UNIVERSITY OF CALIFORNIA
RIVERSIDE

Acceleration of Streaming Applications on FPGAs from High Level Constructs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Abhishek Mitra

December 2008

Dissertation Committee:
Dr. Walid Najjar, Chairperson
Dr. Laxmi Bhuyan
Dr. Vassilis Tsotras

The Dissertation of Abhishek Mitra is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

Acknowledgements

PhD. is an interesting journey, chock full of twists, turns, crossroads, and unchartered territories, and I would like to profusely thank my advisor Prof. Walid Najjar for guiding me through and for ensuring my reaching of the final destination. I am also extremely grateful to Prof. Laxmi Bhuyan for his able guidance, help and encouragement. I am also thankful to Prof. Vasillis Tsotras for his guidance during the last leg of my research. Finally I am thankful to Prof. Satish Tripathi for providing me an opportunity to study at UC Riverside.

Over the course of last five years I have had the honor of working with excellent colleagues, to whom I am grateful for their constructive criticisms and suggestions. I would like to thank my colleagues (Anirban Banerjee, John Cortes, Jason Villarreal, Petko Bakalov, Marcos Vieira, David Sheldon, Edward Fernandez, Scott Sirowy, Joon Lee, Thomas Repantis, Jingnan Yao, and Roger Moussali).

I am grateful and thankful to my parents Anjusree Mitra and Vishwajit Mitra and my sister Ahana Mitra for their constant encouragement. Moreover I am extremely thankful to Poulomi Dasgupta. She was a real catalyst whenever the going got slow or things didnt work out that well. I am grateful to my best friend Dr. Karuppiah Ramkumar for his constant encouragement.

Due thanks go to Terri, Amy, Madie, Vanoohi from CSE Deparment and Kelly and Deja from IEC for helping me out with administrative tasks.

My classmates from college Kumar Kartikeya, Ashish Gupta, Nitin Agarwal, Subhek Garg, Sampan Arora, Sheetendu M. Mani and Abhay P. Singh were always there with a

helping hand, and I would like to thank them for their help and motivation.

Finally I would also like to thank my current and past room mates Shalendra Chhabra, Varun Kohli, Arinder Arora and Anirban Banerjee for all their help outside of the school.

ABSTRACT OF THE DISSERTATION

Acceleration of Streaming Applications on FPGAs from High Level Constructs

by

Abhishek Mitra

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2008
Dr. Walid Najjar, Chairperson

Field Programmable Gate Arrays (FPGA) based Reconfigurable Computing hardware are programmed with specialized Hardware Description Language (HDL). FPGAs are increasingly being made available as co-processors on high-performance computation systems. The generation of HDL from high-level software languages is way too complex for a human developer to handle in a reasonable amount of time due to incompatibilities in the execution paradigm between a traditional CPU and on an FPGA. This error prone process manifests itself as the main impediment to a wider use of reconfigurable platforms in high-performance computing. Compilation frameworks are thus a valuable tool for translating traditional high-level software constructs to HDL for implementation on FPGAs.

This dissertation details how we leverage FPGAs for accelerating PERL Compatible Regular Expressions (PCRE), SNORT Intrusion Detection System (IDS), Common Processing

Functions, and XML Filtering, by compiling high-level software language to HDL.

In this dissertation, we detail the implementation of a tool that translates PCRE code into hardware that is mapped to an FPGA. Our compiler generates VHDL code corresponding to the opcodes generated from regular expressions. We have tuned our hardware implementation to utilize an NFA based regular expression engine using greedy quantifiers in much the same way as the software based PCRE engine does.

The SNORT IDS system, incorporates the software based PCRE engine for regular expression matching on the payload. We benchmark the performance of regular expression based rules from SNORT IDS using software only execution on a multi-processor system. We demonstrate the case when 30% or more number of network packets trigger regular expression matching, the software based IDS cannot maintain 10 Gbps throughout, and thus requires hardware acceleration.

Using our PCRE to HDL compilation system, we implement regular expressions from the SNORT ruleset on to the FPGA. These rulesets are organized into one of 16 banks on the FPGA and all operate in parallel. We have implemented more than two hundred PCRE engines based on a plethora of SNORT IDS regular expression rules. These were mapped to the Xilinx Virtex-4 LX200 FPGA on the SGI RASC RC 100 Blade connected to the SGI ALTIX 4700 supercomputing system as a testbed. We obtain an interface throughput of 12.9 GBits/s and a speedup of 353X over software based PCRE execution. We also show that it is possible to scale down the processing related power consumption of an IDS by two orders of magnitude using an FPGA .

In this dissertation we describe software tools as well as an IDS architecture that leverages reprogrammability of FPGA hardware. Our software tools for Configurable System on a Chip (CSoCs) generates the communication interface between the software running on the CPU and a tightly coupled IP core based co-processing system. Our tool generates hardware wrappers for the IP Cores that makes them look like a C function invocation in the source code. We also use our tool to support partial reconfiguration: the same wrapper is used for a multitude of IP Cores and the user selects the core to be invoked in the program.

We also demonstrate an adaptable regular expression based IDS using Virtex-4 LX 200 FPGAs that have been floor-planned for partial reconfiguration. Our novel design allows partial reprogramming across 16 banks of regular expression rule-sets. We implement 448 different regular expressions on two FPGAs and perform multiple partial and full reconfigurations. We measure the throughout of the integrated Field Programmable Gate Array (FPGA) and multiprocessor SGI Altix system with varying number of reconfigurations per minute. The adaptive IDS can provide better than 10 Gbps throughput even with 32 partial reconfigurations per minute.

In this dissertation we demonstrate a four step approach that converts user profiles expressed as XPath queries into HDL, suitable for implementation on FPGA. We convert XPaths to PCRE, cluster them by their common prefixes, compile the PCRE to HDL and finally synthesize and implement them on FPGA. This hardware is usable for XML filtering in pub-sub applications. Our benchmarks reveal orders of magnitude improvement in running time while running XML filtering on FPGA, when compared to the state of the art

software based XML filtering systems.

Finally, in this dissertation we demonstrate a FPGA based implementation of Prüfer sequence generation hardware for streaming XML document. We match the stream with several Prüfer sequence blocks obtained from twig queries.

Contents

List of Tables **xv**

List of Figures **xvi**

1 Introduction **1**

1.1 FPGAs for Code Acceleration 2

1.2 Regular Expression to HDL 6

1.3 FPGA Reprogrammability 8

1.4 XML Filtering on FPGA 10

1.5 Contributions 12

1.5.1 Compiling PCRE to FPGA and accelerating SNORT IDS 12

1.5.2 Dynamic Co-Processor Interface Automation 15

1.5.3 Adaptive Hardware/Software Regular Expression Based IDS 16

1.5.4 Boosting XML filtering with a scalable FPGA-based architecture . . 18

2 Related Work **21**

2.1 Network Intrusion Detection Systems 21
2.1.1 IDS engines with String Matching 22
2.1.2 IDS engines with Regular Expression Matching 24
2.2 Interfacing IP cores on FPGA 28
2.3 Partial Reconfiguration on FPGA 28
2.4 XML Filtering 34
2.4.1 Software Based Filtering 34
2.4.2 Hardware Based Filtering 36

3 Compiling PCRE to FPGA via opcodes and accelerating SNORT 38

3.1 Regular Expressions, IDS and FPGA Acceleration 38
3.1.1 PCRE 39
3.1.2 SNORT IDS and PCRE 39
3.1.3 Accelerating PCRE on FPGA 40
3.1.4 Finite Automaton on FPGA 42
3.2 SNORT IDS 42
3.2.1 PCRE rules in SNORT 43
3.3 Compiling PERL Compatible Regular Expressions to FPGA 46
3.3.1 PCRE Opcodes 48
3.3.2 PCRE Opcode Frequencies in SNORT Rules 49
3.4 Compilation Flow 50

3.4.1 Compilation Overview 52
3.4.2 Common Prefix Optimization 54
3.4.3 Hardware Implementation of PCRE Opcodes 54
3.4.4 NFA Implementation on FPGA 65
3.5 Experimental Results 69
3.5.1 Software only performance with multi-cpu load balancing 70
3.5.2 Hardware Benchmark and Comparison with Single Threaded Software Execution 75
3.5.3 Single Processor Power Consumption Analysis 80
3.6 Conclusion 82

4 Partial Reconfiguration on FPGA 84

4.1 Dynamic Co-Processor Interface Automation 85
4.2 System Overview for IP Core Wrapper Generation and Partial Reconfiguration 88
4.2.1 The CSoC platform 88
4.2.2 APU (Auxiliary Processing Unit) on Virtex-4 FX 89
4.2.3 IP Cores 90
4.2.4 ROCCC Overview 92
4.2.5 Interface Synthesis 93
4.2.6 Experimental Results 99
4.3 Adaptive Hardware/Software Regular Expression Based IDS 101

4.3.1 The FPGA Architecture 104
4.3.2 Xilinx Partial Reconfiguration Flow 104
4.3.3 The Hardware/Software Integrated Test System 105
4.3.4 Hardware Performance 107
4.3.5 Hardware/Software Performance with Reconfiguration 109
4.4 Conclusion 111

5 Boosting XML filtering with a scalable FPGA-based architecture 113

5.1 XML Pub-sub 113
5.1.1 Using FPGA for XML Filtering 116
5.2 Compilation System Overview 119
5.2.1 XPath Expressions 119
5.2.2 XPath on FPGA 120
5.2.3 Dictionary Replacement 121
5.2.4 XPath to Stack-enhanced Regular Expressions 122
5.2.5 Common Prefix Optimization 125
5.2.6 Area Efficient Character Decoder Hardware 126
5.2.7 Regular Expression to VHDL compilation 127
5.2.8 FPGA Implementation 128
5.3 Twig Profiles on FPGAs 130
5.3.1 Overview of Prüfer Sequences 131

5.3.2 FPGA implementation of Prüfer subsequence matching 133
5.4 Experimental Evaluation 136
5.4.1 Performance and Speedup 142
5.5 Conclusion 143

6 Conclusions **145**

6.1 PCRE to FPGA compiler 145
6.2 Accelerating regular expression of SNORT IDS 146
6.3 Dynamic Co-Processors on FPGA 147
6.4 Adaptive Hardware-Software Regular Expressions based IDS 148
6.5 Scalable Architecture for XML Filtering on FPGA 149

Bibliography **150**

List of Tables

3.1 Example Rules in SNORT DB 2.4 45

3.2 Format of a typical PCRE Rule in SNORT IDS with the optional modifiers . . 45

3.3 Example snippets from SNORT Rules highlighting the use of PCRE operators 46

3.4 Occurences of important PCRE operators in our target SNORT DB 2.4 48

3.5 Simple Quantifiers Occurrence Table 59

3.6 Ranged Quantifiers Occurrence Table 60

3.7 Unbounded Quantifiers Occurrence Table 61

4.1 Area Covered by the Dynamically instantiated IP Cores 100

4.2 The Area Covered by IP Wrappers and Wrapped IP Cores. 101

5.1 PCRE operators used for implementing XPath profiles on FPGA 122

List of Figures

1.1 Demonstration of the throughput of a CPU and FPGA: A Highway Analogy. An FPGA with a ten times slower clock rate can offer 10X the throughput of a dual core CPU by implementing two hundred parallel data paths on the fabric. 3

3.1 A Finite Automata Implemented on FPGA using LUTs. 43

3.2 SNORT IDS and PCRE Engine usage on a software only Implementation. pcre_compile function compiles the regular expression while pcre _execute function runs the engine of the payload 45

3.3 Frequency Distirbution of PCRE opcodes in SNORT DB 2.4. The most frequently occurring opcode is the character-match opcode. 50

3.4 Cumulative Distirbution of PCRE OPCODES in SNORT DB 2.4. The five OPCODES viz. Match, star, Character Class, Alternation and constrained repetition make up for the most frequently occurring cases of OPCODES. . . 51

3.5 The Character(s) Match Opcode. The input (i register) is compared with the data in the ROM and the output (match_out) is triggered high in case of a match. 55

3.6 Implementation Speed in (MHz) of the character-match opcode versus character size sets for the two implementation types. Implementing using IP Core provides a faster clock speed, as compared to the synthesized design for match sizes greater than 2 Bytes. 56

3.7 Implementation area in FPGA slices of the character-match opcode versus character size sets for the two implementation types. The area of the opcode block increases linearly with increasing match size. Moreover the synthesized hardware is more area efficient as compared to the IP Core based block for a given match size 56

