own set of *transition probabilities* for the set of states to which it can transition.

A state that is denoted as an MP state generates (or emits) the base-pair (x_i,x_j) with some *emission probability* $e_v(x_i, x_j)$ and transitions to some state Y with some *transition probability* $t_v(Y)$. Likewise, a state that is an ML (or IL) state generates (or emits) the single residue x_i with some *emission probability* $e_v(x_i)$ and transitions to some state Y with some *transition probability* $t_v(Y)$. The B type bifurcation states represent a fork in the tree structure of the CM and always transition to two distinct S type states without emitting

any residues. S states are also members of the $ROOT$ node of the CM. A D type state is used to represent a residue that is part of the CM, but missing from the target genome sequence. Leaf nodes of the CM are represented as E states and do not emit any residues. For a more detailed discussion on CMs, refer to [23, 25].

5.3.2 Rfam Database

An online database, known as the Rfam database [33], currently contains multiple sequence alignments and CMs for over 600 RNA families. Rfam is an open database that is available for all researchers interested in studying algorithms and architectures related to RNA homology search. Since its inception, the Rfam database has seen continuous growth as more ncRNA families are identified. Figure 5.2 depicts the growth of the Rfam database over the last five years. Estimates suggest that there are several tens of thousands of ncRNAs in the human genome [49, 71]. The vast number of ncRNAs suggests the need for a high-performance alternative to software approaches for homology search.

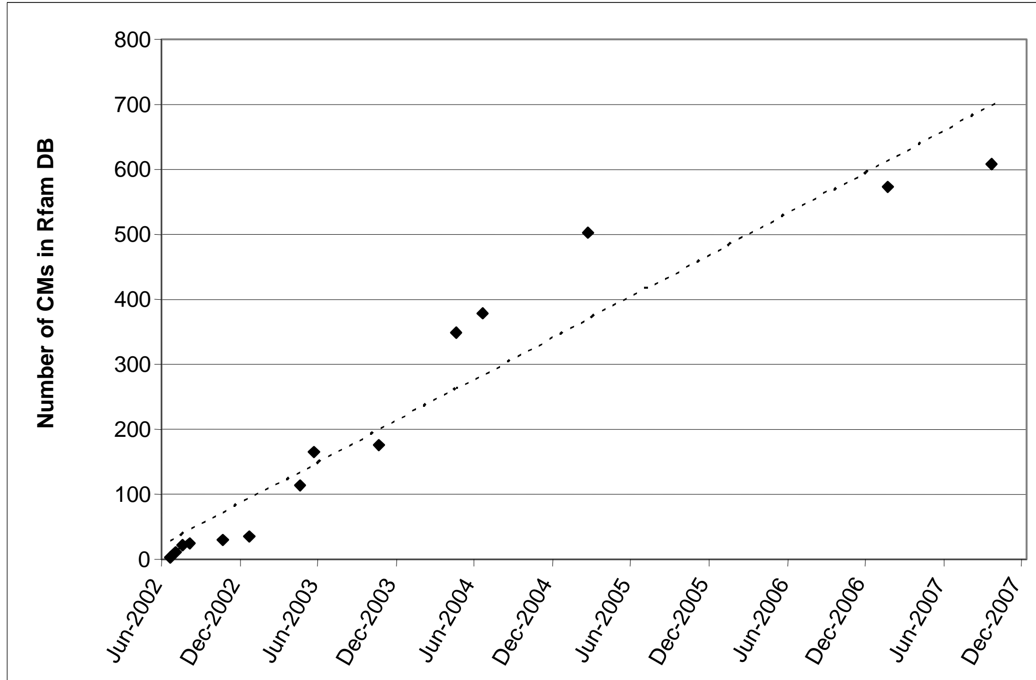


Figure 5.2: The number of covariance models in the Rfam database has continued to increase since its initial release in July of 2002.

5.3.3 Database Search Algorithm

CMs provide a means to represent the consensus secondary structure of an RNA family. This section describes how to utilize a CM to find homologous RNA sequences in a genome database.

Aligning an RNA sequence to a CM can be implemented as a three-dimensional CYK (Cocke-Younger-Kasami) [21, 75, 38] dynamic programming (DP) parsing algorithm. Each state in the CM is represented as a two-dimensional matrix with rows 0 through L where L is the length of the genome database, and columns 0 through W where W is the length of the window (i.e. the longest subsequence of the genome database) which should be aligned to the CM. Figure 5.3 shows an example of the alignment window as it scans across a genome database.

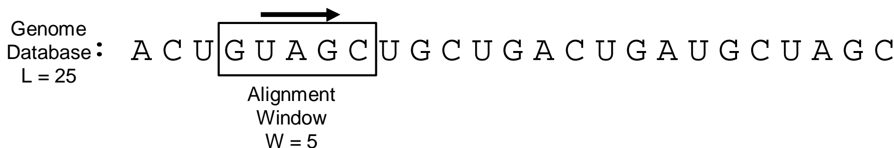


Figure 5.3: An alignment window scans across the genome database. Each window is aligned to the CM via the DP parsing algorithm.

The DP algorithm initializes the three-dimensional matrix for all parse trees rooted at E states of the CM and for all subsequences of zero length. The algorithm then computes the scores for the DP matrix starting from the END nodes of the CM and working towards the $ROOT_0$ node. Figure 5.4 shows the DP recurrences for aligning a genome database sequence $x_1...x_L$ to a CM where:[1]

- $x_1...x_L$ is a sequence of residues (A, C, G, U)
- $x_i...x_j$ is a subsequence of $x_1...x_L$ where $1 \leq i, j \leq L$ and $i \leq j$
- i is the start position of the subsequence $x_i...x_j$
- j is the end position of the subsequence $x_i...x_j$

[1] Figure 5.4 shows the equations used in version 0.81 of the INFERNAL software package. However, it should be noted that as of INFERNAL version 0.4 the emission probabilities for all insert states are set to 0 in lieu of the values stored in the CM.

- d is the length of the subsequence $x_i ... x_j$ where $1 \leq d \leq W$ and $i = j - d + 1$
- $\gamma_v(j, d)$ is the log-odds score for the most likely parse tree rooted at state v that generates the subsequence that ends at location j of the genome sequence and has length d (i.e. the subsequence $x_{j-d+1} ... x_j$)
- M is the number of states in the CM
- v indexes a state from the CM where $0 \leq v < M$
- s_v identifies the state type of v (i.e. MP, ML, etc.)
- C_v is the set of states to which state v can transition
- $t_v(y)$ is the log-odds probability that state v transitions to the state y
- $e_v(x_i)$ is the log-odds probability that state v generates (or emits) the residue x_i
- $e_v(x_j)$ is the log-odds probability that state v generates (or emits) the residue x_j
- $e_v(x_i, x_j)$ is the log-odds probability that state v generates (or emits) the residues x_i and x_j

Initialization: $for\ j = 0\ to\ L,\ v = M - 1\ \ to\ 0:$

$$\gamma_v(j, 0) = \begin{cases} 0 & if\ s_v = E \\ max_{y \in C_v}[\gamma_y(j, 0) + t_v(y)] & if\ s_v \in \{D, S\} \\ \gamma_y(j, 0) + \gamma_z(j, 0) & if\ s_v = B, C_v = (y, z) \\ -\infty & otherwise \end{cases}$$

Recursion: $for\ j = 1\ to\ L,\ d = 1\ to\ W\ (and\ d \leq j),\ v = M - 1\ \ to\ 0:$

$$\gamma_v(j, d) = \begin{cases} -\infty & if\ s_v = E \\ -\infty & if\ s_v = MP\ and\ d < 2 \\ max_{0 \leq k \leq d}[\gamma_y(j - k, d - k) + \gamma_z(j, k)] & if\ s_v = B, C_v = (y, z) \\ max_{y \in C_v}[\gamma_y(j, d) + t_v(y)] & if\ s_v \in \{S, D\} \\ max_{y \in C_v}[\gamma_y(j, d - 1) + t_v(y)] + e_v(x_i) & if\ s_v \in \{ML, IL\} \\ max_{y \in C_v}[\gamma_y(j - 1, d - 1) + t_v(y)] + e_v(x_j) & if\ s_v \in \{MR, IR\} \\ max_{y \in C_v}[\gamma_y(j - 1, d - 2) + t_v(y)] + e_v(x_i, x_j) & if\ s_v = MP\ and\ d \geq 2 \end{cases}$$

Figure 5.4: The initialization and recursion equations for the dynamic programming algorithm

The DP algorithm scores all subsequence of length 0 through W rooted at each of the CM states $M - 1$ down to 0. The final score for a subsequence $x_i ... x_j$ is computed in the start state of the $ROOT_0$ node as $\gamma_0(j, j - i + 1)$ (i.e. $\gamma_0(j, d)$). For example, the final

score for the subsequence $x_{10}...x_{15}$ would be located in $\gamma_0(15,6)$. Generally, subsequences with final scores that are greater than zero represent good alignments to the CM. The DP algorithm has a $O(M_aLW + M_bLW^2)$ time complexity and a $O(M_aW + M_bW^2)$ memory complexity where M_a is the number of non-bifurcation states and M_b is the number of bifurcation states [23].

5.4 Expressing Covariance Models as Task Graphs

To aid in the development of architectures for accelerating the computations described in Section 5.3.3, it is helpful to think of the three-dimensional DP matrix required by the computation as a directed acyclic task graph. With the exception of matrix cells that are a part of bifurcation states, each cell (v, j, d) in the DP matrix can be represented as a single node in a task graph. Nodes are connected using the parent/child relationships that are described in CM, as well as the DP algorithm shown in Figure 5.4. The CM specifies the parent/child relationships between the states v. The DP algorithm specifies the parent/child relationships for cells (j, d) within those states. To connect nodes in the task graph, edges are created from a child node to a parent node (i.e. from nodes in higher numbered states to nodes in lower numbered states). This corresponds to the direction of the computation which starts at state $v = M - 1$ and ends at state $v = 0$. For nodes from non-bifurcation states, the maximum number of incoming edges is six. As can be seen in Figure 5.1(d), the structure of CMs limits the maximum number of children that a state may have to six. The minimum number of incoming edges is one.

5.4.1 Bifurcation States

As mentioned in the previous section, matrix cells from bifurcation states are not represented as a single node in a task graph representation of the DP computation. As the equations in Figure 5.4 show, cells for bifurcation states are treated differently than cells for non-bifurcation states. Unlike non-bifurcation states, the number of children (i.e. incoming edges) that a cell from a bifurcation state may have is not limited to six. Instead, the number of children that a bifurcation cell may have is limited only by the window size W

of the CM. More specifically, the number of individual additions required by a cell in a bifurcation state is $W + 1$. The number of comparisons required to find the maximum of those additions is $\log_2(W + 1)$. For the CMs in the Rfam 8.0 database, W can range from as low as 40 to as high as 1200.

To make the computations required for cells in bifurcation states more like the computations for cells in non-bifurcation states, each bifurcation computation is broken up into a series of smaller computations. It is these smaller computations that are mapped into nodes in the task graph that represents the CM. As mentioned above, the computation for each bifurcation node can be broken down into a set of additions and comparisons. Because non-bifurcation nodes in the task graph can represent as few as one addition and as many as six, it is given that an architecture will need to have resources capable of handling computations in sets of one to six. It is those resources onto which the computations for bifurcation nodes must be mapped. Given the above, the number of addition nodes required in the task graph for a single cell in a bifurcation state can be expressed as:

$$\left\lceil \frac{k+1}{x} \right\rceil \quad \text{where } x \text{ is the number of additions per node and } 0 \leq k \leq d$$

The number of addition nodes required in the task graph for all cells in a single window W of a bifurcation state can be expressed as:

$$\sum_{j=0}^{W} \sum_{k=0}^{j} \left\lceil \frac{k+1}{x} \right\rceil \quad \text{where } x \text{ is the number of additions per node and } 0 \leq k \leq d$$

The number of comparison nodes required in the task graph for a single cell in a bifurcation state is:

$$\sum_{i=1}^{\log_x \left\lceil \frac{k+1}{x} \right\rceil} \left\lceil \frac{\left\lceil \frac{k+1}{x} \right\rceil}{x^i} \right\rceil \quad \text{where } x \text{ is the number of comparisons per node and } 0 \leq k \leq d$$

The number of comparison nodes required in the task graph for all cells in a single window W of a bifurcation state is:

$$\sum_{j=0}^{W}\sum_{k=0}^{j}\sum_{i=1}^{\log_x\left\lceil\frac{k+1}{x}\right\rceil}\left\lceil\frac{\left\lceil\frac{k+1}{x}\right\rceil}{x^i}\right\rceil \quad \text{where } x \text{ is the number of comparisons per node and } 0 \leq k \leq d$$

The total number of nodes required in the task graph for all cells in a single window W of a bifurcation state is:

$$\sum_{j=0}^{W}\sum_{k=0}^{j}\left(\left\lceil\frac{k+1}{x}\right\rceil + \sum_{i=1}^{\log_x\left\lceil\frac{k+1}{x}\right\rceil}\left\lceil\frac{\left\lceil\frac{k+1}{x}\right\rceil}{x^i}\right\rceil\right)$$

5.5 Covariance Model Numeric Representation in Hardware

Before developing architectures to accelerate the algorithm in Section 5.3.3 it is important to understand the range of values that can be generated by the algorithm.

CMs included in the Rfam 8.0 database represent transition and emission probabilities as log-odds probabilities accurate to three decimal places [33]. The INFERNAL software package [37] represents these values as floating-point values and performs floating-point addition to compute the final log-odds score of the most likely parse of an input sequence. Because floating-point units are expensive in terms of hardware resource utilization, it is desirable to avoid floating-point computation in hardware. Fortunately, the DP recurrence, described in Section 5.3.3, requires only floating-point addition, and no floating-point multiplication. This means that all log-odds probabilities can be converted into signed integer values by multiplying them by 1000, which can subsequently be summed quickly and efficiently using integer adders in hardware. Multiplying the CM probabilities by 1000 allows all data in the model to be utilized so there is no data loss during the DP computation in hardware.

To utilize hardware logic and memory resources most efficiently, it is desirable to represent the signed integer scores computed by the DP algorithm with as few bits as

possible. Using too many bits would result in inefficient use of precious fast memory-structures (such as block RAM) and could potentially limit the size of the CMs that can be processed using the architecture. Additionally, too many bits would result in adders with excessively high latencies and degrade the overall performance of the architecture. Using too few bits could cause the adders to become saturated on high scoring sequences, resulting in computational errors, and ultimately incorrect scores being reported for those sequences.

To determine the minimum number of bits required to avoid saturation, one must know the maximum values expected in the DP computation. In this work, the maximum scores expected for all of the CMs in the Rfam 8.0 database were computed as follows. Because there is often a penalty associated with insertions and deletions with respect to the consensus structure, the maximum likelihood path through any CM $\mathcal{M}$ is the consensus path $\vec{\pi}$. To compute the maximum score expected $\gamma_{max}(\vec{\pi}|\mathcal{M})$, let v be a state in $\mathcal{M}$, let C_v be the set of states that are children of the state v, let $t_v(y)$ be the transition probability from state v to state y where $y \in C_v$, and let e_v be the set of emission scores associated with state v. Then for any $\mathcal{M}$ with consensus path $\vec{\pi}$:

$$\gamma_{max}(\vec{\pi}|\mathcal{M}) = \sum_{v=0, v\in\vec{\pi}}^{M-1} t_v(y) + \max\ e_v$$

where $y \in C_v, y \in \vec{\pi}$

That is, the sum of the transition probabilities along the consensus path plus the maximum emission scores for each state along that path produces the maximum score that can be computed for that model.

In computing the maximum scores for each of the CMs in the Rfam 8.0 database, it was found that the maximum score over *all* models is 726.792, which was computed for the model with the Rfam ID RF00228. When converted to a signed integer as described earlier, the maximum value is 726,792 which can be represented using $\lceil \log_2(726, 792) \rceil$ bits $+$ 1 sign bit $= 21$ bits. Using 21-bits ensures that there will be no loss of precision in a hardware architecture for models that have maximum scores of up to $\frac{2^{21-1}}{1000} = 1048.576$.

Therefore, all models in the Rfam 8.0 database can be processed accurately without saturating adders in a hardware architecture. If new CMs are developed in the future that have maximum scores that are greater than 1048.576, additional bits will be required to represent those scores.

A graph illustrating the distribution of maximum scores of all the CMs in the Rfam 8.0 database is shown in Figure 5.5. The maximum scores range from 28.683 (Rfam ID RF00390) to 726.792 (Rfam ID RF00228). The graph also shows the number of bits required to compute scores for the CMs without saturation. For example, using 18 bits to represent probabilities supports only 58% of the CMs in the Rfam 8.0 database. The remaining 42% of the CMs have maximum scores that are greater than 131.072 (i.e. $\frac{2^{18-1}}{1000}$) and will saturate 18-bit adders. Using 19 and 20 bits, the architecture would be capable of supporting 91% and 98% of the CMs respectively. As shown in Figure 5.5, all Rfam 8.0 models can be supported using 21 bits.

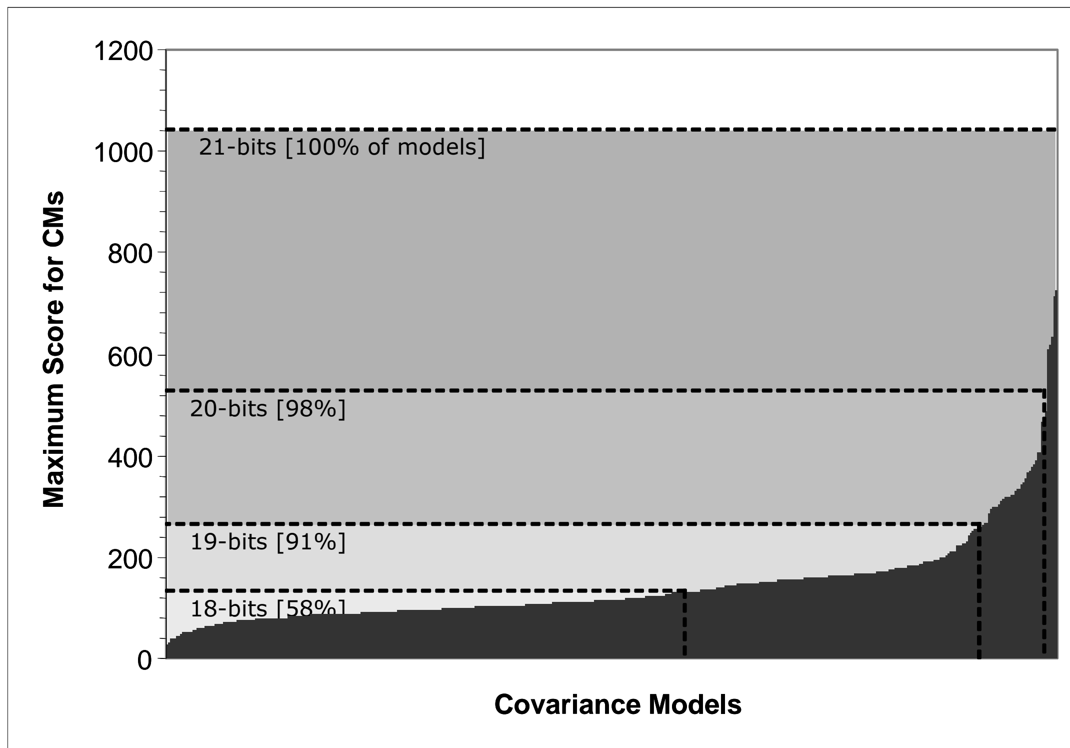


Figure 5.5: Distribution of maximum scores of all CMs in the Rfam 8.0 database

Figure 5.6 illustrates the relationship between the number of states in a CM and the maximum score computable for that CM. The graph shows a linear relationship, which is to be expected, since the scores are computed as a summation over the states in the CM. As the number of states in a CM increases, the maximum score computable also increases. Using the trend seen in Figure 5.6, the maximum number of states that can be supported using 21 bits can be approximated to be 2750 states. This is approximately 1.5× more than the number of states in any currently available CM.

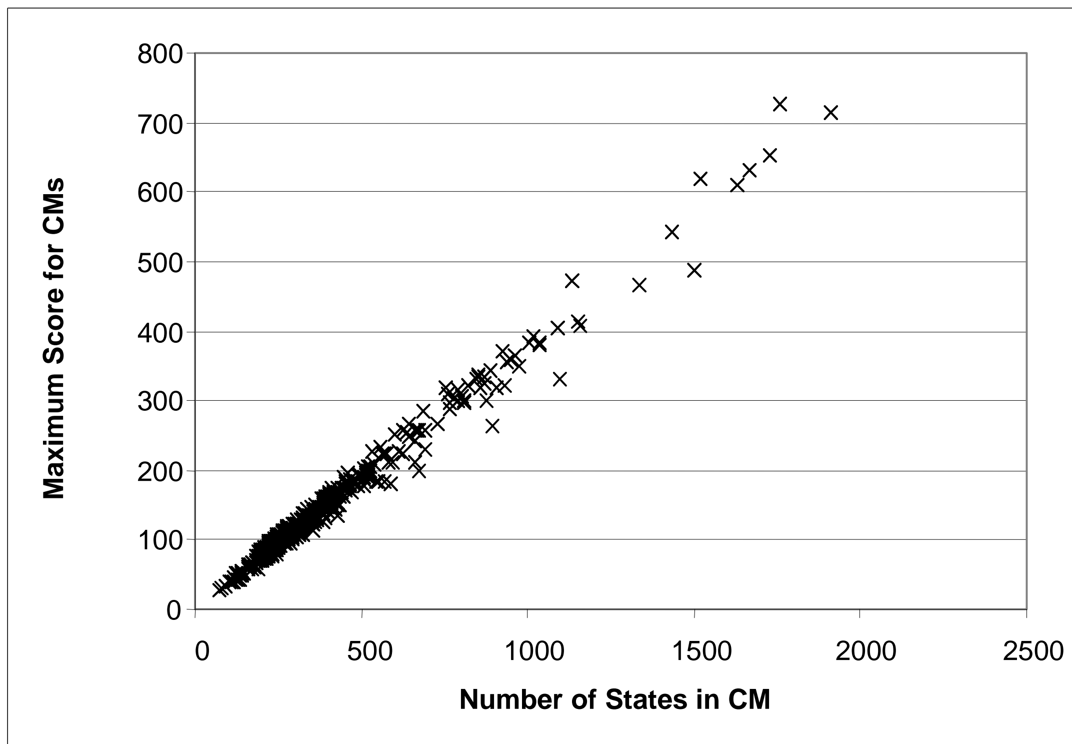


Figure 5.6: Graph depicting the linear relationship between the number of states in a CM and the maximum score computable for that CM

Although positive saturation is of much greater concern for this architecture, it is still important to note that negative saturation is also a possibility. However, unlike when finding the maximum possible score for a CM, there is no single path through a CM that defines the minimum possible score. In fact, there are many paths through a CM, all certain to include some combination of insertion and deletion states, that will result in very low

scores. To determine which of these paths provides the minimum possible score for a CM, a score must be computed for all paths in the CM. However, this is still insufficient to ensure that negative saturation cannot occur. Because the number of insertions is bounded by the window size W, and W can be increased (or decreased) by a knowledgeable computation biologist, it is impossible to definitively compute a minimum score for a CM. Increasing W increases the number of insertions possible, thereby decreasing the minimum possible score.

Instead of trying to determine the minimum score possible for a CM, it is straightforward to show that any sequence that causes negative saturation cannot also exceed a score reporting threshold ρ when $\rho > 0$. Scores that are less than zero are uninteresting since a sequence that generates such a score does not align well to the CM.

The maximum positive value that can be computed for a CM $\mathcal{M}$ has already been defined as $\gamma_{max}(\vec{\pi}|\mathcal{M})$, which can be represented as a two's complement signed integer using k bits. That is, $0 < \gamma_{max}(\vec{\pi}|\mathcal{M}) \leq \frac{2^k}{2}$. In a two's complement system, these k bits are also sufficient to represent $-(\frac{2^k}{2} + 1)$ where $-(\frac{2^k}{2} + 1) < -\gamma_{max}(\vec{\pi}|\mathcal{M}) < 0$. Therefore, for any negative value γ_{nsat} that causes negative saturation, $\gamma_{nsat} < -(\frac{2^k}{2}+1) < -\gamma_{max}(\vec{\pi}|\mathcal{M}) < 0$. From this, it follows that $\gamma_{nsat} + \gamma_{max}(\vec{\pi}|\mathcal{M}) < 0 \leq \rho$. Thus it is shown, that if negative saturation does occur, there is not enough "positive value" (i.e. no path through $\mathcal{M}$) that will allow a sequence to generate a final score that is greater than or equal to ρ. Therefore, negative saturation may cause computational errors, but these errors are ignorable since any score containing one of these errors will never be reported.

5.6 Chapter Summary

This chapter provided a background on RNA secondary structure alignment. Included was a description of covariance models, and the dynamic programming parsing algorithm used to search residue databases for homologous RNA sequences. Additionally, a technique to convert the dynamic programming matrix into a directed task graph was also presented. Finally, covariance models from the Rfam 8.0 database were characterized to determine some of the hardware requirements necessary to develop an architecture that can perform RNA secondary structure alignment.

Chapter 6

The Baseline Architecture

To better understand the level of acceleration achievable using custom hardware, a baseline architecture was developed. The baseline architecture computes the value for each cell of the dynamic programming (DP) matrix immediately after (i.e. on the next clock cycle) all of the cells that it depends on have been computed. That is, the baseline architecture computes all the values for the three-dimensional DP matrix in the fewest possible clock cycles. As a single engine solution, the baseline architecture represents an optimal solution to the DP problem described in Section 5.3.3. However, as discussed later in this chapter, the resource requirements of the baseline architecture make it impractical for even small CMs.

6.1 Overview

The baseline architecture takes advantage of the graph-like structure of covariance models (CMs) and converts the structure directly into a pipelined architecture of processing elements (PEs), where each PE represents a cell in the DP matrix. Provided a CM from the Rfam database, the pipelined architecture can be automatically generated specifically for that CM. The generated hardware can subsequently be programmed onto reconfigurable hardware for an optimal hardware solution for that specific CM.

As mentioned in Section 5.3.3, each state in a CM is represented as a two-dimensional matrix of width $W + 1$ and height $L + 1$, where W is the size of the window sliding over the target sequence in which to align the CM, and L is the length of the target sequence (typically, $W \ll L$). There are M such two-dimensional matrices, where M is the number of states in the CM. For a given CM, there are a total of approximately MWL matrix cell values that need to be computed to score the CM against all possible subsequences of a target sequence that are of length $\leq W$.

However, it is not necessary to have the hardware resources for a PE for each of the MWL cells in the DP matrix. Instead, the number of PEs required can be reduced by observing that each position of the sliding window can be computed independently of all other window positions. This means that the pipelined architecture only needs PEs for a single window position which can be reused for all other window positions. Because of hardware reuse, the actual number of PEs required in the pipeline is approximately $M(W + 1)^2$, which is much less than MWL for large genome databases.

6.2 Processing Elements

To best illustrate the baseline architecture, this chapter presents an example CM, the corresponding architecture, and implementation results. A small example CM, along with its expanded state notation, is shown in Figure 6.1. The CM represents a very small consensus secondary structure consisting of only three residues, where the first and the third residues are base-paired. The CM consists of four nodes and thirteen states, as shown in Figure 6.1.

Figure 6.2 provides a high-level view of how the CM is converted into a pipeline. States at the bottom of the CM are computed first and are thus at the beginning of the pipeline. The states in the $ROOT_0$ node are computed last and are thus at the end of the pipeline. A residue pipeline feeds the target sequence through a pipeline from which PEs can determine what emission score (if any) should be added to their score.

As described earlier, each state in a CM can be represented as a two-dimensional matrix of width and height $W + 1$. For the example CM described here, the window size was configured as $W = 3$, resulting in a total of thirteen 4×4 matrices. One of those

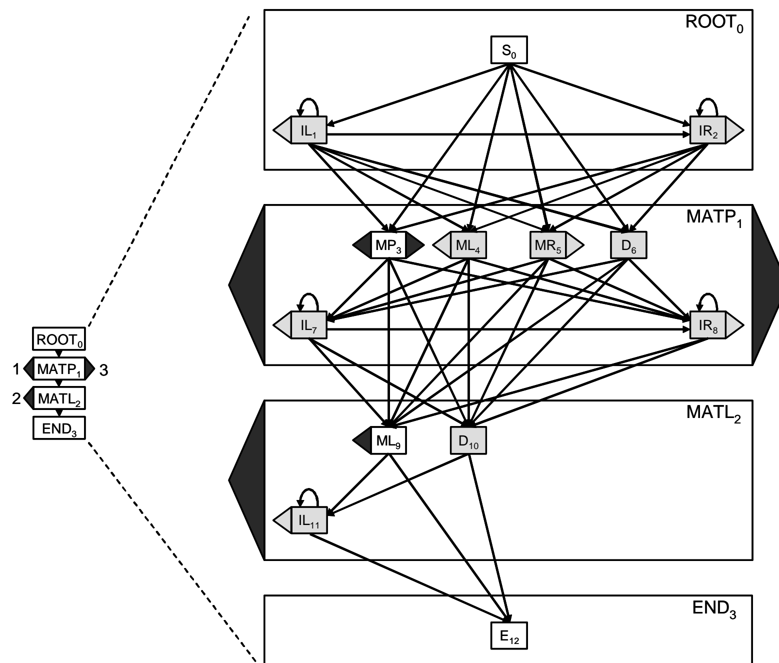


Figure 6.1: A small CM, consisting of four nodes and thirteen states, represents a consensus secondary structure of only three residues

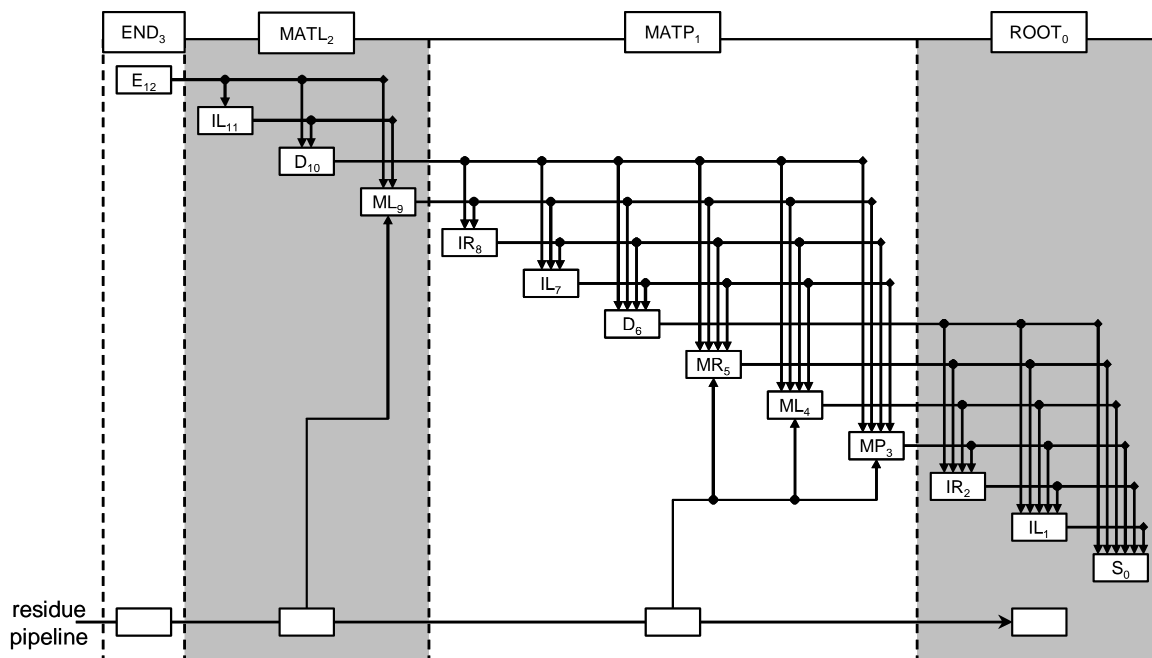


Figure 6.2: A high-level view of a pipeline for the baseline architecture

matrices representing state ML_4 is shown in Figure 6.3. Note that the first column of the matrix is initialized to $-\infty$ as described by the DP algorithm in Section 5.3.3. The dark gray cells in the matrix need not be computed as they represent negative values of i, the starting position of a subsequence. The remaining matrix cells contain the values that are computed as part of the DP computation. They represent the log-odds scores that the subsequences represented by those matrix cells are rooted at the given state.

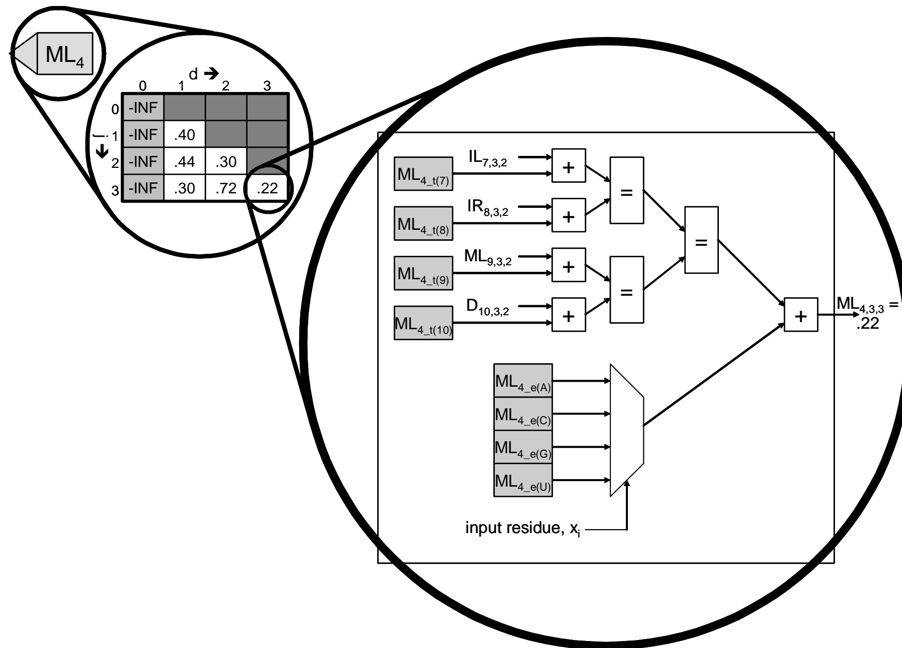


Figure 6.3: Each CM state is represented as a two-dimensional matrix, and each matrix cell is represented as a processing element containing adders and comparators.

Figure 6.3 also shows an example of a PE that is part of the pipelined architecture. The number of children states that a state may depend on ranges from one to six, depending on the structure of the CM. This particular PE is part of a state that has four children states as illustrated in Figure 6.1. Therefore, the maximization portion of the PE requires four adders and three comparators. An additional adder is included to factor the emission probability into the computation, which is dependent on the input residue at location x_i of the target sequence in an ML type state (MR states depend on the residue at location x_j of the target sequence, and MP states depend on the residue at both the x_i and the x_j

locations of the target sequence). The largest PE, which contains inputs for six children states, has a total of seven adders and five comparators.

In the PE shown in Figure 6.3, gray boxes represent the constant values for transition and emission probabilities from the CM. For example, $ML_{4_\mathrm{t}(7)}$ represents the transition probability from state ML_4 to state IL_7. The value $ML_{4_\mathrm{e}(\mathtt{G})}$ represents the probability that the residue G is emitted by state ML_4 of the CM. The other inputs, such as $IL_{7,3,2}$, represent the matrix cell values computed in the child states of state ML_4. More specifically, $IL_{7,3,2}$ represents the score output from the PE for the matrix cell $IL_{v,j,d}$ where v is the state number, j is the position of the last residue of the subsequence, and d is the length of the subsequence.

The output of the PE, $ML_{4,3,3}$ represents the score of the subsequence $x_1 ... x_3$ when rooted at state ML_4. This value is forwarded to the PEs in the pipeline that depend on it. For this CM, the PE $ML_{4,3,3}$ only has a single dependent, $S_{0,3,3}$ as per the structure of the CM shown in Figure 6.1 and the DP algorithm described in Section 5.3.3. Note that states IL_1 and IR_2 are also dependent on state ML_4. However, because both of those state types depend on values from column $d-1$, neither of them contain PEs that are dependent on the last column of a child state.

For the baseline architecture, all values are represented using 16-bit signed integers. This provides a sufficient number of bits to compute the results for small CMs without causing overflow. All adders and comparators in the hardware are implemented as 16-bit adders and comparators.

6.3 Pipeline

Based on the structure of the CM, PEs are created and wired together to create the pipeline for the baseline architecture. Figure 6.4 shows a small portion of the pipeline required for the CM in Figure 6.1. The portion shown represents the first three rows (i.e. $j = 0$ through $j = 2$) of the first three states (i.e. S_0, IL_1, and IR_2) in the CM. The final results for subsequences are output from the $S_{0,j,d}$ PEs which represent the S type states from the $ROOT_0$ node.

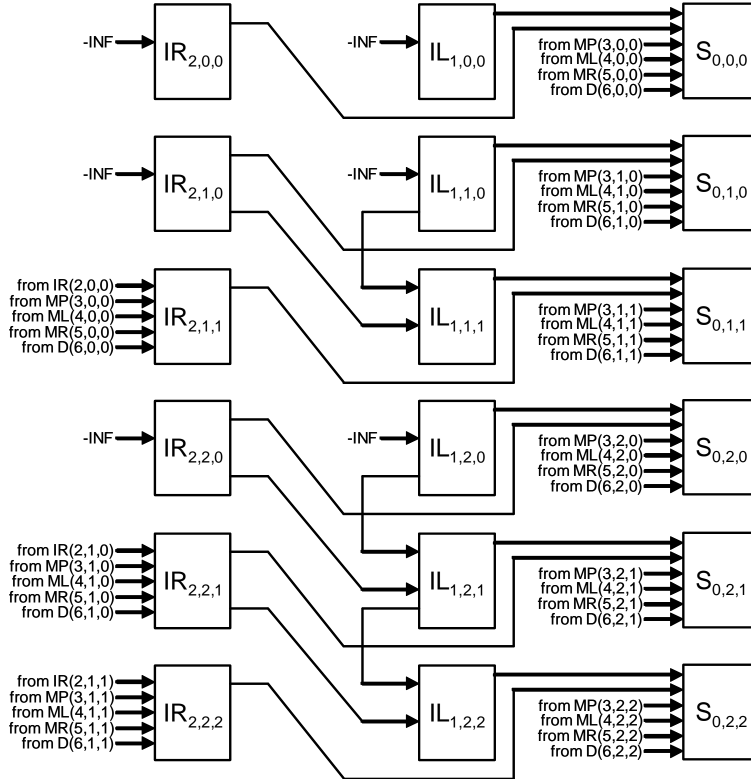


Figure 6.4: 18 of the 130 PEs required to implement the CM shown in Figure 6.1 using the baseline architecture. The full pipeline structure can be automatically generated directly from a CM.

6.4 Implementation Results

An implementation of the CM shown in Figure 6.1 was developed to compare the performance of the baseline architecture to the performance of the INFERNAL (version 0.81) software package. The evaluation system for the INFERNAL software contains dual Intel Xeon 2.8 GHz CPUs and 6 GBytes of DDR2 SDRAM running Linux CentOS 5.0.

The baseline architecture was evaluated on the same system using an FPGA expansion card connected to the system via a 100 MHz PCI-X bus. The FPGA expansion card contains a Xilinx Virtex-II 4000 FPGA. The baseline architecture was built using a combination of Synplicity's Synplify Pro v8.8.0.4 for synthesis and version 9.1i of Xilinx's ISE back-end tools for place-and-route. The implementation of the baseline architecture for the small CM shown in Figure 6.1 occupies 88% of the slices available on the Xilinx

Virtex-II 4000 FPGA and 8% of the block RAMs. The implementation is highly pipelined and can run at over 300 MHz, but was clocked at 100 MHz for the experiments presented here.

By default, INFERNAL's *cmsearch* tool does more work than the baseline architecture presented in this chapter. In particular, INFERNAL's *cmsearch* tool scans both the input database sequence as well as the complement of that sequence. The default parameters also prompt the *cmsearch* tool to output how subsequences mapped to a CM, as opposed to simply outputting the score for each of the high-scoring subsequences. To provide a more direct comparison, the extra work done by INFERNAL can be eliminated by specifying the *–toponly* and *–noalign* options to the *cmsearch* tool. These options prompt the *cmsearch* tool to scan only the input sequence, and to only provide scores for the high-scoring subsequences.

Another option available to INFERNAL's *cmsearch* tool includes the *–noqdb* option. By default, the latest versions of INFERNAL utilizes Nawrocki's Query Dependent Banding (QDB) heuristic [53]. The *–noqdb* option disables the QDB heuristic and provides a more direct comparison to the baseline architecture. For the experiments in this section, the performance with and without QDB are reported.

Table 6.1 compares the performance of INFERNAL with that of the baseline architecture. Testing was done on a randomly generated database sequence of 100 million residues. The first two entries in the table represent the performance of the INFERNAL software package with and without the QDB heuristic. It should be noted that the speedup attainable by the QDB heuristic is dependent on the CM. For the small model tested here, the speedup was only 1.2×. Previous results have shown a speedup between 1.4× and 12.6×, with an average speedup of 4.1× for QDB [53].

Run Type	**Time**	**Speedup**
INFERNAL	17m19.287s	1
INFERNAL (QDB)	14m28.706s	1.2
Baseline Architecture	0m42.434s	24.5

Table 6.1: Performance comparison between INFERNAL and the baseline architecture

The results for the baseline architecture running at 100 MHz shows a 24.5× speedup. As with QDB, these results are quite conservative compared to the possible speedup achievable with the baseline architecture. The baseline architecture processes one residue per clock cycle regardless of the size of the CM. At 100 MHz, the baseline architecture can process a database sequence of 100 million residues in 1 second (plus the latency of the pipeline, ∼60 clock cycles for the CM implemented here). The additional time shown in Table 6.1 ($42.434 - 1 = 41.434$ seconds) represents the time to read the sequence from disk, transfer the sequence to the hardware, and retrieve the results. As the size of the CM increases, the time to send the database sequence to the hardware will stay the same, and the processing time will increase only slightly as the latency of the pipeline increases. Therefore, the degree of acceleration achievable is much greater with larger CMs. Provided a device with sufficient hardware resources, the baseline architecture could achieve a speedup thousands of times faster than Infernal running on a traditional CPU.

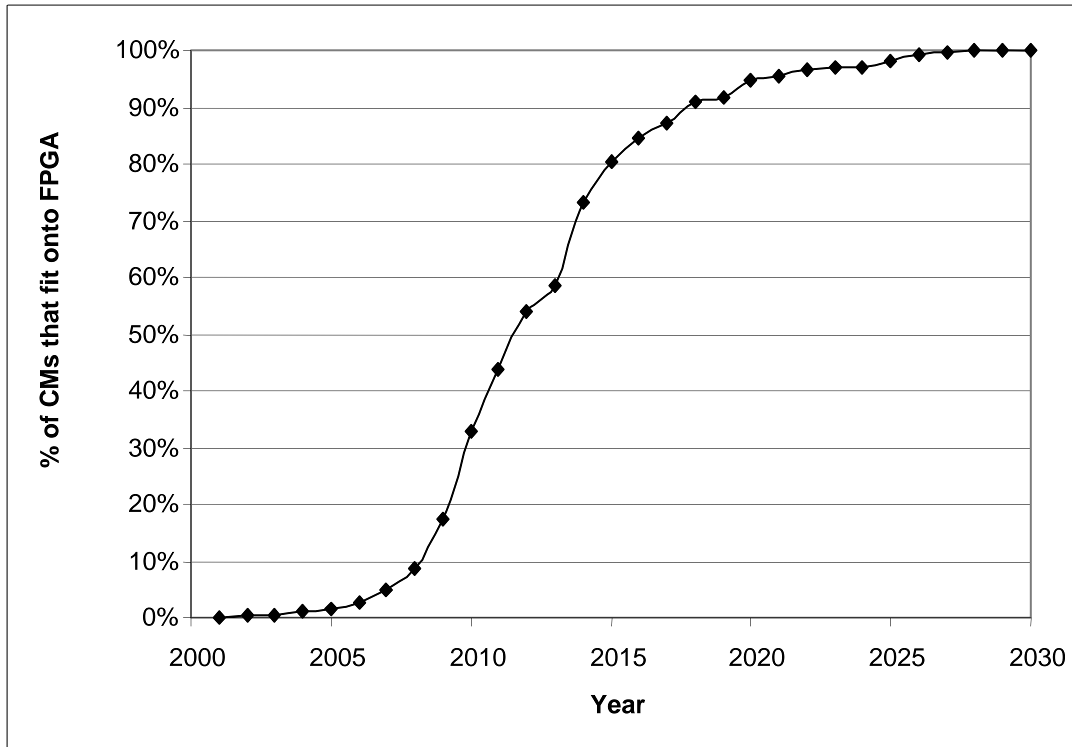


Figure 6.5: Percentage of CMs that will fit onto hardware in a given year

However, the high resource requirement of the baseline architecture makes the approach somewhat impractical for today's technologies. Figure 6.5 charts the percentage of Rfam 8.0 CMs that will fit onto hardware in a given year provided that technology continues to improve at the rate specified by Moore's law. Note that at the time of writing, only about 5% of the CMs in the Rfam 8.0 database can be converted into a pipelined architecture that will fit on today's hardware.

6.5 Expected Speedup for Larger CMs

The experimental results presented in Section 6.4 are for a very small CM. This section estimates the expected speedup that the baseline architecture can achieve when processing much larger CMs. The estimate is based on several factors including the depth of the pipeline required to process the CM, the I/O time required to transmit 100 million residues to the baseline architecture (as measured in Section 6.4), and a clock frequency of 100 MHz for the baseline architecture.

CM	Num PEs	Pipeline Width	Pipeline Depth	Latency (ns)	HW Processing Time (seconds)	Total Time with measured I/O (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Expected Speedup over Infernal	Expected Speedup over Infernal (w/QDB)
RF00001	3539545	39492	195	19500	1.0000195	42.4340195	349492	128443	8236	3027
RF00016	5484002	43256	282	28200	1.0000282	42.4340282	336000	188521	7918	4443
RF00034	3181038	38772	187	18700	1.0000187	42.4340187	314836	87520	7419	2062
RF00041	4243415	44509	206	20600	1.0000206	42.4340206	388156	118692	9147	2797

Table 6.2: Estimated speedup for baseline architecture running at 100 MHz

The results are presented in Table 6.2. The latency of the pipeline is computed as $PipelineDepth \times PE_latency$ where the PE latency is $\frac{10 \text{ cycles}}{100\text{MHz}}$ since each PE has a 10 cycle latency. The hardware processing time is the time, in seconds, that it takes the baseline architecture to process 100 million residues when running at 100 MHz. This includes the latency of the baseline architecture's pipeline. The total time is the expected time, in seconds, to process 100 million residues with the baseline architecture, including the 41.434 seconds (measured in Section 6.4) required to transmit 100 million residues to the baseline architecture. The time required for INFERNAL to process a database of 100 million residues

was estimated from the measured time required to process 1 million residues on the test machine described in Section 6.4. The expected speedup over INFERNAL for four different CMs from the Rfam database is shown in Table 6.2. The baseline architecture exhibits an estimated speedup of over $9,000\times$ over the INFERNAL software for CM RF00041. A speedup of over $4,000\times$ is estimated for CM RF00016 when INFERNAL is run with the QDB heuristic. Results for additional CMs from the Rfam 8.0 database are shown in Appendix C, where the speedup exceeds $13,000\times$ for some CMs.

6.6 Chapter Summary

This chapter introduced a baseline architecture capable of searching genome databases for homologous RNA sequences. The architecture consists of a pipeline of processing elements, each of which represents a single computation from the three-dimensional dynamic programming matrix. The processing elements are connected based on the the structure of the covariance model. An implementation of the baseline architecture was developed for a small CM and showed a $24.5\times$ improvement over the INFERNAL software package. Additionally, the performance of the baseline architecture was estimated for covariance models from the Rfam database. The baseline architecture showed an estimated speedup of over $9,000\times$ for one of the models tested in this chapter. Other results for models in Appendix C show speedups in excess of $13,000\times$.

Chapter 7

The Processor Array Architecture

Chapter 6 described a baseline architecture for accelerating RNA secondary structure alignment in hardware. By unrolling the alignment computation into individual processing elements, the baseline architecture can potentially process genome databases many thousands of times faster than software approaches such as INFERNAL. However, the baseline architecture is limited by the steep resource requirements needed for the vast number of computations in the alignment. This chapter introduces a second architecture that employs a stored program model, where alignment computations are divided up onto an array of processing elements (PEs) that can be used to perform the computation. The number of PEs in the processor array can be increased or decreased to make the best use of available hardware resources. PEs in the processor array architecture are similar in structure to those described in Chapter 6 (e.g. Figure 6.3) with the only difference being that the constant transition and emission probabilities from the baseline architecture are replaced with rewritable registers. This allows each PE in the processor array to act as a shared resource that can compute the result of any computation in the dynamic programming (DP) matrix.

Because the number of PEs is far less than the number of computations required to align an RNA sequence to a covariance model (CM), computations must be scheduled onto the available PEs. The remainder of this chapter describes the processor array architecture and a scheduling algorithm capable of scheduling the necessary DP computations onto the processor array.

7.1 Overview

The processor array architecture generalizes the approach taken with the baseline architecture presented in Chapter 6. The baseline architecture utilizes individual PEs for each computation in the graph-like structure of CMs. This limits the size of the CMs that can be processed using the baseline architecture as the number of PEs is bounded by physical hardware resources. The processor array architecture described in this chapter employs PEs using a more general technique so that any PE can be utilized to compute the result of any computation in the graph-like structure of a CM.

The processor array architecture consists of three main components, an array of processing modules (PMs), a multi-port shared memory structure, and a reporting module. Inputs to the processor array architecture consist of a stream of residues, which is

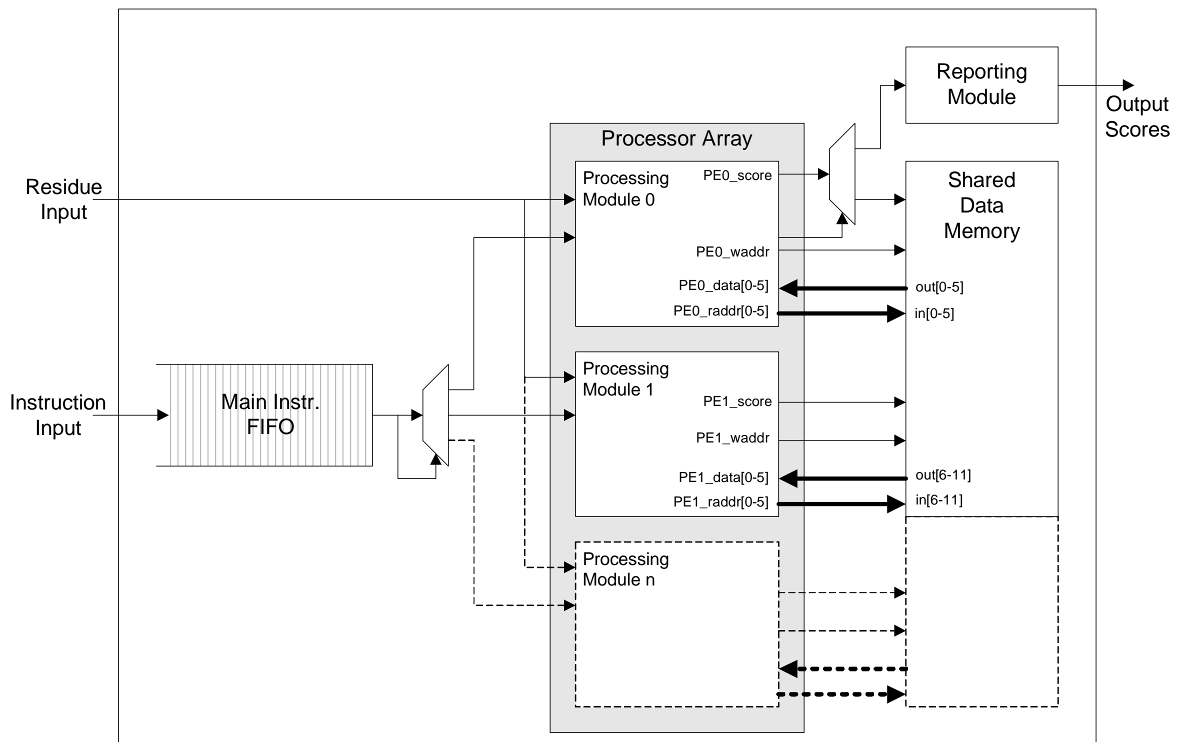


Figure 7.1: A high-level block diagram of processor array architecture with two processing modules. The number of processing modules can be scaled as shown with dashed lines.

replicated and sent to each PM, and a stream of instructions, which is divided up so instructions are sent to the appropriate PM. The mapping of instructions to PMs is discussed in Section 7.5.2. The architecture outputs only the scores for high scoring alignments. A high-level block diagram of the processor array architecture is illustrated in Figure 7.1. Each of the main components for the processor array architecture is described in more detail in the following sections.

7.2 Processing Modules

The processing module is where all of the computation required for the RNA alignment takes place. A block diagram of a PM is shown in Figure 7.2.

The core of the PM is similar to the PEs used in the baseline architecture. Each PE consists of seven saturating adders and five comparators. The first six saturating adders are used to sum results from previous computations with transition probabilities associated with the state of the current computation. The five comparators in the PE determine the maximum value from those summations and pass that value to the final adder in the PE. The final saturating adder adds an emission probability which is selected from the local

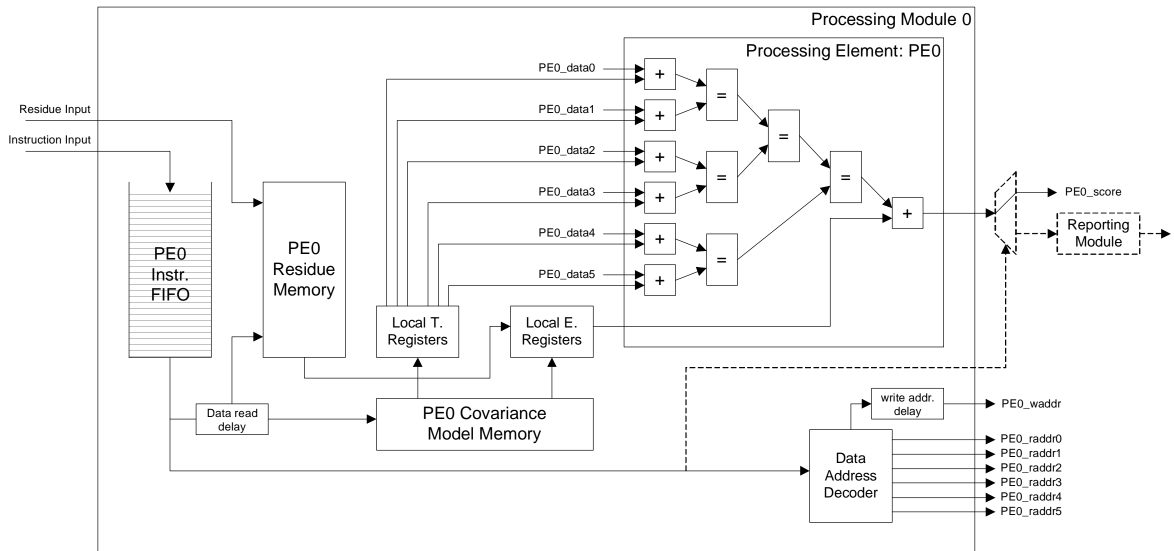


Figure 7.2: Block diagram of a single PM. Dashed lines represent components that are only needed on the first PM.

emission probability memory. When a computation does not require the addition of an emission probability, like those for start (S), delete (D), and bifurcation (B) states, a value of zero is presented to the adder so as not to affect the result of the maximization portion of the PE. Likewise, not all states in a CM have six children states, thus not all computations require all six adders in the maximization portion of the circuit. In such cases, the largest negative number possible (dependent on the number of bits used in the architecture) is presented to the unused adders so as not to affect the result of the maximization portion of the PE.

In addition to the PE core, PMs also contain a variety of memories for different functions of the PM. An instruction FIFO collects and stores instructions from an input stream until they are ready to be executed. A dual ported residue memory stores a single window's (W) worth of residues. Only one port is used when executing computations for the single emission states ML and MR. Both ports are required when executing computations for the pairwise emission state MP.

Each PM also contains its own memory for storing CM information such as transition and emission probabilities. Transition and emission probabilities are loaded into the memory during initialization and reused throughout the computation. Because each PM has its own CM memory, it has unrestricted access to the probabilities required for the alignment computation. Additionally, the CM memory for each PM stores only the probabilities required by the computations that are to be executed on that PM. During an alignment computation, the necessary transition and emission probabilities are read from the CM memory and stored in local registers where they are immediately available to the PE. This design makes effective use of distributed memories, such as block RAMs on an FPGA or embedded SRAM on an ASIC.

Data computed by each PM is stored in a shared memory that is shared by all PMs in the processor array architecture. For each instruction executed, each PM may issue up to six reads to the shared memory. The data returned from the shared memory is used as input to the PE adders. Because each PM only produces a single result for each instruction executed, only a single write interface to the shared memory is required.

Scaling the performance of the processor array architecture can be accomplished by increasing the number of PMs in the array. A discussion on the scalability of the processor array architecture is presented later in Section 7.6.4.

7.2.1 Instruction Format

The instruction format utilized by the processor array architecture is a long instruction word that contains all of the information necessary to complete the computation. The instruction format has fields for the state value v, the starting position of a test alignment i, the end position of a test alignment j, six read address pointers $raddr_0$ through $raddr_5$, and a write address pointer $waddr$ where the result is to be written. An example of the long instruction word is shown below:

$$[v, i, j, raddr_0, raddr_1, raddr_2, raddr_3, raddr_4, raddr_5, waddr]$$

The number of bits required for each of the fields in the instruction word is dependent on a number of factors. The values for v, i, and j must be large enough to handle the largest CMs that are to be processed. Using 10 bits for each of v, i, and j allows the processor array architecture to process CMs with up to 1024 states and window sizes W up to 1024. The number of bits required for each of the read address pointers and the write address pointer is dependent on the number of processors in the processor array architecture. More details on the number of bits required for each pointer are discussed in Section 7.6.4.

The analysis presented later in Section 7.6 is based on fixed-length instructions as described above. However, it would also be possible to use variable-length instructions for the processor array architecture. Based on an analysis of the CMs in the Rfam 8.0 database, the average number of reads required for a computation is approximately 4, where $\sim 42\%$ of computations required 3 reads and $\sim 40\%$ of computations require 6 reads. Eliminating the unnecessary read addresses from instructions could help to reduce the bandwidth required for instruction execution considerably.

7.2.2 Executing Instructions

Instruction execution on the processor array architecture works similarly to that of a traditional instruction pipeline, with stages for fetching and decoding instructions, reading and writing data memory, and execution. On each clock cycle a single instruction is read from the instruction FIFO of the PM. The seven memory addresses (one write address, and up to six read addresses) are sent to the data address decoder where they are divided onto the respective read of the shared memory interface. The data address decoder also extracts the write address, that is the shared memory location where the result of the instruction should be stored, and sends it to a delay module. The write address is not needed until the computation has completed. Other portions of the instruction that indicate the state number as well as the i and j residue positions for the computation are also delayed until the data from the shared memory becomes available.

The data returning from the shared memory represents computations from the CM states that are children of the current CM state computation being executed. Synchronously with the arrival of data from the shared memory, the necessary transition and emission probabilities are read from the CM memory and stored in local registers. Additionally, one or two residues are read from the residue memory and used to select the emission probability that will be used, if any, by the PE to complete the computation.

Upon completion of the computation by the PE, the result is written to shared memory in the location specified by the instruction and previously stored in the delay unit. Computations for state $v = 0$ are not stored in the shared memory structure. Instead, they are sent to a reporting module for further processing.

7.3 Shared Memory Structure

One of the pivotal components of the processor array architecture is the shared memory structure. Throughout the alignment computation, results computed on one PM may be required by other PMs in the processing array. The shared memory structure provides a means for communicating results between PMs.

The shared memory structure is a large memory bank composed of many smaller memories, each with its own read and write interfaces. This approach allows a large number of independent reads and writes per clock cycle and increases the effective bandwidth of the memory interface. The shared memory is partitioned in such a way so that each PM writes results to a designated region of the memory. However, each PM can read any location from any of the available memories in the structure. Furthermore, each PM can issue up to six memory reads per clock cycle. Therefore, if the processor array has multiple PMs, the number of concurrent memory reads on any given clock cycle may be as high as $6p$ where p is the number of PMs in the processing array.

The size and number of each memory required in the shared memory structure is dependent on the CM being processed. However, since each PM can issue up to six simultaneous reads to the shared memory structure, the minimum number of individual memories required for each PM is six. A discussion on the required size of those memories is presented in Section 7.6.3.

7.3.1 Writing Results to the Shared Memory Structure

As previously mentioned, each PM in the processor array architecture is allocated a portion of the shared memory structure where it writes its results. Additionally, because each PM only needs to write a single result per clock cycle, the write interface of the shared memory structure is considerably simpler than the read interface. Figure 7.3 illustrates the write interface to one region of the shared memory structure. The region shown is for a single PM and consists of six individual memories. In the example configuration, the value being written is an 18-bit value and the write address is a 15-bit value. The most significant bits are used to select which of the available memories in which to write the data. The remaining bits of the write address are used to address the individual memories.

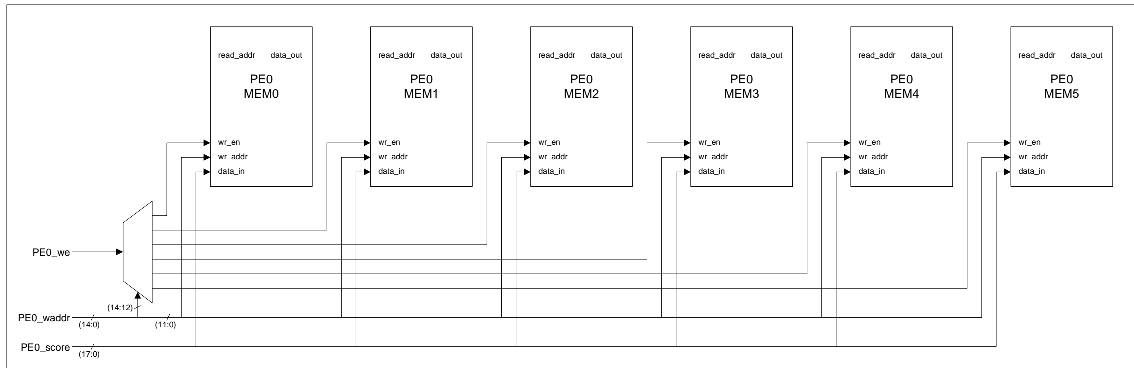


Figure 7.3: Write interface configuration for a single PM with six individual memories

7.3.2 Reading Data from the Shared Memory Structure

The read interface for the shared memory structure allows any PM in the processor array architecture to read a value from any memory in the structure. To accomplish this, the available memories in the shared memory structure are interfaced to the PMs via a pair of switching fabrics. There are many types of switching fabrics available, each with its own pros and cons. The requirements for the shared memory structure are that the switching fabric be non-blocking and that it be scalable. Non-blocking behavior is desirable so that all memory reads have the same known latency. This ensures that all PMs in the processing array can operate continuously without running the risk of desynchronization with other PMs. As described later, synchronization between PMs is built into the schedules for each PM. Using a switching fabric with blocking behavior (i.e. buffering) could stall a memory read for one PM while allowing another to proceed. This could desynchronize the PMs producing unknown results. The switching fabric must also be scalable so that as more PMs are added to the processing array, more read interfaces can be added to the shared memory structure.