3.8 The Character Class Match Opcode. 57

3.9 The * repetition Match Opcode. 57

3.10 The + repetition Match Opcode. 58

3.11 The quantifier Match Opcode. 59

3.12 Speed of unrestricted counters in (MHz) with increasing count size and the two implementation types. For counts until 8 bit the synthesized counters are faster, while for 9 and 10 bit counters, the IP Core is slightly faster. 61

3.13 Speed of restricted counters in (MHz) with increasing sizes and the two implementation types. The 4 bit and 6 bit restricted counters are faster when synthesized while the performance of the synthesized and IP core is comparable for higher count sizes 62

3.14 Variation of Area of un-restricted counters on FPGA with increasing count size and the two implementation types. The area of the counters implemented using IP Core is lesser when compared to the area of synthesized counters. . 63

3.15 Variation of Area of counters with restricted count on FPGA with increasing count size and the two implementation types. Restricting the count i.e. counting before the maximum range of the counter, imposes substantial penalty on the size of the counter which results in more than doubling the area of counts up to 8 bit and almost triples the area for the 9 and 10 bit counters. 64

3.16 The backreference Opcode. 65

3.17 Multiple NFA engines executing in parallel on a FPGA. 66

3.18 The NFA derived from the SNORT Rule 'NetBus\s+\d+\x2E\d+'. This NFA occupies 71 slices and can run at 331MHz. The NFA controller implements flip-flops to enable subsequent stages in the NFA and generates the match output at the final flop. The NFA controller occupies 2 slices in this regular expression. 66

3.19 Architecture of parallel PCRE Engines on Virtex-4 LX 200 FPGA. Each of the sixteen byte-wide bank obtains a character from the Memory Interface Module and sends them to the 14 NFA engines on that bank. The BRAM is utilized by NFAs implementing the back reference opcode. 67

3.20 Overall system using SNORT IDS and PCRE Engines on FPGA 69

3.21 Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of minimal malicious activity in the network payload dump. 71

3.22 Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of minimal malicious activity in the tcp payload dump. 72

3.23 Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of moderate amounts of malicious activity in the tcp payload dump. 73

3.24 Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of moderate amounts of malicious activity in the tcp payload dump. 73

3.25 Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of high amounts of malicious activity in the tcp payload dump. 73

3.26 Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of high amounts of malicious activity in the tcp payload dump. 74

3.27 Throughput of the PCRE engines on the SGI RASC RC100 Blade as function of the number of regular expressions. The speedup is in comparison to software execution on a 3.0 GHz Xeon. The throuput improvement is 353x using 200 regular expressions. 77

3.28 Area (in slices) occupied by PCRE engines on the Virtex-4 FPGA on SGI RASC RC100 Blade. The lower dark section is the fixed area cost dedicated to the RASC Core services on the FPGA (11,012 slices). 79

3.29 Picture of the RASC RC100 Blade usable on the SGI Altix 4700 81

4.1 System Architecture of the dynamic co-processor system on FPGA 89

4.2 An example Floating Point IP Core, demonstrating the I/O interface 91

4.3 ROCCC system overview 92

4.4 The C function call to the co-processor and the #pragma directive 94

4.5 Data flow using FSL from the Virtex-4 APU to the static wrapper 94

4.6 Data flow using FSL from the Virtex-4 APU to the static wrapper 95

4.7 A compiler generated dynamic wrapper for CORDIC engine 96

4.8 The Partial Reconfiguration Module Generation Flowchart for FPGA 97

4.9 SLICE macros placed on the Dynamic / Static logic boundary 98

4.10 SLICE usage for various IP Cores, and PR Block occupancy 98

4.11 Architecture of the Sixteen Partial Reconfigurable Area blocks on the Virtex-4 LX 200 FPGA. Each PR block consists of fourteen NFA engines. A PR Block, expanded on the right hand side of the figure, obtains one byte payload data through the SLICE Macro each clock cycle and outputs 14-bit match data on completion of a match. 103

4.12 Using regular expression engines with an integrated FPGA hardware and multiprocessor software flow 106

4.13 System Throughput with increasing number of FPGA reconfigurations / minute. During the FPGA reconfiguration, the software based PCRE engines are utilized. Data is plotted for both partial one bank reconfiguration and complete FPGA reconfiguration. 108

4.14 Projected System throughput with increasing numbers of DPR per minute. With DPR, only the regexes corresponding to the NFA bank being reprogrammed need to be run in software. 111

5.1 An XML Publish Subscribe System. A published XML document stream is parsed and filtered through multiple subscriber profiles. 115

5.2 An example XML tree 119

5.3 Compilation Flow of XPath expressions to FPGAs. The XPATH profiles go through a four step compilation process to generate the HDL. The lower gray section denotes the hardware flow for converting HDL to a bitstream for the FPGA. 121

5.4 The block diagram for XPath <a0>//<b0>, showing the implementation of the ancestor-descendant axis 123

5.5 The block diagram for XPath <a0>/<b0>, showing the implementation of the parent-child axis. The additional hardware includes the tag filter, stack and TOS match blocks 125

5.6 Block diagram of the Character Match Hardware Block for a tag <a0>. The hardware is a 8-bit x 4 comparator block. 127

5.7 Block diagram of the Character Pre-Decoder Hardware Block for a tag <a0>. The hardware is a 1-bit x 4 comparator block. 128

5.8 An example FPGA organization denoting the input / output data path with sixteen XPath expressions. 129

5.9 XML document, Tree and Prufer sequence representation. 130

5.10 The block diagram for twig matching hardware. This block generates Prüfer sequence of the XML tags from a streaming document and matches it with Prüfer sequences of the queries in twig form. This figure is an example of the query a0[b0]/c0. 135

5.11 The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of two tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of two tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware. 138

5.12 The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of four tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware. 139

5.13 The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of four tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware. 140

5.14 The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags and eight tags long sequence matching paths for twig queries. The graph at the bottom demonstrates the variation of clock speed of the FPGA hardware in MHz with increasing number four tags and eight tags long sequence matching paths for twig queries. 141

Chapter 1

Introduction

Stream data processing and inspection involves executing a set of pre-defined processing steps on the streamed sequence of data. Examples of network oriented stream content processing applications include IDS (Intrusion Detection Systems), XML (eXtensible Markup Language) data filtering, video transcoding, image compression, etc. Due to the nature of stream content processing, it involves very low storage overhead during the actual processing step. Moreover stream data processing is highly parallelizable because a stream can be processed across multiple independent engine threads. In fact multiple streams can be processed by multiple parallel threads, all independent of each other. Such kind of applications are amenable for hardware acceleration using silicon devices such as FPGA (Field Programmable Gate Array), ASICs (Application Specific Integrated Circuit) and GP-GPUs (General Purpose Graphics Processing Unit) [127].

1.1 FPGAs for Code Acceleration

FPGA based reconfigurable computing systems are being commonly used to speed up CPU (Central Processing Unit) intensive applications. FPGAs allow speedup of slow sequential software by efficient hardware implementation. Streaming applications are highly suitable for speedup on FPGAs, because the required processing steps can be implemented efficiently as a datapath on the FPGA. Moreover a datapath implemented on FPGAs alleviate the inefficiencies of Von Neumann computing paradigm, by trimming down load store and branch instructions from the traditional CPU datapath.

Modern FPGA devices benefit from Moores' Law, latest silicon processes and feature size. FPGAs provide immense amounts of programmable logic blocks which can be exploited for implementing parallel datapaths from CPU intensive algorithms. Another important advantage of FPGA with respect to other available acceleration devices is its reconfigurablity. FPGAs are programmable hardware devices, and various innovative and custom digital circuits can be implemented on them. FPGAs can be also be reprogrammed to obtain a multitude of hardware capabilities, at different times, as required by the implementation scenario. One drawback of an FPGA when compared to an ASIC or a CPU is its low clock speed, usually an order slower than the highest performing CPUs. Nevertheless, FPGAs can more than make up for the slow clock rate by implementing several parallel datapaths.

Figure 1.1 demonstrates a simple explanation of how an FPGA, compares to a dual core CPU operating at ten times the clock rate of the FPGA. A datapath could be analogous to a

highway and the traffic on the highway is analogous to data moving on the datapath and the speed limit is analogous to the clock speed of the device. A dual core CPU implements two parallel data path at a clock rate of 100 units. The datapath could be two threads of regular expression matching engines. On an FPGA, several of these engines can be optimized and implemented. Each engine operates in parallel, running at a clock rate of 10 units. If we measure the overall throughput of both the approaches, it can be ascertained that the FPGA provides ten times the throughput than what is provided by the CPU.

Figure 1.1: Demonstration of the throughput of a CPU and FPGA: A Highway Analogy. An FPGA with a ten times slower clock rate can offer 10X the throughput of a dual core CPU by implementing two hundred parallel data paths on the fabric.

FPGAs are increasingly being made available as co-processors on high-performance computation systems. In one kind of configuration, the FPGAs are available in blades, which are directly plugged into the backplane of a server. The SGI RASC Blade [4] available on SGI Altix 4700 [151] shared memory computing system is one such example. The available maximum data throughput for this blade is 6.4 GBytes/s. In other kinds of implementations, FPGAs are usually packaged in modules, which are dropped in CPU sockets on server

motherboards. Examples include bridged FSB-FPGA [123] on Intel Xeon platforms, Quick-path interconnect [191] on new Intel platforms and Hypertransport [12] interfaced FPGA on AMD Opteron platforms. One of the first systems of this kind is the Cray XD-1 supercomputer [1]. The Intel FSB FPGA architecture provides upwards of 8.5GBytes/s throughput. The Xtremedata [5] XD1000 coprocessor device [210] using a single channel 8-bit Hypertransport provides 0.5 GBytes/s throughput.

High density FPGAs such as Xilinx Virtex-4LX 200 [195], Virtex-5 LX [207] and Altera Stratix II EP2S80F [9] containing millions of logic gates, abundant high speed dual port memory hardware ALU blocks on current feature sized silicon fabric (90nm, 65nm, 40nm) have been used in these accelerator systems. Modern FPGAs also integrate a (hard or soft) processor core, with the reconfigurable fabric. These FPGAs provide multiple specialized I/O transceivers [206] [10] which can operate from 600Mbps to 6 Gbps. The acceleration architecture involves streaming input data from a host processor to the FPGA, which is then processed by the various hardware datapaths, with the eventual streaming out of the results back to the host processor. The host processor is relegated to the simple task of setting up DMA based data transfer from memory to the FPGA, thus resulting in very high throughputs [10] [206]. A typical FPGA based application acceleration scenario includes profiling / inspecting the software code to ascertain the slowest executing program components. These program components are transformed to equivalent HDL (Hardware Description Languages) for implementation on FPGA. The HDL code is synthesized and implemented on FPGA. Simulation after Place and Route can provide an accurate representation of the theoretical

speedup, provided by the FPGA. Various application from a wide range of domain have been successfully accelerated on FPGAs. The application domain for utilizing FPGA based code acceleration include computer databases [209], regular expressions [116] [162], molecular dynamics [186] [185], image and signal processing [73] [117] [115], bio-infomatics [55][44], option pricing models [211], signature detection, [168], etc.

The transformation from a high level language description to HDL is a very challenging task, due to incompatibilities in the execution paradigm between a traditional CPU and on an FPGA. Architectural limitations on an FPGA based accelerator includes lack of pointer based memory addressing, lack of dynamic memory allocation, limited stack size, lack of an ISA (Instruction Set Architecture) and absence of cached memory access. Thus, to accelerate a program written in high level language on FPGA the above mentioned limitations have to be factored in before conversion to HDL. As an example, pointers need to be converted to array accesses in C loop nests and regular expression that demand nesting on a stack need to be simplified. Manual translation from high level programming language to HDL has been used in the past, but it quickly manifests itself as an impediment to a wider use and is prone to errors. Compilation frameworks thus are a valuable tool for translating traditional high level description languages to FPGA.