Given these requirements, a Banyan switch [32] was selected to interface the PMs to the memories in the shared memory structure. Banyan switches are non-blocking and scalable [34]. Additionally, Banyan switches can be easily pipelined, allowing for very high-speed designs. However, the Banyan switch does have two restrictions that need to be considered. The first restriction is that no two inputs to the Banyan switch can be routed

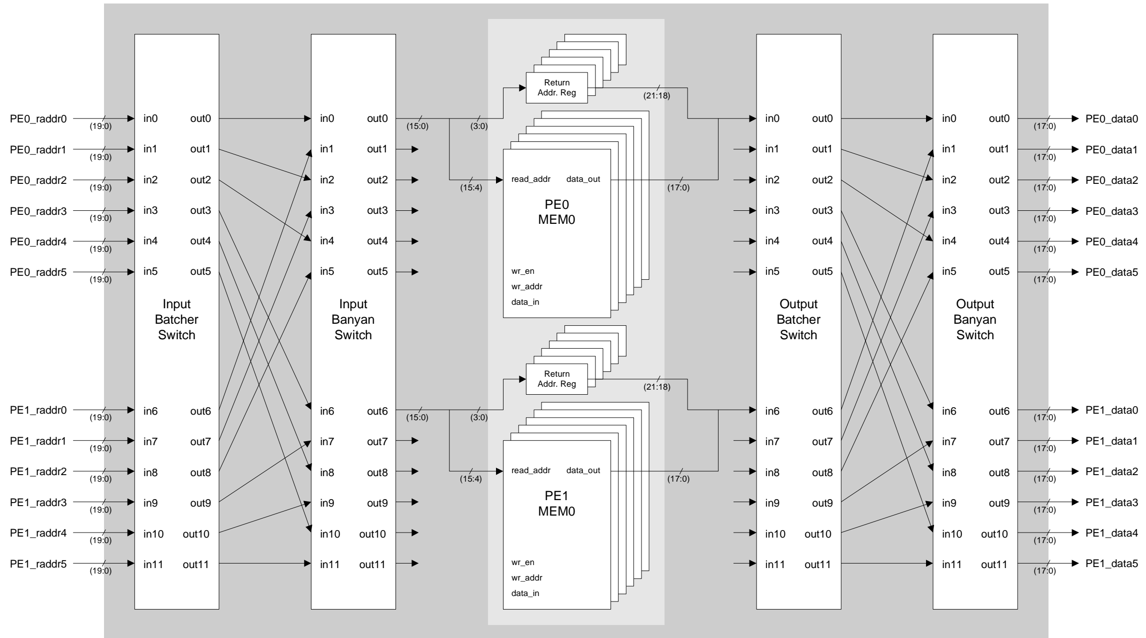


Figure 7.4: Switched read interface for two PMs, each with six individual memories

to the same output port. This would result in contention not only at the output of the Banyan switch, but also at the memory since each memory only has a single read interface. This type of contention is prevented by ensuring that no two PMs are ever scheduled to read the same memory interface on any given clock cycle. Section 7.5.2 provides more details on removing memory contention from the schedule.

The second restriction of the Banyan network is that certain inputs can cause internal collisions [34]. An internal collision in the Banyan switch can result in one or more of the inputs being routed to the wrong output port. As shown by Batcher, these types of collisions can be avoided by pre-sorting all inputs to the switch [9]. A Batcher switch, which performs a merge sort on its inputs, can be used to pre-sort memory read prior to sending them through the Banyan switch. On each clock cycle, the Batcher switch can take up to k inputs. Those k inputs are then sorted in ascending order and output on the first k outputs of the Batcher switch. The sorted outputs of the Batcher switch can then pass through the Banyan switch with no collisions.

Figure 7.4 shows the configuration of the Batcher and Banyan switches as well as the individual memories. To perform a read on one of the memory interfaces, a PM combines a 16-bit memory address with a 4-bit return address. The most significant bits of the memory address are used to route the read through the switching fabric to the appropriate memory. The remaining bits of the memory address are used to read the required value from the memory. The 4-bit return address is used to route the newly read data back to the appropriate interface of the PM that issued the read. The memory structure in Figure 7.4 illustrates the structure required when using two PMs, each with six memories. The number of bits required to route memory reads from PMs to the appropriate memory, and the resulting value back to the PM, varies with the number and size of individual memories in the shared memory structure and the number of PMs in the processor array.

Removal of the Banyan Switch

To conserve hardware resources and reduce the latency of memory reads, it is possible to remove the Banyan switch from the shared memory structure. As mentioned in the previous section, the Batcher switch sorts k inputs in ascending order onto its first k outputs. Those outputs are then passed into the Banyan switch which routes the values to the appropriate output port. If all k input ports to the Batcher switch are provided a value to sort, then all k output ports of the Batcher switch will have a value once the sort is complete. Because all memory reads on the processor array architecture are scheduled, the schedule can be modified to ensure that all input ports to the Batcher switch are busy on all cycles. The end result is that all memory reads can be sorted to the appropriate ports without the need of the Banyan switch. This can be accomplished by inserting dummy reads into the schedule to ensure that all memories in the shared memory structure are being read even if the resulting value is unused.

7.4 Reporting Module

The reporting module is a small component that resides at the output of the first processing module. The function of the reporting module is to compare the scores that are output

from the processing module to some threshold value and report any scores that exceed that threshold. Along with the score, the reporting module also reports the starting position i and ending position j of the high-scoring alignment. Since the final scores for an alignment are computed in state $v = 0$ at the root of the CM, and because of the way computations are assigned to the available processing modules (discussed later in Section 7.5.2), only the first processing module requires a reporting module.

7.5 Scheduling Computations

The number of PEs that can be utilized by the processor array architecture is dependent on the implementation platform. Platforms with more hardware resources can accommodate more PEs than platforms with fewer hardware resources. To determine how each of the available PEs are used, a polynomial-time scheduling algorithm is employed to determine the ordering of computations and how those computations are distributed among the available PEs.

As shown in Chapter 6, the DP computation required to align a target sequence to a CM can be thought of as a directed task graph. A survey paper by Kwok and Ahmad provides an in-depth survey of static scheduling algorithms for mapping directed task graphs to multiple processors [43]. In this survey, the authors note that there are only three special cases for which there currently exists optimal, polynomial-time scheduling algorithms. Those cases, as enumerated by Kwok and Ahmad [43], are: (1) scheduling tree-structured task graphs with uniform computation costs onto an arbitrary number of processors [36], (2) scheduling arbitrary task graphs with uniform computation costs on two processors [22], and (3) scheduling an interval-ordered task graph [27] with uniform node weights to an arbitrary number of processors [68].

By using PEs that have constant computational latency, the problem of scheduling the task graph that represents the DP matrix for RNA alignment fits into the first special case listed above. An algorithm developed by T.C. Hu [36] provides an optimal, linear-time scheduling algorithm for such problems, and a good starting point for scheduling the DP matrix computations onto the available PEs.

7.5.1 Optimal Scheduling of Directed Task Graphs

Hu's scheduling algorithm [36] constructs optimal schedules for tree-structured directed acyclic task graphs where each graph node takes unit computation time. The algorithm works for an arbitrary number of processors and can therefore be used regardless of the number of PEs available on a given platform. The scheduling algorithm runs in linear time in terms of the number of DP matrix cells in a CM that need to be scheduled.

For the processor array architecture, a schedule is constructed for a single window position of the CM over the input genome database. That schedule can then be reused for each window position of the CM. Hu's scheduling algorithm processes each cell in the DP matrix as a graph node in a task graph. The first stage of the scheduling algorithm involves labeling each of the nodes in the graph with a distance value. Starting from the exit node of the graph (all $S_{0,j,d}$ cells of $ROOT_0$ are exit nodes and all $E_{v,j,d}$ cells are entry nodes), each node is labeled with its distance from the exit node. The distance of a node is the number of graph edges between that node and the exit node. If multiple paths exist, a node is assigned the distance of the longest path.

Once all the nodes in the graph have been labeled, an optimal schedule for a platform with p processors can be constructed as follows: (1) schedule the p (or fewer) nodes with the greatest distance labels and no predecessors. If there are more than p nodes with no predecessors, nodes with higher distance labels should be scheduled first. If there are fewer

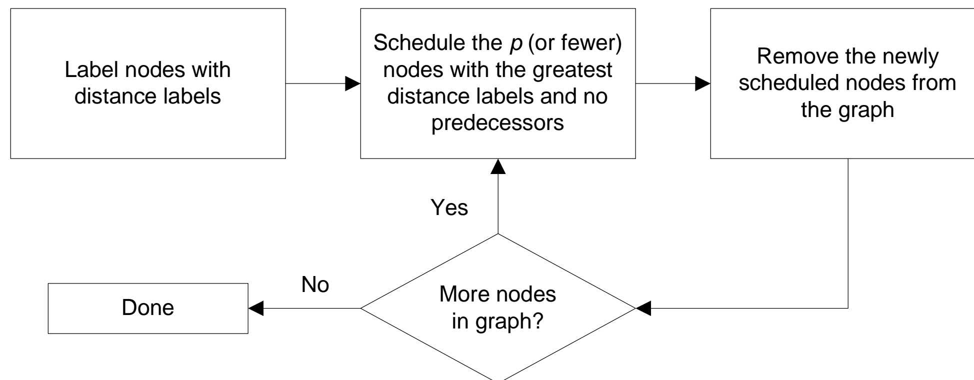


Figure 7.5: Flow Diagram of Hu's scheduling algorithm

than p nodes with no predecessors, then fewer than p nodes must be scheduled and some processors will go unused during that time slot; (2) remove the nodes that were scheduled in step (1) from the graph; (3) repeat steps (1) and (2) until all nodes are scheduled and there are no more nodes in the graph. Figure 7.5 provides a flow diagram of Hu's scheduling algorithm. Figure 7.6(a) illustrates an example task graph where each node is labeled with a distance d from the end node N0. A schedule developed using Hu's algorithm for an unlimited number of processors (i.e. $p = \infty$) is shown in Figure 7.6(b). Figure 7.6(c) shows a schedule for the same task graph when only two processors are available.

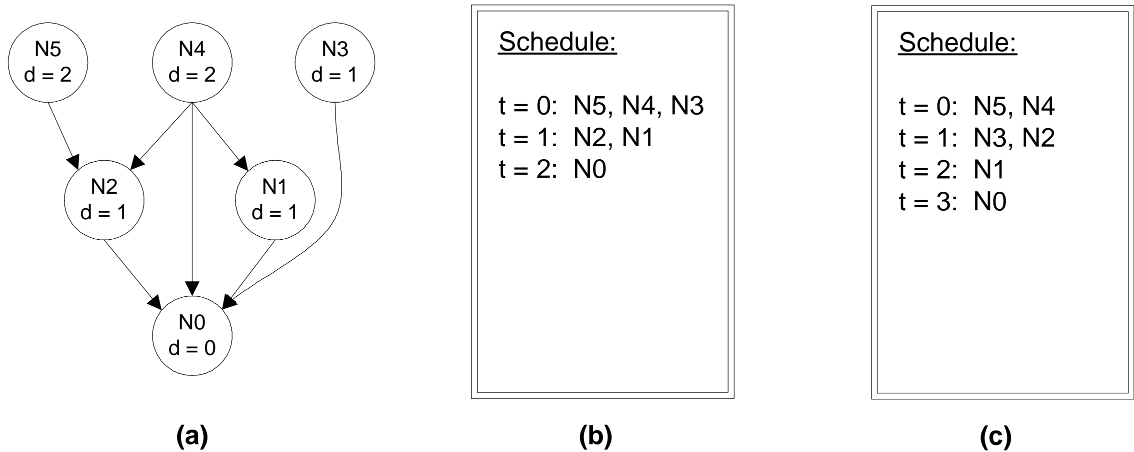


Figure 7.6: (a) An example task graph with distance labels; (b) schedule for task graph in (a) using unlimited processors; (c) schedule for task graph in (a) using two processors

7.5.2 Scheduling Task Graphs on Finite Resources with Computational Latency

While Hu's scheduling algorithm does provide a good starting point for scheduling the DP matrix computations, there are a few shortcomings with Hu's algorithm as it relates the desired application. The first of these shortcomings is that the algorithm does not account for physical hardware resources or computational latencies. For example, the example schedule in Figure 7.6(b) shows that graph nodes N2 and N1 can be scheduled at time $t = 1$, and that the graph node N0 can be scheduled in the subsequent time slot $t = 2$.

However, if the computation represented by node N0 is dependent upon the result of the computation represented by node N1, and there is some latency associated with computing the result of node N1, then node N0 cannot be scheduled until time $t_{N0} = t_{N1} + l$, where l is the computational latency.

Another hardware restriction that Hu's scheduling algorithm does not take into account is memory. The problem originates from the arbitrary choice of p computations from the pool of available computations at each time slot. In scheduling computations arbitrarily, Hu's algorithm does not account for the possibility that multiple computations may require accessing different data from the same memory bank. Without memory arbitration, this could lead to potential memory conflicts. With memory arbitration, additional control would be required to ensure that the PEs do not become desynchronized.

Finally, Hu's algorithm does not take advantage of any problem-specific knowledge. For example, if there are more than p computations that can be scheduled in a particular time slot, each with an identical distance label, Hu's algorithm does not differentiate between them. Instead, p of the available nodes are chosen arbitrarily and scheduled. Choosing nodes more intelligently by utilizing additional information associated with each node may help to reduce the hardware resources required and the total time required to complete the computation.

The remainder of this section describes several modifications that were made to Hu's scheduling algorithm to produce an algorithm suitable for scheduling the DP matrix computations for RNA alignment onto an array of processors while accounting for hardware restrictions. Instead of specifying some number of processors to Hu's scheduling algorithm and allowing the algorithm to choose the ordering of computations arbitrarily, Hu's scheduling algorithm is run on a task graph assuming an unlimited number of processors. The output is an optimal schedule for the task graph that provides the earliest possible time that each node can be scheduled (see Figure 7.6(b)). From that schedule, restrictions can be introduced and intelligent decisions can be made regarding which of the available computations should be scheduled in a given time slot. A flow diagram of the modifications to Hu's schedule is shown in Figure 7.7.

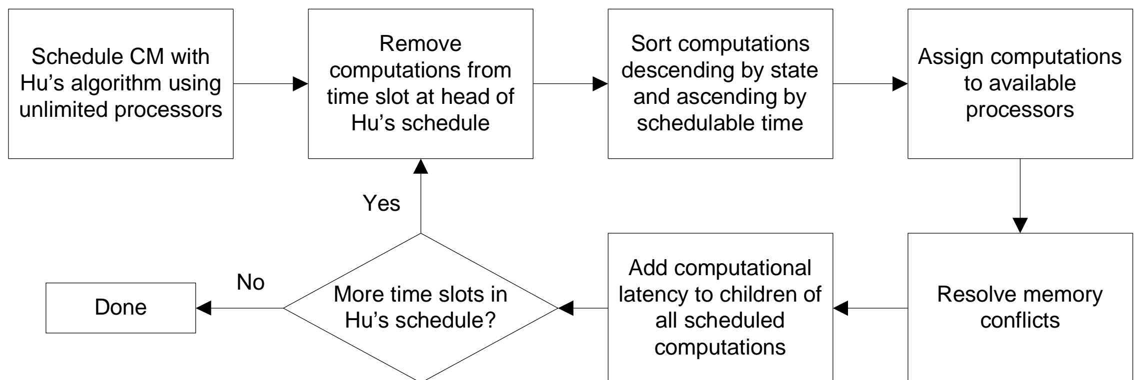


Figure 7.7: Flow Diagram of Modified Hu's scheduling algorithm

Accounting for Computational Latency

The first modification to Hu's scheduling algorithm was developed to account for the computational latency of the PEs used in the architecture. Although the time to complete each computation is uniform, this time must still be represented in the schedule. This is because each time slot in the schedule is the equivalent of a single clock cycle in hardware, and without the appropriate delays built into the schedule a computation may be scheduled to execute prior to the availability of a required result from a previous computation. Accounting for the latency of a computation can be done by recursively traversing the task graph and setting the scheduled time for each child of a node n to $t_{childNode} = max(t_{childNode}, t_n + l)$. Figure 7.8(b) shows the effect of adding computational latency equal to 10 time units into the schedule. Note that the scheduled times for the nodes in Figure 7.8(b) are no longer $t = 0, 1$ and 2 as they were in Hu's original algorithm (Figure 7.5), but instead $t = 0, 10$ and 20.

Accounting for computational latency in the *initial* Hu schedule will still result in an optimal schedule with regards to how quickly the hardware platform can correctly produce the result of the desired computations. If no other changes are made to the initial Hu schedule, then a modified Hu schedule that accounts for the computational latency of the hardware will be exactly $ScheduleLength_{Hu} \times l$ in length. However, throughout

the development of a schedule, computations may be delayed for various other reasons as described in the next few sections. Each time the scheduled time of a computation is adjusted for other conflicts, all descendants of that computation in the task graph need to be adjusted to account for the computational latency. Because not all conflicts are resolved optimally, the final modified schedule may no longer be optimal.

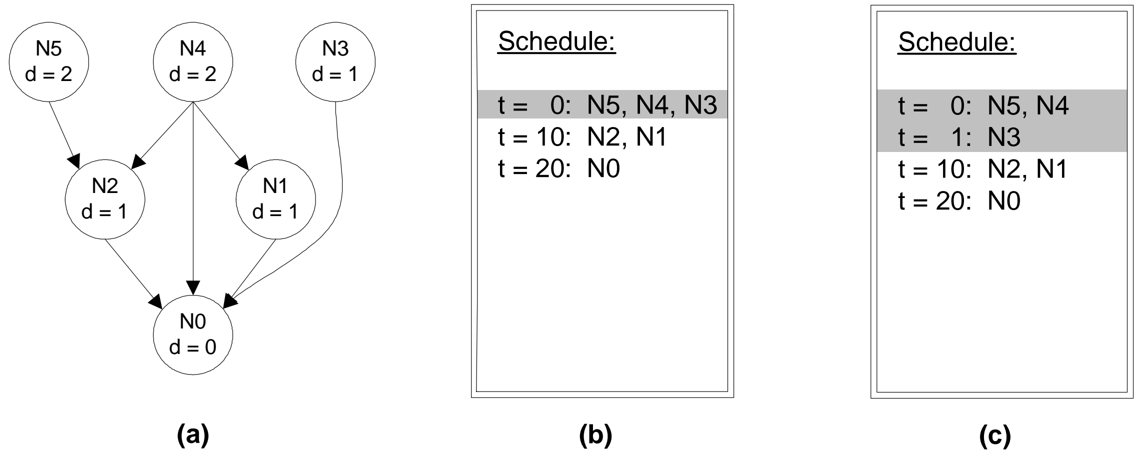


Figure 7.8: (a) An example task graph with distance labels; (b) schedule for task graph in (a) using unlimited processors and accounting for a 10 time unit computational latency; (c) schedule for task graph in (a) using two processors and accounting for a 10 time unit computational latency

Processor Assignment

The schedule produced in the previous section accounts for the computational latency. However, it still assumes that there are an infinite number of processors on which to schedule the computation. From the assumption of an unlimited number of processors, a schedule can be obtained for a given CM that completes the work as quickly as possible. A bound on the number of useful processors can then be obtained from that schedule, assuming every computation is completed on a different processor. While a finite number of processors now suffices, instantiation of so many processors (see Section 6.4) will likely exceed any reasonable resource availability. Thus, we must consider the issues that arise from scheduling a CM to execute on a number of processors that is relatively small compared to the maximum

number of processors that could be used. Thus, the next modification to the schedule accounts for the limited number of processors, or PEs, available in the architecture. Starting with the schedule produced in Figure 7.8(b), if the number of processors available is limited to two processors, the schedule must be altered to reduce the number of computations at time $t = 0$. All three computations represented by nodes N5, N4, and N3 cannot be computed in the same time slot. Instead, one of those computations must be delayed and scheduled at a later time. In the example shown in Figure 7.8(c), the computation for node N3 is delayed one time unit and rescheduled at time $t = 1$. Node N3 is not dependent on nodes N5 or N4, so it is not necessary to delay node N3 more than a single time unit. Node N0, which is dependent on node N3, is unaffected by N3's newly scheduled time because it is scheduled for time $t = 20$ which is greater than $t_{N3} + l$ where l is still 10 time units. Because this adjustment is dependent on the actual number of PEs available, as the architecture is scaled to include more PEs, new schedules will need to be generated.

In the above example, the computation for node N3 was arbitrarily chosen to be delayed while nodes N5 and N4 remained scheduled at time $t = 0$. However, when scheduling the DP matrix computations for aligning an RNA sequence to a CM, the choice of which computations should be assigned to which PEs, and which computations should be delayed, is based on a number of factors.

The first factor in determining the ordering and processor assignment of computations is based on how CM information, such as transition and emission probabilities for each CM state, is stored by the processor array architecture. If all of the CM information were to be stored in a single memory in the processor array architecture, then PEs could not efficiently retrieve the probabilities required if computations for multiple different CM states were scheduled in a single time slot. This is because different CM states require different transition and emission probabilities and scheduling multiple different states in a single time slot would result in memory contention. Instead, all computations for a particular CM state are assigned to the same PE regardless of where they occur in the schedule. This allows the CM information to be divided among several smaller memories that are local to each PE. Each PE stores only the portion of the CM information that is required

for the states that it is scheduled to process. This eliminates any contention that may have occurred by storing CM information in a global memory used by all PEs. Computations are assigned to a PE based on their CM state number, $PE\# = v \ \% \ p$, where v is the CM state number to which the computation is a member, and p is the number of PEs available.

Other information about the computations is also used to determine the order in which computations should be completed and which computations are delayed. As shown in Figure 5.4, the DP parsing algorithm treats CMs as a tree structure and computes values starting at the bottom of the tree and ending at the top of the tree. That is, the algorithm computes values from state $v = M - 1$ down to $v = 0$. Computations in each time slot are scheduled in a similar fashion, where those with higher state values are scheduled first. If there are multiple computations with the same state value in a given time slot, then those with the earliest schedulable time are scheduled first. Note that the earliest schedulable time may not be the actual time that a computation is scheduled due to the limited hardware resources. As an example, refer back to Figures 7.8(b) and (c) where the earliest schedulable time for node N3 was $t = 0$, but the actual time the node was scheduled was $t = 1$.

Eliminating Memory Contention

The last consideration when developing a schedule for the processor array architecture is memory contention. As described in Section 7.3.2, no two computations can access the same memory interfaces in any given time slot of the schedule. To prevent memory conflicts, computations are first assigned to processors as described in the previous sections. Then, for each instruction in a time slot, unique memory addresses are provided from which to read previous results in the alignment computation. Each address assigned is bounded to the region of memory that is associated with the computation that produced the result. This, in turn, assigns write locations to the computations that generated the results.

In general, more than one computation in the alignment may require the same result from a previous computation. When scheduled in different time slots, there is no conflict since each of the computations can access the result from memory uncontested during its time slot. However, in some circumstance two computations may require the same result

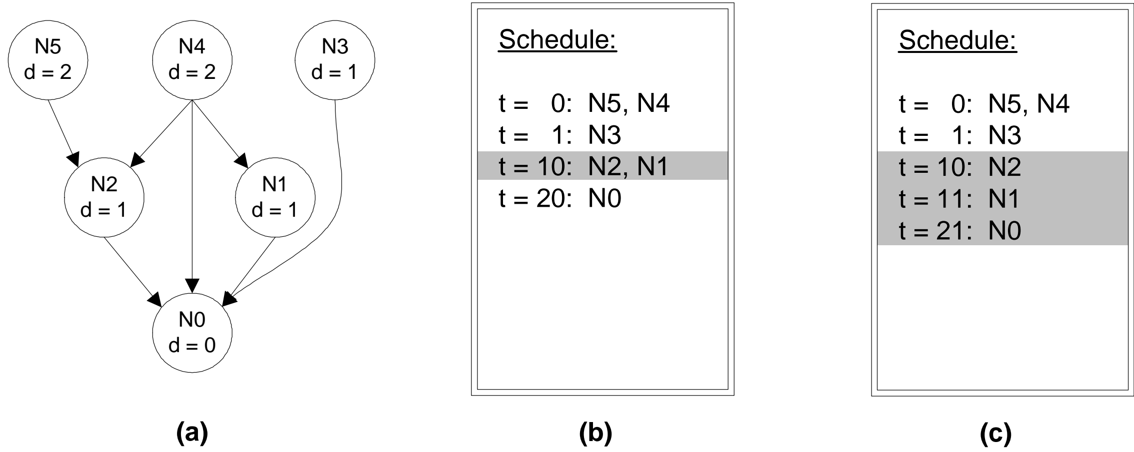


Figure 7.9: (a) An example task graph with distance labels; (b) schedule for task graph as shown in Figure 7.8, but with a memory conflict at time $t = 10$; (c) schedule for task graph with memory conflict resolved. Note that N1 was moved to $t = 11$ and its dependent N0 was moved to $t_{N0} = t_{N1} + l = 21$.

in the same time slot. When this occurs, one of the computations needs to be moved to a different time slot. The example shown in Figure 7.9(b) illustrates a schedule with a time conflict at time $t = 10$ where both node N2 and N1 need the result of node N4. Because both computations cannot access the same memory in the same time slot, node N1 is moved to the next time slot $t = 11$. While this technique may not produce an optimal scheduling of the computations, it is likely that an optimal approach is NP-complete.

7.6 Architecture Analysis

This section provides an analysis of the processor array architecture and the proposed scheduled technique. From more than 600 CMs in the Rfam database, four were selected to illustrate the behavior of the processor array architecture and the scheduler. The four CMs chosen are of average size with varying numbers of bifurcation states. The characteristics of the four models (RF00001, RF00016, RF00034, RF00041) are shown in Table B.1 of Appendix B. The complete results for the analysis done in this section are located in Appendix D.

7.6.1 Running Time

The first thing to consider when analyzing the effectiveness of the scheduler is the running time, or the length of the schedule required to produce results. The schedules for the processor array architecture are developed for a single window W of the DP computation. The schedule length, in clock cycles, for a single window of four different CMs is shown in Figure 7.10. As should be expected, as the number of processors in the processor array increases, the length of the schedule decreases. This is because the total number of computations in the DP matrix stays the same regardless of the number of processors. However, with more processors available, more computations can be done in parallel which decreases the time required for the computation. The most dramatic decreases in schedule length occur between one and sixteen processors. The number of bifurcation states in a model does not appear to have an affect on the decreasing schedule length.

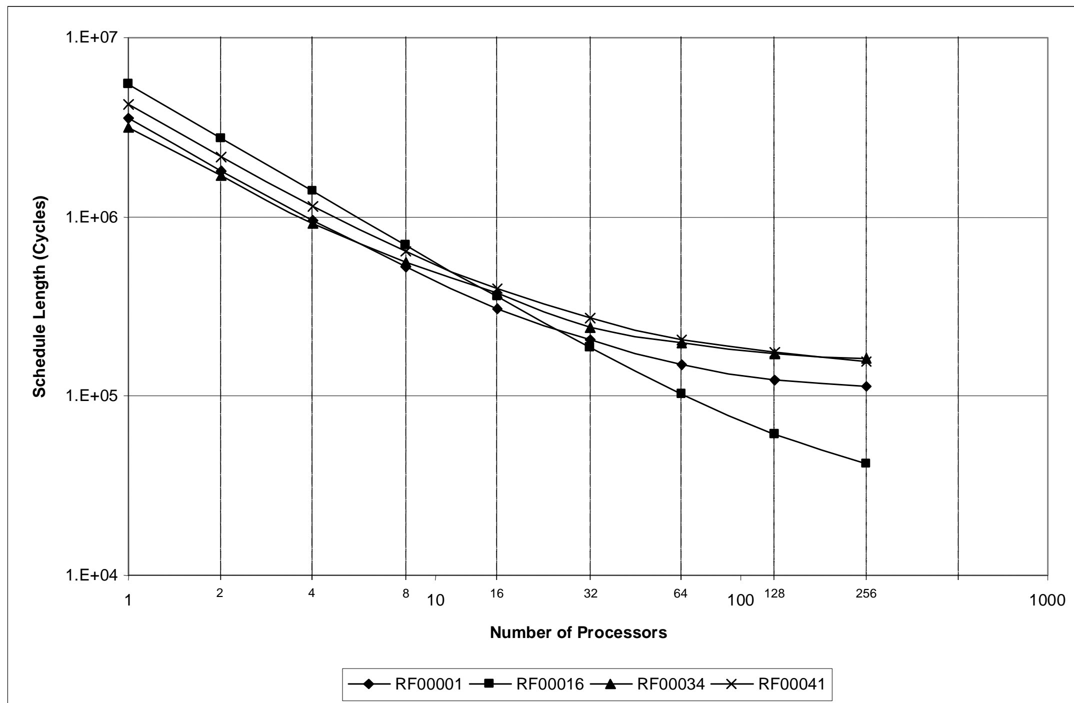


Figure 7.10: The length of the scheduled computation decreases as the number of processors available for the computation increases (shown on a log-log scale).

7.6.2 Scheduling Efficiency

Section 7.6.1 illustrated that as the number of processors in the processor array increases, the length of the scheduled computation decreases. Although adding more processors consistently decreases the schedule length, it is far from linear. As more processors are added, there are diminishing decreases in the schedule length. This can be illustrated as the speedup and efficiency of a schedule as more processors are added. The speedup is a measure of how much faster a computation can be done with p processors than with a single processor. The speedup is measured as $\frac{ScheduleLength_{1_proc}}{ScheduleLength_{p_procs}}$. The efficiency of the schedule is a measure of how much idle time must be inserted into the schedule to ensure no conflicts during the computation. This is measured as $\frac{NumComputations}{TotalCycles}$ where $NumCompuations$ is the number of computations that need to completed for a single window of the specified CM and $TotalCycles$ is the total number of clock cycles, among all processors, that are required to compute the results, including any idle time in the schedule.

Figure 7.11 illustrates both the speedup and the schedule efficiency of the four example CMs. As additional processors are added to the processor array the speedup increases. However, there are diminishing returns as the schedule is divided over more processors. This is especially true for models RF00001, RF00034, and RF00041, which have 1, 3, and 2 bifurcations states respectively. CM RF00016 has no bifurcation nodes and shows much greater speedup with the addition of more processors. This behavior is likely due to the combination of the way computations are assigned to processors ($PE\# = v \ \% \ p$) and the fact that bifurcation states require a much larger number of computations than non-bifurcation states. These two factors result in a much larger number of computations, those for the bifurcation states, being assigned a single processor. This, in turn, could result in a large number of idle slots being inserted into the schedules for each of the other processors to ensure that all schedules remain synchronized. However, it was noted in Section 7.5.2 that computations are assigned to processors in such a way so as to eliminate the need to unnecessarily duplicate CM transition and emission probabilities. Because bifurcation states do not require probabilities other than those from its children states, it may be possible to evenly distribute bifurcation computations over all available processors in the processor

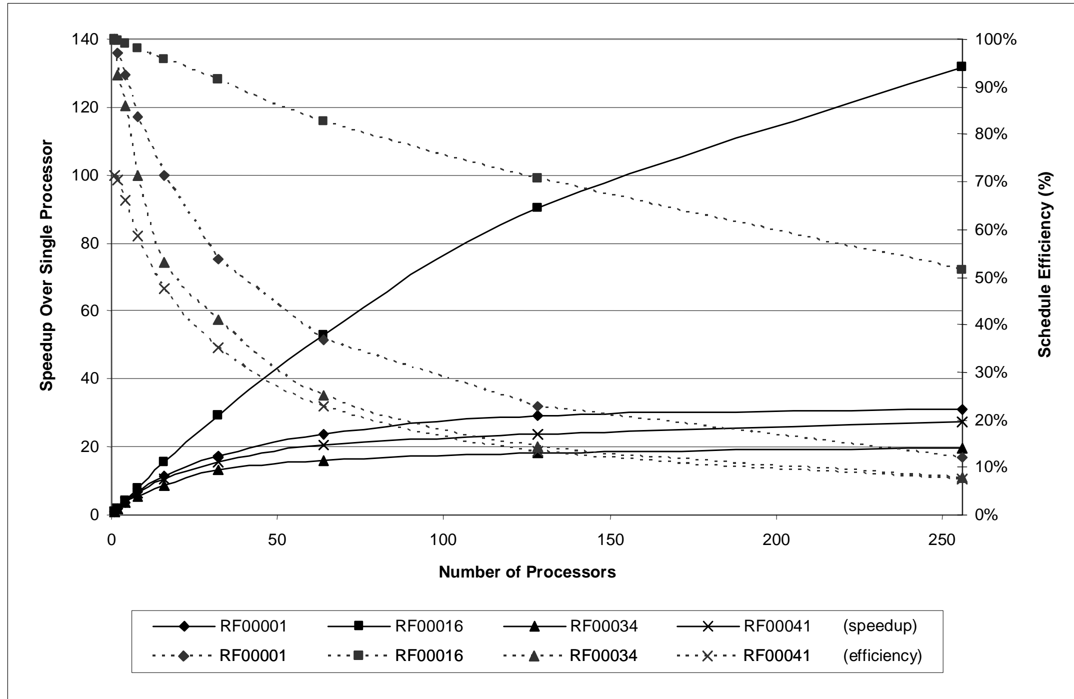


Figure 7.11: The speedup shows the diminishing returns as more processors are added to the processors array. The efficiency decreases as more processors are added to the processors array, indicating that more idle time is inserted into the schedule as the number of processors increases.

array. This could allow CMs with bifurcations nodes to achieve speedup similar to that of CMs without bifurcation nodes.