1.2 Regular Expression to HDL

An example of a widely used inspection and parsing application is regular expression. In regular expression matching process, the input stream is inspected for the existence of one or more member strings of a given regular expression. Regular expressions are usually implemented as one of DFA or NFA in software based systems. DFA implementation is unsuitable for hardware due to the state space explosion of implementing a DFA, and the accompanying memory requirements. NFA based implementations try to mimic parallelism on software based execution by using a stack. Since the software can evaluate only one transition at a given time, a stack based implementation can store the other transitions for future evaluation at a later time. On the other hand FPGA provides inherent hardware based parallelism, which allows an automata to evaluate more than one state transition at the same time. Therefore NFA based implementations are extremely suitable on FPGAs. One area which has seen a rapid growth in use of regular expressions is Intrusion Detection System (IDS). Increase in malicious activities using computer networks as a medium, has also resulted in an increased deployment of IDS that scan and intercept network packets containing signatures of such activities. SNORT IDS, one the most popular open source IDS uses PERL compatible regular expressions (PCRE) for its regular expression based rules. Network payload data are streamed to a PCRE engine and is tested with a regular expression deemed suitable by SNORT. The NFA based regular expression model used by PCRE imposes a high demand on the computation power needed to execute regular expression matches. With current net-

work data links approaching 10 Gbps and higher, software based regular expression engines working on network payloads are unable to cope up with the link throughput while looking for malicious signatures within network packets, especially during an active attack. Thus a FPGA based implementation of regular expression rules can result in speeding up of an IDS. More so, since a network payload is frequently tested on more than one regular expression rule, an FPGA based accelerator can parallelize the regular expression matching, by testing the payload through the required rules simultaneously.

Our compilation tool converts a PCRE to HDL via PCRE opcodes. This tool solves a very important limitation towards implementation of PCRE on FPGA. This tools uses the front end parser of the PCRE compiler which produces opcodes based on the regular expression operands which are in turn obtained from a regular expression. In the original software implementation, these opcodes are executed by a software based PCRE engine running on a processor. In order to accelerate PCRE, our tool allows us to implement the opcodes on a FPGA based PCRE engine. The current implementation of the tool operates within the limitations of FPGA hardware and does not currently support regular expression operations that could require nesting / recursion. Our tool compiles the PCRE opcodes obtained from a regular expression and creates HDL code for each of them. Our compilation tool then integrates the hardware opcode blocks with a NFA controller which is then implemented on a FPGA. Moreover multiple regular expression rules are connected to the same input data stream, allowing a network payload to be tested against multiple rules at the same time. Multiple input data streams can also be matched in parallel banks on suitable FPGA hardware

that supports multiple input streams, for example SGI RASC Blade.

In a load balanced software implementation of SNORT, multiple IDS processes generate multiple network payload test requests, which are processed by multiple instances of PCRE engines in parallel. We solve the problem of accelerating such systems by implementing multiple banks of regular expressions, with each regular expression bank catering to one of the IDS process. With a 128-bit wide input data bus (available on SGI RASC Blade), a single FPGA can cater to sixteen 8-bit input payload threads. Additional FPGAs can implement many more regular expression banks or can replicate the banks, as deemed necessary.

1.3 FPGA Reprogrammability

Unlike ASICs and hard silicon devices, FPGAs allow reprogrammability of the hardware. This powerful feature increases the versatility of FPGAs, and increases the number of applications that can be accelerated. FPGA based accelerators typically utilize a fixed width input / output data interface. The accelerator hardware block connects to this interface and thus is able to send / receive data from the host processor. Interfacing the available library of accelerator blocks to the FPGA interface is a time-consuming and tedious task which almost always, needs to be taken care of manually. The system designer is left with the task of interfacing each and every accelerator blocks, (usually available as IP cores) to the data interface. In order to solve this problem, we have developed a tool that automatically generates the communication interface between the data interface and a tightly coupled IP core based

accelerator (co-processor) system on a CSoC (Configurable Systems on a Chip) i.e. Virtex-4 FX FPGA. It generates hardware wrappers for the IP core that makes the hardware look like a C function invocation in the host processor source code. Thus a 'C' function call in the host processor can change the functionality of the FPGA base co-processor by reprograming the required bitstream. The SGI RASC library uses a similar mechanism by which an API call can reprogram the Virtex-4 LX 200 FPGA on the blade with a new bitstream.

Partial reconfiguration on the FPGA makes it possible to create a system that allows reconfiguration of pre-assigned parts of the FPGA without affecting the static parts, or inducing a system-wide reset. It is a very powerful tool to overcome the area limitation of a single FPGA platform across multiple applications. The system designer is usually left with the task of generating the interface between static and dynamic regions of the FPGA as required for partial reconfiguration. We extend the aforementioned interface generation tool to support partial reconfiguration, by generating an interface wrapper that delineates the static and dynamic regions on the FPGA. This feature is useful on CSoCs that implement basic hardware peripherals along with a co-processor on the same reconfigurable fabric.

Another application for utilizing partial reconfiguration on an FPGA are situations that demand quick adaptability. Software defined cognitive radios have used this functionality for a while to adapt the hardware towards external changes in the air interface.

There is a limitation to the number of regular expressions that can be implemented on an FPGA. Fast changing network activity scenario, can lead to the IDS selecting among different rulesets in a short period of time. Reconfiguration of the FPGA with the type of intrusion

detection engines required at the moment enables adaptability to change with network conditions. Partial reprogrammability of the FPGA can be used to cater to such dynamic situations, when only few rulesets of regular expression rules changes over short intervals of time. IDS that employ these FPGA can maintain execution through software threads during the brief moment, when the FPGA is being reconfigured Moreover an IDS with two or more FPGAs can maintain execution on the other FPGA(s) while one of them is either partially or fully reconfigured with a different set of regular expression engines. In order to implement an adaptable IDS that allows a single bank of regular expression to be re-programmed keeping the other banks intact, we have added modularity to our aforementioned regular expression architecture. With our modular architecture, regular expression banks can be quickly swapped in order to adapt to changing networks scenarios.

1.4 XML Filtering on FPGA

Streaming XML filtering is being used abundantly for publish/subscribe applications (or simply pub-sub). In pub-sub, the message transmission on the internet is guided by the message content, rather than its destination IP address. Selective deliverance of parts of XML documents, is obtained by filtering the document through multiple filters described in high level language such as XPath. In pub-sub systems, each individual subscriber interest is described by an XPath expression. XPath expressions consists of a sequence of XML tags and the relationship between the tags are expressed as axes. XPath infers a a tree based naviga-

tion over an XML document, and involves a parent-child axis and ancestor-descendant axis. XPath profiles can be easily converted to PCRE. The XML tags are converted to character match blocks, while the axes define the regular expression sequence. We have used a conversion process to convert XPaths to PCRE. Thereafter we have employed our PCRE to HDL compiler to implement XPath profiles on hardware. The only other addition required would be a XML tag STACK to verify a parent-child axis. Pub-sub involves streaming the same document across multiple XPath profiles, and thus all the subscriber profiles can execute in parallel on a FPGA. Moreover XPath profiles share commonality in their prefix, and hence are optimized to share the common prefix and reduce the area occupied on FPGAs.

Implementing XPath profiles on FPGAs mainly involves implementing character matching blocks to identify XML tags in the input document stream. The character matching hardware block compares sequences of characters from the input stream to a given sequence that define an XML tag. The implemented character matching blocks for the XML tags consist of many redundant blocks, the prime examples being the open tag '<', close tag '>', and end tag '/' characters. It is possible to simplify the design with a 8-bit stream ASCII decoder. This decoder can be used to decode the 8-bit XML data input into one of each 256 1-bit output per clock cycle. Identifying an XML tag in the input stream would thus involve a simple controller, one that checks for a given sequence of 1-bit decoder symbols. Due to this simplification, the XML filter design using a ASCII decoder is area efficient and it runs at a higher clock rate when compared to the design using character match blocks.

XML queries are also formed from twig structures, which include multiple path queries

in a single query. Such queries can be converted to a sequence of tags / tree nodes also known as a Prüfer sequence. An XML document is also converted to its Prüfer sequence. Thus the problem of matching a twig query would now involve matching the given Prüfer sub-sequence of the twig with the Prüfer sequence of the document.

1.5 Contributions

1.5.1 Compiling PCRE to FPGA and accelerating SNORT IDS

We present a novel method to compile PCRE Operation Codes (opcodes) directly to VHDL for parallel implementation on FPGA hardware. We implement the PCRE regular expressions from the SNORT IDS using a two stage translation process. In the first stage, the SNORT IDS rulesets are compiled using the PCRE compiler to generate PCRE opcodes. In the second stage the PCRE opcodes are translated to VHDL hardware blocks suitable for implementation on FPGA and connected together using a NFA based control logic. Our system maintains the execution semantics of the software based regular expression engine on the FPGA hardware, thus ensuring compatibility with the SNORT IDS ruleset. The interface throughput suffices for wire-speed payload scanning of even the fastest available ethernet interfaces. Our design is a compile once, NFA based design, with re-compilation necessary only for new and updated rules. We obtain more than 350X speedup with our FPGA based regular expression engine architecture when compared to a baseline state of the art CPU, the Intel Xeon 5160. Our design can sustain a throughput of 12.9 Gbps.

The specific contributions include:

- Compilation of PCRE opcode to hardware. We modified the PCRE compiler v6.7 by adding an opcode dump module to the PCRE compilation library. We then process the generated opcodes and related operands using successive compilation steps and convert them to VHDL blocks. The VHDL opcode blocks are tied together in an NFA and are synthesized and implemented on the FPGA.

- Quantitative Evaluation of PCRE opcodes. We investigate the frequency of the opcodes which make up regular expression rules in SNORT DB2.4 using the modified version of the PCRE compiler. We have removed the NETBIOS rules from the database since it is suggested by the security community [54] that this ruleset be disabled because the rules may generate a lot of false positives, and these rules are not pertinent to Internet based traffic. We have ascertained, using the results, that the character match is the most prominent opcode (34%) followed by the Kleene closure operator '*' (26%). Our design is currently limited to the frequently occurring opcodes in SNORT DB. Moreover due to hardware limitations, we concentrate on regular expressions that do not involve backtracking.

- Implementation Details of the PCRE opcodes. We discuss detailed FPGA based implementation of important PCRE opcodes. We compare two important implementation paradigms i.e. utilizing IP core based opcode generation and secondly by synthesizing the opcodes from VHDL code. We provide data on the variation of area and speed of

important opcodes generated using the two paradigms. We provide additional details with respect to quantifier opcodes. We provide data on the size of counters required to deal with the specific range of counts that occur after the compilation of SNORT rules. We also document the variation of FPGA area with increasing number of regular expressions.

- Implementation of SNORT Regular Expression Rules on FPGA Platform. We have designed an FPGA based regular expression IDS with clock speed of 155MHz and 128bit payload I/O per clock cycle. It is possible to run regular expression based IDS on the Virtex-4 LX 200 FPGA at a maximum sustained throughputs of 19.84 Gbps. We obtain more than 350X speedup with our FPGA based regular expression engine architecture when compared to a baseline state of the art CPU viz. the Intel Xeon 5160 and our design can sustain a throughput of 12.9 Gbps on the SGI RASC Blade.

- Detailed Benchmark of a Load Balanced IDS with Regular Expression. We conduct detailed benchmark of software based regular expression IDS, under various network scenarios, using three different multiprocessor systems. We demonstrate, with a first of a kind comprehensive experiment, that, even on a thirty core Distributed Shared Memory (DSM) system, a software based IDS using regular expression, can reduce the network throughput from 10 Gbps to nearly 150 Mbps when the amount of malicious activity increases in the network. We demonstrate the case when 30% or more number of network packets trigger regular expression matching, the software based IDS cannot

maintain 10 Gbps throughout, and thus requires hardware acceleration.

- Comparative Evaluation of CPU and FPGAs. Multi-CPU load balanced IDS approach[63] entails huge cost in terms of the hardware infrastructure, with the cost of components like memory, motherboards, disk drives and also power consumption and cooling costs adding up. We provide an analysis into the projected performance of a FPGA vis-a-vis a high end dual core processor while executing regular expression matches, and in this dissertation, we estimate the power savings enabled by the use of FPGAs in such designs.

1.5.2 Dynamic Co-Processor Interface Automation

In this dissertation. we describe a software tool for automatically generating the communication interface between the software running on the CPU and a tightly coupled IP core based co-processing system on the Virtex-4 FX FPGA. We use the software tool to extend our compiler for FPGA-based reconfigurable systems, ROCCC [72] which leverages the huge wealth of IP cores by allowing the user to import these cores into the software source code. Our tool generates hardware wrappers for the core that makes it look like a C function invocation in the source code. Using this tool, the compiler automatically generates a wrapper structure that would hide the timing and stateful nature of the IP Cores and makes each available to the C language compiler, as an un-timed side-effect free function call.