7.6.3 Memory Requirements

One of the main challenges in considering an architecture to accelerate the RNA alignment computations was determining how to handle the memory requirements of the computations. This section provides an analysis on the memory requirements and behaviors for the processor array architecture and the scheduler.

Live Memory Requirements

Figure 7.12 illustrates the maximum amount of live memory required for a schedule as the number of processors increases. The live memory is the number of DP matrix cells that need

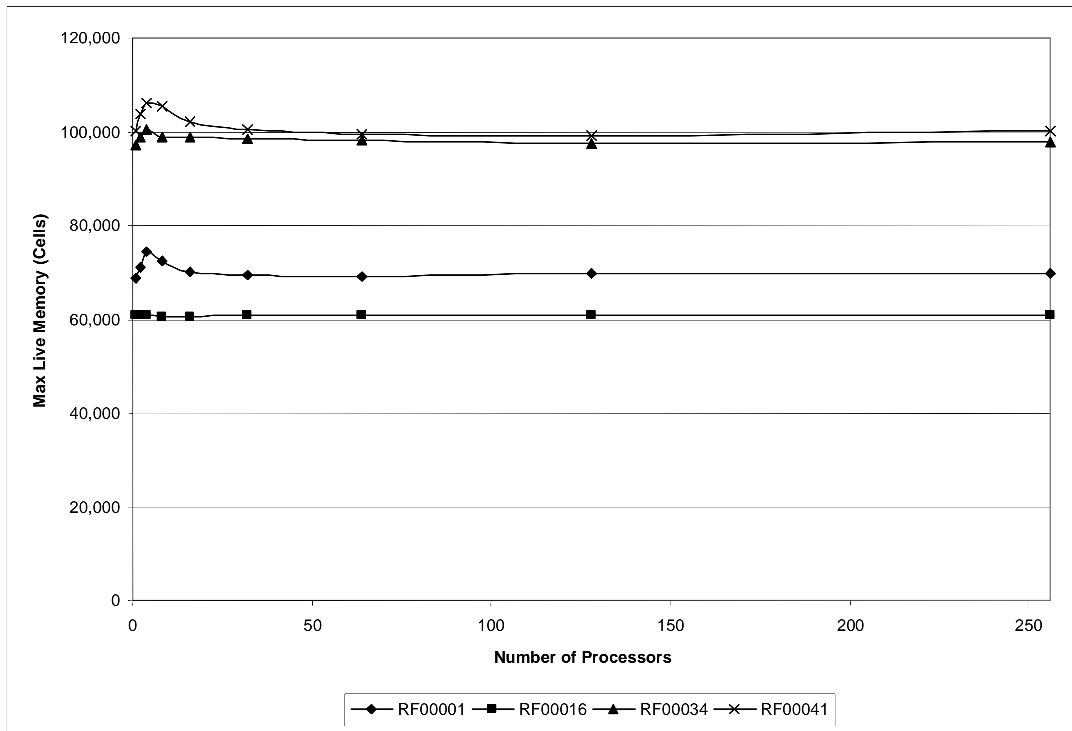


Figure 7.12: The maximum amount of live memory remains fairly consistent regardless of the number of processors.

to be stored throughout the computation because another computation is still dependent on the value from that cell. Once a stored value no longer has any dependencies, its memory location can be reclaimed.

Increasing the number of processors has very little affect on the maximum amount of live memory required throughout an alignment computation. This means that the total memory required for processor array architecture is independent of the number of processors in the array.

Average Memory/Processor

Figure 7.12 illustrates the maximum amount of live memory required for a computation for the complete array of processors. However, as discussed in Section 7.3 each processor in the processor array architecture has its own memory in the shared memory structure to which it can write data. Figure 7.13 illustrates the average memory required per processor

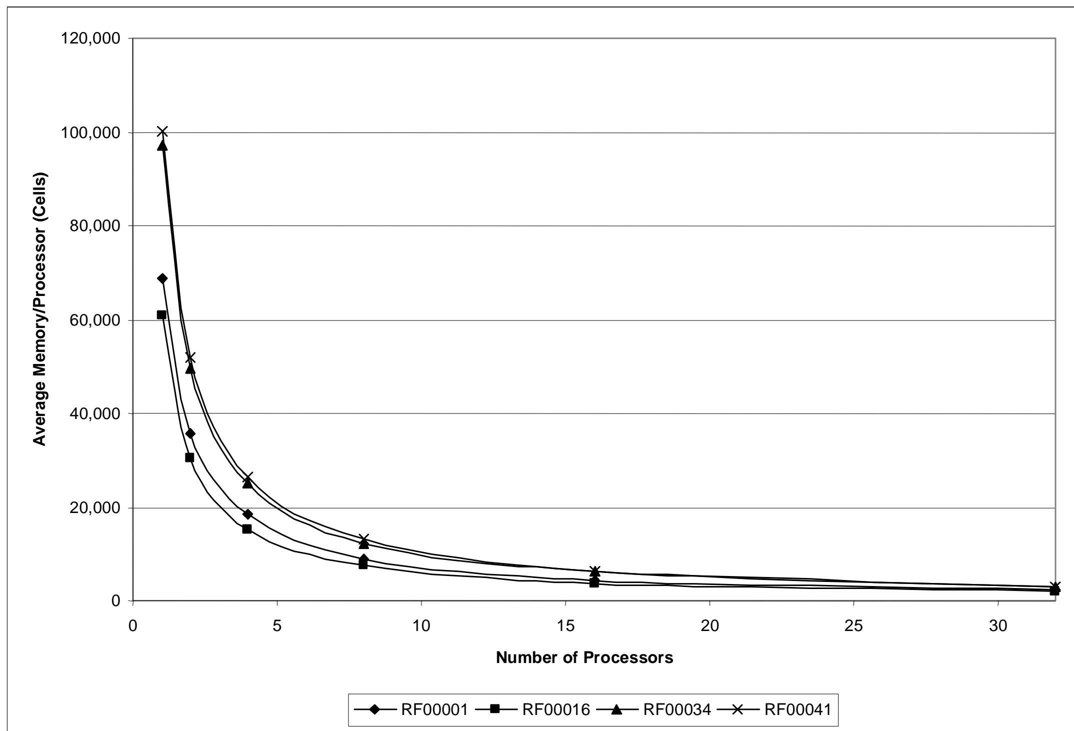


Figure 7.13: As the number of processors in the processor array architecture increases, the average memory required per processor decreases.

as the number of processors increases. When using a single processor, all of the memory required must be associated with that one processor. This can be verified by noting that the average memory required for a single processor, as shown in Figure 7.13, is equivalent to the maximum live memory required for a single processor in Figure 7.12.

Figure 7.12 also shows that as the number of processors in the processor array increases, and hence the total number of individual memories, the amount of memory required for each processor decreases. This is because the total number of DP matrix cells that need to be stored does not vary much as the number of processors increases, thus the same number of cells are being stored over a larger number of memories. The decrease in efficiency shown in Figure 7.11 does not affect the number of cells that need to be stored throughout the computation.

Memory Traces

To understand the behavior of the memory usage throughout an RNA alignment computation, several computations were simulated for two different CMs from the Rfam database. Memory traces were created for the duration of the simulated computation to monitor the required memory throughout the computation. The models chosen to illustrate memory usage were RF00016 and RF00034, which have 0 and 3 bifurcation states respectively. Each of the models were run three times each to simulate 16, 32, and 64 processor configurations. The memory traces for CM RF00016 are shown in Figure 7.14. The memory traces for CM RF00034 are shown in Figure 7.15.

The overall profile of each memory trace is a result of the structure of the CM. Therefore, the memory traces for the two CMs have very distinct curves. However, comparing the traces for the two CMs helps to reveal some commonalities in the behavior of the memory usage. First, note that the profile of the memory traces remain fairly consistent regardless of the number of processors. The time scale of the profile is simply compressed

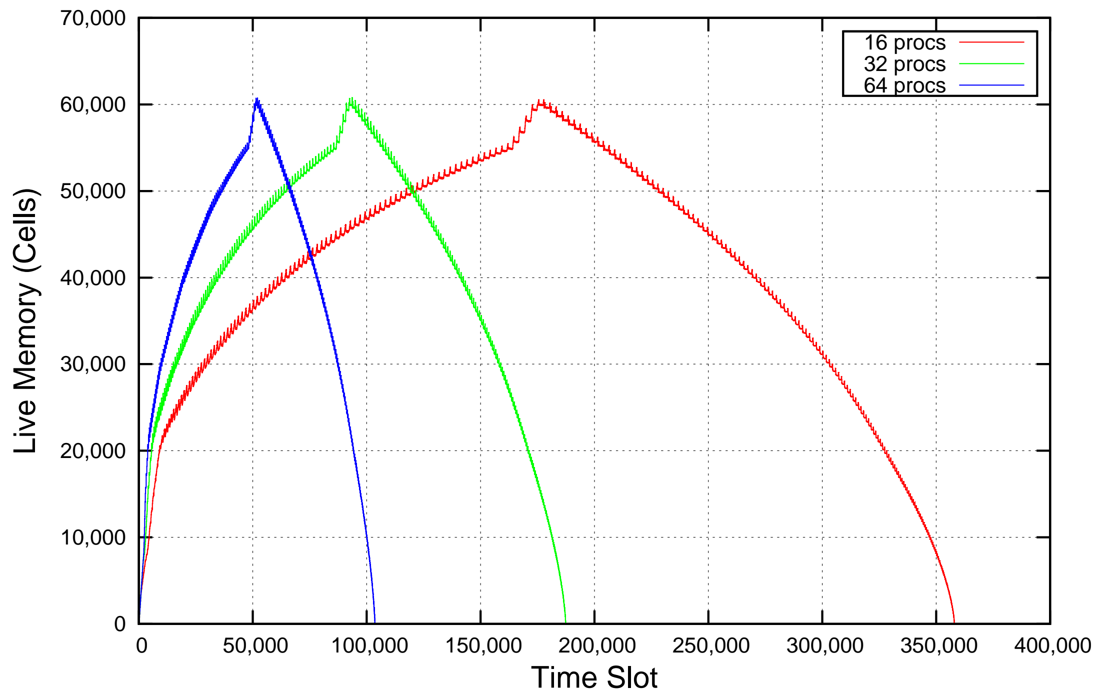


Figure 7.14: Memory trace for CM RF00016

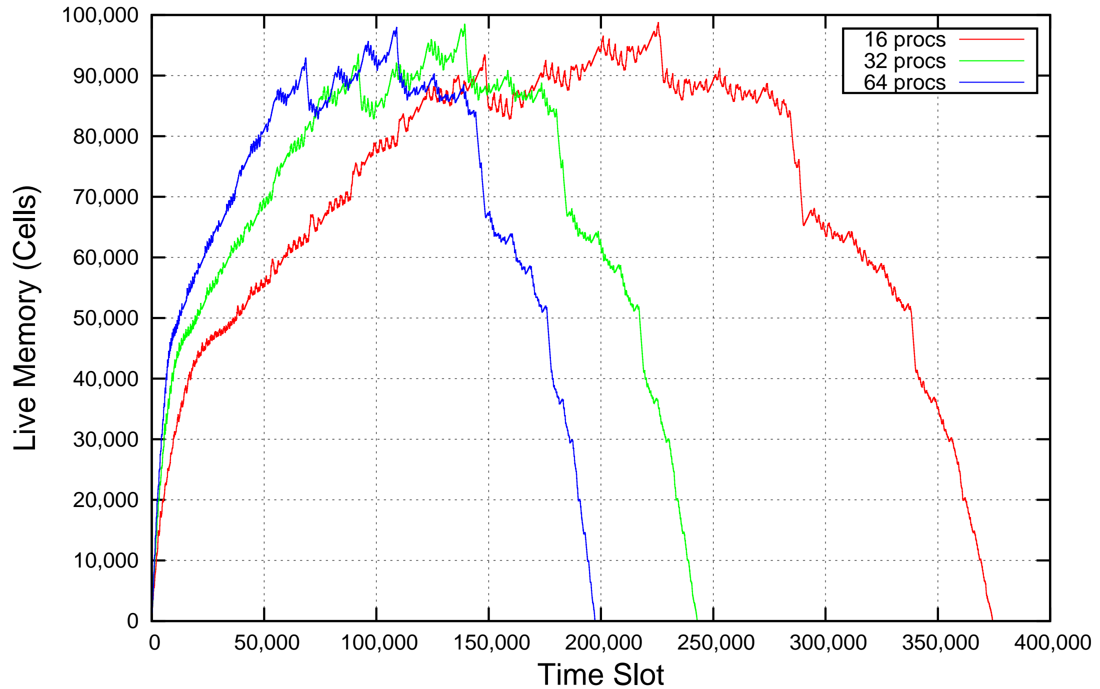


Figure 7.15: Memory trace for CM RF00034

or expanded depending on whether there are more or fewer processors. Also note that the peak of each profile (i.e. the maximum live memory) is approximately equivalent regardless of the number of processors. This means that the total memory required for the processor array architecture is independent of the number of processors in the array. This was also shown earlier in Figure 7.12.

7.6.4 Scalability of Architecture

This section provides an analysis on the scalability of the processor array architecture. It includes a discussion on the logic requirements and the I/O requirements.

Logic Requirements

To determine the logic requirements for different sized processor array architectures, the main components were scaled according to the number of processors in the array. For the processing elements discussed in Section 7.2, this entailed finding the resource requirements

for a single processing element and then multiplying those requirements by the number of processors desired. For the shared memory structure, scaling the resource requirements entailed not only increasing the number of ports on the switching fabrics, but also increasing the width of each port. The increased width of each port is due to the additional bits required to address the larger number of ports. Note that the number of processors chosen was determined by the number of ports available on the switches of the shared memory structure. The switches scale as a power of two and the number of processors was chosen to use as many of those ports as possible. The number of processors chosen is simply $\left\lfloor \frac{NumSwitchPorts}{6} \right\rfloor$. The results of the scalability analysis can be seen in Table 7.1.

In terms of the individual memories required, if each processor in the processor array architecture is given six memories as described in Section 7.3, then the total number of individual memories required is $6p$ where p is the number of processors. One of the largest FPGAs available today, the Xilinx Virtex-5, contains over 600 1024x18bit block RAMs, allowing for a total of over 600,000 18-bit memory locations (or 300,000 21-bit memory locations if using the numeric representation discussed in Section 5.5). As shown in Section 7.6.3, and further illustrated in Appendix D, the live memory requirements (i.e. the number of memory locations required) rarely exceeds 100,000 entries for the CMs tested. Furthermore, as the number of processors is increased, the average memory required per processor decreases. Because the maximum amount of live memory stays fairly consistent regardless of the number of processors (7.12), the average memory required per processor has an almost perfect inverse relationship to the number of processors. That is, doubling the number of processors reduces the average memory required per processor by half. However, because the memory requirements differ for each CM, no general statement can be made regarding the ability, or the inability, of a hardware device to handle all CMs. Each CM must be considered individually with respect to the resources available. To determine if a device can support a CM using a specified number of processors, the following must be true: $p \times \left\lceil \frac{AverageMemPerProc}{6 \times NumEntriesPerBRAM} \right\rceil \leq NumBRAMsAvailable$.

The logic requirements for the processor array architecture, in terms of both look-up tables (LUTs) and the equivalent gate count, are shown in Table 7.1 and illustrated in

Processor Array Configuration									
Num Procs	**1**	**2**	**5**	**10**	**21**	**42**	**85**	**170**	**341**
Num Memories	6	12	30	60	126	252	510	1020	2046
Switch Ports	8	16	32	64	128	256	512	1024	2048
Port Address Bits	3	4	5	6	7	8	9	10	11
In Switch Port Size	16	18	20	22	24	26	28	30	32
Out Switch Port Size	21	22	23	24	25	26	27	28	29
Resource Requirements in LUTs									
Processing Elements	528 (15%)	1056 (10%)	2640 (8%)	5280 (6%)	11088 (5%)	22176 (4%)	44880 (3%)	89760 (2%)	180048 (2%)
Banyan Switch (in)	396 (11%)	1186 (11%)	3286 (10%)	4344 (5%)	6364 (3%)	10404 (2%)	18515 (1%)	34562 (1%)	66913 (1%)
Batcher Switch (in)	889 (26%)	3286 (30%)	11292 (33%)	35005 (40%)	102323 (44%)	285696 (47%)	762749 (48%)	1994099 (49%)	5098195 (50%)
Banyan Switch (out)	517 (15%)	1442 (13%)	3767 (11%)	4729 (5%)	6620 (3%)	10404 (2%)	17874 (1%)	32639 (1%)	62453 (1%)
Batcher Switch (out)	1130 (33%)	3927 (36%)	12735 (38%)	37699 (43%)	105914 (46%)	285696 (47%)	744706 (47%)	1899186 (47%)	4747949 (47%)
Total Size	3460	10897	33720	87057	232309	614376	1588725	4050246	10155559
Resource Requirements in Gates									
Processing Elements	11598	23196	57990	115980	243558	487116	985830	1971660	3954918
Banyan Switch (in)	5451	16335	45319	59859	87339	142299	254453	472481	913541
Batcher Switch(in)	11937	44279	149115	469425	1368370	3782106	10168566	26570852	67901931
Banyan Switch (out)	7137	19919	52045	65241	90923	142299	245487	448320	858036
Batcher Switch (out)	15303	53245	169293	507093	1418588	3799730	9917324	25246972	63014048
Total Size	51426	156974	473762	1217598	3208778	8353550	21571660	54710285	136642474

Table 7.1: Configuration and resource requirements for different sized processor arrays.

Figures 7.16 and 7.17. While the resource requirements for the processing elements and the Banyan switch scale linearly, the Batcher switch does not. Moreover, as shown in Figure 7.17 the logic requirements for the Batcher switch quickly dwarf the logic requirements for other key components of the architecture. Note that the logic requirements for the Batcher switch closely follow the total logic requirements of the complete architecture. When using a single processor in the processor array, the Batcher switch accounts for almost 60% of the logic resources. This quickly grows to about 90% of the total resources when using only 21 processors.

Given the rapid growth of the Batcher switch, the logic requirements quickly become the limiting factor in terms of the number of processors that can be deployed on today's devices. Even though the Xilinx Virtex-5 has enough block RAMs to support up to 100 processors, with only 200K LUTs, the device is limited to fewer than 20 processors. However, with ASIC devices currently supporting over 1.5 billion transistors [66], 340 or more processors on an ASIC is possible.

Section 7.3.2 presented the shared memory structure used in the processor array architecture. The possibility of removing the Banyan switch from the shared memory structure by inserting dummy memory reads was also discussed. However, upon looking at

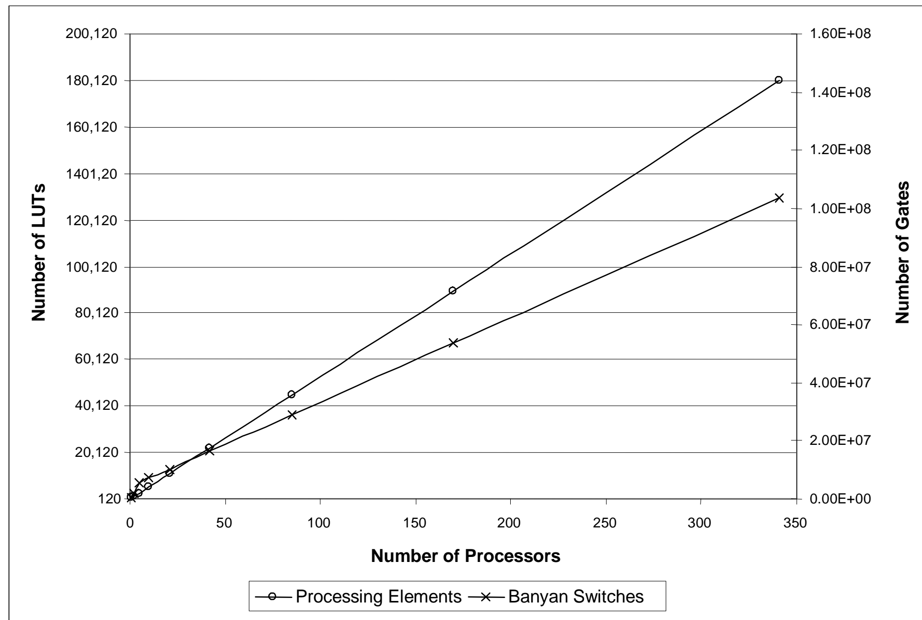


Figure 7.16: Resource requirements for the processing elements and Banyan switches in different sized processor arrays.

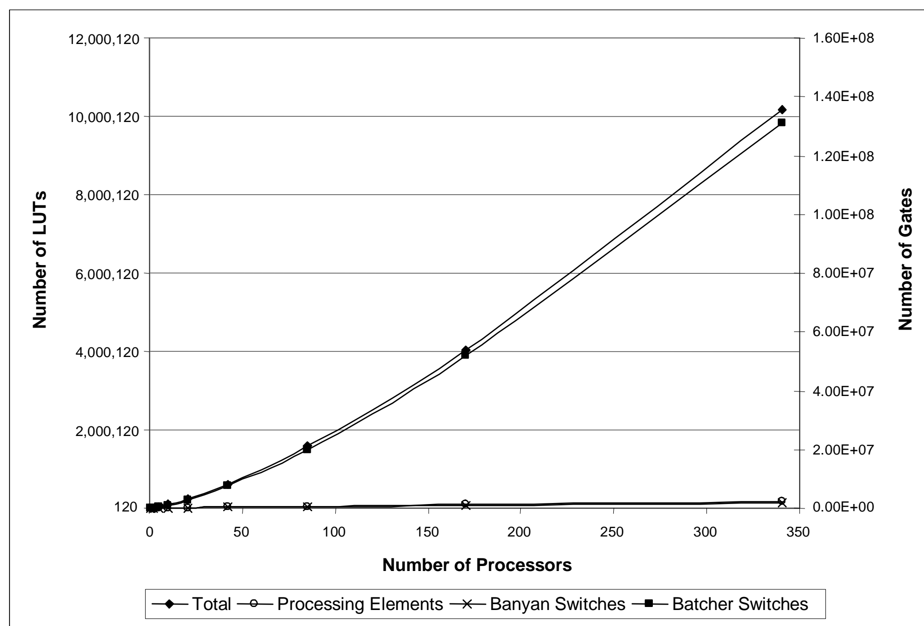


Figure 7.17: Resource requirements for different sized processor arrays. Note that the Batcher switches account for the majority of the resources.

the results in Table 7.1 and the graphs in Figures 7.16 and 7.17, it is clear that the resources required by the Banyan switches are insignificant when compared to the Batcher switches. Based on the results presented in this section, eliminating the Batcher switches from the shared memory structure should take precedence over eliminating the Banyan switches.

I/O Requirements

This section examines the I/O requirements of the processor array architecture. There are three components to the I/O: the score output, the residue input, and the instruction input.

As described in Section 7.4, only a single processing element (PE0) outputs alignment scores. Furthermore, scores are only output for state $v = 0$, meaning that the number of potential scores that need to be output consists of only a small fraction of the overall computation. Because only scores that exceed some predetermined threshold are output, the number of scores output will generally be much less than the total number of scores in state $v = 0$. For the CMs listed in Appendix B, the size of the output scores for a single window W is typically less than a few hundred KBytes. This translates to an average required output bandwidth of approximately 5 MBps for a single processor configuration and approximately 400 MBps for a 256 processor configuration when running the processor array architecture at 250 MHz.

The residue input consists of a single new residue for each window W of the computation. Because each residue can be represented with only a few bits of information, and each window may contain millions of instructions, the residue input is insignificant when compared to the instruction input. When running the processor array architecture at 250 MHz, the bandwidth required for the residue input is on average less than 1 KByte for a single processor configuration and approximately 5 KBytes for a 256 processor configuration.

The instruction input is the main component of the I/O and is the limiting factor for processor array architecture. The instruction input consists of a stream of instructions for each processor in the processor array that identifies the computations for each processor. As the number of processors increases, the number of instructions that must be streamed to

Processor Array Configuration									
Num Procs	**1**	**2**	**5**	**10**	**21**	**42**	**85**	**170**	**341**
Num Memories	6	12	30	60	126	252	510	1020	2046
Switch Ports	8	16	32	64	128	256	512	1024	2048
Port Address Bits	3	4	5	6	7	8	9	10	11
In Switch Port Size	16	18	20	22	24	26	28	30	32
Out Switch Port Size	21	22	23	24	25	26	27	28	29
Instruction I/O Requirements @ 250 MHz									
Instruction Size	121	127	133	139	145	151	157	163	169
I/O Requirements (bits)	121	254	665	1390	3045	6342	13345	27710	57629
I/O Requirements (Gbps)	30.25	63.5	166.25	347.5	761.25	1585.5	3336.25	6927.5	14407.25

Table 7.2: The instruction sizes and bandwidth required for streaming instructions to the processor array architecture.

the processor array architecture also increases. Additionally, increasing the number of processors also increases the size of each instruction that must be streamed to the architecture. As described in Section 7.2.1, each instruction consists of up to six memory addresses from which to read memory and a single memory address to which the result should be written. As the number of processors increases, the number of address bits required to access each of the individual memories in the shared memory structure also increases.

Table 7.2 shows the instruction sizes required for different configurations of the processor array architecture. Also shown in Table 7.2 is the number of instruction bits per clock cycle that must be provided to the processor array to fully utilize all of the available processors. This value can be used to determine the required input bandwidth for streaming instructions to the processor array architecture. The bandwidth requirements shown in Table 7.2 are for a processor array architecture running at 250 MHz.

At first glance the bandwidth required for streaming instructions to the processor array seem a bit daunting. However, the actual schedules tend to be only several hundreds of MBytes in size. When divided over a couple of processors, the size of the schedule assigned to each processor is typically around 50 MBytes. When using a large number of processors in the processor array, the size of the schedule assigned to each processor can be less than 1 MByte. Furthermore, the same schedule is reused for each window of the computation. Therefore, providing each processor in the processor array architecture with a small SRAM

or DRAM for schedule storage could provide the bandwidth necessary to keep all of the processors in the processor array architecture busy.

7.6.5 Comparison to Software

To determine the effectiveness of the processor array architecture, this section compares its estimated runtime to that of the INFERNAL software package. While some results are presented in this section, additional results can be found in Appendix E.

The evaluation system for the INFERNAL software contains dual Intel Xeon 2.8 GHz CPUs and 6 GBytes of DDR2 SDRAM running Linux CentOS 5.0. INFERNAL was run with the *–toponly* and *–noalign* options to ensure that INFERNAL was not doing more work than the processor array architecture. Results were collected for both the standard version of INFERNAL as well as INFERNAL using the Query Dependent Banding (QDB) heuristic [53]. The runtime represents the time it took, in seconds, for INFERNAL to process a randomly generated database of 1 million residues. The database search algorithm that is executed by INFERNAL is equivalent to the algorithm presented in Section 5.3.3 which does not require a traceback algorithm. Since no traceback is required, the contents of the residue database have no affect on the runtime.

The results for the processor array architecture are an estimate based on several factors including: the number of computations required to process a database of 1 million residues, the efficiency of the schedule, the number of processors available, and the clock frequency of the processor array. For the estimates in this section, a clock frequency of 250 MHz was used.

Table 7.3 shows the results for four different CMs using 16, 32, and 64 processors. In the table, *Total Computations* represents the total number of computations required to compute the results for a database of 1 million residues. This value includes all of the computations, including the expanded computations required for scheduling the DP matrix cells from bifurcation states. The efficiency value was determined in Section 7.6.2 for a single window of the alignment computation. The total number of cycles required for processing 1 million residues, including idle time, is estimated as $\frac{TotalComputations}{Efficiency}$. Because

those cycles are divided evenly across p processors, the actual length of the schedule is $\frac{TotalCycles}{p}$. The time to process the schedule varies with the frequency at which the processor array architecture can process the schedule. For the comparison presented in this section, the frequency of the processor array architecture is assumed to be 250 MHz. The times listed in Table 7.3 for the processor array architecture are computed as $\frac{ScheduleLength}{250MHz}$. For comparison the table lists the time required for both the standard INFERNAL software package and the INFERNAL software package when using the QDB heuristic. The speedup represents the speedup of the processor array architecture over the INFERNAL software package.

Figure 7.18 illustrates the estimated time to align 1 million residues to a CM when using between 1 and 256 processors in the processor array. Results for four different CMs are shown. The shortest bars represent the time required when using 256 processors. The longest bar represents the time required when using a single processor. The bars in between represent the time required when using 128, 64, 32, 16, 8, 4, and 2 processors.

Figure 7.19 compares the estimated time for the processor array architecture to the time for the INFERNAL software package. The bars showing the results for the processor array architecture are the same as those shown in Figure 7.18 but on an expanded time

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
16 Processors									
RF00001	52178338429	71.4	73072036714	4567002295	18.27	3494.92	1284.43	191.31	70.31
RF00016	62121487523	95.74	64884170889	4055260681	16.22	3360.00	1885.21	207.14	116.22
RF00034	50108566254	53.09	94373229739	5898326859	23.59	3148.36	875.20	133.44	37.10
RF00041	59389512497	66.69	89048404795	5565525300	22.26	3881.56	1186.92	174.36	53.32
32 Processors									
RF00001	52178338429	53.75	97071025316	3033469541	12.13	3494.92	1284.43	288.03	105.85
RF00016	62121487523	91.44	67929980421	2122811888	8.49	3360.00	1885.21	395.70	222.02
RF00034	50108566254	40.92	122439740504	3826241891	15.30	3148.36	875.20	205.71	57.18
RF00041	59389512497	49.1	120945137629	3779535551	15.12	3881.56	1186.92	256.75	78.51
64 Processors									
RF00001	52178338429	36.85	141581036608	2212203697	8.85	3494.92	1284.43	394.96	145.15
RF00016	62121487523	82.66	75149663664	1174213495	4.70	3360.00	1885.21	715.37	401.38
RF00034	50108566254	25.17	199009881845	3109529404	12.44	3148.36	875.20	253.12	70.36
RF00041	59389512497	32.05	185255436066	2894616189	11.58	3881.56	1186.92	335.24	102.51

Table 7.3: Estimated runtime and speedup of processor array architecture over INFERNAL software package. Estimate is based on a processor array running at 250 MHz. Additional results are in Appendix E.

scale. The bars showing the results of the INFERNAL software package contain the results for both the standard INFERNAL software and the INFERNAL software when using the QDB heuristic. The shorter bar represents the results when using the QDB heuristic. The longer bar represents the results when using the standard INFERNAL software.