We extend this tool to support partial reconfiguration: the same static wrapper is used for multiple cores and the user selects a given core to be invoked in the 'C' program. We support

run-time reconfiguration by automating the generation of the interface between static and dynamic regions of the FPGA. The user can switch between multiple functional units by calling the appropriate C function in the code, thus entailing the use of the same hardware wrapper for multiple IP Cores. Utilizing our software tool along with the ROCCC infrastructure we have been able to automatically configure multiple IP-cores on the fabric viz. FP (Floating point) Adder, FP Multiplier, Integer divider CORDIC engine and an FFT engine.

Moreover using partial reconfiguration we have been able to overcome the area limitation of a single FPGA platform (Virtex-4 FX12), using five different IP Cores. We have allocated a region of 1800 slices for the co-processor, thus resulting in a reduction in the floor area by 2656 slices due to partial reconfiguration. Moreover the area dedicated for the hardware wrapper is no more than 14 slices, quite miniscule, when compared to the actual IP Core area.

1.5.3 Adaptive Hardware/Software Regular Expression Based IDS

FPGAs can be reprogrammed to change it's functionality. This allows reconfiguration of the FPGA with the type of the intrusion detection engines required at the moment i.e. adaptability to the current network conditions. Additionally Xilinx FPGAs and some custom developed FPGAs [161] also support partial reconfiguration flow, so that a part of the FPGA could be reconfigured. This reduces the hardware re-programming time, when only a part of the FPGA needs to be modified. IDS that employ these FPGA can maintain execution through software threads while the FPGA is reconfigured during the brief moment. Moreover an IDS

with two or more FPGAs can maintain execution on the other FPGAs while one of them is either partially or fully reconfigured with a different set of regular expression engines. An FPGA system supporting partial reconfiguration can respond to new types of network attack much faster than FPGA systems that only support full reconfiguration.

With our proof of concept hardware system, our novel design allows partial reprogramming across 16 banks of regular expression rule-sets can successfully maintain throughput at 10 Gbps scale even under a range of partial and full reconfiguration scenarios running on a proof-of-concept platform. We use our PCRE to HDL compiler to compile regular expression based rules to VHDL. Similar rules are grouped together in banks of rule-sets. We implement our adaptive IDS on a Virtex-4 LX 200 FPGA that has been floor-planned for partial reconfiguration across 16 banks of regular expression rule-sets.

We have also benchmarked our proof of concept FPGA accelerated regular expression based IDS test-bed using a thirty-two core SGI Altix 4700 supercomputer with a RASC Blade consisting of two FPGAs. We implement 448 different regular expressions in 32 modular rule-sets, on the two FPGAs. Such an architecture is a first of a kind demonstration of an adaptable hardware/software regular expression based IDS. We show that by utilizing our architecture, it is possible to avert concerted attacks and also to adapt towards changing network activities, by performing multiple partial and full reconfigurations. We measure the throughout of the integrated Field Programmable Gate Array (FPGA) and multiprocessor SGI Altix system with varying number of reconfigurations per minute. The maximum sustainable throughput of our design is 19.84 Gbps per FPGA. Our adaptive IDS can provide

better than 10 Gbps throughput even with 32 partial reconfigurations per minute. Our system can also sustain 10 Gbps throughput with four full-reconfigurations per minute. Our IDS design can be extended to similar FPGA accelerated multi-processor system.

1.5.4 Boosting XML filtering with a scalable FPGA-based architecture

The growing amount of XML encoded data exchanged over the Internet increases the importance of XML based publish/subscribe (pub-sub) and content based routing systems. The input in such systems typically consists of a stream of XML documents and a set of user subscriptions expressed as XML queries. The pub-sub system then filters the published documents and passes them to the subscribers.

Pub-sub systems are characterized by very high input XML data rates and therefore the processing time is critical. Given the high volumes of messages and profiles, the filtering process becomes a critical performance requirement for pub-sub systems. Since pub-sub XML filtering involves multiple parallel queries processed over a single document data-stream, it is possible to utilize FPGAs for improving the filtering performance. Each query can be implemented on the FPGA unit as a hardware datapath circuit and with appropriate optimizations it is possible to fit thousands of queries on a single FPGA chip. This results in accelerated query processing and leads to substantial savings in general purpose computation infrastructure, and thus reducing the amount of power required by the infrastructure.

We utilize a four step approach that converts user profiles expressed as XPath queries into hardware description language, suitable for implementation on FPGA. The first step involves

conversion of an XPath query to PERL compatible regular expressions. The regular expressions are clustered by their common prefixes in order to produce more compact representation on the board and are then translated to VHDL using our "PCRE to VHDL" compiler. Moreover, in order to support parent-child relationships, we introduce the use of stacks and modify the regular expression hardware to use them. The highly optimized VHDL code is then deployed on the Virtex-4 LX 200 FPGA on SGI RASC Blade. The stream of documents is forwarded to the RASC Blade where it is processed with high degree of parallelism. Our experimental evaluation reveals that this architecture achieves orders of magnitude improvement in the terms of running time compared to the state of the art software based XML filtering systems.

We investigated the XPath filter architecture and came to the conclusion that most of the FPGA area was being used by XML tag match blocks, which in turn consist of 8-bit character match blocks. In order to further improve the area efficiency of the XPath hardware we incorporated a stream ASCII decoder, which would decode the incoming XML stream at the input and produce 256 1-bit outputs. The character decoder hardware block simplifies the design of the XML tag match blocks by replacing a 8-bit character match comparator with a 1-bit comparator. Moreover using multiple 1-bit data lines instead of routing the 8-bit input stream over the FPGA, reduces the routing overhead, which in turn leads to a design with faster clock speed. The average area improvement by using a character decoder at the input over distributed character matching blocks is 1.5X while the clock speed improvement is 2.5X.

We have also with described our hardware implementation of streaming Prüfer sequence conversion of an XML document. We also describe how we can execute twig pattern matching using the generated Prüfer sequence. Our hardware can accurately match parent-child relationship in the twig patterns.

Chapter 2

Related Work

In this chapter we describe related work with respect to implementation of efficient IDS systems using string and regular expression based approaches. We also discuss various hardware based approaches for accelerating string and regular expression matching. We also discuss software based load balanced IDS systems, and hardware based approaches to accelerate SNORT IDS.

2.1 Network Intrusion Detection Systems

Network Intrusion Detection Systems originated as software string matchers which worked with the NIC drivers and LIBPCAP [184] to filter malicious packets from the ingress / egress links on a location. Very soon the throughput of the software based NIDS began to dwindle, mainly due to the serial execution approach on a Processor. Moreover executing Regular Expression engine in addition warranted a hardware oriented approach to deal with ever

increasing number of rules which a network payload goes through.

2.1.1 IDS engines with String Matching

Initial versions of SNORT started with string based pattern matching on a ruleset comprising of string matches. These string matching algorithms are continuously being optimized for Software execution, on newer processors. A very important algorithm for state based string matching is the Aho Corasick [6] method. Worst case performance improvement over Aho Corasick was improved with [179] by Tuck, Sherwood, et al, by utilizing path-compression on the Aho Corasick algorithm. Commentz - Walter [51] and Wu [193], propose similar string matching algorithms which preprocess the data structure.

Various improved versions of string matching algorithms are implemented on a variety of Hardware such as FPGA and ASICs. Since hardware execution provides orders of magnitude improvement over software execution of string matching algorithms, thus it is imperative to utilize optimized hardware for current multi GBits/s rate network interfaces. FPGAs provide the dual benefits of fast optimized hardware execution along with great flexiblity to compile and re-program the hardware quickly and efficiently. The use of parallel bloom filter [35] architecture on FPGA has been introduced by [58]. [168] have documented a method to compile C code of bloom filter based text scanners to VHDL and achieve high throughput (18 Gbps) on Virtex II FPGA. [175] and [88] detail on a high throughout design of the Aho-Corasick engine for string matching based IDS on Application specific silicon by converting the Aho Corasick algorithm into multiple binary state machines. Also known as

the bit-split optimization for string matching this optimized engine is documented in detail in [174]. [163], detail a CAM-based pattern matcher FPGA design that additionally pre-decodes characters. [164] also target Virtex-2 FPGA with a fine grained pipelined string matching hardware to achieve 10Gbps throughput. [47] detail out a silicon to implement a hardware based string matching coprocessor for SNORT IDS that runs at 7 Gbps. Their ASIC design provides a high performance platform for pattern matching. [16][15] demonstrate an FPGA implementation of the Knuth-Morris-Pratt algorithm for string matching suitable for IDS applications at 2.4Gbps. Area constraints on string matching hardware on FPGAs have been solved in [213] by utilizing bit-level hardware sharing CAM. The authors also mention on the performance and space efficiency of many other approaches towards efficient high speed hardware implementation of string matching which are proposed in an 2.88Gbps FPGA based hardware in [48], and in an 2Gbps FPGA based hardware in [68]. Novel hardware oriented methods namely Hash Boyer Moore algorithm implemented on Intel IXP network processors with a throughput in close vicinity of 2Gbps have been researched by [125]. A programmable systolic array based FPGA implementation of Knuth-Morris-Pratt string matching by [18] provides 2.4Gbps throughput for use as an IDS. [49] have developed a Platform FGPA based embedded intrusion detection system that includes the network packet decoder as well as the SNORT rules based IDS on a single platform FPGA chip.

2.1.2 IDS engines with Regular Expression Matching

Research initiatives over the past decade have resulted in optimized Regular Expression engines in software as well as hardware which result in fast execution, in order to keep up with increasing data rates of network interfaces.

Song, et al.[160] have also identified the core problem with NIDS, that the throughput reduces drastically while processing malicious packets. Software optimizations have been proposed to mainly target the 1Gbps throughput barrier on general purpose processors. [212] suggest optimizing techniques on DFAs generated from regular expressions to reduce their execution times and achieve 50 to 700 times speedup. But their method also asks for rewriting of the SNORT rulesets, which may not be supported by the community due to their adherence to PCRE standards. [95] have demonstrated graph theoretic algorithm to generate D2FA from DFA by combining multiple transitions in order to reduce the memory requirements of DFAs by more than 95%. Their design enhances Cisco network appliance by reducing embedded memory requirements. [94] have categorized three deficiencies viz. Insomnia, Amnesia and Acalculia, in DFA based execution paradigm and have proposed relevant mechanisms to deal with such problems. Networked cluster based approach for load balancing NIDS has been well documented in[183],[189] and [190]. Katashita et al.[91] explore the potential of porting SNORT rule-set 2.3 using a space efficient NFA hardware on FPGAs and suggest a theoretical maximum throughput of 10 Gbps on a Virtex2-8000 with 64bit datapath. DFA based regular expression engines have been targeted towards FPGAs mainly for parallel execution on smaller FPGAs by Moscola et al. in[122] and Lockwood et al. in[104][103]. The authors

utilize the JLex library to generate description of regular expressions from SpamAssassin rules.