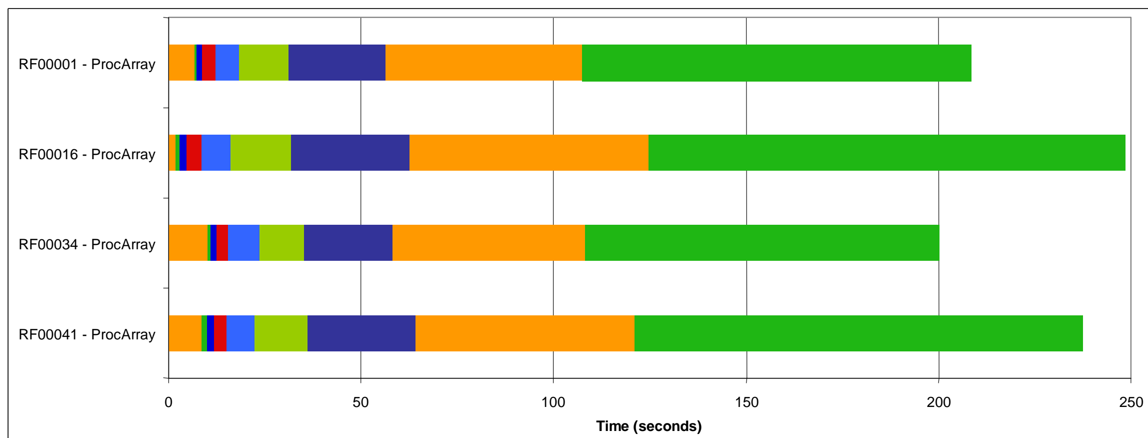


Figure 7.18: The estimated time to align a 1 million residue database to four different CMs using varying numbers of processors. The shortest bar represents the time to compute the results using 256 processors. The longest bar represents the time to compute the results using a single processor.

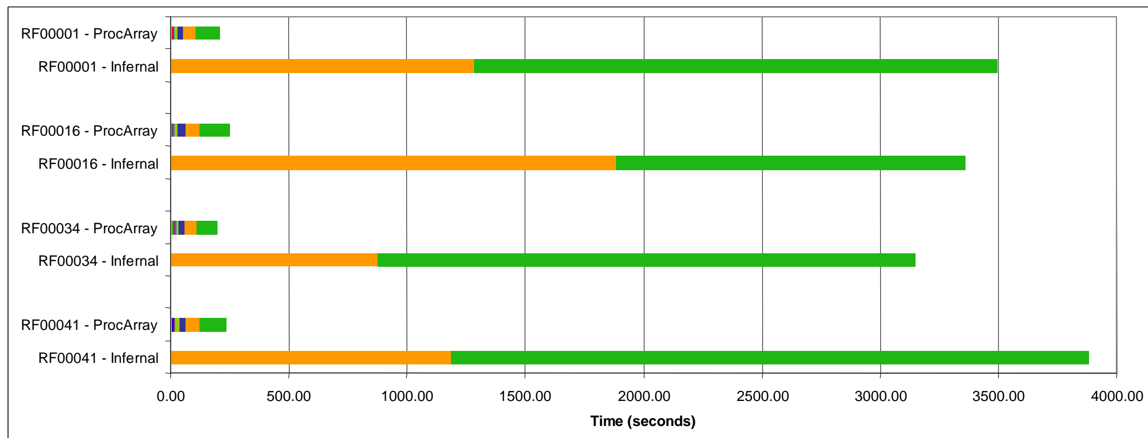


Figure 7.19: A comparison of the estimated time to align a 1 million residue database using the processor array architecture versus the time required for INFERNAL. The shorter bar in the INFERNAL categories represents the time when using the QDB heuristic. The longer bar represents the time without QDB.

Based on the results in this section, as well as the additional results in Appendix E, when running at 250 MHz the processor array architecture can achieve an estimated speedup of 11× to 18× over the standard Infernal software package. With 64 processors, the processor array architecture can achieve an estimated speedup of 250× to 970×. If provided enough hardware resource to employ 256 processors, the processor array architecture can achieve up to an estimated 2,350× over the Infernal software package.

7.7 Chapter Summary

This chapter described a second architecture for performing RNA secondary structure alignments on a residue database. The architecture presented in this chapter utilizes an array of processing elements and a multi-port shared memory structure to share data between processors. Additionally, a scheduling algorithm was presented that takes as input a covariance model and outputs a schedule for the processor array architecture. The scheduling algorithm accounts for all processor and memory conflicts thereby eliminating the need for buffering and/or conflict resolution in hardware.

An analysis of the processor array architecture and the schedule was also presented. As should be expected, the time required to perform an alignment computation decreases as the number of processors in the processor array increases. The memory required to process a sequence with a specified CM stays almost constant as the number of processors in the array is increased.

Additionally, the scalability of the architecture was examined. The hardware resources for many of the key components in the architecture scale linearly. However, the Batcher switch in the shared memory structure does not, and quickly becomes the limiting component of the architecture.

Finally, the processor array architecture was compared to the Infernal software package. The speedup achievable by the processor array was estimated based on the schedule length for an alignment. The estimated speedup for the processor array architecture over the Infernal software package ranged from under 20× to over 2,350×.

Chapter 8

Summary and Future Work

8.1 Dissertation Summary

The work described in this dissertation focused on the design of high-speed architectures for different parsing problems. Due to the wide variety of applications that require parsing, and the numerous parsing algorithms that have been developed, it is unlikely that any single architecture could suffice for all parsing problems. Therefore, this work focused on two areas that are most likely to benefit from accelerated parsing techniques.

In the area of network applications, this work introduced high-speed architectures for pattern matching as well as parsing for large grammars that define regular languages. In the area of bioinformatics, this work introduced two high-speed architectures for accelerating the computationally complex parsing algorithms used for RNA alignment.

8.2 Future Work

The work presented in this dissertation, particularly the work in Chapter 7, has many aspects that can be extended beyond what is presented in this dissertation. Below is a summary of optimizations and changes that can be made to the processor array architecture discussed in Section 7.1 and to the scheduling techniques discussed in Section 7.5.

- The scheduling algorithm presented in Section 7.5 creates a schedule for a single window W of the RNA alignment computation. That same schedule can then be reused for each subsequent window of the alignment computation. However, this approach does not take advantage of one very important aspect of the dynamic programming (DP) alignment algorithm discussed in Section 5.3.3: a high percentage of the computations required for any given window in the alignment computation have already been computed in a previous window. By reusing the same schedule for each window, a large portion of the computations are being computed up to W times.

  To eliminate the need to recompute data, a dual-schedule scheme may be appropriate. In the proposed dual-scheduled scheme, one schedule can be developed to compute the first W rows of the DP computation (rows 0 through $W-1$), and a second schedule can be developed to compute one or more rows beyond the first schedule, starting at row W. The first W rows in the DP matrix are not the complete width of the matrix, making them different from the remainder of the rows in the matrix. All rows $\geq W$ are the same width (i.e. $W+1$) and can be scheduled similarly. During computation, the first schedule is used only once, whereas the second schedule is repeated as many times as necessary to process the entire input. The second schedule for the computation can be developed for a single row and repeated $L-W+1$ times, or for multiple rows and repeated $\frac{L-W+1}{R}$ times, where L is the length of the input sequence and R is the number of full rows scheduled in the second schedule.

- The RNA alignment algorithm has steep memory requirements, whether it is realized in software or in hardware. The scheduling algorithm presented in Section 7.5 does not make any attempt to minimize the memory requirements for the computation. With further analysis it may be possible to discover optimizations that can be made to the scheduling algorithm which can help reduce the amount of live memory required throughout the computation. Making the best use of available memory resources on any given platform may enable the use of additional processing modules.

- The instruction format described in Section 7.2.1 requires a large number of bits for each instruction in the schedule. This increases the input bandwidth required to keep

all of the processors in the processor array architecture busy. Future implementations may be able to reduce the size of each instruction, and thus the required input bandwidth, by taking advantage of the structure of covariance models (CMs). For example, the state type and location of a cell in the DP matrix is used to determine the location of its dependencies. It may be possible to extend this method of addressing the three-dimensional DP matrix into the processor array architecture's shared memory structure while still keeping the memory requirements low. This would eliminate of all of the read addresses from the instruction format presented in Section 7.2.1. Another possible method for reducing the size of an instruction includes using variable-length instructions as mentioned in Section 7.2.1.

- The processor array architecture and scheduler presented in Chapter 7 were customized specifically for the RNA alignment algorithm discussed in Section 5.3.3. However, throughout the development of both the architecture and the scheduler many opportunities to generalize the approach presented themselves.

  Future realizations of the processor array architecture may be generalizable to accelerate any DP computation, not just the RNA alignment discussed in this work. One vision of such an approach would start with the recurrence relation for the DP computation. From the specified recurrence relation, information regarding the computation required for each step of the recurrence, the number of concurrent memory reads required for each computation, and the relationships between each computation could be extracted. This information could then be used to generate the necessary hardware and memory required for the processor array. Additionally, the relationships specified by the recurrence could be used to generate a directed task graph which could subsequently be scheduled onto the generated processor array.

  Generalizing the processor array architecture as described above would result in a very powerful architecture useful for many different applications. DP algorithms are used extensively throughout the field of bioinformatics, and such an approach would create a single high-speed solution that could benefit many researchers in the field of bioinformatics.

Appendix A

Email Grammar in Lex/Yacc Style Format

This appendix contains the complete Lex/Yacc style grammar used to generate the ALPS email processor described in Section 4.3 of this work.

```
%%

start:          message;

ALPHA:          x41.5A | x61.7A | A.Z | a.z;
CR:             x0D;
LF:             x0A;
DIGIT:          x30.39;
DQUOTE:         x22;
HTAB:           x09;
SP:             x20;
NO-WS-CTL:      d1.8 | d11 | d12x | d14.31 | d127;
txt:            d1.9 | d11 | d12x | d14.127;

CRLF:           CR LF;
WSP:            SP | HTAB;
LWSP:           WSP LWSP | CRLF WSP LWSP | ;
FWS:            FWS1 FWS3;
FWS1:           FWS2 CRLF | ;
FWS2:           WSP FWS2 | ;
FWS3:           WSP FWS2;
FWSopt:         FWS | ;
CFWS:           CFWS1 CFWS2;
CFWS1:          FWSopt comment CFWS | ;
CFWS2:          FWSopt comment | FWS;
CFWSopt:        CFWS | ;
```

```
quoted-pair:    "\" txt;
ctext:          NO-WS-CTL | d33.39 | d42.91 | d93.126;
ccontent:       ctext | quoted-pair | comment;
comment:        "(" comment1 ")";
comment1:       comment2 FWSopt;
comment2:       FWSopt ccontent comment2 | ;

atext:          ALPHA | DIGIT | "!" | "#" | "$" | "%" | "amp" | "'"
                | "*" | "+" | "-" | "/" | "=" | "?" | "^" | "_"
                | "`" | "{" | "staff" | "}" | "~";
atom:           CFWSopt atext1 CFWSopt;
atext1:         atext atext1b;
atext1b:        atext atext1b | ;
dot-atom:       CFWSopt dot-atom-text CFWSopt;
dot-atom-text:  atext1 dot-atom-text1;
dot-atom-text1: "." atext1 dot-atom-text1 | ;

qtext:          NO-WS-CTL | d33 | d35.91 | d93.126;
qcontent:       qtext | quoted-pair;
quoted-string:  CFWSopt DQUOTE quoted-string1 FWSopt DQUOTE CFWSopt;
quoted-string1: FWSopt qcontent quoted-string1 | ;
word:           atom | quoted-string;
phrase:         word1;
word1:          word word1b;
word1b:         word word1b | ;
utext:          NO-WS-CTL | d33.126;
unstructured:   unstructured1 FWSopt;
unstructured1:  FWSopt utext unstructured1 | ;

date-time:      date-time1 date FWS time CFWSopt;
date-time1:     day-of-week "," | ;
day-of-week:    FWSopt day-name;
day-name:       "Mon" | "Tue" | "Wed" | "Thu" | "Fri" | "Sat" | "Sun";
date:           day month year;
year:           DIGIT DIGIT DIGIT DIGIT year1;
year1:          DIGIT year1 | ;
month:          FWS month-name FWS;
month-name:     "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun"
                | "Jul" | "Aug" | "Sep" | "Oct" | "Nov" | "Dec";
day:            FWSopt DIGIT day1;
day1:           DIGIT | ;
time:           time-of-day FWS zone;
time-of-day:    hour ":" minute time-of-day1;
time-of-day1:   ":" second | ;
minute:         DIGIT DIGIT;
second:         DIGIT DIGIT;
hour:           DIGIT DIGIT;
zone:           zone1 zone2;
zone1:          "+" | "-";
zone2:          DIGIT DIGIT DIGIT DIGIT;

address:        mailbox | group;
mailbox:        name-addr | addr-spec;
```

```
name-addr:       name-addr1 angle-addr;
name-addr1:      display-name | ;
angle-addr:      CFWSopt "<" addr-spec ">" CFWSopt;
group:           display-name ":" group1 ";" CFWSopt;
group1:          mailbox-list | CFWS | ;
display-name:    phrase;
mailbox-list:    mailbox mailbox-list1;
mailbox-list1:   "," mailbox mailbox-list1 | ;
address-list:    address address-list1;
address-list1:   "," address address-list1 | ;
addr-spec:       local-part "@" domain;
local-part:      dot-atom | quoted-string;
domain:          dot-atom | domain-literal;
domain-literal: CFWSopt "[" domain-lit1 FWSopt "]" CFWSopt;
domain-lit1:     FWSopt dcontent domain-lit1 | ;
dcontent:        dtext | quoted-pair;
dtext:           NO-WS-CTL | d33.90 | d94.126;

message:         fields message1;
message1:        CRLF body | CRLF multipart-body | ;
body:            body1 text1;
body1:           text1 CRLF | ;
text1:           txt text1 | ;

fields:          fields1 fields4;
fields1:         trace fields2 fields1 | ;
fields2:         fields3 fields2 | ;
fields3:         resent-date | resent-from | resent-sender | resent-to
                 | resent-cc | resent-bcc | resent-msg-id;
fields4:         fields5 fields4 | ;
fields5:         orig-date | from | sender | reply-to | to | cc | bcc
                 | message-id | in-reply-to | references | subject
                 | comments | keywords | MIME-message-headers | optional-field;

orig-date:       "Date:" date-time CRLF;
from:            "From:" mailbox-list CRLF;
sender:          "Sender:" mailbox CRLF;
reply-to:        "Reply-To:" address-list CRLF;
to:              "To:" address-list CRLF;
cc:              "Cc:" address-list CRLF;
bcc:             "Bcc:" bcc1 CRLF;
bcc1:            address-list | CFWSopt;

message-id:      "Message-ID:" msg-id CRLF;
in-reply-to:     "In-Reply-To:" msg-id1 CRLF;
msg-id1:         msg-id msg-id1b;
msg-id1b:        msg-id msg-id1b | ;
references:      "References:" msg-id1 CRLF;
msg-id:          CFWSopt "<" id-left "@" id-right ">" CFWSopt;
id-left:         dot-atom-text | no-fold-quote;
id-right:        dot-atom-text | no-fold-lit;
no-fold-quote:   DQUOTE no-fold-quote1 DQUOTE;
no-fold-quote1: no-fold-quote2 no-fold-quote1 | ;
no-fold-quote2: qtext | quoted-pair;
```

```
no-fold-lit:    "[" no-fold-lit1 "]";
no-fold-lit1:   no-fold-lit2 no-fold-lit1 | ;
no-fold-lit2:   dtext | quoted-pair;

subject:        "Subject:" unstructured CRLF;
comments:       "Comments:" unstructured CRLF;
keywords:       "Keywords:" phrase keywords1 CRLF;
keywords1:      "," phrase keywords1 | ;

resent-date:    "Resent-Date:" date-time CRLF;
resent-from:    "Resent-From:" mailbox-list CRLF;
resent-sender:  "Resent-Sender:" mailbox CRLF;
resent-to:      "Resent-To:" address-list CRLF;
resent-cc:      "Resent-Cc:" address-list CRLF;
resent-bcc:     "Resent-Bcc:" resent-bcc1 CRLF;
resent-bcc1:    address-list | CFWSopt;
resent-msg-id:  "Resent-Message-ID:" msg-id CRLF;

trace:          returnopt trace1;
trace1:         received trace1b;
trace1b:        received trace1b | ;
returnopt:      return | ;
return:         "Return-Path:" path CRLF;
path:           CFWSopt "<" path1 ">" CFWSopt;
path1:          CFWSopt | addr-spec;
received:       "Received:" name-val-list CRLF ";" date-time CRLF;
name-val-list:  CFWSopt name-val-list1;
name-val-list1: name-val-pair name-val-list2 | ;
name-val-list2: CFWS name-val-pair name-val-list2 | ;
name-val-pair:  item-name CFWS item-value;
item-name:      ALPHA item-name1;
item-name1:     item-name2 item-name1 | ;
item-name2:     item-name3 item-name4;
item-name3:     "-" | ;
item-name4:     ALPHA | DIGIT;
item-value:     angle-addr1 | addr-spec | atom | domain | msg-id;
angle-addr1:    angle-addr angle-addr1b;
angle-addr1b:   angle-addr angle-addr1b | ;

optional-field: field-name ":" unstructured CRLF;
field-name:     ftext1;
ftext1:         ftext ftext1b;
ftext1b:        ftext ftext1b | ;
ftext:          d33.57 | %d59.126;

attribute:      token;
composite-type: "message" | "multipart" | extension-token;
content:        "Content-Type" ":" type "/" subtype content1;
content1:       ";" parameter content1 | ;
description:    "Content-Description" ":" text0;
text0:          txt text0 | ;
discrete-type:  "text" | "image" | "audio" | "video" | "application"
                | extension-token;
encoding:       "Content-Transfer-Encoding" ":" mechanism;
```

```
entity-headers: entity_h1 entity_h2 entity_h3 entity_h4;
entity_h1:      content CRLF | ;
entity_h2:      encoding CRLF | ;
entity_h3:      id CRLF | ;
entity_h4:      description CRLF | ;
extension-token:        x-token;
id:             "Content-ID" ":" msg-id;
mechanism:      "7bit" | "8bit" | "binary" | "quoted-printable"
                | "base64" | x-token;
MIME-message-headers:   entity-headers fields version CRLF;
parameter:      attribute "=" value;
subtype:        extension-token;
token:          text1;
transport-padding:      LWSP transport-padding | ;
type:           discrete-type | composite-type;
value:          token | quoted-string;
version:        "MIME-Version" ":" DIGIT1 "." DIGIT1;
DIGIT1:         DIGIT DIG1b;
DIG1b:          DIGIT DIG1b | ;
x-token:        x-token1 token;
x-token1:       "X-" | "x-";

boundary:       boundary1 bcharsnospace;
boundary1:      bchars boundary1 | ;
bchars:         bcharsnospace | " ";
bcharsnospace:  DIGIT | ALPHA | "'" | "(" | ")" | "+" | "_" | ","
                | "-" | "." | "/" | ":" | "=" | "?";
body-part:      message;
close-delimiter:        delimiter "--";
dash-boundary:  "--" boundary;
delimiter:      CRLF dash-boundary;
discard-text:   text0 CRLF discard-text | ;
encapsulation:  delimiter transport-padding CRLF body-part;
epilogue:       discard-text;
multipart-body: mpart-body1 dash-boundary transport-padding CRLF body-part mpart-body2
                close-delimiter transport-padding mpart-body3;
mpart-body1:    preamble CRLF | ;
mpart-body2:    encapsulation mpart-body2 | ;
mpart-body3:    CRLF epilogue | ;
preamble:       discard-text;

%%
```

Appendix B

Covariance Model Data

This appendix contains a listing of the covariance models (CMs) that were used for the analyses performed in Chapters 6 and 7 along with their characteristics. All of the models listed are from the Rfam 8.0 database [33]. Additional results from the analysis on the baseline architecture are in Appendix C. Additional results from the analysis on the processor array architecture are in Appendix D.

CM	Num Nodes	Num States	W	Num ROOT	Num MATP	Num MATL	Num MATR	Num BIF	Num BEGL	Num BEGR	Num END
RF00001	91	366	137	1	34	31	20	1	1	1	2
RF00005	61	230	130	1	21	29	1	2	2	2	3
RF00006	74	280	184	1	20	44	8	0	0	0	1
RF00008	45	171	105	1	15	20	4	1	1	1	2
RF00014	66	275	104	1	31	22	3	2	2	2	3
RF00015	120	430	175	1	31	72	3	3	3	3	4
RF00016	107	352	176	1	11	59	35	0	0	0	1
RF00019	88	307	139	1	15	62	9	0	0	0	1
RF00021	91	368	137	1	37	31	13	2	2	2	3
RF00026	99	310	210	1	5	4	88	0	0	0	1
RF00027	54	253	105	1	31	11	10	0	0	0	1
RF00031	43	193	90	1	22	13	6	0	0	0	1
RF00032	22	82	41	1	6	9	5	0	0	0	1
RF00033	84	291	114	1	16	56	6	1	1	1	2
RF00034	104	343	131	1	18	61	11	3	3	3	4
RF00035	94	344	132	1	26	54	4	2	2	2	3
RF00037	22	94	45	1	10	9	1	0	0	0	1
RF00038	118	394	151	1	14	72	30	0	0	0	1
RF00039	43	160	72	1	11	26	4	0	0	0	1
RF00041	102	377	146	1	29	61	2	2	2	2	3
RF00042	72	282	122	1	25	39	2	1	1	1	2
RF00043	60	234	93	1	21	22	11	1	1	1	2
RF00046	94	292	114	1	4	88	0	0	0	0	1
RF00047	52	220	89	1	22	22	6	0	0	0	1
RF00048	51	187	78	1	12	23	14	0	0	0	1
RF00049	76	241	103	1	5	68	1	0	0	0	1
RF00051	59	250	100	1	25	22	10	0	0	0	1
RF00052	52	220	90	1	22	22	6	0	0	0	1
RF00053	62	268	107	1	28	24	8	0	0	0	1
RF00054	85	268	106	1	5	77	1	0	0	0	1
RF00055	82	259	172	1	5	75	0	0	0	0	1
RF00056	101	411	153	1	39	43	13	1	1	1	2
RF00057	77	261	103	1	13	48	10	1	1	1	2
RF00060	114	400	151	1	20	55	37	0	0	0	1
RF00063	82	295	114	1	17	19	44	0	0	0	1
RF00065	112	388	145	1	18	17	75	0	0	0	1
RF00066	38	139	84	1	9	24	3	0	0	0	1
RF00068	93	289	113	1	4	84	3	0	0	0	1
RF00069	74	232	130	1	4	68	0	0	0	0	1
RF00070	64	199	136	1	3	57	2	0	0	0	1

Table B.1: Covariance Model Data

Appendix C

Additional Results for the Baseline Architecture

This appendix contains additional results comparing the estimated performance of the baseline architecture to the INFERNAL (version 0.81) software package.

The evaluation system for the INFERNAL software contains dual Intel Xeon 2.8 GHz CPUs and 6 GBytes of DDR2 SDRAM running Linux CentOS 5.0. INFERNAL was run with the *–toponly* and *–noalign* options to ensure that INFERNAL was not doing more work than the baseline architecture. Results were collected for both the standard version of INFERNAL as well as INFERNAL using the Query Dependent Banding (QDB) heuristic [53]. The time required for INFERNAL to process a randomly generated database of 1 million residues was measured. For this appendix, the time required to process 1 million residues was used to estimate the time required to process 100 million residues.

As described in Section 6.5, these results for the baseline architecture are an estimate based on several factors including: the depth of the pipeline required to process a given covariance model (CM), the measured time required to transmit 100 million residues to the baseline architecture, and a clock frequency of 100 MHz for the baseline architecture.

- **CM -** the covariance model from the Rfam 8.0 database.
- **Num PEs -** the number of processing elements required for the baseline architecture pipeline.

- **Pipeline Width -** the maximum width of the pipeline.
- **Pipeline Depth -** the depth of the pipeline. The depth of the pipeline is used to define the latency of the pipeline.
- **Latency -** the latency of the pipeline for a pipeline running at 100 MHz.
- **HW Processing Time -** the time, in seconds, that it takes the baseline architecture to process 100 million residues.
- **I/O Latency -** the measured time, in seconds, to send 100 million residues to the baseline architecture.
- **Total Time with measured I/O -** the expected time, in seconds, to process 100 million residues with the baseline architecture on the test system. The I/O on the test system required 41.434 seconds to transmit 100 million residues (25 MBytes) to the baseline architecture.
- **Infernal Time (seconds) -** the time required for INFERNAL to process 100 million residues. The number is based on the results in Appendix E, which measured the time for INFERNAL to process 1 million residues.
- **Infernal Time (QDB) (seconds) -** the time required for INFERNAL to process 1 million residues when using the QDB heuristic. The number is based on the results in Appendix E. which measured the time for INFERNAL to process 1 million residues with the QDB heuristic.
- **Expected Speedup over Infernal -** the estimated speedup of the baseline architecture over the INFERNAL software package.
- **Expected Speedup over Infernal (w/QDB) -** the estimated speedup of the baseline architecture over the INFERNAL software package using the QDB heuristic.

CM	Num PEs	Pipeline Width	Pipeline Depth	Latency (ns)	HW Processing Time (seconds)	Total Time with measured I/O (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Expected Speedup over Infernal	Expected Speedup over Infernal (w/QDB)
RF00001	3539545	39492	195	19500	1.0000195	42.4340195	349492	128443	8236	3027
RF00005	2093665	26583	162	16200	1.0000162	42.4340162	236010	87388	5562	2059
RF00006	4758405	41957	257	25700	1.0000257	42.4340257	330608	230154	7791	5424
RF00008	984853	15681	141	14100	1.0000141	42.4340141	134133	85132	3161	2006
RF00014	1568965	23659	141	14100	1.0000141	42.4340141	222702	79799	5248	1881
RF00015	7138305	66415	230	23000	1.000023	42.434023	564719	159732	13308	3764
RF00016	5484002	43256	282	28200	1.0000282	42.4340282	336000	188521	7918	4443
RF00019	2987475	30681	226	22600	1.0000226	42.4340226	254675	145867	6002	3438
RF00021	3639478	41427	189	18900	1.0000189	42.4340189	379347	122324	8940	2883
RF00026	6865523	49359	308	30800	1.0000308	42.4340308	304166	162165	7168	3822
RF00027	1404743	20629	158	15800	1.0000158	42.4340158	200735	126783	4731	2988
RF00031	788082	13394	132	13200	1.0000132	42.4340132	123939	74592	2921	1758
RF00032	70377	2463	62	6200	1.0000062	42.4340062	21821	14962	514	353
RF00033	1954810	23037	189	18900	1.0000189	42.4340189	210954	85281	4971	2010
RF00034	3181038	38772	187	18700	1.0000187	42.4340187	314836	87520	7419	2062
RF00035	3165067	38533	183	18300	1.0000183	42.4340183	323801	106147	7631	2501
RF00037	96771	3341	66	6600	1.0000066	42.4340066	30798	22066	726	520
RF00038	4525738	36601	268	26800	1.0000268	42.4340268	330370	138539	7785	3265
RF00039	420126	8155	114	11400	1.0000114	42.4340114	74698	48491	1760	1143
RF00041	4243415	44509	206	20600	1.0000206	42.4340206	388156	118692	9147	2797
RF00042	2170975	28389	178	17800	1.0000178	42.4340178	243994	100369	5750	2365
RF00043	1046417	17343	130	13000	1.000013	42.434013	154047	65760	3630	1550
RF00046	1917744	20339	207	20700	1.0000207	42.4340207	169687	83352	3999	1964
RF00047	879727	14354	140	14000	1.000014	42.434014	135865	84975	3202	2003
RF00048	576080	9767	128	12800	1.0000128	42.4340128	90967	53942	2144	1271
RF00049	1292725	16405	178	17800	1.0000178	42.4340178	130538	68781	3076	1621
RF00051	1260808	17862	158	15800	1.0000158	42.4340158	175105	100940	4127	2379
RF00052	899466	14615	141	14100	1.0000141	42.4340141	139274	83901	3282	1977
RF00053	1546318	21105	168	16800	1.0000168	42.4340168	205787	118720	4850	2798
RF00054	1522615	17848	190	19000	1.000019	42.434019	147889	79855	3485	1882
RF00055	3851677	34687	253	25300	1.0000253	42.4340253	230660	139126	5436	3279
RF00056	4953865	50268	217	21700	1.0000217	42.4340217	442269	155548	10423	3666
RF00057	1432755	19811	159	15900	1.0000159	42.4340159	165748	67965	3906	1602
RF00060	4593080	36693	264	26400	1.0000264	42.4340264	353784	148386	8337	3497
RF00063	1934547	20575	195	19500	1.0000195	42.4340195	197771	95906	4661	2260
RF00065	4109991	33312	256	25600	1.0000256	42.4340256	311563	132844	7342	3131
RF00066	495049	8839	121	12100	1.0000121	42.4340121	73631	52869	1735	1246
RF00068	1865044	20008	205	20500	1.0000205	42.4340205	166359	82574	3920	1946
RF00069	1976008	22185	203	20300	1.0000203	42.4340203	156322	90767	3684	2139
RF00070	1852791	21084	199	19900	1.0000199	42.4340199	141281	100989	3329	2380

Table C.1: Estimated speedup for baseline architecture running at 100 MHz compared to INFERNAL

Appendix D

Additional Results for the Processor Array Architecture

This appendix contains a complete set of results for the processor array architecture presented in Chapter 7. Tables D.1 through D.9 present results for 40 different covariance models (CMs) from the Rfam 8.0 database [33] for a single processor up to 256 processors. In those tables, the columns are as follows:

- **CM -** the covariance model from the Rfam 8.0 database.
- **Time -** the time, in seconds, to generate a schedule for a single window W of the CM (on Intel Xeon 2.8 GHz CPU with 6 GBytes of DDR2 SDRAM running Linux CentOS 5.0).
- **Num Computations -** the number of computations (i.e. the number of nodes in the task graph) that need to be completed for the specified CM.
- **Cycles To Compute -** the number of clock cycles (i.e. the number of time slots in the final schedule) that it takes to compute the results for a single window W of the specified CM, including any idle time in the schedule.
- **Total Cycles -** the total number of clock cycles, among all processors, that are required to compute the result (i.e. $p \times CyclesToCompute$ where p is the number of processors), including any idle time in the schedule.

- **Schedule Efficiency -** the efficiency of the schedule that is generated for the specified CM. The efficiency of the schedule is a measure of the number of idle tasks inserted into the schedule ($\frac{NumComputations}{TotalCycles}$). The fewer idle tasks inserted into the schedule, the higher the efficiency.
- **Max Live Memory -** the maximum number of intermediary values that must be stored throughout the alignment computation. The numbers in the tables represent generic memory locations. The actual size of the memory required will depend on the size/precision of the intermediary values as described in Section 5.5. If two bytes are used to represent values, then the required memory will be $2 \times MaxLiveMemory$ bytes.
- **Average Memory -** the average amount of memory required for each processor in the processor array architecture.
- **Speedup -** the speedup over a processor array with only a single processor (this column is not shown in Table D.1, as the value is simply 1).
- **Processor Efficiency -** the efficiency of the processors in the processor array architecture as compared to a processor array with only a single processor ($\frac{Speedup}{p}$ where p is the number of processors in the processor array).
- **Schedule Size/Proc -** the size, in MB, of the schedule for each processor in the processor array architecture. The size is based on the instruction format discussed in Section 7.2.1.
- **Total Schedule Size -** the total size, in MB, of the schedules for all processors in the processor array architecture. The size is based on the instruction format discussed in Section 7.2.1.