Hardware oriented state of the art architectures utilize FPGAs, ASICs as well as GPGPUs. Tarari Inc. [182] have demonstrated a 6.2 Gbps using a state of the art GPU, to power a malware scanning acceleration engine. FPGAs have been utilized on various IDS architectures, due to their ability to execute parallel regular expression based scanning engines, and the possibility to compile regular expression based rules to hardware, thus leading to immense flexibility vis-a-vis hardware generation and modifications. Compilation of regular expressions to hardware circuits were proposed more than two decades ago in a seminal work by [66]. Regular expressions can be implemented on a given hardware by utilizing two paradigms which are DFA and NFA. DFA entails faster execution, by sacrificing the space requirements, which could exponentially blow up, while NFAs entails $O(n^2)$ space requirements but by processing one character at a time.With the advent of FPGAs with several hundred megahertz clock rates, and high speed I/O interface to the host processors, it has been made possible to speed up NFAs on FGPAs. It is also possible to enable parallel matching paths using NFAs on FPGA. Current research on NFAs used for regular expression Matching have resulted in optimization of speed and area on FPGAs. Sidhu, Prasanna provide a highly detailed work on implementing and optimizing NFAs for use on FPGAs [154]. In fact they propose a fast algorithm that generates the NFA on the FPGA hardware, rather than compiling it from software. Generation of several regular expression operators including single character match, alternation, concatenation, and Kleene closure have been detailed

in this work. [169] improves the FGPA based NFA / Regular Expression model by implementing very fast partial character decoders on the hardware. [120] implement a combined pipelined character grid string matching as described in [15] combined with NFA based regular expression matching. [84] proposed optimized space usage of NFAs via common prefix sharing, as well as a design philosophy built on [154] by including some additional regular expression operators i.e. '?', '.', and '[ ]'. Their design involves a pipelined broadcast tree for ensuring maximum throughput. [43] develop new hardware structures to implement FSM based regular expression engine. On their actual hardware test they obtain a 4Gbps sustained throughput on a 133MHz Virtex-II FPGA. [101] propose various optimization methods including prefix infix and postfix sharing of regular expressions on an older version of the SNORT ruleset. Overall their methods bring about 20% reduction in on chip area, but its effect on clock speed is not discussed. [177] propose utilizing Block RAM resources on the FPGA for storing LUT data, and hence free up LUT resources on the chip. It results in a savings of 26% power compared to LUT based implementation of control logic for Finite State Machines. [32] touch upon a VHDL generation scheme of NFAs from SNORT ruleset. They utilize extensive size optimization on NFAs including prefix sharing, Character Class Sharing and Static Pattern sharing. Their design results in a throughout of 2Gbps on a Virtex-4 FPGA, which suffice for 1GbE or slower network connections. [162] describe three building blocks that optimize constrained repetitions in regular expressions. They utilized their VHDL based hardware blocks to generate overall area efficient IDS systems on FPGA. The maximum throughput corresponding to their improved design is 3.2Gbps on a Virtex-4

FPGA. They also propose a space efficient FGPA specific counter namely the SRL16. [121] demonstrate a state of the art 10+ Gbps capable regular expression architecture on Virtex-4 FPGA. Their design involves character pre-decoding and pipelining to optimize the space speed up the design. They further re-use segments of the pipelined character match grids, in a timed segment matching (TSM) grid. [91] explore the potential of porting SNORT rule-set 2.3 using a space efficient NFA hardware on FPGAs and suggest a theoretical maximum throughput of 10Gbps on a Virtex2-8000 with 64bit datapath. DFA based regular expression engines have been targeted towards FPGAs mainly for parallel execution on smaller FPGAs by [122], [104] and [103]. The authors utilize the JLex library to generate description of regular expressions from SpamAssassin rules. The authors also propose the use of DFAs by providing data on their compactness when compared to NFAs. Since DFAs can have only one active stage the hardware may be faster. [17] demonstrate a microcontroller architecture on hardware that executes specifically regular expressions using DFA and on-chip memories . Moscola, Lockwood et al. also implement a reconfigurable regex parser based router in[119], and the design can be implemented on Virtex-4 FPGA to run on 12.9 Gbps. Baker, Jung and Prasanna demonstrate a microcontroller architecture on hardware that executes specifically regular expressions using DFA and on-chip memories[17].

2.2 Interfacing IP cores on FPGA

Intellectual Property cores have been available for a while for FPGA based systems and have been successfully used by developers of such systems. Xilinx Logicore series of IP Cores are a library of highly available cores and have been extremely popular with designs based on Virtex series FPGA[46] [195] [200]. The XILINX IPIF module attempts to target connectivity of IP Cores to FPGA [203] [201], but does so only for the slower peripheral busses. Targeting IP Cores to the FPGA peripheral bus using wrappers is discussed in [109] [98] [201]. Since IP-cores provide a black / gray box paradigm, system verification and integration maybe an issue. These have been documented in light of popular simulation and programming tools in [188] and the advantages and challenges in development of interface synthesis has been targeted in [152]. IP Core Reuse has been effectively discussed in light of a co-design paradigm in [65]. An automatic generator of interface synthesis for PowerPC software to custom software accelerators is detailed in [136]. Standards based IP bus interfaces such as the VSIA (Virtual Socket Interface Alliance) specify interface standards allow IP Cores to fit into virtual sockets [187]. However, the current condition is that numerous standards exist and no standard is adopted widely [166].

2.3 Partial Reconfiguration on FPGA

Two popular FPGA configuration mechanisms required for Partial-Reconfiguration (PR), Dynamic Partial Reconfiguration (DPR) or Run Time Reconfiguration (RTR) along with their

performances are discussed in [204] [34] [98]. [148] discuss the design space of reconfigurable hardware systems among various applications and the associated methodologies on various platforms. Details on the methodology of PR for Xilinx FPGA is demonstrated in this tutorial [80] .

Since development of a PR system on a FPGA entails working with a birds-eye view of the chip for layout and interface planning, thus the use of graphical environment leads to proper and efficient floor-planning and the process is documented in [60].

An early toolkit (PARBIT) targeted at the Virtex-E FPGA for enabling columnar partial-reconfiguration is treated in [77]. High speed dynamic packet processing circuit modules can be plugged-in on an FPGA using PARBIT and design methodology documented in [78]. Reconfiguration interfaces, modules and tools have been discussed in [173] [25]. Dynamic partial reconfiguration is used for evolvable hardware systems, in which the hardware circuit is able to self optimize, and can reprogram itself with an improved circuit bitstream [150]. [69] describe the rationale for using PR for supporting a multitude of cryptographic algorithms on the same FPGA. [76] demonstrate a twin flow path based run-time programmable architecture that can run one java program flow on one flow while the other is reconfigured. Using DPR [52] [53] demonstrate different behavior and controllers on a mobile robot. [172] use DPR to achieve multiple PR modules for implementing varioius image filtering algorithms on FPGA. In order to obtain hardware multitasking, [171] develop heuristic algorithms for obtaining the best location for re-placing PR blocks on a floor-planned reconfigurable device. [14] demonstrate a self-reconfiguration focussed multiprocessor cores processing system on

FPGA. [62] demonstrate an on-line multi-level fault tolerant design using PR and hardware migration on the ORCA 2CA FPGAs [124]. [134] demonstrate a system that uses PR to enable multiple test modules on an FPGA.[132] describe strategies for failure recovery based including task migration among PR slots on the FPGA. [135] use JCAP that uses the external JTAG port on Spartan-3 FPGA to support configuration readback for failure detection and task migration. [133] demonstrate a FPGA based hardware design of a low power capacity based level measurement system in a storage tank, that can swap the data processing modules. [131] optimize power dissipation of FPGA by adapting the on-chip signal lines using 2D placement of the PR Module. They use XPower tool to estimate power savings. [82] demonstrate 2-Dimensional PR hardware modules for Virtex-II FPGA each containing functional logic blocks and communication blocks. [83] solve signal routing problems that arise due to the use of TBUFs at the PR module boundaries. They solve the problem by using FPGA slices.

[156] have developed a bitstream generation tool for PR and bitstream reallocation. [126] demonstrate a PR system for a adaptive lossless image compression. [37] present a method to work around singe even upsets faults using PR. [81] provide a tool to visualize the space time effect of an FPGA application using PR.

[29] provide a FPGA partition scheme to implement various functional modules using PR. [64] provide a tool for implementing PR on an embedded CSoC. [28] have developed a emulator that allows efficient scheduling of PR blocks on a FPGA. [42] use SystemC to model and simulate PR on FPGA. [214] use PR to demonstrate an adaptable DWT hardware.

[114] use PR to implement an adaptable cryptographic hardware with certain countermeasures against attacks. [20] [22]develop tools to select FPGA PR task graph for JPEG applications to improve on execution time. They further optimize the performance of task chains by selecting the granularity of data-parallelism. [21] formulate exact and heuristic partition and scheduling schemes using task graphs for DPR on FPGA. [130] improve their work by enhancing it with pre-fetching and instruction forecasting. [23] [19] also demonstrate optimal sharing of bandwidth among PR tasks by assigning suitable clock frequency to each task.

[100] demonstrate a bitstream prefetching scheme to overlap the computation on host processor with reconfiguration of PR block and thus reduce the reconfiguration overhead. [170] demonstrate the implementation details of a user-level process that configures the bitstream on a CSoC at run-time. [112] define API interface for using PR cores on FPGA through a RTOS.

[79] add hardware based fast routing to quickly route new PR hardware on FPGA. [192] demonstrate a dynamic instruction set computer which can modify its instruction set on demand, and allocate hardware resources using DPR. [89] [90] demonstrate vertically placed PR blocks along a horizontal communication channel PR design that can be relocated on the FPGA using bitstream manipulations. [137] demonstrate hardware-software task relocation, and use PR modules for dynamically swapping hardware blocks on FPGA. The system is demonstrated with encryption algorithms like DES, AES, etc. [96] demonstrate the concept of dynamic partial reconfiguration to alleviate area limitations.

As described in [27], DPR can provide versatility and reliability in FPGAs. [40] demon-

strate a lightweight network protocol for implementing a remote bitstream server and network protocol for supporting DPR. [92] developed an analytical model for placement costs considering the constraints for PR modules on FPGAs.

[45] show reduction in PR module bitstream size for difference based PD by automating the creation of designs at placement level to be as similar as possible. [113] demonstrate an overlay based design that allows swapping of IP PR blocks by subsequent reconfiguration calls. [111] demonstrate the applicability of DPR for efficient circuit-switched network for use in WDM networks. [110] discuss the DPR problem as slot based allocation and demonstrate a slot-oriented architecture called 'Erlangen Slot machine'. [149] discuss programming PR module bitstreams from an embedded OS running on a NIOS [8] soft-processor on a Virtex-II FPGA. The system is called Aquarius, and also support full reconfiguration of the FPGA. [143] detail their CAD tools that can be used with FPGA design tools to generate and control PR designs. They demonstrate a PR Viterbi decoder design.

[153] demonstrate applicability of DPR towards on-the-fly routing of neural synaptic connectivity on a Virtex-5 FPGA and thus realizing a spiking neural model on chip. [155] describe the organization of dynamically modifiable processing pipeline on an FPGA using DPR. [50] describe the use of PR for utilizing a variety of co-processing cores for accelerating video processing in driver assistance systems on vehicles. They demonstrate their design with an 'AddressEngine' co-processor for processing pixels in a video stream. [36] develop algorithms to analyze tasks from their Control Data Flow Graphs and extract temporal reusable modules. These can be used as temporally placrf PR modules on FPGAs. [215]

demonstrate an automatic adaptation method for programming clock frequency and dynamically programmed PR modules to achieve power efficiency. They demonstrate their design using IDWT (Inverse Discrete Wavelet Transform) hardware design. [181] demonstrate a DPR architecture for use by automotive controller to support a multitude of in-vehicle functionality, and interfaces with CAN (Controller Area Network) protocol. [56] demonstrate using DPR to realize virtual pipelines, which are pipelines that are implemented incrementally on the FPGA. Large designs, too big to be incorporated on a single FPGA can be supported by overlapping reconfiguration with pipeline stage execution. [67] demonstrate DPR to reconfigure hardware configuration registers on a reconfigurable TPM (Trusted Platform Module) platform.

[26] present on-demand reconfiguration, diagnosis and recovery techniques for DPR hardware. [129] demonstrate a DPR architecture for RSoC (Reconfigurable System on Chip). Their architecture is tune for use in space applications. They demonstrate PR Modules for packet forwarding network on FPGA for use by specific codecs for data compression. [105] demonstrate a low power self-reconfigurable platform based on an Atmel FPGA which supports multiple profiles on onboard memories. [57] realize a reconfigurable cryptographic system by separating algorithm blocks which are PR Blocks and the system architecture into two components. [61] characterize the performance bounds of a DPR / RTR system with an analytical model. They verify their analysis on a Cray XD1[2] [1] system. [180] show the applicability of DPR for speeding up JPEG encoding on FPGA, when compared to software execution. [38] demonstrate improvement in recovery time for TMR (Triple Modular Re-

dundancy) implementation on FPGA using DPR. [140] demonstrate additional improvement in recovery time for TMR by using voting and check point states, and reconfiguration of only the faulty domain. [39] demonstrate use of DPR in a FPGA based system, in order to be able to cope with SEU (Single Event Upsets) faults by re-programming the partial bitstream of the faulty module. [31] [30] demonstrate a design flow to realize partial reconfiguration by a algorithm control data flow graph, in which vertices represent PR module operations and fixed hardware operations. They used their methodology for efficient mapping, code generation and resource estimation for a telecom application module (MC-CDMA).

Since DPR is a relatively new technology, [147] discuss a framework of steps required for obtaining certifiable DPR designs for avionics, health, etc based applications.