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Schedule Size /Proc(MB)	Total Schedule Size (MB)
RF00001	90.77	3539545	3539555	3539555	99.99	68746	68746	53.54	53.54
RF00005	51.41	2093665	2093675	2093675	99.99	64822	64822	31.67	31.67
RF00006	111.15	4758405	4758415	4758415	99.99	63228	63228	71.97	71.97
RF00008	21.24	984853	984864	984864	99.99	30365	30365	14.90	14.90
RF00014	37.04	1568965	1568976	1568976	99.99	48801	48801	23.73	23.73
RF00015	216.88	7138305	7138315	7138315	99.99	186178	186178	107.97	107.97
RF00016	121.57	5484002	5484012	5484012	99.99	60919	60919	82.95	82.95
RF00019	64.31	2987475	2987485	2987485	99.99	45247	45247	45.19	45.19
RF00021	91.73	3639478	3639489	3639489	99.99	87192	87192	55.05	55.05
RF00026	155.56	6865523	6865533	6865533	99.99	67680	67680	103.84	103.84
RF00027	31.57	1404743	1404753	1404753	99.99	33218	33218	21.25	21.25
RF00031	16.68	788082	788092	788092	99.99	21247	21247	11.92	11.92
RF00032	1.06	70377	70387	70387	99.98	3752	3752	1.06	1.06
RF00033	42.5	1954810	1954820	1954820	99.99	47492	47492	29.57	29.57
RF00034	78.04	3181038	3181049	3181049	99.99	97069	97069	48.11	48.11
RF00035	78.74	3165067	3165078	3165078	99.99	82303	82303	47.87	47.87
RF00037	1.65	96771	96781	96781	99.98	5359	5359	1.46	1.46
RF00038	98.84	4525738	4525748	4525748	99.99	51965	51965	68.45	68.45
RF00039	7.82	420126	420136	420136	99.99	12227	12227	6.35	6.35
RF00041	114.29	4243415	4243425	4243425	99.99	100324	100324	64.18	64.18
RF00042	52	2170975	2170985	2170985	99.99	50298	50298	32.84	32.84
RF00043	22.84	1046417	1046428	1046428	99.99	28082	28082	15.83	15.83
RF00046	37.74	1917744	1917755	1917755	99.99	28471	28471	29.01	29.01
RF00047	18.7	879727	879737	879737	99.99	22566	22566	13.31	13.31
RF00048	10.9	576080	576090	576090	99.99	14417	14417	8.71	8.71
RF00049	24.8	1292725	1292736	1292736	99.99	23307	23307	19.55	19.55
RF00051	27.09	1260808	1260819	1260819	99.99	27856	27856	19.07	19.07
RF00052	18.99	899466	899476	899476	99.99	22945	22945	13.60	13.60
RF00053	34.29	1546318	1546329	1546329	99.99	33326	33326	23.39	23.39
RF00054	29.58	1522615	1522625	1522625	99.99	25285	25285	23.03	23.03
RF00055	81.22	3851677	3851687	3851687	99.99	48674	48674	58.26	58.26
RF00056	126.02	4953865	4953875	4953875	99.99	83928	83928	74.93	74.93
RF00057	30.32	1432755	1432766	1432766	99.99	34674	34674	21.67	21.67
RF00060	103.21	4593080	4593090	4593090	99.99	52284	52284	69.47	69.47
RF00063	41.07	1934547	1934558	1934558	99.99	29404	29404	29.26	29.26
RF00065	91.31	4109991	4110001	4110001	99.99	46918	46918	62.16	62.16
RF00066	9.28	495049	495059	495059	99.99	13082	13082	7.49	7.49
RF00068	36.49	1865044	1865055	1865055	99.99	28043	28043	28.21	28.21
RF00069	39.38	1976008	1976018	1976018	99.99	31112	31112	29.89	29.89
RF00070	36.98	1852791	1852801	1852801	99.99	29603	29603	28.02	28.02

Table D.1: Results for processor array architecture using 1 processor

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	90.68	3539545	1820222	3640444	97.22	75435	37718	1.9446	97.22	28.90	57.79
RF00005	51.43	2093665	1080025	2160050	96.92	67847	33924	1.9385	96.92	17.15	34.29
RF00006	111.58	4758405	2387850	4775700	99.63	63200	31600	1.9928	99.63	37.91	75.81
RF00008	21.27	984853	510080	1020160	96.53	33464	16732	1.9308	96.54	8.10	16.20
RF00014	37.69	1568965	805407	1610814	97.4	50985	25493	1.9481	97.4	12.79	25.57
RF00015	219.42	7138305	3840389	7680778	92.93	195532	97766	1.8587	92.93	60.97	121.93
RF00016	121.35	5484002	2752527	5505054	99.61	60778	30389	1.9924	99.61	43.70	87.39
RF00019	64.37	2987475	1501391	3002782	99.49	45229	22615	1.9898	99.49	23.83	47.67
RF00021	91.49	3639478	1864457	3728914	97.6	91346	45673	1.9520	97.6	29.60	59.20
RF00026	154.83	6865523	3445679	6891358	99.62	67568	33784	1.9925	99.62	54.70	109.40
RF00027	31.46	1404743	703932	1407864	99.77	33240	16620	1.9956	99.77	11.17	22.35
RF00031	16.77	788082	397789	795578	99.05	21193	10597	1.9812	99.05	6.31	12.63
RF00032	1.07	70377	35904	71808	98	3748	1874	1.9604	98.02	0.57	1.14
RF00033	42.51	1954810	1007208	2014416	97.04	50651	25326	1.9408	97.04	15.99	31.98
RF00034	78.25	3181038	1716295	3432590	92.67	105360	52680	1.8534	92.67	27.25	54.49
RF00035	79.03	3165067	1676144	3352288	94.41	90201	45101	1.8883	94.41	26.61	53.22
RF00037	1.65	96771	49396	98792	97.95	5333	2667	1.9593	97.96	0.78	1.57
RF00038	99.19	4525738	2270072	4540144	99.68	51946	25973	1.9937	99.68	36.04	72.07
RF00039	7.86	420126	211179	422358	99.47	12238	6119	1.9895	99.47	3.35	6.70
RF00041	114.63	4243415	2158518	4317036	98.29	105997	52999	1.9659	98.29	34.27	68.53
RF00042	51.81	2170975	1115516	2231032	97.3	55479	27740	1.9462	97.3	17.71	35.42
RF00043	22.99	1046417	541224	1082448	96.67	29242	14621	1.9334	96.67	8.59	17.18
RF00046	37.8	1917744	962804	1925608	99.59	28458	14229	1.9918	99.59	15.28	30.57
RF00047	18.72	879727	442308	884616	99.44	22584	11292	1.9890	99.44	7.02	14.04
RF00048	10.91	576080	289796	579592	99.39	14436	7218	1.9879	99.39	4.60	9.20
RF00049	24.81	1292725	649021	1298042	99.59	23242	11621	1.9918	99.59	10.30	20.61
RF00051	27.14	1260808	633193	1266386	99.55	27873	13937	1.9912	99.56	10.05	20.10
RF00052	19.03	899466	453187	906374	99.23	22966	11483	1.9848	99.23	7.19	14.39
RF00053	34.61	1546318	778208	1556416	99.35	33291	16646	1.9870	99.35	12.35	24.71
RF00054	29.62	1522615	765672	1531344	99.42	25249	12625	1.9886	99.43	12.16	24.31
RF00055	81.24	3851677	1933302	3866604	99.61	48632	24316	1.9923	99.61	30.69	61.38
RF00056	126.27	4953865	2549245	5098490	97.16	92332	46166	1.9433	97.16	40.47	80.94
RF00057	30.34	1432755	739903	1479806	96.82	37241	18621	1.9364	96.82	11.75	23.49
RF00060	103.25	4593080	2302728	4605456	99.73	52272	26136	1.9946	99.73	36.56	73.11
RF00063	41.08	1934547	972462	1944924	99.46	29381	14691	1.9893	99.46	15.44	30.88
RF00065	91.41	4109991	2062482	4124964	99.63	46897	23449	1.9927	99.63	32.74	65.48
RF00066	9.3	495049	249732	499464	99.11	13086	6543	1.9824	99.11	3.96	7.93
RF00068	36.53	1865044	936219	1872438	99.6	28009	14005	1.9921	99.6	14.86	29.72
RF00069	39.38	1976008	993568	1987136	99.43	31018	15509	1.9888	99.44	15.77	31.55
RF00070	36.99	1852791	929770	1859540	99.63	29501	14751	1.9928	99.63	14.76	29.52

Table D.2: Results for processor array architecture using 2 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	90.9	3539545	955529	3822116	92.6	81918	20480	3.7043	92.6	15.89	63.54
RF00005	51.64	2093665	594620	2378480	88.02	75864	18966	3.5210	88.02	9.89	39.54
RF00006	111.03	4758405	1200196	4800784	99.11	63308	15827	3.9647	99.11	19.95	79.81
RF00008	21.27	984853	276991	1107964	88.88	36319	9080	3.5556	88.88	4.60	18.42
RF00014	37.14	1568965	432526	1730104	90.68	56554	14139	3.6275	90.68	7.19	28.76
RF00015	219.01	7138305	2205496	8821984	80.91	207493	51873	3.2366	80.91	36.67	146.67
RF00016	121.61	5484002	1384436	5537744	99.02	60833	15208	3.9612	99.02	23.02	92.06
RF00019	64.66	2987475	756719	3026876	98.69	45265	11316	3.9479	98.69	12.58	50.32
RF00021	91.78	3639478	1003565	4014260	90.66	103569	25892	3.6266	90.66	16.68	66.74
RF00026	155.4	6865523	1735207	6940828	98.91	67677	16919	3.9566	98.91	28.85	115.39
RF00027	31.46	1404743	354550	1418200	99.05	33254	8314	3.9621	99.05	5.89	23.58
RF00031	16.97	788082	200724	802896	98.15	21228	5307	3.9262	98.15	3.34	13.35
RF00032	1.07	70377	18488	73952	95.16	3765	941	3.8072	95.17	0.31	1.23
RF00033	42.66	1954810	533660	2134640	91.57	53340	13335	3.6630	91.57	8.87	35.49
RF00034	78.44	3181038	924464	3697856	86.02	109249	27312	3.4410	86.02	15.37	61.48
RF00035	78.99	3165067	921743	3686972	85.84	98955	24739	3.4338	85.84	15.32	61.30
RF00037	1.66	96771	25304	101216	95.6	5340	1335	3.8247	95.61	0.42	1.68
RF00038	99.41	4525738	1142595	4570380	99.02	51954	12989	3.9609	99.02	19.00	75.98
RF00039	7.78	420126	106972	427888	98.18	12257	3064	3.9275	98.18	1.78	7.11
RF00041	114.65	4243415	1148605	4594420	92.36	114350	28588	3.6944	92.36	19.10	76.38
RF00042	51.52	2170975	589320	2357280	92.09	60095	15024	3.6839	92.09	9.80	39.19
RF00043	22.89	1046417	285215	1140860	91.72	32094	8024	3.6689	91.72	4.74	18.97
RF00046	37.79	1917744	485713	1942852	98.7	28471	7118	3.9483	98.7	8.07	32.30
RF00047	18.62	879727	223216	892864	98.52	22592	5648	3.9412	98.52	3.71	14.84
RF00048	10.96	576080	146830	587320	98.08	14449	3612	3.9235	98.08	2.44	9.76
RF00049	24.88	1292725	327840	1311360	98.57	23334	5834	3.9432	98.57	5.45	21.80
RF00051	27.21	1260808	319749	1278996	98.57	27875	6969	3.9432	98.57	5.32	21.26
RF00052	19.02	899466	228619	914476	98.35	23046	5762	3.9344	98.35	3.80	15.20
RF00053	34.19	1546318	392068	1568272	98.6	33313	8328	3.9440	98.6	6.52	26.07
RF00054	29.61	1522615	386622	1546488	98.45	25267	6317	3.9383	98.45	6.43	25.71
RF00055	81.46	3851677	974827	3899308	98.77	48720	12180	3.9511	98.77	16.21	64.83
RF00056	127	4953865	1344198	5376792	92.13	95946	23987	3.6854	92.13	22.35	89.39
RF00057	30.45	1432755	391639	1566556	91.45	39757	9939	3.6584	91.45	6.51	26.04
RF00060	102.81	4593080	1158310	4633240	99.13	52283	13071	3.9653	99.13	19.26	77.03
RF00063	41.05	1934547	490673	1962692	98.56	29410	7353	3.9427	98.56	8.16	32.63
RF00065	91.58	4109991	1037855	4151420	99	46914	11729	3.9601	99	17.25	69.02
RF00066	9.25	495049	126885	507540	97.53	13132	3283	3.9016	97.54	2.11	8.44
RF00068	36.53	1865044	472346	1889384	98.71	28025	7006	3.9485	98.71	7.85	31.41
RF00069	39.45	1976008	500935	2003740	98.61	31065	7766	3.9447	98.61	8.33	33.31
RF00070	37.02	1852791	470044	1880176	98.54	29571	7393	3.9418	98.54	7.81	31.26

Table D.3: Results for processor array architecture using 4 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	91.25	3539545	528279	4226232	83.75	84104	10513	6.7002	83.75	9.18	73.43
RF00005	51.7	2093665	349005	2792040	74.98	81207	10151	5.9990	74.98	6.06	48.51
RF00006	111.02	4758405	606182	4849456	98.12	63525	7941	7.8498	98.12	10.53	84.26
RF00008	21.52	984853	160472	1283776	76.71	36723	4590	6.1373	76.71	2.79	22.31
RF00014	37.14	1568965	246029	1968232	79.71	59180	7398	6.3772	79.71	4.27	34.20
RF00015	221.93	7138305	1239665	9917320	71.97	215430	26929	5.7583	71.97	21.54	172.31
RF00016	122.43	5484002	700234	5601872	97.89	61110	7639	7.8317	97.89	12.17	97.33
RF00019	64.86	2987475	384331	3074648	97.16	45617	5702	7.7732	97.16	6.68	53.42
RF00021	92.07	3639478	573825	4590600	79.28	111727	13966	6.3425	79.28	9.97	79.76
RF00026	155.02	6865523	877031	7016248	97.85	68039	8505	7.8282	97.85	15.24	121.91
RF00027	31.47	1404743	180497	1443976	97.28	33500	4188	7.7827	97.28	3.14	25.09
RF00031	16.84	788082	102234	817872	96.35	21447	2681	7.7087	96.35	1.78	14.21
RF00032	1.07	70377	9732	77856	90.39	3881	485	7.2325	90.4	0.17	1.35
RF00033	42.71	1954810	292538	2340304	83.52	51686	6461	6.6823	83.52	5.08	40.66
RF00034	78.61	3181038	558237	4465896	71.22	112083	14010	5.6984	71.22	9.70	77.59
RF00035	79.27	3165067	545370	4362960	72.54	100468	12559	5.8035	72.54	9.48	75.81
RF00037	1.67	96771	13197	105576	91.66	5458	682	7.3336	91.66	0.23	1.83
RF00038	99.12	4525738	578762	4630096	97.74	52475	6559	7.8197	97.74	10.06	80.45
RF00039	7.81	420126	54852	438816	95.74	12497	1562	7.6594	95.74	0.95	7.62
RF00041	115.64	4243415	645430	5163440	82.18	123154	15394	6.5746	82.18	11.21	89.71
RF00042	51.8	2170975	329103	2632824	82.45	63363	7920	6.5967	82.45	5.72	45.75
RF00043	22.96	1046417	159467	1275736	82.02	34055	4257	6.5620	82.02	2.77	22.17
RF00046	37.99	1917744	247277	1978216	96.94	28828	3604	7.7555	96.94	4.30	34.37
RF00047	18.67	879727	114048	912384	96.42	22946	2868	7.7137	96.42	1.98	15.85
RF00048	11.04	576080	75370	602960	95.54	14714	1839	7.6435	95.54	1.31	10.48
RF00049	24.95	1292725	167149	1337192	96.67	23479	2935	7.7340	96.67	2.90	23.23
RF00051	27.19	1260808	162789	1302312	96.81	28286	3536	7.7451	96.81	2.83	22.63
RF00052	19.12	899466	116779	934232	96.27	23163	2895	7.7024	96.27	2.03	16.23
RF00053	34.47	1546318	199527	1596216	96.87	33635	4204	7.7500	96.87	3.47	27.73
RF00054	29.67	1522615	197020	1576160	96.6	25568	3196	7.7283	96.6	3.42	27.39
RF00055	81.96	3851677	495908	3967264	97.08	48957	6120	7.7669	97.08	8.62	68.93
RF00056	127.38	4953865	733609	5868872	84.4	101968	12746	6.7527	84.4	12.75	101.97
RF00057	30.82	1432755	215227	1721816	83.21	39179	4897	6.6570	83.21	3.74	29.92
RF00060	102.99	4593080	586151	4689208	97.95	52871	6609	7.8360	97.95	10.18	81.47
RF00063	41.13	1934547	249050	1992400	97.09	29855	3732	7.7677	97.09	4.33	34.62
RF00065	94.18	4109991	525896	4207168	97.69	47476	5935	7.8152	97.69	9.14	73.10
RF00066	9.24	495049	65513	524104	94.45	13303	1663	7.5567	94.45	1.14	9.11
RF00068	36.69	1865044	240400	1923200	96.97	28297	3537	7.7581	96.97	4.18	33.42
RF00069	39.71	1976008	254633	2037064	97	31217	3902	7.7603	97	4.42	35.39
RF00070	37.54	1852791	238879	1911032	96.95	29745	3718	7.7562	96.95	4.15	33.20

Table D.4: Results for processor array architecture using 8 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	91.2	3539545	309805	4956880	71.4	85218	5326	11.4251	71.4	5.62	89.84
RF00005	51.98	2093665	230130	3682080	56.86	84917	5307	9.0978	56.86	4.17	66.74
RF00006	111.25	4758405	312572	5001152	95.14	64120	4008	15.2234	95.14	5.67	90.65
RF00008	21.41	984853	103169	1650704	59.66	36862	2304	9.5461	59.66	1.87	29.92
RF00014	37.33	1568965	153412	2454592	63.91	61716	3857	10.2272	63.92	2.78	44.49
RF00015	223.21	7138305	731916	11710656	60.95	222465	13904	9.7529	60.95	13.27	212.26
RF00016	122.14	5484002	357993	5727888	95.74	61503	3844	15.3188	95.74	6.49	103.82
RF00019	65.11	2987475	198072	3169152	94.26	46196	2887	15.0828	94.26	3.59	57.44
RF00021	92.84	3639478	358290	5732640	63.48	111684	6980	10.1579	63.48	6.49	103.90
RF00026	155.68	6865523	451164	7218624	95.1	68542	4284	15.2174	95.1	8.18	130.84
RF00027	31.52	1404743	92578	1481248	94.83	33919	2120	15.1737	94.83	1.68	26.85
RF00031	16.76	788082	52714	843424	93.43	21755	1360	14.9503	93.43	0.96	15.29
RF00032	1.08	70377	5438	87008	80.88	4058	254	12.9435	80.89	0.10	1.58
RF00033	43.1	1954810	174732	2795712	69.92	50873	3180	11.1875	69.92	3.17	50.67
RF00034	79	3181038	374443	5991088	53.09	114490	7156	8.4954	53.09	6.79	108.59
RF00035	79.48	3165067	309037	4944592	64.01	102304	6394	10.2417	64.01	5.60	89.62
RF00037	1.67	96771	7001	112016	86.39	5541	346	13.8239	86.39	0.13	2.03
RF00038	99.71	4525738	296884	4750144	95.27	53433	3340	15.2442	95.27	5.38	86.10
RF00039	7.83	420126	28792	460672	91.19	12882	805	14.5921	91.2	0.52	8.35
RF00041	115.26	4243415	397660	6362560	66.69	123335	7708	10.6710	66.69	7.21	115.32
RF00042	51.86	2170975	202850	3245600	66.88	65280	4080	10.7024	66.89	3.68	58.83
RF00043	23.02	1046417	97111	1553776	67.34	35857	2241	10.7756	67.34	1.76	28.16
RF00046	37.89	1917744	128207	2051312	93.48	29324	1833	14.9583	93.48	2.32	37.18
RF00047	18.69	879727	59119	945904	93	23527	1470	14.8808	93	1.07	17.14
RF00048	10.95	576080	39520	632320	91.1	15120	945	14.5772	91.1	0.72	11.46
RF00049	24.93	1292725	86903	1390448	92.97	23784	1487	14.8756	92.97	1.58	25.20
RF00051	27.26	1260808	83932	1342912	93.88	29054	1816	15.0219	93.88	1.52	24.34
RF00052	19.11	899466	60565	969040	92.82	23543	1471	14.8514	92.82	1.10	17.56
RF00053	34.3	1546318	102647	1642352	94.15	34129	2133	15.0645	94.15	1.86	29.77
RF00054	29.72	1522615	102148	1634368	93.16	25931	1621	14.9061	93.16	1.85	29.62
RF00055	81.94	3851677	256197	4099152	93.96	49164	3073	15.0341	93.96	4.64	74.30
RF00056	127.2	4953865	429114	6865824	72.15	107543	6721	11.5444	72.15	7.78	124.44
RF00057	30.55	1432755	127856	2045696	70.03	39735	2483	11.2061	70.03	2.32	37.08
RF00060	103.65	4593080	300227	4803632	95.61	53957	3372	15.2987	95.61	5.44	87.07
RF00063	41.25	1934547	128706	2059296	93.94	30646	1915	15.0308	93.94	2.33	37.32
RF00065	91.87	4109991	269852	4317632	95.19	48585	3037	15.2306	95.19	4.89	78.26
RF00066	9.3	495049	34333	549328	90.11	13694	856	14.4193	90.12	0.62	9.96
RF00068	36.7	1865044	124528	1992448	93.6	28748	1797	14.9770	93.6	2.26	36.11
RF00069	39.73	1976008	132286	2116576	93.35	31459	1966	14.9375	93.35	2.40	38.36
RF00070	37.2	1852791	124782	1996512	92.8	30004	1875	14.8483	92.8	2.26	36.19

Table D.5: Results for processor array architecture using 16 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	93.78	3539545	205777	6584864	53.75	87256	2727	17.2009	53.75	3.88	124.29
RF00005	52.21	2093665	169626	5428032	38.57	86009	2688	12.3429	38.57	3.20	102.45
RF00006	111.36	4758405	162819	5210208	91.32	65138	2036	29.2252	91.32	3.07	98.34
RF00008	21.59	984853	70954	2270528	43.37	37643	1176	13.8803	43.37	1.34	42.86
RF00014	37.55	1568965	105917	3389344	46.29	65069	2033	14.8133	46.29	2.00	63.97
RF00015	228.57	7138305	529156	16932992	42.15	226986	7093	13.4900	42.15	9.99	319.61
RF00016	122.18	5484002	187399	5996768	91.44	62456	1952	29.2638	91.44	3.54	113.19
RF00019	65.13	2987475	104614	3347648	89.24	47126	1473	28.5572	89.24	1.97	63.19
RF00021	92.8	3639478	244978	7839296	46.42	112807	3525	14.8564	46.42	4.62	147.97
RF00026	155.4	6865523	235125	7524000	91.24	69911	2185	29.1995	91.24	4.44	142.02
RF00027	31.65	1404743	48988	1567616	89.61	34824	1088	28.6755	89.61	0.92	29.59
RF00031	16.77	788082	28213	902816	87.29	22354	699	27.9336	87.29	0.53	17.04
RF00032	1.09	70377	3172	101504	69.33	4459	139	22.1901	69.34	0.06	1.92
RF00033	43.12	1954810	114334	3658688	53.42	52108	1628	17.0975	53.42	2.16	69.06
RF00034	79.76	3181038	242901	7772832	40.92	118716	3710	13.0961	40.92	4.58	146.71
RF00035	79.59	3165067	213938	6846016	46.23	106393	3325	14.7944	46.23	4.04	129.22
RF00037	1.66	96771	4001	128032	75.58	5876	184	24.1892	75.59	0.08	2.42
RF00038	99.33	4525738	156377	5004064	90.44	55483	1734	28.9413	90.44	2.95	94.45
RF00039	7.84	420126	15832	506624	82.92	13665	427	26.5371	82.92	0.30	9.56
RF00041	115.88	4243415	270050	8641600	49.1	127219	3976	15.7135	49.1	5.10	163.11
RF00042	52.24	2170975	137541	4401312	49.32	67569	2112	15.7843	49.32	2.60	83.07
RF00043	23.16	1046417	62245	1991840	52.53	37297	1166	16.8114	52.53	1.17	37.60
RF00046	37.9	1917744	68668	2197376	87.27	30048	939	27.9279	87.27	1.30	41.48
RF00047	18.82	879727	31570	1010240	87.08	24549	767	27.8662	87.08	0.60	19.07
RF00048	10.99	576080	21624	691968	83.25	16142	504	26.6412	83.25	0.41	13.06
RF00049	24.9	1292725	47084	1506688	85.79	24238	757	27.4560	85.79	0.89	28.44
RF00051	27.45	1260808	44568	1426176	88.4	30245	945	28.2898	88.4	0.84	26.92
RF00052	19.15	899466	32358	1035456	86.86	24222	757	27.7976	86.86	0.61	19.54
RF00053	34.53	1546318	55097	1763104	87.7	35121	1098	28.0656	87.7	1.04	33.28
RF00054	29.77	1522615	55108	1763456	86.34	26492	828	27.6298	86.34	1.04	33.29
RF00055	82.26	3851677	136717	4374944	88.03	49833	1557	28.1727	88.03	2.58	82.58
RF00056	127.61	4953865	283771	9080672	54.55	112011	3500	17.4573	54.55	5.36	171.40
RF00057	30.78	1432755	84246	2695872	53.14	40973	1280	17.0069	53.14	1.59	50.88
RF00060	103.82	4593080	157818	5050176	90.94	56127	1754	29.1037	90.94	2.98	95.32
RF00063	41.33	1934547	68694	2198208	88	32309	1010	28.1620	88	1.30	41.49
RF00065	92.16	4109991	141937	4541984	90.48	50670	1583	28.9565	90.48	2.68	85.73
RF00066	9.33	495049	19394	620608	79.76	14442	451	25.5264	79.76	0.37	11.71
RF00068	36.99	1865044	66693	2134176	87.38	29313	916	27.9648	87.38	1.26	40.28
RF00069	39.56	1976008	71340	2282880	86.55	31934	998	27.6986	86.55	1.35	43.09
RF00070	37.23	1852791	67875	2172000	85.3	30291	947	27.2973	85.3	1.28	41.00

Table D.6: Results for processor array architecture using 32 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	92.69	3539545	150066	9604224	36.85	90310	1411	23.5867	36.85	2.95	188.48
RF00005	53.12	2093665	136567	8740288	23.95	87417	1366	15.3308	23.95	2.68	171.53
RF00006	112.06	4758405	91220	5838080	81.5	67368	1053	52.1642	81.5	1.79	114.57
RF00008	21.78	984853	54568	3492352	28.2	38246	598	18.0484	28.2	1.07	68.54
RF00014	38.15	1568965	82806	5299584	29.6	67315	1052	18.9476	29.6	1.63	104.00
RF00015	238.35	7138305	419915	26874560	26.56	232379	3631	16.9994	26.56	8.24	527.41
RF00016	123.14	5484002	103658	6634112	82.66	65156	1018	52.9049	82.66	2.03	130.19
RF00019	64.87	2987475	57980	3710720	80.5	48885	764	51.5261	80.5	1.14	72.82
RF00021	93.56	3639478	186408	11930112	30.5	116610	1822	19.5243	30.5	3.66	234.13
RF00026	155.65	6865523	127202	8140928	84.33	72185	1128	53.9735	84.33	2.50	159.77
RF00027	31.61	1404743	27143	1737152	80.86	36575	571	51.7538	80.86	0.53	34.09
RF00031	16.86	788082	15820	1012480	77.83	24127	377	49.8162	77.83	0.31	19.87
RF00032	1.12	70377	2234	142976	49.22	4868	76	31.5072	49.22	0.04	2.81
RF00033	43.39	1954810	83096	5318144	36.75	54651	854	23.5248	36.75	1.63	104.37
RF00034	80.38	3181038	197402	12633728	25.17	123266	1926	16.1146	25.17	3.87	247.94
RF00035	80.51	3165067	169008	10816512	29.26	110206	1722	18.7274	29.26	3.32	212.27
RF00037	1.68	96771	2722	174208	55.54	6389	100	35.5551	55.55	0.05	3.42
RF00038	99.46	4525738	86112	5511168	82.11	58585	915	52.5565	82.11	1.69	108.16
RF00039	7.88	420126	9857	630848	66.59	14715	230	42.6231	66.59	0.19	12.38
RF00041	117.31	4243415	206822	13236608	32.05	131057	2048	20.5173	32.05	4.06	259.77
RF00042	52.62	2170975	106109	6790976	31.96	70439	1101	20.4600	31.96	2.08	133.27
RF00043	23.56	1046417	48633	3112512	33.61	39492	617	21.5168	33.62	0.95	61.08
RF00046	38.16	1917744	39108	2502912	76.62	31327	489	49.0374	76.62	0.77	49.12
RF00047	18.9	879727	18616	1191424	73.83	26080	408	47.2570	73.83	0.37	23.38
RF00048	11.01	576080	12715	813760	70.79	17775	278	45.3079	70.79	0.25	15.97
RF00049	24.98	1292725	27060	1731840	74.64	25339	396	47.7729	74.64	0.53	33.99
RF00051	27.33	1260808	25224	1614336	78.1	32275	504	49.9849	78.1	0.50	31.68
RF00052	19.18	899466	18849	1206336	74.56	25795	403	47.7201	74.56	0.37	23.67
RF00053	34.38	1546318	31248	1999872	77.32	36717	574	49.4857	77.32	0.61	39.25
RF00054	29.82	1522615	31651	2025664	75.16	27685	433	48.1067	75.16	0.62	39.75
RF00055	82.29	3851677	76939	4924096	78.22	51230	800	50.0616	78.22	1.51	96.64
RF00056	129.21	4953865	212909	13626176	36.35	116636	1822	23.2676	36.35	4.18	267.41
RF00057	31.02	1432755	63033	4034112	35.51	43165	674	22.7304	35.51	1.24	79.17
RF00060	103.48	4593080	86803	5555392	82.67	60128	940	52.9140	82.67	1.70	109.02
RF00063	41.35	1934547	38576	2468864	78.35	34649	541	50.1493	78.35	0.76	48.45
RF00065	92.24	4109991	78281	5009984	82.03	54466	851	52.5032	82.03	1.54	98.32
RF00066	9.4	495049	11930	763520	64.83	15669	245	41.4970	64.83	0.23	14.98
RF00068	36.82	1865044	38096	2438144	76.49	30732	480	48.9567	76.49	0.75	47.85
RF00069	39.73	1976008	40185	2571840	76.83	33142	518	49.1730	76.83	0.79	50.47
RF00070	37.36	1852791	39423	2523072	73.43	31455	491	46.9980	73.43	0.77	49.52

Table D.7: Results for processor array architecture using 64 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	93.7	3539545	121668	15573504	22.72	95261	744	29.0919	22.72	2.48	317.31
RF00005	54.27	2093665	122556	15687168	13.34	89708	701	17.0834	13.34	2.50	319.63
RF00006	111.95	4758405	55660	7124480	66.78	70684	552	85.4907	66.78	1.13	145.16
RF00008	22.44	984853	49374	6319872	15.58	40074	313	19.9470	15.58	1.01	128.77
RF00014	38.81	1568965	72227	9245056	16.97	71285	557	21.7228	16.97	1.47	188.37
RF00015	249.49	7138305	365015	46721920	15.27	239760	1873	19.5562	15.27	7.44	951.96
RF00016	122.89	5484002	60703	7769984	70.57	69448	543	90.3417	70.57	1.24	158.31
RF00019	65.44	2987475	35708	4570624	65.36	51918	406	83.6643	65.36	0.73	93.13
RF00021	95.94	3639478	162781	20835968	17.46	124426	972	22.3582	17.46	3.32	424.53
RF00026	156.99	6865523	77329	9898112	69.36	75852	593	88.7834	69.36	1.58	201.67
RF00027	31.76	1404743	16385	2097280	66.97	39537	309	85.7341	66.97	0.33	42.73
RF00031	16.91	788082	11044	1413632	55.74	26676	208	71.3593	55.74	0.23	28.80
RF00032	1.12	70377	1779	227712	30.9	5151	40	39.5655	30.91	0.04	4.64
RF00033	44.21	1954810	69346	8876288	22.02	59154	462	28.1894	22.02	1.41	180.85
RF00034	82.33	3181038	173868	22255104	14.29	127440	996	18.2958	14.29	3.54	453.45
RF00035	81.98	3165067	144224	18460672	17.14	115750	904	21.9456	17.14	2.94	376.14
RF00037	1.69	96771	2037	260736	37.11	6900	54	47.5115	37.11	0.04	5.31
RF00038	100.41	4525738	51482	6589696	68.67	63573	497	87.9093	68.67	1.05	134.27
RF00039	7.99	420126	6992	894976	46.94	16455	129	60.0881	46.94	0.14	18.24
RF00041	118.65	4243415	177304	22694912	18.69	138759	1084	23.9330	18.69	3.61	462.41
RF00042	53.51	2170975	89823	11497344	18.88	73600	575	24.1696	18.88	1.83	234.26
RF00043	23.99	1046417	41608	5325824	19.64	42989	336	25.1497	19.64	0.85	108.51
RF00046	38.19	1917744	24660	3156480	60.75	34719	271	77.7678	60.75	0.50	64.31
RF00047	18.85	879727	11914	1524992	57.68	28770	225	73.8406	57.68	0.24	31.07
RF00048	11.1	576080	8940	1144320	50.34	19626	153	64.4396	50.34	0.18	23.32
RF00049	25.06	1292725	17351	2220928	58.2	28401	222	74.5050	58.2	0.35	45.25
RF00051	27.68	1260808	15564	1992192	63.28	35467	277	81.0087	63.28	0.32	40.59
RF00052	19.29	899466	11985	1534080	58.63	28241	221	75.0501	58.63	0.24	31.26
RF00053	34.6	1546318	19539	2500992	61.82	39511	309	79.1406	61.82	0.40	50.96
RF00054	29.91	1522615	20182	2583296	58.94	31151	243	75.4447	58.94	0.41	52.63
RF00055	82.26	3851677	47395	6066560	63.49	54703	427	81.2678	63.49	0.97	123.61
RF00056	131.45	4953865	177817	22760576	21.76	123040	961	27.8594	21.76	3.62	463.75
RF00057	31.56	1432755	52780	6755840	21.2	47133	368	27.1460	21.2	1.08	137.65
RF00060	104.3	4593080	51728	6621184	69.36	66450	519	88.7931	69.36	1.05	134.91
RF00063	41.71	1934547	24202	3097856	62.44	39436	308	79.9338	62.44	0.49	63.12
RF00065	92.94	4109991	46780	5987840	68.63	60977	476	87.8581	68.63	0.95	122.00
RF00066	9.42	495049	8227	1053056	47.01	17500	137	60.1749	47.01	0.17	21.46
RF00068	36.98	1865044	24056	3079168	60.56	34239	267	77.5297	60.57	0.49	62.74
RF00069	39.84	1976008	25012	3201536	61.72	36092	282	79.0028	61.72	0.51	65.23
RF00070	37.52	1852791	24611	3150208	58.81	33403	261	75.2835	58.81	0.50	64.19

Table D.8: Results for processor array architecture using 128 processors

CM	Time	Num Computations	Cycles To Compute	Total Cycles	Schedule Efficiency (%)	Max Live Memory	Average Memory	Speedup	Processor Efficiency (%)	Schedule Size/Proc(MB)	Total Schedule Size (MB)
RF00001	96.2	3539545	113561	29071616	12.17	100775	394	31.1688	12.17	2.40	614.14
RF00005	57.82	2093665	121052	30989312	6.75	91483	357	17.2957	6.75	2.56	654.65
RF00006	112.76	4758405	38294	9803264	48.53	77749	304	124.2601	48.53	0.81	207.09
RF00008	23.56	984853	49374	12639744	7.79	41600	163	19.9470	7.79	1.04	267.01
RF00014	40.7	1568965	65815	16848640	9.31	73295	286	23.8392	9.31	1.39	355.93
RF00015	273.44	7138305	344335	88149760	8.09	245910	961	20.7307	8.09	7.27	1862.16
RF00016	123.49	5484002	41627	10656512	51.46	76988	301	131.7417	51.46	0.88	225.12
RF00019	65.38	2987475	25032	6408192	46.61	58139	227	119.3466	46.61	0.53	135.37
RF00021	98.99	3639478	152213	38966528	9.34	127514	498	23.9105	9.34	3.22	823.17
RF00026	157.81	6865523	53046	13579776	50.55	83257	325	129.4260	50.55	1.12	286.87
RF00027	32.05	1404743	11225	2873600	48.88	45115	176	125.1450	48.88	0.24	60.70
RF00031	17.02	788082	7953	2035968	38.7	29327	115	99.0937	38.7	0.17	43.01
RF00032	1.16	70377	1779	455424	15.45	5151	20	39.5655	15.45	0.04	9.62
RF00033	45.73	1954810	64462	16502272	11.84	65644	256	30.3252	11.84	1.36	348.61
RF00034	86.94	3181038	161144	41252864	7.71	131424	513	19.7404	7.71	3.40	871.47
RF00035	85.69	3165067	133655	34215680	9.25	118215	462	23.6810	9.25	2.82	722.81
RF00037	1.75	96771	2037	521472	18.55	6900	27	47.5115	18.55	0.04	11.02
RF00038	100.95	4525738	35563	9104128	49.71	71717	280	127.2600	49.71	0.75	192.32
RF00039	8.09	420126	5731	1467136	28.63	17578	69	73.3094	28.63	0.12	30.99
RF00041	122.13	4243415	155666	39850496	10.64	144368	564	27.2598	10.64	3.29	841.84
RF00042	55.51	2170975	82246	21054976	10.31	78330	306	26.3962	10.31	1.74	444.79
RF00043	24.63	1046417	36033	9224448	11.34	43776	171	29.0408	11.34	0.76	194.87
RF00046	38.67	1917744	18149	4646144	41.27	42210	165	105.6673	41.27	0.38	98.15
RF00047	19.15	879727	8631	2209536	39.81	32088	125	101.9276	39.81	0.18	46.68
RF00048	11.3	576080	7068	1809408	31.83	22113	86	81.5068	31.83	0.15	38.22
RF00049	25.36	1292725	13200	3379200	38.25	34430	134	97.9345	38.25	0.28	71.39
RF00051	27.64	1260808	10951	2803456	44.97	40577	159	115.1328	44.97	0.23	59.22
RF00052	19.41	899466	8772	2245632	40.05	32435	127	102.5394	40.05	0.19	47.44
RF00053	35.02	1546318	13885	3554560	43.5	46645	182	111.3669	43.5	0.29	75.09
RF00054	30.39	1522615	15156	3879936	39.24	38497	150	100.4635	39.24	0.32	81.96
RF00055	83.28	3851677	33298	8524288	45.18	61972	242	115.6732	45.18	0.70	180.08
RF00056	136.14	4953865	160055	40974080	12.09	131261	513	30.9511	12.09	3.38	865.58
RF00057	32.73	1432755	47551	12173056	11.76	51272	200	30.1311	11.76	1.00	257.16
RF00060	104.72	4593080	35457	9076992	50.6	74945	293	129.5397	50.6	0.75	191.75
RF00063	42.18	1934547	17306	4430336	43.66	46898	183	111.7854	43.66	0.37	93.59
RF00065	93.06	4109991	32200	8243200	49.85	69015	270	127.6398	49.85	0.68	174.14
RF00066	9.54	495049	6751	1728256	28.64	17735	69	73.3312	28.64	0.14	36.51
RF00068	37.24	1865044	17750	4544000	41.04	41667	163	105.0735	41.04	0.37	95.99
RF00069	40.21	1976008	18114	4637184	42.61	41556	162	109.0879	42.61	0.38	97.96
RF00070	37.8	1852791	17983	4603648	40.24	36952	144	103.0307	40.24	0.38	97.25

Table D.9: Results for processor array architecture using 256 processors

Appendix E

Additional Results for the Speedup of the Processors Array Architecture Over Infernal

This appendix contains additional results comparing the estimated performance of the processor array architecture to the INFERNAL (version 0.81) software package.

The evaluation system for the INFERNAL software contains dual Intel Xeon 2.8 GHz CPUs and 6 GBytes of DDR2 SDRAM running Linux CentOS 5.0. INFERNAL was run with the *–toponly* and *–noalign* options to ensure that INFERNAL was not doing more work than the processor array architecture. Results were collected for both the standard version of INFERNAL as well as INFERNAL using the Query Dependent Banding (QDB) heuristic [53]. The running time represents the time it took, in seconds, for INFERNAL to process a randomly generated database of 1 million residues.

As described in Section 7.6.5, the results for the processor array architecture are an estimate based on several factors including: the number of computations required to process a database of 1 million residues, the efficiency of the schedule, the number of processors available, and a clock frequency of 250 MHz.

Tables E.1 through E.9 compare the results for 40 different covariance models (CMs) from the Rfam 8.0 database [33] for a single processor up to 256 processors. In those tables, the columns are as follows:

- **CM -** the covariance model from the Rfam 8.0 database.
- **Total Computations -** the total number of computations required to compute the results for a database of 1 million residues.
- **Efficiency -** the efficiency of the schedule being used to process the 1 million residues.
- **Total Cycles -** the total number of cycles required to process 1 million residues, including idle time. This value is estimated as $\frac{TotalComputations}{Efficiency}$.
- **Schedule Length -** the length of the schedule after the computations have been divided up over p processors. This value is estimated as $\frac{TotalCycles}{p}$.
- **Time (seconds) -** the estimated time to process the schedule on a processor array running at 250 MHz. This value assumes that the I/O interface can provide p instructions to the processor array architecture on each clock cycle.
- **Infernal Time (seconds) -** the time required for INFERNAL to process 1 million residues.
- **Infernal Time (QDB) (seconds) -** the time required for INFERNAL to process 1 million residues when using the QDB heuristic.
- **Speedup over Infernal -** the estimated speedup of the processor array architecture over the INFERNAL software package.
- **Speedup over Infernal (w/QDB) -** the estimated speedup of the processor array architecture over the INFERNAL software package using the QDB heuristic.

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	99.99	52178485844	52178485844	208.71	3494.92	1284.43	16.75	6.15
RF00005	33258736474	99.99	33258895328	33258895328	133.04	2360.1	873.88	17.74	6.57
RF00006	51610209630	99.99	51610318091	51610318091	206.44	3306.08	2301.54	16.01	11.15
RF00008	19070963221	99.99	19071176228	19071176228	76.28	1341.33	851.32	17.58	11.16
RF00014	30832331395	99.99	30832547560	30832547560	123.33	2227.02	797.99	18.06	6.47
RF00015	84412280385	99.99	84412398638	84412398638	337.65	5647.19	1597.32	16.73	4.73
RF00016	62121487523	99.99	62121600801	62121600801	248.49	3360	1885.21	13.52	7.59
RF00019	42836989875	99.99	42837133264	42837133264	171.35	2546.75	1458.67	14.86	8.51
RF00021	54266150218	99.99	54266314233	54266314233	217.07	3793.47	1223.24	17.48	5.64
RF00026	65192108534	99.99	65192203490	65192203490	260.77	3041.66	1621.65	11.66	6.22
RF00027	26710573271	99.99	26710763417	26710763417	106.84	2007.35	1267.83	18.79	11.87
RF00031	17471198130	99.99	17471419823	17471419823	69.89	1239.39	745.92	17.73	10.67
RF00032	3401927493	99.98	3402410879	3402410879	13.61	218.21	149.62	16.03	10.99
RF00033	34594976155	99.99	34595153129	34595153129	138.38	2109.54	852.81	15.24	6.16
RF00034	50108566254	99.99	50108739529	50108739529	200.43	3148.36	875.2	15.71	4.37
RF00035	48979650328	99.99	48979820554	48979820554	195.92	3238.01	1061.47	16.53	5.42
RF00037	4277899983	99.98	4278342047	4278342047	17.11	307.98	220.66	18.00	12.89
RF00038	59731445866	99.99	59731577848	59731577848	238.93	3303.7	1385.39	13.83	5.80
RF00039	11606572815	99.99	11606849079	11606849079	46.43	746.98	484.91	16.09	10.44
RF00041	59389512497	99.99	59389652454	59389652454	237.56	3881.56	1186.92	16.34	5.00
RF00042	35994743344	99.99	35994909144	35994909144	143.98	2439.94	1003.69	16.95	6.97
RF00043	22722910361	99.99	22723149226	22723149226	90.89	1540.47	657.6	16.95	7.23
RF00046	33463069269	99.99	33463261210	33463261210	133.85	1696.87	833.52	12.68	6.23
RF00047	19709105827	99.99	19709329864	19709329864	78.84	1358.65	849.75	17.23	10.78
RF00048	14693415254	99.99	14693670313	14693670313	58.77	909.67	539.42	15.48	9.18
RF00049	24958696885	99.99	24958909262	24958909262	99.84	1305.38	687.81	13.08	6.89
RF00051	25147720759	99.99	25147940162	25147940162	100.59	1751.05	1009.4	17.41	10.03
RF00052	19928085927	99.99	19928307482	19928307482	79.71	1392.74	839.01	17.47	10.53
RF00053	28834432030	99.99	28834637149	28834637149	115.34	2057.87	1187.2	17.84	10.29
RF00054	28567465732	99.99	28567653353	28567653353	114.27	1478.89	798.55	12.94	6.99
RF00055	44630129995	99.99	44630245867	44630245867	178.52	2306.6	1391.26	12.92	7.79
RF00056	65402881033	99.99	65403013057	65403013057	261.61	4422.69	1555.48	16.91	5.95
RF00057	28050515347	99.99	28050730705	28050730705	112.20	1657.48	679.65	14.77	6.06
RF00060	60643374584	99.99	60643506616	60643506616	242.57	3537.84	1483.86	14.58	6.12
RF00063	33808046397	99.99	33808238632	33808238632	135.23	1977.71	959.06	14.62	7.09
RF00065	56497860699	99.99	56497998164	56497998164	225.99	3115.63	1328.44	13.79	5.88
RF00066	11729497999	99.99	11729734935	11729734935	46.92	736.31	528.69	15.69	11.27
RF00068	32830122196	99.99	32830315828	32830315828	131.32	1663.59	825.74	12.67	6.29
RF00069	30259011817	99.99	30259164949	30259164949	121.04	1563.22	907.67	12.92	7.50
RF00070	27124136529	99.99	27124282925	27124282925	108.50	1412.81	1009.89	13.02	9.31

Table E.1: Processor array architecture with 1 processor compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	97.22	53665744909	26832872454	107.33	3494.92	1284.43	32.56	11.97
RF00005	33258736474	96.92	34313289720	17156644860	68.63	2360.1	873.88	34.39	12.73
RF00006	51610209630	99.63	51797793195	25898896598	103.60	3306.08	2301.54	31.91	22.22
RF00008	19070963221	96.53	19754657639	9877328819	39.51	1341.33	851.32	33.95	21.55
RF00014	30832331395	97.4	31654722103	15827361051	63.31	2227.02	797.99	35.18	12.60
RF00015	84412280385	92.93	90827162206	45413581103	181.65	5647.19	1597.32	31.09	8.79
RF00016	62121487523	99.61	62359959638	31179979819	124.72	3360	1885.21	26.94	15.12
RF00019	42836989875	99.49	43056474826	21528237413	86.11	2546.75	1458.67	29.57	16.94
RF00021	54266150218	97.6	55599678656	27799839328	111.20	3793.47	1223.24	34.11	11.00
RF00026	65192108534	99.62	65437426789	32718713395	130.87	3041.66	1621.65	23.24	12.39
RF00027	26710573271	99.77	26769917720	13384958860	53.54	2007.35	1267.83	37.49	23.68
RF00031	17471198130	99.05	17637378935	8818689468	35.27	1239.39	745.92	35.14	21.15
RF00032	3401927493	98	3471100067	1735550034	6.94	218.21	149.62	31.43	21.55
RF00033	34594976155	97.04	35649844991	17824922495	71.30	2109.54	852.81	29.59	11.96
RF00034	50108566254	92.67	54071081024	27035540512	108.14	3148.36	875.2	29.11	8.09
RF00035	48979650328	94.41	51876909411	25938454706	103.75	3238.01	1061.47	31.21	10.23
RF00037	4277899983	97.95	4367241169	2183620584	8.73	307.98	220.66	35.26	25.26
RF00038	59731445866	99.68	59921578660	29960789330	119.84	3303.7	1385.39	27.57	11.56
RF00039	11606572815	99.47	11668234960	5834117480	23.34	746.98	484.91	32.01	20.78
RF00041	59389512497	98.29	60419889045	30209944523	120.84	3881.56	1186.92	32.12	9.82
RF00042	35994743344	97.3	36990487791	18495243896	73.98	2439.94	1003.69	32.98	13.57
RF00043	22722910361	96.67	23505322328	11752661164	47.01	1540.47	657.6	32.77	13.99
RF00046	33463069269	99.59	33600289658	16800144829	67.20	1696.87	833.52	25.25	12.40
RF00047	19709105827	99.44	19818637328	9909318664	39.64	1358.65	849.75	34.28	21.44
RF00048	14693415254	99.39	14782991831	7391495915	29.57	909.67	539.42	30.77	18.24
RF00049	24958696885	99.59	25061352431	12530676216	50.12	1305.38	687.81	26.04	13.72
RF00051	25147720759	99.55	25258977974	12629488987	50.52	1751.05	1009.4	34.66	19.98
RF00052	19928085927	99.23	20081135867	10040567934	40.16	1392.74	839.01	34.68	20.89
RF00053	28834432030	99.35	29022731005	14511365503	58.05	2057.87	1187.2	35.45	20.45
RF00054	28567465732	99.42	28731240165	14365620083	57.46	1478.89	798.55	25.74	13.90
RF00055	44630129995	99.61	44803092045	22401546023	89.61	2306.6	1391.26	25.74	15.53
RF00056	65402881033	97.16	67312277367	33656138683	134.62	4422.69	1555.48	32.85	11.55
RF00057	28050515347	96.82	28971681071	14485840536	57.94	1657.48	679.65	28.61	11.73
RF00060	60643374584	99.73	60806777443	30403388722	121.61	3537.84	1483.86	29.09	12.20
RF00063	33808046397	99.46	33989394329	16994697164	67.98	1977.71	959.06	29.09	14.11
RF00065	56497860699	99.63	56703686568	28351843284	113.41	3115.63	1328.44	27.47	11.71
RF00066	11729497999	99.11	11834105288	5917052644	23.67	736.31	528.69	31.11	22.34
RF00068	32830122196	99.6	32960277797	16480138899	65.92	1663.59	825.74	25.24	12.53
RF00069	30259011817	99.43	30429417141	15214708571	60.86	1563.22	907.67	25.69	14.91
RF00070	27124136529	99.63	27222939253	13611469626	54.45	1412.81	1009.89	25.95	18.55

Table E.2: Processor array architecture with 2 processors compared to Infernal

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	92.6	56343869668	14085967417	56.34	3494.92	1284.43	62.03	22.80
RF00005	33258736474	88.02	37783140822	9445785205	37.78	2360.1	873.88	62.46	23.13
RF00006	51610209630	99.11	52069857153	13017464288	52.07	3306.08	2301.54	63.49	44.20
RF00008	19070963221	88.88	21454918342	5363729586	21.45	1341.33	851.32	62.52	39.68
RF00014	30832331395	90.68	33998935525	8499733881	34.00	2227.02	797.99	65.50	23.47
RF00015	84412280385	80.91	104322214722	26080553680	104.32	5647.19	1597.32	54.13	15.31
RF00016	62121487523	99.02	62730264285	15682566071	62.73	3360	1885.21	53.56	30.05
RF00019	42836989875	98.69	43401955352	10850488838	43.40	2546.75	1458.67	58.68	33.61
RF00021	54266150218	90.66	59854307726	14963576932	59.85	3793.47	1223.24	63.38	20.44
RF00026	65192108534	98.91	65907173028	16476793257	65.91	3041.66	1621.65	46.15	24.61
RF00027	26710573271	99.05	26966452236	6741613059	26.97	2007.35	1267.83	74.44	47.02
RF00031	17471198130	98.15	17799613611	4449903403	17.80	1239.39	745.92	69.63	41.91
RF00032	3401927493	95.16	3574738082	893684520	3.57	218.21	149.62	61.04	41.85
RF00033	34594976155	91.57	37777492390	9444373098	37.78	2109.54	852.81	55.84	22.57
RF00034	50108566254	86.02	58249622411	14562405603	58.25	3148.36	875.2	54.05	15.02
RF00035	48979650328	85.84	57056169531	14264042383	57.06	3238.01	1061.47	56.75	18.60
RF00037	4277899983	95.6	4474397543	1118599386	4.47	307.98	220.66	68.83	49.32
RF00038	59731445866	99.02	60320638437	15080159609	60.32	3303.7	1385.39	54.77	22.97
RF00039	11606572815	98.18	11821009004	2955252251	11.82	746.98	484.91	63.19	41.02
RF00041	59389512497	92.36	64302068972	16075517243	64.30	3881.56	1186.92	60.36	18.46
RF00042	35994743344	92.09	39083678343	9770919586	39.08	2439.94	1003.69	62.43	25.68
RF00043	22722910361	91.72	24773736966	6193434241	24.77	1540.47	657.6	62.18	26.54
RF00046	33463069269	98.7	33901183399	8475295850	33.90	1696.87	833.52	50.05	24.59
RF00047	19709105827	98.52	20003422727	5000855682	20.00	1358.65	849.75	67.92	42.48
RF00048	14693415254	98.08	14980101109	3745025277	14.98	909.67	539.42	60.73	36.01
RF00049	24958696885	98.57	25318483627	6329620907	25.32	1305.38	687.81	51.56	27.17
RF00051	25147720759	98.57	25510493477	6377623369	25.51	1751.05	1009.4	68.64	39.57
RF00052	19928085927	98.35	20260639431	5065159858	20.26	1392.74	839.01	68.74	41.41
RF00053	28834432030	98.6	29243811679	7310952920	29.24	2057.87	1187.2	70.37	40.60
RF00054	28567465732	98.45	29015373515	7253843379	29.02	1478.89	798.55	50.97	27.52
RF00055	44630129995	98.77	45182039649	11295509912	45.18	2306.6	1391.26	51.05	30.79
RF00056	65402881033	92.13	70986530217	17746632554	70.99	4422.69	1555.48	62.30	21.91
RF00057	28050515347	91.45	30670074870	7667518717	30.67	1657.48	679.65	54.04	22.16
RF00060	60643374584	99.13	61173615277	15293403819	61.17	3537.84	1483.86	57.83	24.26
RF00063	33808046397	98.56	34299907006	8574976752	34.30	1977.71	959.06	57.66	27.96
RF00065	56497860699	99	57067363131	14266840783	57.07	3115.63	1328.44	54.60	23.28
RF00066	11729497999	97.53	12025454883	3006363721	12.03	736.31	528.69	61.23	43.96
RF00068	32830122196	98.71	33258575988	8314643997	33.26	1663.59	825.74	50.02	24.83
RF00069	30259011817	98.61	30683677565	7670919391	30.68	1563.22	907.67	50.95	29.58
RF00070	27124136529	98.54	27525042232	6881260558	27.53	1412.81	1009.89	51.33	36.69

Table E.3: Processor array architecture with 4 processors compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	83.75	62301161188	7787645148	31.15	3494.92	1284.43	112.19	41.23
RF00005	33258736474	74.98	44352712867	5544089108	22.18	2360.1	873.88	106.42	39.41
RF00006	51610209630	98.12	52597759281	6574719910	26.30	3306.08	2301.54	125.71	87.51
RF00008	19070963221	76.71	24859390061	3107423758	12.43	1341.33	851.32	107.91	68.49
RF00014	30832331395	79.71	38678479945	4834809993	19.34	2227.02	797.99	115.16	41.26
RF00015	84412280385	71.97	117274842768	14659355346	58.64	5647.19	1597.32	96.31	27.24
RF00016	62121487523	97.89	63456691218	7932086402	31.73	3360	1885.21	105.90	59.42
RF00019	42836989875	97.16	44086951437	5510868930	22.04	2546.75	1458.67	115.53	66.17
RF00021	54266150218	79.28	68447779926	8555972491	34.22	3793.47	1223.24	110.84	35.74
RF00026	65192108534	97.85	66623329514	8327916189	33.31	3041.66	1621.65	91.31	48.68
RF00027	26710573271	97.28	27456571593	3432071449	13.73	2007.35	1267.83	146.22	92.35
RF00031	17471198130	96.35	18131620513	2266452564	9.07	1239.39	745.92	136.71	82.28
RF00032	3401927493	90.39	3763452078	470431510	1.88	218.21	149.62	115.96	79.51
RF00033	34594976155	83.52	41417202222	5177150278	20.71	2109.54	852.81	101.87	41.18
RF00034	50108566254	71.22	70347995088	8793499386	35.17	3148.36	875.2	89.51	24.88
RF00035	48979650328	72.54	67517134770	8439641846	33.76	3238.01	1061.47	95.92	31.44
RF00037	4277899983	91.66	4667137558	583392195	2.33	307.98	220.66	131.98	94.56
RF00038	59731445866	97.74	61108780176	7638597522	30.55	3303.7	1385.39	108.13	45.34
RF00039	11606572815	95.74	12122910404	1515363800	6.06	746.98	484.91	123.23	80.00
RF00041	59389512497	82.18	72265895371	9033236921	36.13	3881.56	1186.92	107.42	32.85
RF00042	35994743344	82.45	43652195051	5456524381	21.83	2439.94	1003.69	111.79	45.99
RF00043	22722910361	82.02	27702564821	3462820603	13.85	1540.47	657.6	111.21	47.48
RF00046	33463069269	96.94	34518256366	4314782046	17.26	1696.87	833.52	98.32	48.29
RF00047	19709105827	96.42	20440742197	2555092775	10.22	1358.65	849.75	132.94	83.14
RF00048	14693415254	95.54	15379012744	1922376593	7.69	909.67	539.42	118.30	70.15
RF00049	24958696885	96.67	25817223157	3227152895	12.91	1305.38	687.81	101.12	53.28
RF00051	25147720759	96.81	25975547837	3246943480	12.99	1751.05	1009.4	134.82	77.72
RF00052	19928085927	96.27	20698342763	2587292845	10.35	1392.74	839.01	134.58	81.07
RF00053	28834432030	96.87	29764887790	3720610974	14.88	2057.87	1187.2	138.28	79.77
RF00054	28567465732	96.6	29572082758	3696510345	14.79	1478.89	798.55	100.02	54.01
RF00055	44630129995	97.08	45969459029	5746182379	22.98	2306.6	1391.26	100.35	60.53
RF00056	65402881033	84.4	77483164603	9685395575	38.74	4422.69	1555.48	114.16	40.15
RF00057	28050515347	83.21	33709759263	4213719908	16.85	1657.48	679.65	98.34	40.32
RF00060	60643374584	97.95	61912572227	7739071528	30.96	3537.84	1483.86	114.29	47.93
RF00063	33808046397	97.09	34819082525	4352385316	17.41	1977.71	959.06	113.60	55.09
RF00065	56497860699	97.69	57833701242	7229212655	28.92	3115.63	1328.44	107.74	45.94
RF00066	11729497999	94.45	12417915841	1552239480	6.21	736.31	528.69	118.59	85.15
RF00068	32830122196	96.97	33853834552	4231729319	16.93	1663.59	825.74	98.28	48.78
RF00069	30259011817	97	31193974745	3899246843	15.60	1563.22	907.67	100.23	58.20
RF00070	27124136529	96.95	27976762020	3497095252	13.99	1412.81	1009.89	101.00	72.19

Table E.4: Processor array architecture with 8 processors compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	71.4	73072036714	4567002295	18.27	3494.92	1284.43	191.31	70.31
RF00005	33258736474	56.86	58491367242	3655710453	14.62	2360.1	873.88	161.40	59.76
RF00006	51610209630	95.14	54243071599	3390191975	13.56	3306.08	2301.54	243.80	169.72
RF00008	19070963221	59.66	31964684347	1997792772	7.99	1341.33	851.32	167.85	106.53
RF00014	30832331395	63.91	48236126353	3014757897	12.06	2227.02	797.99	184.68	66.17
RF00015	84412280385	60.95	138481499146	8655093697	34.62	5647.19	1597.32	163.12	46.14
RF00016	62121487523	95.74	64884170889	4055260681	16.22	3360	1885.21	207.14	116.22
RF00019	42836989875	94.26	45442031192	2840126950	11.36	2546.75	1458.67	224.18	128.40
RF00021	54266150218	63.48	85476077445	5342254840	21.37	3793.47	1223.24	177.52	57.24
RF00026	65192108534	95.1	68545006589	4284062912	17.14	3041.66	1621.65	177.50	94.63
RF00027	26710573271	94.83	28165282359	1760330147	7.04	2007.35	1267.83	285.08	180.06
RF00031	17471198130	93.43	18698089554	1168630597	4.67	1239.39	745.92	265.14	159.57
RF00032	3401927493	80.88	4205847185	262865449	1.05	218.21	149.62	207.53	142.30
RF00033	34594976155	69.92	49476721511	3092295094	12.37	2109.54	852.81	170.55	68.95
RF00034	50108566254	53.09	94373229739	5898326859	23.59	3148.36	875.2	133.44	37.10
RF00035	48979650328	64.01	76517933799	4782370862	19.13	3238.01	1061.47	169.27	55.49
RF00037	4277899983	86.39	4951826937	309489184	1.24	307.98	220.66	248.78	178.25
RF00038	59731445866	95.27	62693193727	3918324608	15.67	3303.7	1385.39	210.79	88.39
RF00039	11606572815	91.19	12726713205	795419575	3.18	746.98	484.91	234.78	152.41
RF00041	59389512497	66.69	89048404795	5565525300	22.26	3881.56	1186.92	174.36	53.32
RF00042	35994743344	66.88	53812014877	3363250930	13.45	2439.94	1003.69	181.37	74.61
RF00043	22722910361	67.34	33740194176	2108762136	8.44	1540.47	657.6	182.63	77.96
RF00046	33463069269	93.48	35793721971	2237107623	8.95	1696.87	833.52	189.63	93.15
RF00047	19709105827	93	21191712927	1324482058	5.30	1358.65	849.75	256.45	160.39
RF00048	14693415254	91.1	16127864764	1007991548	4.03	909.67	539.42	225.61	133.79
RF00049	24958696885	92.97	26845439027	1677839939	6.71	1305.38	687.81	194.50	102.48
RF00051	25147720759	93.88	26785343986	1674083999	6.70	1751.05	1009.4	261.49	150.74
RF00052	19928085927	92.82	21469530129	1341845633	5.37	1392.74	839.01	259.48	156.32
RF00053	28834432030	94.15	30625192951	1914074559	7.66	2057.87	1187.2	268.78	155.06
RF00054	28567465732	93.16	30664187489	1916511718	7.67	1478.89	798.55	192.91	104.17
RF00055	44630129995	93.96	47497670918	2968604432	11.87	2306.6	1391.26	194.25	117.16
RF00056	65402881033	72.15	90645318406	5665332400	22.66	4422.69	1555.48	195.16	68.64
RF00057	28050515347	70.03	40050690483	2503168155	10.01	1657.48	679.65	165.54	67.88
RF00060	60643374584	95.61	63423335701	3963958481	15.86	3537.84	1483.86	223.13	93.58
RF00063	33808046397	93.94	35988153668	2249259604	9.00	1977.71	959.06	219.82	106.60
RF00065	56497860699	95.19	59352191108	3709511944	14.84	3115.63	1328.44	209.98	89.53
RF00066	11729497999	90.11	13015563463	813472716	3.25	736.31	528.69	226.29	162.48
RF00068	32830122196	93.6	35072797912	2192049870	8.77	1663.59	825.74	189.73	94.17
RF00069	30259011817	93.35	32411558149	2025722384	8.10	1563.22	907.67	192.92	112.02
RF00070	27124136529	92.8	29228155831	1826759739	7.31	1412.81	1009.89	193.35	138.21