2.4 XML Filtering

2.4.1 Software Based Filtering

Among the many works related to the XML filtering the XFilter [11] is one of the first of its kind. It defines a Finite State Machine (FSM) for each XPath profile, where every tag in the profile becomes a state in the FSM. The last tag of the user profile becomes the accept state in the FSM. The machines are then executed concurrently for each message in the input. When a machine finds the current state as an open tag in the user profile, it makes a transition forward to the next element. When it finds the state as a closed tag it makes a transition backward. Finally, if an accepted state is reached, the document is reported as

a match to the corresponding profile's subscriber. Later, the *YFilter* [59] system improved the matching performance by combining all profiles into a single Nondeterministic Finite Automata (NFA). Common profile prefixes are combined and represented with a single set of states. This allows dramatic reduction in the number of states needed to represent the set of user profiles. It also improves the filtering performance of the system by processing common profile paths only once.

Other FSM-based approaches use different techniques for building the machine as well as different types of machines. For example, [75] builds a single deterministic push down automata using a lazy approach, [70][128] employs a lazily built Deterministic Finite Automata (DFA), [107] builds a transducer, which employs a DFA with a set of buffers. [138] [139] demonstrate a hierarchical organization of push down transducers with buffers in an engine named XSQ processing streaming XML data.

All these solutions are similar in the sense that they traverse the provided input document in a top-down fashion (i.e. in-order traversal) while advancing the state machine for each XML element (or attribute) read. Another proposed approach is to use a bottom-up traversal of the document. This idea takes into consideration the fact that an XML document typically has its more selective elements located at its leaves and uses them to perform early pruning in the query space. Examples of systems which utilize the bottom-up approach include FiST [142, 97] and BUFF [118].

The NFA based approaches discussed above are entirely *software* based solutions using the standard *von Neumann organization*. None of them takes advantage of specialized archi-

tectures to overcome the bottleneck problem which appears during XML document filtering.

2.4.2 Hardware Based Filtering

Previous works [108, 102, 165] that have used FPGAs for processing XML documents have mainly dealt with the problem of XML parsing which in turn is transformed to implementing regular expressions on FPGAs. In particular, [108] proposes the *ZuXA* engine to parse XML documents. This engine employs state machines for efficient parsing based on set of rules. The paper however does not provide any discussion how this engine can be adapted to work with the XPath profiles used in the pub-sub systems.

There is also a large amount of research related to implementing regular expressions on FPGAs [154, 101]. Here we build on our previous works [116] where we compiled PERL Compatible Regular Expressions (PCRE) to VHDL for accelerating intrusion-detection system rules using FPGAs. However, XPath query evaluation is more complex than plain regular expressions. To this end we introduce appropriate stacks that are implemented on the FPGA device.

The works in [165, 102] propose the use of a mixed hardware/software architecture to solve simple XPath queries having only parent-child axis. A finite state machine implemented in FPGAs is facilitated to parse the XML document and to provide partial evaluation of XPath predicates. The results are then reported to the software part for further processing. Similarly to the *ZuXA* engine, this architecture can only supports simple XPath queries with only parent-child axis.

There are also approaches that use specialized parallel architectures for XML processing [99, 106]. In particular, [99] uses the Cell Broadband Engine multi-processor which consists of 8 independent processors (SPEs) that run the same software. This approach achieve parallelism by parsing (eight) XML documents in parallel at a time. Each processor implements the FSM of the *ZuXA* engine [108]. In addition to be only suitable for XML parsing, this solution is a combination of hardware-software approach.

To the best of our knowledge our system is the first one to provide an *entirely* hardware solution to the XML filtering problem in pub-sub systems. It is also the first one able to efficiently evaluate complex XPath queries with different types of navigation directions (parent-child "/" as well as ancestor-descendant "//" axis) over the stream of XML documents. While parallelism can be achieved with multi-core machines (as a software-hardware solution), FPGAs offer a viable alternative due to their power efficiency (less power consumption and cooling costs) [167, 85] as well as higher throughput. The work in [74], quantitatively demonstrates the benefits of using FPGAs over general purpose CPUs for streaming applications. While multi-core systems come with 2 and 4 CPUs it is not always feasible to achieve proportional speed-up due to the bottleneck in shared cache memory and the FSB.

Chapter 3

Compiling PCRE to FPGA via opcodes and accelerating SNORT

We discuss our compilation flow for converting PCRE to VHDL. We discuss SNORT IDS and the regular expressions subsysem. We discuss how our tool can be used to accelerate regular expressions from SNORT IDS. In this chapter, we also demonstrates the mechanism of a reconfigurable FPGA based streaming architecture that provides throughput greater than 10Gbps while matching PCREs.

3.1 Regular Expressions, IDS and FPGA Acceleration

Regular expressions are a systematic and effective process to encompass multiple similar strings in to a single expression or a set. Regular expressions are an extension to the string matching process. Unlike string matching wherein we look for a particular string in a given

input stream, in regular expression matching process, the input stream is inspected for the existence of one or more string members of a given regular expression. PERL is a very popular string oriented language with a rich set of regular expressions and it is widely used for both regular expression matching and string matching. Due to its popularity, PERL is frequently used for creating regular expressions based rules for IDS.

3.1.1 PCRE

PCRE is a open source 'C' based software library which is independent of the original PERL distribution. PCRE is solely relegated to the compilation and software execution of PERL regular expressions. PCRE software consists of two parts namely the PCRE compiler and the PCRE Engine. The PCRE compiler compiles PERL based regular expressions into a set of op-codes, which would then be suitable for execution on the PCRE engine. The engine executes the regular expression represented as opcodes with a given string to recognize whether the regular expression matches the string.

3.1.2 SNORT IDS and PCRE

Increase in malicious activities using computer networks as a medium, has also resulted in an increased deployment of intrusion detection systems(IDS) to scan and intercept network packets containing signatures of such activities. The SNORT[146] software is a popular and widely used open source IDS for securing the network of an organization from malicious activities.

Deep payload inspection systems, like SNORT and BRO, use regular expressions for their high expressibility and compactness. Since SNORT IDS uses both fast string matching as well as regular expression matching, we would like to emphasize that this chapter concentrates on accelerating the regular expression matching subsystem of SNORT. The SNORT Intrusion Detection System (IDS) system, in addition to string matching, uses the PCRE (Perl Compatible Regular Expressions) engine for regular expression matching on the payload. The popularity of PERL in the user and developer community makes it highly suitable for creating regular expressions based rules for SNORT IDS.

SNORT uses the software based PCRE engine, which can match the payload with a single regular expression at a time, and needs to do so for multiple rules in the ruleset. Therefore the throughput of the SNORT IDS system dwindles with increasing number of regular expressions, and increasing payload throughput.

In fact, with increasing use of regular expression based rules in Intrusion Detection Systems (IDS), it has been impossible for traditional CPUs to keep up with 10 Gbps throughput due to the high computation complexity involved. Thus it has been necessary to utilize hardware based accelerators for running regular expression engines.

3.1.3 Accelerating PCRE on FPGA

Multiple PCRE engines can be implemented on hardware, which can operate in parallel to inspect, network payload across multiple regular expressions rules. Regular expression rules for IDS are dynamic (due to ever changing network activity levels and scenarios), and thus

is the acceleration hardware needs to be reconfigurable. FPGA based hardware acceleration of regular expression matching is an excellent choice to fulfill both of the aforementioned conditions, since they allow parallel high speed hardware blocks to be implemented and also allow reconfigurability.

To effectively convert multiple regular expressions to VHDL, and thus implement them on FPGA, a compilation framework is required. Since no compilation framework existed that could directly take a PCRE to VHDL, we implemented one, based on the PCRE compiler. The original PCRE compiler generated opcodes for the software based PCRE engine. We adapted the front-end of the PCRE compiler and used the generated opcodes to generate VHDL hardware module, one opcode at a time. Thus our back-end compilation plug-in effectively compiles the PCRE opcodes to VHDL and thus into hardware blocks.

In this chapter we first demonstrate our benchmark performance of regular expression based rules from SNORT IDS using software only execution. We demonstrate the case when 30% or more number of network packets trigger regular expression matching, the software based IDS cannot maintain 10 Gbps throughout, and thus requires hardware acceleration.

In order to demonstrate the efficacy of our compilation flow and the regular expression matching architecture, we have also implemented a proof of concept two hundred PCRE engines based on a plethora of SNORT IDS regular expression rules. These were mapped to the Xilinx Virtex-4 LX200 FPGA on the SGI RASC RC 100 Blade connected to the SGI ALTIX 4700 supercomputing system as a testbed. We obtain an interface throughput of 12.9 GBits/s and a speedup of 353X over software based PCRE execution. We also show that it is

possible to scale down the processing related power consumption of an IDS by two orders of magnitude using an FPGA .

3.1.4 Finite Automaton on FPGA

The basic building block of regular expression engines implemented on FPGA are finite automata. An example of a finite automata is depicted in Figure 3.1. The language of the depicted finite automata matches a string with even numbers of zeros. The states of the finite automata are encoded as '0' corresponding to S1 and '1' corresponding to S2. The first lookup table of size four elements is addressed by the current state of the automata and the current input data. As an example, if the automata is in state S1 and receives an input character '1', the corresponding address in the LUT is '01'. The data stored at address '01' is '0' and is routed to the next LUT on the right which contains two locations. Since the corresponding data at address '0' in the second LUT is S1 hence the finite automata selects state S1 as its next state. The gray section in Figure 3.1 demonstrates the state transitions as the automata processes and accepts an example string "0110100".

3.2 SNORT IDS

In this section we describe the mechanism by which SNORT IDS utilizes the PCRE compiler for translating the regular expression based rules from the SNORT database. We also describe the regular expression operations that are obtained from the rules in SNORT database.

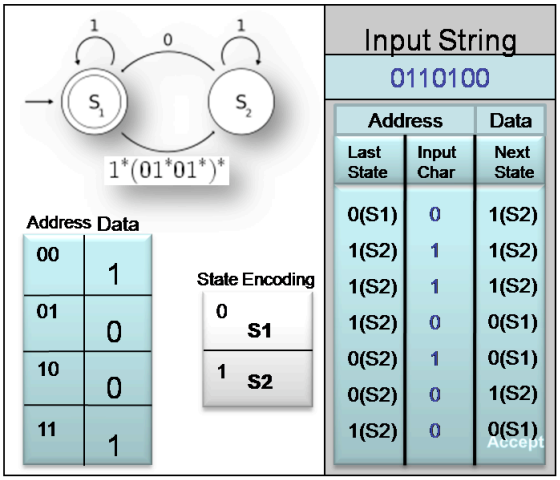

Figure 3.1: A Finite Automata Implemented on FPGA using LUTs.

The SNORT [146][144] software is an is an open source (GPL), popular and widely used IDS for securing the network of an organization from malicious activities. SNORT IDS interacts with the TCP/IP stack on a computer or a security appliance. It identifies signatures of malicious activities such as buffer overflow, denial of service, man in the middle and other attacks on the network packets and thus avoid potential contingencies. SNORT IDS is based on a community driven ruleset wherein the rules are updated frequently by the security community thus capturing the signatures of the newest vulnerabilities and malicious packets.

3.2.1 PCRE rules in SNORT

Intrusion Detection Systems (IDS) such as SNORT and BRO [41] started as string matching engines for deep payload inspection of network packets using a database of signature strings known as the rulesets. As the database of string based signatures expanded, the efficiency of the rules started to dwindle. Regular expressions are therefore being increasingly used

to chart out new rules, due to their higher expressibility and compactness. A single regular expression can encompass tens and hundreds of individual string representations, and thus they have become a highly popular method for constructing signatures for IDS. PERL based regular expressions are being increasingly utilized for charting out the SNORT ruleset due to their compact representation, excellent expressibility and wide usage across the community. The SNORT IDS utilizes a plugin oriented architecture to enable regular expression matching as well as various additional features. The results in this chapter are based on the VRT Certified Rules for SNORT v2.4 dated 2007-09-11. SNORT supports PCRE libraries 6.0 and later, and we have developed our PCRE to vhdl compiler on PCRE lib v6.7. Table 3.1 exemplifies two different PCRE rules from the SNORT database v2.4. More than four thousand such rules make up the SNORT PCRE rulesets. SNORT uses a two-stage process for detecting malicious activity in network payloads. In the standard configuration, all the non-PCRE pattern matching rules are matched simultaneously that are loaded in at runtime into a fast optimized set-wise pattern matching engine. The packets qualified as malicious would then be matched with the relevant PCRE rules depending on the nature of the payload. [145].