Table E.5: Processor array architecture with 16 processors compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	53.75	97071025316	3033469541	12.13	3494.92	1284.43	288.03	105.85
RF00005	33258736474	38.57	86226538563	2694579330	10.78	2360.1	873.88	218.97	81.08
RF00006	51610209630	91.32	56510517095	1765953659	7.06	3306.08	2301.54	468.03	325.82
RF00008	19070963221	43.37	43967126038	1373972689	5.50	1341.33	851.32	244.06	154.90
RF00014	30832331395	46.29	66605295478	2081415484	8.33	2227.02	797.99	267.49	95.85
RF00015	84412280385	42.15	200236956597	6257404894	25.03	5647.19	1597.32	225.62	63.82
RF00016	62121487523	91.44	67929980421	2122811888	8.49	3360	1885.21	395.70	222.02
RF00019	42836989875	89.24	48001460592	1500045643	6.00	2546.75	1458.67	424.45	243.10
RF00021	54266150218	46.42	116887205896	3652725184	14.61	3793.47	1223.24	259.63	83.72
RF00026	65192108534	91.24	71444728189	2232647756	8.93	3041.66	1621.65	340.59	181.58
RF00027	26710573271	89.61	29807532074	931485377	3.73	2007.35	1267.83	538.75	340.27
RF00031	17471198130	87.29	20014766498	625461453	2.50	1239.39	745.92	495.39	298.15
RF00032	3401927493	69.33	4906563909	153330122	0.61	218.21	149.62	355.78	243.95
RF00033	34594976155	53.42	64749118389	2023409950	8.09	2109.54	852.81	260.64	105.37
RF00034	50108566254	40.92	122439740504	3826241891	15.30	3148.36	875.2	205.71	57.18
RF00035	48979650328	46.23	105942613480	3310706671	13.24	3238.01	1061.47	244.51	80.15
RF00037	4277899983	75.58	5659837044	176869908	0.71	307.98	220.66	435.32	311.90
RF00038	59731445866	90.44	66044472288	2063889759	8.26	3303.7	1385.39	400.18	167.81
RF00039	11606572815	82.92	13996201963	437381311	1.75	746.98	484.91	426.96	277.17
RF00041	59389512497	49.1	120945137629	3779535551	15.12	3881.56	1186.92	256.75	78.51
RF00042	35994743344	49.32	72973708042	2280428376	9.12	2439.94	1003.69	267.49	110.03
RF00043	22722910361	52.53	43252739370	1351648105	5.41	1540.47	657.6	284.92	121.63
RF00046	33463069269	87.27	38342419686	1198200615	4.79	1696.87	833.52	354.05	173.91
RF00047	19709105827	87.08	22633074886	707283590	2.83	1358.65	849.75	480.24	300.36
RF00048	14693415254	83.25	17649238242	551538695	2.21	909.67	539.42	412.33	244.51
RF00049	24958696885	85.79	29089689681	909052803	3.64	1305.38	687.81	358.99	189.16
RF00051	25147720759	88.4	28446104245	888940758	3.56	1751.05	1009.4	492.45	283.88
RF00052	19928085927	86.86	22941007377	716906481	2.87	1392.74	839.01	485.68	292.58
RF00053	28834432030	87.7	32876874259	1027402321	4.11	2057.87	1187.2	500.75	288.88
RF00054	28567465732	86.34	33086150373	1033942199	4.14	1478.89	798.55	357.59	193.08
RF00055	44630129995	88.03	50693326424	1584166451	6.34	2306.6	1391.26	364.01	219.56
RF00056	65402881033	54.55	119886615908	3746456747	14.99	4422.69	1555.48	295.12	103.80
RF00057	28050515347	53.14	52779853436	1649370420	6.60	1657.48	679.65	251.23	103.02
RF00060	60643374584	90.94	66678506554	2083703330	8.33	3537.84	1483.86	424.47	178.03
RF00063	33808046397	88	38415772816	1200492901	4.80	1977.71	959.06	411.85	199.72
RF00065	56497860699	90.48	62436238748	1951132461	7.80	3115.63	1328.44	399.21	170.21
RF00066	11729497999	79.76	14704443993	459513875	1.84	736.31	528.69	400.59	287.64
RF00068	32830122196	87.38	37567617101	1173988034	4.70	1663.59	825.74	354.26	175.84
RF00069	30259011817	86.55	34958205076	1092443909	4.37	1563.22	907.67	357.73	207.72
RF00070	27124136529	85.3	31797231604	993663488	3.97	1412.81	1009.89	355.45	254.08

Table E.6: Processor array architecture with 32 processors compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	36.85	141581036608	2212203697	8.85	3494.92	1284.43	394.96	145.15
RF00005	33258736474	23.95	138843098251	2169423410	8.68	2360.1	873.88	271.97	100.70
RF00006	51610209630	81.5	63320489247	989382644	3.96	3306.08	2301.54	835.39	581.56
RF00008	19070963221	28.2	67626860604	1056669697	4.23	1341.33	851.32	317.35	201.42
RF00014	30832331395	29.6	104144152447	1627252382	6.51	2227.02	797.99	342.14	122.60
RF00015	84412280385	26.56	317798538160	4965602159	19.86	5647.19	1597.32	284.32	80.42
RF00016	62121487523	82.66	75149663664	1174213495	4.70	3360	1885.21	715.37	401.38
RF00019	42836989875	80.5	53207499667	831367182	3.33	2546.75	1458.67	765.83	438.64
RF00021	54266150218	30.5	177882995833	2779421810	11.12	3793.47	1223.24	341.21	110.03
RF00026	65192108534	84.33	77302816077	1207856501	4.83	3041.66	1621.65	629.56	335.65
RF00027	26710573271	80.86	33031184906	516112264	2.06	2007.35	1267.83	972.34	614.13
RF00031	17471198130	77.83	22445936695	350717761	1.40	1239.39	745.92	883.47	531.71
RF00032	3401927493	49.22	6911263413	107988491	0.43	218.21	149.62	505.17	346.38
RF00033	34594976155	36.75	94117108501	1470579820	5.88	2109.54	852.81	358.62	144.98
RF00034	50108566254	25.17	199009881845	3109529404	12.44	3148.36	875.2	253.12	70.36
RF00035	48979650328	29.26	167386338276	2615411536	10.46	3238.01	1061.47	309.51	101.46
RF00037	4277899983	55.54	7701112939	120329890	0.48	307.98	220.66	639.87	458.45
RF00038	59731445866	82.11	72737315561	1136520556	4.55	3303.7	1385.39	726.71	304.74
RF00039	11606572815	66.59	17428065026	272313516	1.09	746.98	484.91	685.77	445.18
RF00041	59389512497	32.05	185255436066	2894616189	11.58	3881.56	1186.92	335.24	102.51
RF00042	35994743344	31.96	112594312774	1759286137	7.04	2439.94	1003.69	346.72	142.63
RF00043	22722910361	33.61	67588094587	1056063978	4.22	1540.47	657.6	364.67	155.67
RF00046	33463069269	76.62	43673773783	682402715	2.73	1696.87	833.52	621.65	305.36
RF00047	19709105827	73.83	26692259872	417066561	1.67	1358.65	849.75	814.41	509.36
RF00048	14693415254	70.79	20755647822	324306997	1.30	909.67	539.42	701.24	415.83
RF00049	24958696885	74.64	33436708978	522448578	2.09	1305.38	687.81	624.65	329.13
RF00051	25147720759	78.1	32199090535	503110790	2.01	1751.05	1009.4	870.11	501.58
RF00052	19928085927	74.56	26726932941	417608327	1.67	1392.74	839.01	833.76	502.27
RF00053	28834432030	77.32	37291923946	582686312	2.33	2057.87	1187.2	882.92	509.36
RF00054	28567465732	75.16	38005724956	593839452	2.38	1478.89	798.55	622.60	336.18
RF00055	44630129995	78.22	57056457379	891507147	3.57	2306.6	1391.26	646.83	390.14
RF00056	65402881033	36.35	179898153838	2810908654	11.24	4422.69	1555.48	393.35	138.34
RF00057	28050515347	35.51	78979951609	1234061744	4.94	1657.48	679.65	335.78	137.69
RF00060	60643374584	82.67	73348976725	1146077761	4.58	3537.84	1483.86	771.73	323.68
RF00063	33808046397	78.35	43145743505	674152242	2.70	1977.71	959.06	733.41	355.65
RF00065	56497860699	82.03	68869585879	1076087279	4.30	3115.63	1328.44	723.83	308.63
RF00066	11729497999	64.83	18090545203	282664769	1.13	736.31	528.69	651.22	467.59
RF00068	32830122196	76.49	42918325493	670598836	2.68	1663.59	825.74	620.19	307.84
RF00069	30259011817	76.83	39383108242	615361066	2.46	1563.22	907.67	635.08	368.76
RF00070	27124136529	73.43	36936788553	577137321	2.31	1412.81	1009.89	611.99	437.46

Table E.7: Processor array architecture with 64 processors compared to INFERNAL

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	22.72	229577406768	1793573490	7.17	3494.92	1284.43	487.14	179.03
RF00005	33258736474	13.34	249197166947	1946852867	7.79	2360.1	873.88	303.07	112.22
RF00006	51610209630	66.78	77272932065	603694782	2.41	3306.08	2301.54	1369.10	953.11
RF00008	19070963221	15.58	122379732278	956091658	3.82	1341.33	851.32	350.73	222.60
RF00014	30832331395	16.97	181678131990	1419360406	5.68	2227.02	797.99	392.26	140.55
RF00015	84412280385	15.27	552498640947	4316395632	17.27	5647.19	1597.32	327.08	92.51
RF00016	62121487523	70.57	88016555083	687629337	2.75	3360	1885.21	1221.59	685.40
RF00019	42836989875	65.36	65537543916	512012062	2.05	2546.75	1458.67	1243.50	712.22
RF00021	54266150218	17.46	310673060649	2427133286	9.71	3793.47	1223.24	390.74	126.00
RF00026	65192108534	69.36	93988293650	734283544	2.94	3041.66	1621.65	1035.59	552.12
RF00027	26710573271	66.97	39878861194	311553603	1.25	2007.35	1267.83	1610.76	1017.34
RF00031	17471198130	55.74	31339181399	244837355	0.98	1239.39	745.92	1265.52	761.65
RF00032	3401927493	30.9	11007285239	85994416	0.34	218.21	149.62	634.37	434.97
RF00033	34594976155	22.02	157086863534	1227241121	4.91	2109.54	852.81	429.73	173.73
RF00034	50108566254	14.29	350568384683	2738815505	10.96	3148.36	875.2	287.38	79.89
RF00035	48979650328	17.14	285680290300	2231877268	8.93	3238.01	1061.47	362.70	118.90
RF00037	4277899983	37.11	11526206508	90048488	0.36	307.98	220.66	855.04	612.61
RF00038	59731445866	68.67	86971908205	679468033	2.72	3303.7	1385.39	1215.55	509.73
RF00039	11606572815	46.94	24724973250	193163854	0.77	746.98	484.91	966.77	627.59
RF00041	59389512497	18.69	317630908088	2481491469	9.93	3881.56	1186.92	391.05	119.58
RF00042	35994743344	18.88	190625846183	1489264423	5.96	2439.94	1003.69	409.59	168.49
RF00043	22722910361	19.64	115650091073	903516337	3.61	1540.47	657.6	426.24	181.96
RF00046	33463069269	60.75	55078002531	430296895	1.72	1696.87	833.52	985.87	484.27
RF00047	19709105827	57.68	34165404396	266917222	1.07	1358.65	849.75	1272.54	795.89
RF00048	14693415254	50.34	29186864573	228022379	0.91	909.67	539.42	997.35	591.41
RF00049	24958696885	58.2	42879551920	334996499	1.34	1305.38	687.81	974.17	513.30
RF00051	25147720759	63.28	39735699737	310435154	1.24	1751.05	1009.4	1410.16	812.89
RF00052	19928085927	58.63	33988253096	265533227	1.06	1392.74	839.01	1311.27	789.93
RF00053	28834432030	61.82	46636386456	364346769	1.46	2057.87	1187.2	1412.03	814.61
RF00054	28567465732	58.94	48468076274	378656846	1.51	1478.89	798.55	976.41	527.23
RF00055	44630129995	63.49	70294409791	549175076	2.20	2306.6	1391.26	1050.03	633.34
RF00056	65402881033	21.76	300494108009	2347610219	9.39	4422.69	1555.48	470.98	165.65
RF00057	28050515347	21.2	132266014498	1033328238	4.13	1657.48	679.65	401.01	164.43
RF00060	60643374584	69.36	87420846469	682975363	2.73	3537.84	1483.86	1295.01	543.16
RF00063	33808046397	62.44	54137976167	422952939	1.69	1977.71	959.06	1168.99	566.88
RF00065	56497860699	68.63	82311652315	643059784	2.57	3115.63	1328.44	1211.25	516.45
RF00066	11729497999	47.01	24950698304	194927331	0.78	736.31	528.69	944.34	678.06
RF00068	32830122196	60.56	54202185955	423454578	1.69	1663.59	825.74	982.15	487.50
RF00069	30259011817	61.72	49025770977	383013836	1.53	1563.22	907.67	1020.34	592.45
RF00070	27124136529	58.81	46117814630	360295427	1.44	1412.81	1009.89	980.31	700.74

Table E.8: Processor array architecture with 128 processors compared to Infernal

CM	Total Computations	Efficiency	Total Cycles	Schedule Length	Time (seconds)	Infernal Time (seconds)	Infernal Time (QDB) (seconds)	Speedup over Infernal	Speedup over Infernal (w/QDB)
RF00001	52178338429	12.17	428560342735	1674063839	6.70	3494.92	1284.43	521.92	191.81
RF00005	33258736474	6.75	492278068038	1922961203	7.69	2360.1	873.88	306.83	113.61
RF00006	51610209630	48.53	106327332394	415341142	1.66	3306.08	2301.54	1989.98	1385.33
RF00008	19070963221	7.79	244759464556	956091658	3.82	1341.33	851.32	350.73	222.60
RF00014	30832331395	9.31	331099069791	1293355741	5.17	2227.02	797.99	430.47	154.25
RF00015	84412280385	8.09	1042393433314	4071849349	16.29	5647.19	1597.32	346.72	98.07
RF00016	62121487523	51.46	120714466779	471540886	1.89	3360	1885.21	1781.39	999.49
RF00019	42836989875	46.61	91886176728	358930378	1.44	2546.75	1458.67	1773.85	1015.98
RF00021	54266150218	9.34	581007348285	2269559954	9.08	3793.47	1223.24	417.86	134.74
RF00026	65192108534	50.55	128947820998	503702426	2.01	3041.66	1621.65	1509.65	804.87
RF00027	26710573271	48.88	54640246188	213438462	0.85	2007.35	1267.83	2351.20	1485.01
RF00031	17471198130	38.7	45135912652	176312159	0.71	1239.39	745.92	1757.38	1057.67
RF00032	3401927493	15.45	22014570479	85994416	0.34	218.21	149.62	634.37	434.97
RF00033	34594976155	11.84	292046647164	1140807215	4.56	2109.54	852.81	462.29	186.89
RF00034	50108566254	7.71	649826210473	2538383635	10.15	3148.36	875.2	310.08	86.20
RF00035	48979650328	9.25	529490226316	2068321197	8.27	3238.01	1061.47	391.38	128.30
RF00037	4277899983	18.55	23052413016	90048488	0.36	307.98	220.66	855.04	612.61
RF00038	59731445866	49.71	120157801620	469366413	1.88	3303.7	1385.39	1759.66	737.90
RF00039	11606572815	28.63	40531699570	158326951	0.63	746.98	484.91	1179.49	765.68
RF00041	59389512497	10.64	557735109624	2178652772	8.71	3881.56	1186.92	445.41	136.20
RF00042	35994743344	10.31	349091287202	1363637841	5.45	2439.94	1003.69	447.32	184.01
RF00043	22722910361	11.34	200308581602	782455397	3.13	1540.47	657.6	492.19	210.11
RF00046	33463069269	41.27	81071424813	316685253	1.27	1696.87	833.52	1339.56	658.00
RF00047	19709105827	39.81	49501696381	193366001	0.77	1358.65	849.75	1756.58	1098.63
RF00048	14693415254	31.83	46150505325	180275411	0.72	909.67	539.42	1261.50	748.05
RF00049	24958696885	38.25	65242358981	254852965	1.02	1305.38	687.81	1280.52	674.71
RF00051	25147720759	44.97	55916942665	218425557	0.87	1751.05	1009.4	2004.17	1155.31
RF00052	19928085927	40.05	49753017297	194347724	0.78	1392.74	839.01	1791.56	1079.26
RF00053	28834432030	43.5	66282432667	258915753	1.04	2057.87	1187.2	1987.01	1146.32
RF00054	28567465732	39.24	72795774849	284358496	1.14	1478.89	798.55	1300.20	702.06
RF00055	44630129995	45.18	98772581801	385830398	1.54	2306.6	1391.26	1494.57	901.47
RF00056	65402881033	12.09	540955976733	2113109284	8.45	4422.69	1555.48	523.24	184.03
RF00057	28050515347	11.76	238324412860	930954738	3.72	1657.48	679.65	445.10	182.51
RF00060	60643374584	50.6	119845381738	468146022	1.87	3537.84	1483.86	1889.28	792.41
RF00063	33808046397	43.66	77424329852	302438788	1.21	1977.71	959.06	1634.80	792.77
RF00065	56497860699	49.85	113314886897	442636277	1.77	3115.63	1328.44	1759.70	750.30
RF00066	11729497999	28.64	40948623861	159955562	0.64	736.31	528.69	1150.80	826.31
RF00068	32830122196	41.04	79987429390	312450896	1.25	1663.59	825.74	1331.08	660.70
RF00069	30259011817	42.61	71010140371	277383361	1.11	1563.22	907.67	1408.90	818.06
RF00070	27124136529	40.24	67395608508	263264096	1.05	1412.81	1009.89	1341.63	959.01

Table E.9: Processor array architecture with 256 processors compared to Infernal

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools.* Addison-Wesley, 1986.

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] Srinivas Aluru, Natsuhiko Fuamura, and Kishan Mehrotra. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing*, 63(3):264–272, 2003.

[4] Antonio Carzaniga and Alexander L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, AZ, October 2001.

[5] Antonio Carzaniga and David S. Rosenblum and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[6] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[7] Zachary Baker, Hong-Jip Jung, and Viktor K. Prasanna. Regular Expression Software Deceleration for Intrusion Detection Systems. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL)*, Madrid, Spain, August 2006.

[8] Zachary K. Baker and Viktor K. Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[9] K. E. Batcher. Sorting Networks and their Applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[10] Joao Bispo, Ioannis Sourdis, Joao M.P. Cardoso, and Stamatis Vassiliadis. Regular Expression Matching for Reconfigurable Packet Inspection. In *Proceedings of International Conference on Field Programmable Technology (FPT)*, Bangkok, Thailand, December 2006.

[11] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware. *IEEE Micro*, Volume 22(Number 3):66–74, February 2002.

[12] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Michael P. S. Brown. Small Subunit Ribosomal RNA Modeling Using Stochastic Context-Free Grammars. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, pages 57–66. AAAI Press, 2000.

[14] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A Routing Scheme for Content-Based Networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[15] Young H. Cho and William H. Mangione-Smith. High-Performance Context-Free Parser for Polymorphic Malware Detection. In *Advanced Networking and Communications Hardware Workshop*, Madison, WI, June 2005. Lecture Notes in Computer Science (LNCS).

[16] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *12th Conference on Field Programmable Logic and Applications*, pages 452–461, Montpellier, France, September 2002. Springer-Verlag.

[17] Chu-Sing Yang and Mon-Yen Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of USENIX Symposium on Internet Technologies & Systems (USITS)*, Boulder, CO, October 1999.

[18] Cristian Ciressan, Eduardo Sanchez, Martin Rajman, and Jean-Cédric Chappelier. An FPGA-Based Coprocessor for the Parsing of Context-Free Grammars. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 2000.

[19] Cristian Ciressan, Eduardo Sanchez, Martin Rajman, and Jean-Cédric Chappelier. An FPGA-Based Syntactic Parser for Real-Life Unrestricted Context-Free Grammars. *Lecture Notes in Computer Science*, 2147:590, 2001.

[20] Christopher R. Clark and David E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 956–959, Lisbon, Portugal, 2003.

[21] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes.* Courant Institute of Mathematical Sciences, New York University, 1969.

[22] E.G. Coffman and R.L. Graham. Optimal Scheduling for Two-Processor Systems. *Acta Informatica*, 1(3):200–213, 1972.

[23] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[24] Sean R. Eddy. Computational Analysis of RNAs. *Cold Spring Harbor Symposia on Quantitative Biology*, 71(1):117–128, 2006.

[25] Sean R. Eddy and Richard Durbin. RNA Sequence Analysis Using Covariance Models. *Nucleic Acids Research*, 22(11):2079–2088, April 1994.

[26] Stephen G. Eick, John W. Lockwood, James Moscola, Chip Kastner, Andrew Levine, Mike Attig, Ron Loui, and Doyle J. Weishar. Transformation Algorithms for Data Streams. In *Proceedings of IEEE Aerospace Conference*, Big Sky, MT, USA, March 2005.

[27] P.C. Fishburn. *Interval Orders and Interval Graphs.* John Wiley & Sons, New York, 1985.

[28] R. Franklin, D. Carver, and B. L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *IEEE Symposium on Field-programmable Custom Computing Machines*, Napa Valley, CA, April 2002. IEEE.

[29] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, November 1996.

[30] N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, November 1996.

[31] G. D. Ritchie and F. K. Hanna. Semantic Networks - A General Definition and a Survey. *Information Technology: Research and Development*, 2(4):187–231, 1983.

[32] L. Rodney Goke and G. J. Lipovski. Banyan Networks for Partitioning Multiprocessor Systems. In *ISCA '73: Proceedings of the 1st Annual Symposium on Computer Architecture*, pages 21–28, New York, NY, USA, 1973.

[33] Sam Griffiths-Jones, Simon Moxon, Mhairi Marshall, Ajay Khanna, Sean R. Eddy, and Alex Bateman. Rfam: Annotating Non-Coding RNAs in Complete Genomes. *Nucleic Acids Research*, 33, 2005.

[34] H. Ahmadi and W. Denzel. A Survey of Modern High-Performance Switching Techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, 1989.

[35] HMMER Website. http://hmmer.janelia.org [July 2007].

[36] Te C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(6):841–848, 1961.

[37] Infernal Website. http://infernal.janelia.org [October 2007].

[38] T. Kasami. An Efficient Recognition and Syntax Algortihm for Context-Free Languages. Technical Report Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford MA, 1965.

[39] Charles M. Kastner. HAIL: An Algorithm for the Hardware-Accelerated Identification of Languages. Master's thesis, Washington University, St. Louis, MO, USA, May 2006.

[40] Chip Kastner, Adam Covington, Andrew Levine, and John Lockwood. HAIL: A Hardware-Accelerated Algorithm for Language Identification. In *Proceedings of 15th International Conference on Field-Programmable Logic and Applications (FPL)*, Tampere, Finland, August 2005.

[41] Kimberly C. Claffy and George C. Polyzos and Hans-Werner Braun. Application of Sampling Methodologies to Network Traffic Characterization. In *Proceedings of SIGCOMM*, pages 194–203, 1993.

[42] Andreas Koulouris, Nectarios Koziris, Theodore Andronokos, George Papakonstantinou, and Panayotis Tsanakas. A Parallel Parsing VLSI Architecture for Arbitrary Context Free Grammars. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, Tainan, Taiwan, December 1998.

[43] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[44] Hans-Peter Lenhof, Knut Reinert, and Martin Vingron. A Polyhedral Approach to RNA Sequence Structure Alignment. *Journal of Computational Biology*, 5(3):517–530, 1998.

[45] Tong Liu and Bertil Schmidt. Parallel RNA Secondary Structure Prediction Using Stochastic Context-Free Grammars. *Concurrency and Computation: Practice & Experience*, 17(14):1669–1685, 2005.

[46] John W Lockwood. An open platform for development of network processing modules in reprogrammable hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, CA, January 2001.

[47] John W. Lockwood, Stephen G. Eick, Justin Mauger, John Byrnes, Ron Loui, Andrew Levine, Doyle J. Weishar, and Alan Ratner. Hardware accelerated algorithms for semantic processing of document streams. In *IEEE Aerospace Conference (Aero'06)*, page 10.0802, Big Sky, MT, March 2006.

[48] T.M. Lowe and Sean R. Eddy. tRNAscan-SE: a program for improved detection of transfer RNA genes in genomic sequence. *Nucleic Acids Research*, 25(5):955–964, March 1997.

[49] John S. Mattick. Challenging the dogma: the hidden layer of non-protein-coding RNAs in complex organisms. *BioEssays*, 25:930–939, 2003.

[50] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2003. IEEE.

[51] James Moscola, Young H. Cho, and John W. Lockwood. A Scalable Hybrid Regular Expression Pattern Matcher. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, April 2006.

[52] James Moscola, Young H. Cho, and John W. Lockwood. Reconfigurable Context-Free Grammar based Data Processing Hardware with Error Recovery. In *Proceedings of International Parallel & Distributed Processing Symposium (IPDPS/RAW)*, Rhodes Island, Greece, April 2006.

[53] Eric P. Nawrocki and Sean R. Eddy. Query-Dependent Banding (QDB) for Faster RNA Similarity Searches. *PLoS Computational Biology*, 3(3), 2007.

[54] Val Oliva. Traffic Monitoring for High-Performance Computing: Unlocking the Knowledge Within Your Network. Technical report, Foundry Networks, January 2003.

[55] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Science of the U.S.A.*, 85(8):2444–2448, April 1988.

[56] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2001. IEEE.

[57] P. Resnick. RFC 2822: Internet Message Format, April 2001.

[58] Elena Rivas and Sean R. Eddy. Noncoding RNA gene detection using comparative sequence analysis. *BMC Bioinformatics*, 2(1), October 2001.

[59] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA 1999 conference*, http://www.snort.org/, November 1999. USENIX.

[60] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 344–360. Springer–Verlag, 1997.

[61] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, number 2820 in Lecture Notes in Computer Science, pages 155–172, 2003.

[62] Bertil Schmidt, Heiko Schroder, and Manfred Schimmler. Massively Parallel Solutions for Molecular Sequence Analysis. In *Proceedings of the IPDPS'02 1st International Workshop on High Performance Computational Biology*, Alamitos, CA, 2002.

[63] David Schuehler and John Lockwood. A Modular System for FPGA-based TCP Flow Processing in High-Speed Networks. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 301–310, Antwerp, Belgium, August 2004.

[64] David Searls. The Linguistics of DNA. *American Scientist*, 80:579–591, 1992.

[65] J. Brian Sharkey, Doyle Weishar, John W. Lockwood, Ron Loui, Richard Rohwer, John Byrnes, Krishna Pattipati, David Cousins, Michael Nicolletti, and Stephen Eick. Information processing at very high-speed data ingestion rates. In Robert Popp and John Yin, editors, *Emergent Information Technologies and Enabling Policies for Counter Terrorism*, pages 75–104. IEEE Press/Wiley, 2006. ISBN: 0-471-77615-7.

[66] Robert Shiveley. Dual-Core Intel Itanium 2 Processors Deliver Unbeatable Flexibility and Performance to the Enterprose. *Technology@Intel Magazine*, August 2006.

[67] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In *13th Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003. Springer-Verlag.

[68] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A Survey. *IEEE Computer*, 27(6):17–26, 1994.

[69] Gisela Storz. An Expanding Universe of Noncoding RNAs. *Science*, 296(5571):1260–1263, May 2002.

[70] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, Queensland, Australia, September 2007.

[71] S. Washietl, I. L. Hofacker, M. Lukasser, A. Huttenhofer, and P. F. Stadler. Mapping of conserved RNA secondary structures predicts thousands of functional noncoding RNAs in the human genome. *Nature Biotechnology*, 23:1390–1390, 2003.

[72] Katsuhiro Watanabe, Nobuhiko Tsuruoka, and Ryutaro Himeno. Performance of Network Intrusion Detection Cluster System. In *Proceedings of High Performance Computing*, number 2858 in Lecture Notes in Computer Science, pages 278–287, 2003.

[73] Zasha Weinberg and Walter L. Ruzzo. Faster Genome Annotation of Non-coding RNA Families without Loss of Accuracy. In *Eighth Annual International Conference on Research in Computational Molecular Biology (RECOMB 2004)*, pages 243–251, San Diego, CA, March 2004.

[74] Zasha Weinberg and Walter L. Ruzzo. Sequence-based heuristics for faster annotation of non-coding RNA families. *Bioinformatics*, 22(1):35–39, 2006.

[75] D.H. Younger. Recognition and Parsing of Context-Free Languages in Time O(n3). *Information and Control*, 10(2):189–208, 1967.

Vita

James M. Moscola

Education

Ph.D. Computer Engineering, Washington University, May 2008
M.S. Computer Science, Washington University, August 2003
B.S. Computer Engineering, Washington University, May 2001
B.S. Physical Science, Muhlenberg College, May 2000

Professional Societies

Institute of Electrical and Electronics Engineers
Eta Kappa Nu Electrical and Computer Engr. Honor Society
Golden Key International Honor Society

Conference Papers

Phillip H. Jones, James Moscola, Young H. Cho, John W. Lockwood. Adaptive Thermoregulation for Applications on Reconfigurable Devices. *Proceedings of Field Programmable Logic and Applications (FPL)*, (Amsterdam, Netherlands), August 27-29 2007.

James Moscola, Young H. Cho, John W. Lockwood. Hardware-Accelerated Parser for Extraction of Metadata in Semantic Network Content. *Proceedings of IEEE Aerospace Conference*, (Big Sky, MT), March 3-10, 2007.

James Moscola, Young H. Cho, John W. Lockwood. A Reconfigurable Architecture for Multi-Gigabit Speed Content-Based Routing. *Proceedings of Hot Interconnects 14 (HotI)*, (Stanford, CA), August 23-25, 2006.

Young H. Cho, James Moscola, John W. Lockwood. Context-Free Grammar based Token Tagger in Reconfigurable Devices. *Proceedings of International Workshop on Data Engineering (ICDE/SeNS)*, (Atlanta, GA), April 3-7, 2006.

Stephen G. Eick, John W. Lockwood, James Moscola, Chip Kastner, Andrew Levine, Mike Attig, Ron Loui, Doyle J. Weishar. Transformation Algorithms for Data Streams. *Proceedings of IEEE Aerospace Conference*, (Big Sky, MT), March 5-12, 2005.

John W. Lockwood, James Moscola, David Reddick, Matthew Kulig, Tim Brooks. Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection. *Proceedings of International Working Conference on Active Networks (IWAN)*, (Kyoto, Japan), December, 2003.

John W. Lockwood, James Moscola, Matthew Kulig, David Reddick, Tim Brooks. Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. *Proceedings of Military and Aerospace Programmable Logic Devices (MAPLD)*, Paper E10, (Washington D.C.), September 9-11, 2003.

John Lockwood, Chris Neely, Chris Zuver, Dave Lim, James Moscola. An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall. *Proceedings of Field-Programmable Logic and Applications (FPL)*, Paper 14B, (Lisbon, Portugal), September 1-3, 2003.

David V. Schuehler, James Moscola, John Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System. *Proceeding of Hot Interconnects 11 (HotI-11)*, pp. 89-94, (Stanford, CA, USA), August 2003.

James Moscola, Michael Pachos, John Lockwood, Ronald P. Loui. FPsed: A Streaming Content Search-and-Replace Module for an Internet Firewall.

Proceeding of Hot Interconnects 11 (HotI-11), pp.122-129, (Stanford, CA, USA), August 2003.

James Moscola, John Lockwood, Ronald P. Loui, Michael Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31-38, (Napa, CA, USA), April 9-11, 2003.

Journal Papers

James Moscola, Young H. Cho, John W. Lockwood. Reconfigurable Content-based Router using Hardware-Accelerated Language Parser. *Transactions on Design Automation of Electronic Systems (TODAES)*, Volume 13, Number 1, April 2008.

David V. Schuehler, James Moscola, John W. Lockwood. Architecture for a Hardware-Based, TCP/IP Content-Processing System. *IEEE Micro*, Volume 24, Number 1, January/February 2004, pp. 62-69.

Short Papers

Phillip H. Jones, James Moscola, Young H. Cho, John W. Lockwood. Changing Output Quality for Thermal Management. *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA, USA), April 23-25 2007.

James Moscola, Young H. Cho, John W. Lockwood. Implementation of Network Application Layer Parser for Multiple TCP/IP Flows in Reconfigurable Devices. *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, (Madrid, Spain), August 28-30, 2006.

James Moscola, Young H. Cho, John W. Lockwood. A Scalable Hybrid Regular Expression Pattern Matcher. *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA), April 24-26, 2006.

James Moscola, Young H. Cho, John W. Lockwood. Reconfigurable Context-Free Grammar based Data Processing Hardware with Error Recovery. *Proceedings of International Parallel & Distributed Processing Symposium (IPDPS/RAW)*, (Rhodes Island, Greece), April 25-26, 2006.

Young H. Cho, James Moscola, John W. Lockwood. Context-Free Grammar based Token Tagger in Reconfigurable Devices. *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA)*, (Monterey, CA), February 22-24, 2006.

Haoyu Song, Jing Lu, John Lockwood, James Moscola. Secure Remote Control of Field-programmable Network Devices. *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA), April 20-23, 2004.

Technical Reports

James Moscola. FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet. *Master's Thesis*, Washington University, August 2003.

Patents

John Lockwood, Ronald Loui, James Moscola, Michael Pachos. Methods, Systems, and Devices Using Reprogrammable Hardware for High-Speed Processing of Streaming Data to Find a Redefinable Pattern and Respond Thereto. Issued August 2006.

May 2008

Hardware-Accelerated Parsing, Moscola, Ph.D. 2008