The PCRE engine is used as a plugin by SNORT IDS to run a regular expression match on the intercepted payload as depicted in Figure 3.2. Upon encountering a pre-qualified payload SNORT invokes the PCRE compiler on selected regular expressions from the pertinent ruleset. The PCRE compiler takes in a PCRE regular expression rule and compiles it to op-codes. The compiled rule composed of opcodes along with the network payload is run through a PCRE execution engine, and the match result is returned back to SNORT. There-

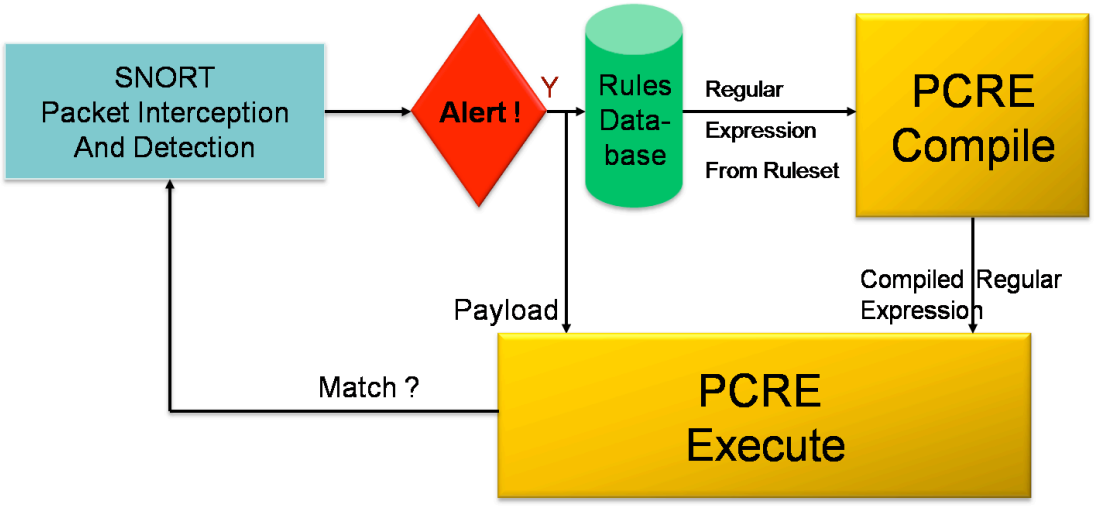

Figure 3.2: SNORT IDS and PCRE Engine usage on a software only Implementation. pcre_compile function compiles the regular expression while pcre _execute function runs the engine of the payload

Table 3.1: Example Rules in SNORT DB 2.4

Ruleset	**Regular Expression Rule**	**Implication**
backdoor	^Netbus\s+\d+\x2E\d+ (Netbus Trojan)	Captures the header of the Netbus trojan i.e. Netbus followed by one or more spaces, one or more digits, character '.' and one or more digits
web-misc	^[^\x3e\x3f\x26]{63} (Buffer Overflow)	Captures a McAfee specific buffer overflow attack sequence i.e. Any 63 characters other than >, ? or &

after the IDS may make additional countermeasures or raise an alarm based on a positive match.

Table 3.2 highlights the format of a typical PCRE rule in a SNORT IDS ruleset. The commonly used PCRE flags include *'i'* for case insensitive match, *'s'* for inclusion of new-lines in the dot operator, *'m'* for enabling anchors to match immediately following a newline,

Table 3.2: Format of a typical PCRE Rule in SNORT IDS with the optional modifiers

pcre:"/<regex>/[ismxAEGRUB]";

Table 3.3: Example snippets from SNORT Rules highlighting the use of PCRE operators

Operator	Snippet	Implication
"^"	^NetBus	Netbus at the start of line
"{}"	[\x26]"{**63**}"	'&' Exactly Sixty Three Times
"{**n,x**}"	[^\n]"{**244,255**}"	Any character but newline, more than 244 but less than 255 times
"*"	[^\r\n]*	Any character but CR,LF Zero or more number of times
"+"	\s+	White Space, one or more number of times
"**1,****2,****3,**" "**4,****5,****6** "	(\x22 | \x27) ... \1	If ' " ' i.e. (\x22) was matched earlier then match ' " ', **alternatively** if ' ' ' i.e. (\x27) was matched, then match ' ' '
"[^**...**]"	[^\r\n]	Any character but CR,LF

and *'x'* to ignore whitespace between regular expression token.

3.3 Compiling PERL Compatible Regular Expressions to FPGA

In this section we present information on PCRE and our method of compiling PCRE opcodes directly to VHDL for parallel implementation on FPGA hardware.

PERL is a very popular string oriented language with a rich set of regular expressions, thus making it highly suitable for creating regular expressions based rules for SNORT IDS. The PCRE software consists of two parts: the PCRE compiler and the PCRE Engine. The PCRE compiler compiles PERL based regular expressions into a set of op-codes, which are then executed by the software based PCRE engine. The engine executes the regular expression represented as opcodes with a given string to recognize whether the regular expression

matches the string. [1]

The PCRE engine executes a greedy quantifier matching NFA which conforms to the PERL regular expression semantics. The important PCRE operators including the anchors, ranged quantifiers, repetitions, back-references, character classes, and their occurences in the SNORT rulesets are highlighted in Table 3.4 and example rule snippets demonstrating their behavior is shown in Table 3.3.

Regular expressions can produce different results depending on the execution engine. As an example a greedy NFA would provide a different result when compared to a non greedy execution engine. PERL uses an NFA-based greedy quantifier match strategy as default and the SNORT rules have been generated by the community to adhere to the PERL regular expression standards. Thus it is extremely important for the accelerated regular expression engine in hardware to adhere to the PERL regular expression standards in order to successfully detect malicious activity. It is possible that non-conforming implementations may result in false negatives, which could result in potential security issues. As an example a greedy quantifier regular expression engine, using the regular expression **/test.*test/** on the string "This test is testing greedy and lazy tests", would match and return "*test is testing greedy and lazy test*" while a lazy quantifier match would return "*test is test*" i.e. up until at the fourth word which is "testing".

We implement the PCRE regular expressions to hardware using a two stage translation process. In the first stage, the regular expression is compiled using the PCRE compiler to

[1]The current revision of PCRE library is v7.2 as of writing this chapter. Since we had already started work with PCRE v6.7 and the differences between the versions are minor, we have limited ourselves to PCRE v6.7.

Table 3.4: Occurences of important PCRE operators in our target SNORT DB 2.4

Regular Expression Operator	**Occurences**	**PCRE Opcode**
Character Match	8117	OP_CHAR 21
Repetition Star "*"	6276	OP_STAR 33
Character Class "[ ]"	2912	OP_CLASS 59
Alternation "|"	2365	OP_ALT 65
Ranged Quantifiers "{**n,x**}"	2011	OP_UPTO 30
Repetition "**+**"	1218	OP_PLUS 26
Back References "**\1,\2,\3,\4,\5,\6** "	262	OP_REF 62

generate PCRE opcodes. A regular expression is translated by the PCRE compiler into an unique series of regular expression opcodes that are processed by the PCRE engine with the input data. To that end, we modified the PCRE compiler v6.7 by adding an opcode dump module to the PCRE compilation library. In the second stage the PCRE opcodes are translated to VHDL hardware blocks suitable for implementation on FPGA and connected together using a NFA based control logic. Our system maintains the execution semantics of the software based regular expression engine on the FPGA hardware, thus ensuring compatibility with software based PCRE execution. We then process the generated opcodes and related operands using successive compilation steps and convert them to VHDL blocks. The VHDL opcode blocks are tied together in an NFA and are synthesized and implemented on the FPGA.

3.3.1 PCRE Opcodes

The PCRE opcodes are defined in the *pcre_internal.h* file which is part of the PCRE library package. These opcodes are instructions for the software based PCRE engine. Each of the PCRE opcodes have an equivalent hardware implementation. As discussed earlier, since the

hardware implementation on an NFA is inherently parallel in nature, it is possible to process one new character every clock cycle. The match output of the preceding opcode block in the NFA engine enables the succeeding opcode block. The input character received each cycle is copied to the input of each opcode. Moreover the output character bus of the opcode can be used to gather the matched string from each of the opcode for debugging purposes. The most important and commonly occuring opcodes in the SNORT rules database can be classified into three groups: character-match opcodes, repetition opcodes and quantifier opcodes. We detail the implementation details of these opcode groups, in the following four subsections.

3.3.2 PCRE Opcode Frequencies in SNORT Rules

The PCRE opcodes are at the heart of the machine that matches a particular regular expression in a packet. Using the SNORT v2.4 database we have measured the frequency of occurrence of the various opcodes. These results are shown in Figure 3.3. It can be observed that that following opcodes namely *character-match*, *star*, *character class*, *alternation*, *quantifier*, *plus*, and *back reference* make up more than 90% of the total opcodes. Therefore we concentrate on the hardware development and optimization of these opcodes. A cumulative distribution of PCRE opcodes in SNORT database is charted out in Figure 3.4. Currently our PCRE to VHDL compiler is limited to the aforementioned opcodes tabulated in Table 3.4.

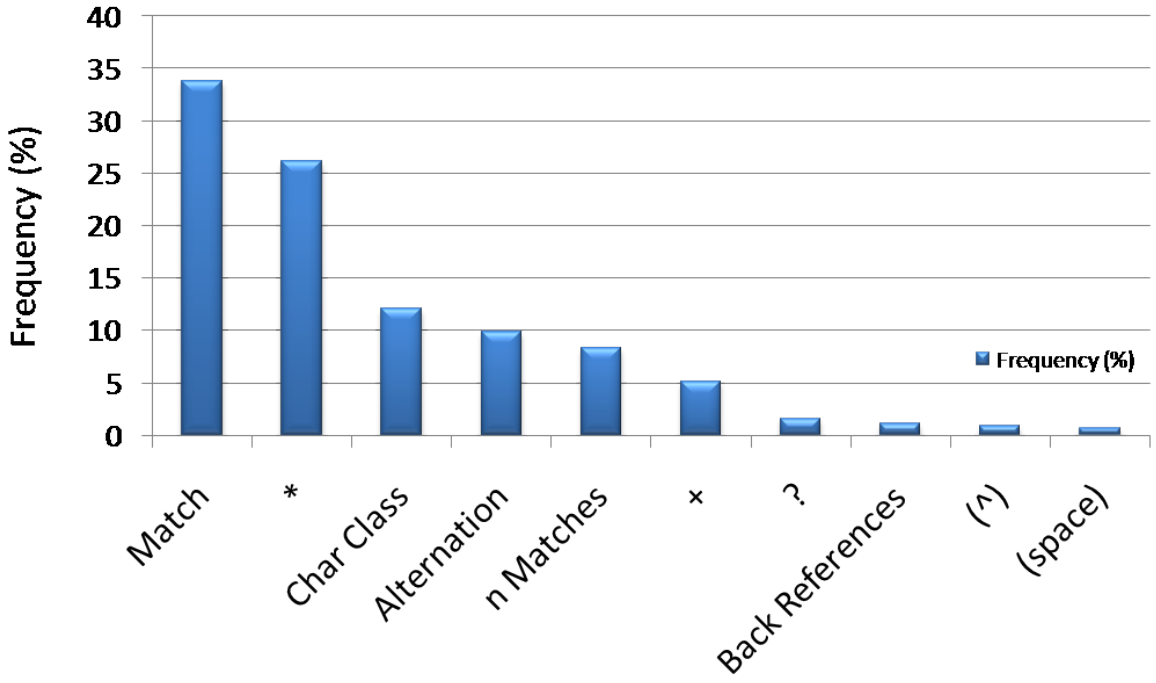

Figure 3.3: Frequency Distirbution of PCRE opcodes in SNORT DB 2.4. The most frequently occurring opcode is the character-match opcode.

3.4 Compilation Flow

At the heart of our automated compiling system is the conversion of regular expressions from the SNORT database to engines on FPGA. We detail the implementation of our tool that translates PCRE code into hardware that is mapped to an FPGA for accelerating regular expression matching subsystem of SNORT. Our compiler generates VHDL code corresponding to the opcodes generated for the SNORT regular expression rules. We have tuned our hardware implementation to utilize an NFA based regular expression engine using greedy quantifiers in much the same way as the software based PCRE engine does. Our system implements a regular expression only once for each new rule in the SNORT ruleset thus

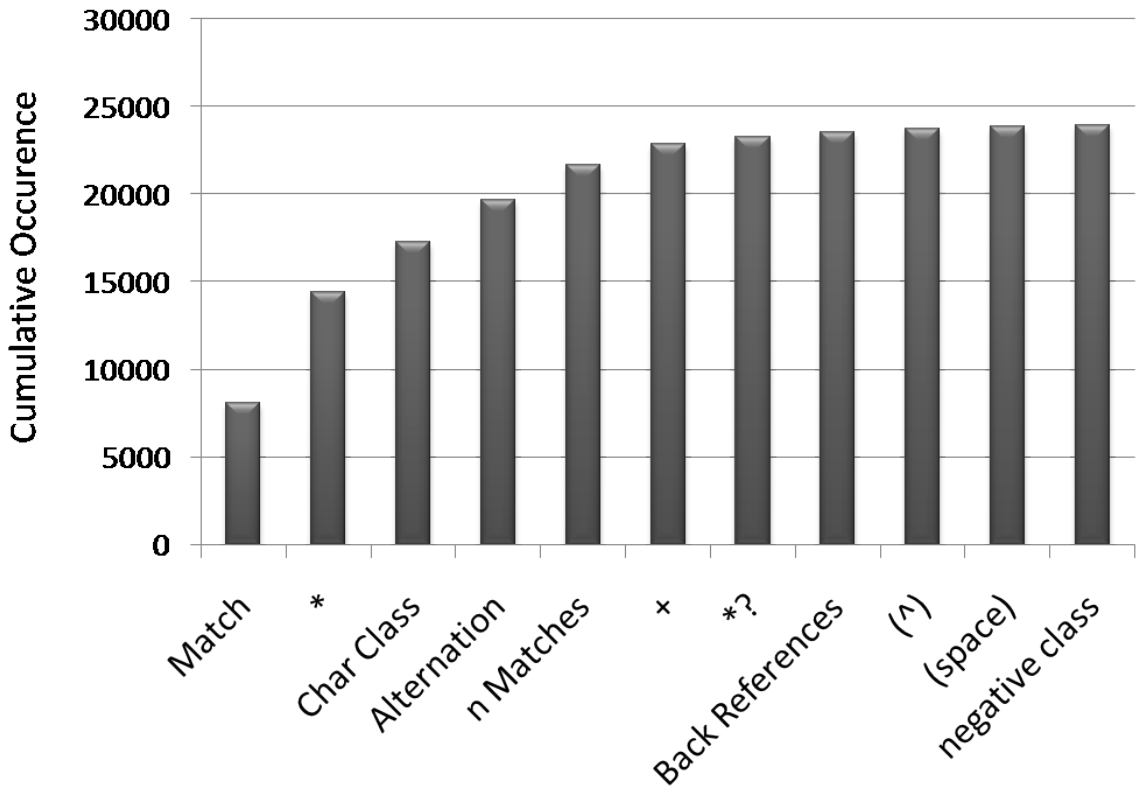

Figure 3.4: Cumulative Distirbution of PCRE OPCODES in SNORT DB 2.4. The five OPCODES viz. Match, star, Character Class, Alternation and constrained repetition make up for the most frequently occurring cases of OPCODES.

resulting in a fast system that scales well with new updates.

3.4.1 Compilation Overview

The SNORT IDS accesses the rules by rulesets when enabling PCRE based IDS. Each of the rulesets are available as separate files in the available SNORT database. As a first step, our compiler script extracts all rules from the SNORT database that have a *pcre* field and stores them into local ruleset files for further processing. These rules contain the various regular expressions which are used by SNORT IDS.

The compilation starts with extraction of opcodes and operands from PCRE rules. Thereafter the extracted opcodes and operands are processed by the opcode to VHDL compilation script, to generate opcode blocks such as comparators, counters, etc. The next script processes the sequence of opcodes and generates a NFA based control logic in VHDL. A final script combines them to a VHDL entity that interacts with the memory interface module.

The initial compilation step involves invoking PCRE compiler v6.7 that translates the SNORT REs into PCRE opcodes. We have added a mechanism for the PCRE compiler to emit compiled opcodes from regular expressions to a local database. An AWK script parses the database and extracts the opcodes and operands corresponding to the regular expression based rules.

In the second step, scripts are invoked to extract case-sensitive character-match opcodes from the regular expression. The case sensitive character-matching blocks are implemented as comparator circuits on hardware. These blocks receive characters from the payload mem-

ory via the memory controller module. Alternation opcodes involve unrolling the NFA into multiple parallel character match paths. Also extracted are the corresponding token / character match blocks for repetition, quantifier and back-reference opcodes. Finally the counters, quantifiers and back reference opcodes are generated and connected to their respective character match blocks.

In the third step, a compiler script combines the generated opcode blocks, into one NFA engine, that corresponds to the original regular expression. In this stage, the opcodes defining the regular expression operators are iteratively analyzed by the script, and the control structure for the NFA is generated in VHDL. The NFA generating script converts the sequence of opcodes into an extensive set of *if - else* statements in VHDL. The generated control structure is a series of flip-flops that enable an opcode block, based upon the match output of the previous opcode. Additionally the NFA controller enables the write signal on the back-reference opcode when the reference is first matched, and thereafter enables the read signal when a back referenced is enabled.

In the fourth step, a script generates a payload buffer and a matchdata buffer. The payload buffer receives TCP/IP payload from the software and inserts a character into each regular expression bank every cycle. The matchdata buffer is connected to the match output of each of the NFA engines, and the data is sent back to the software once the complete payload has streamed through the regular expression engines.

The NFA control structure is thereafter tied together with the opcode blocks in a single VHDL file. The VHDL files are collected together and are mapped to the payload buffer via

the memory interface module.

3.4.2 Common Prefix Optimization

Since the SNORT IDS regular expressions are based on a collection of similar rules grouped by the rulesets, some of which contain regular expressions that share a common prefix. These prefixes are a potential point of design consideration which may lead to conservation of on-chip area, and are discussed in [101]. We extracted the common prefixes from regular expression in the rule sets of SNORT IDS, and compiled them together into a single hardware block. The optimized design resulted in a savings of $\sim$26% area on the chip. However, its impact on the circuit clock frequency was more than 50% lower because of the resultant circuit size that implements the prefix tree. We have thus chosen not to enable the Common Prefix Optimizations since it constrains the digital clock routes on the FPGA to operate at a lower frequency, thus affecting the throughput of the regular expression engines.

3.4.3 Hardware Implementation of PCRE Opcodes

In this subsection we describe the detailed implementation of PCRE opcodes on FPGA hardware. We compare two important implementation paradigms i.e. utilizing IP Core based opcode generation and secondly by synthesizing the opcodes from VHDL code. We provide data on the variation of area and speed of important opcodes generated using the two paradigms. We provide additional details with respect to quantifier opcodes. We provide data on the size of counters required to deal with the specific range of counts that occur after the

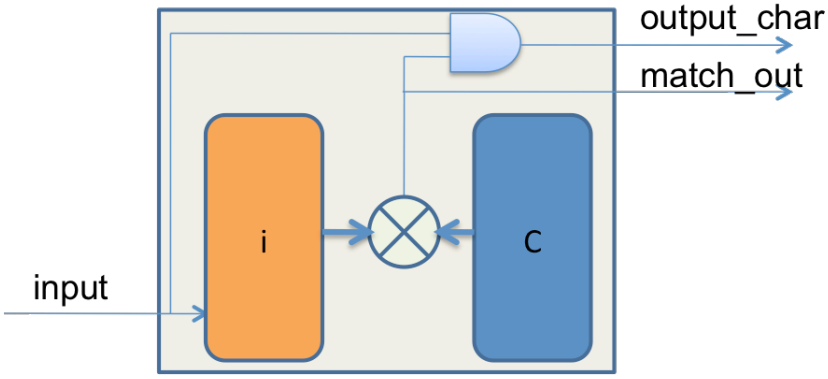

Figure 3.5: The Character(s) Match Opcode. The input (i register) is compared with the data in the ROM and the output (match_out) is triggered high in case of a match.

compilation of SNORT rules.

Character-matching Opcodes

The character-match opcode is a comparator hardware blocks which match a single character or a string. Since the PCRE hardware is not modified at run-time, we can use ROM based hardware comparators which have smaller area and faster speed than RAM based comparators. The structure of the character-match opcode is shown in Figure 3.5.

We compare the performance (clock speed) and areas of the character-match opcodes using two implementations: the Xilinx Logicore IPCORE and synthesized VHDL. Our building blocks consist of 1, 2, 4, 8, 16 and 32-byte ROM based comparator blocks. As depicted in Figure 3.6, the IPCORE implementation is marginally faster than synthesized VHDL, mainly because IP Cores are heavily optimized by the FPGA vendors. The tradeoff in sizes can be seen in Figure 3.7: the IPCORE blocks consume marginally more areas than the synthesized versions. All the speed and area figures have been reported using the Xilinx ISE 9.2i tool af-

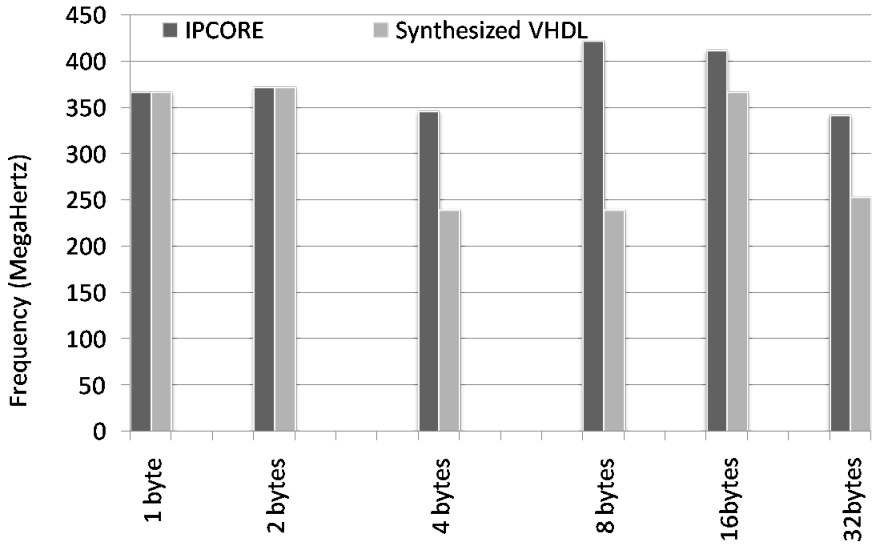

Figure 3.6: Implementation Speed in (MHz) of the character-match opcode versus character size sets for the two implementation types. Implementing using IP Core provides a faster clock speed, as compared to the synthesized design for match sizes greater than 2 Bytes.

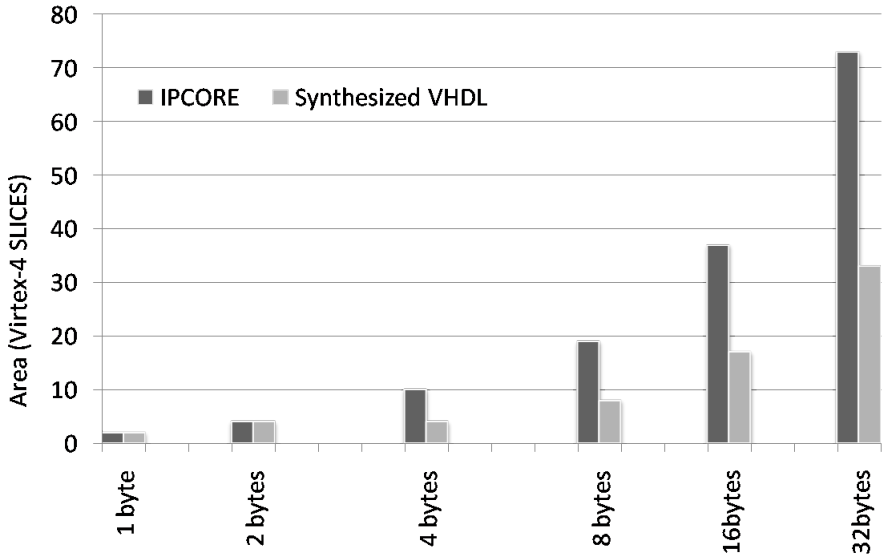

Figure 3.7: Implementation area in FPGA slices of the character-match opcode versus character size sets for the two implementation types. The area of the opcode block increases linearly with increasing match size. Moreover the synthesized hardware is more area efficient as compared to the IP Core based block for a given match size

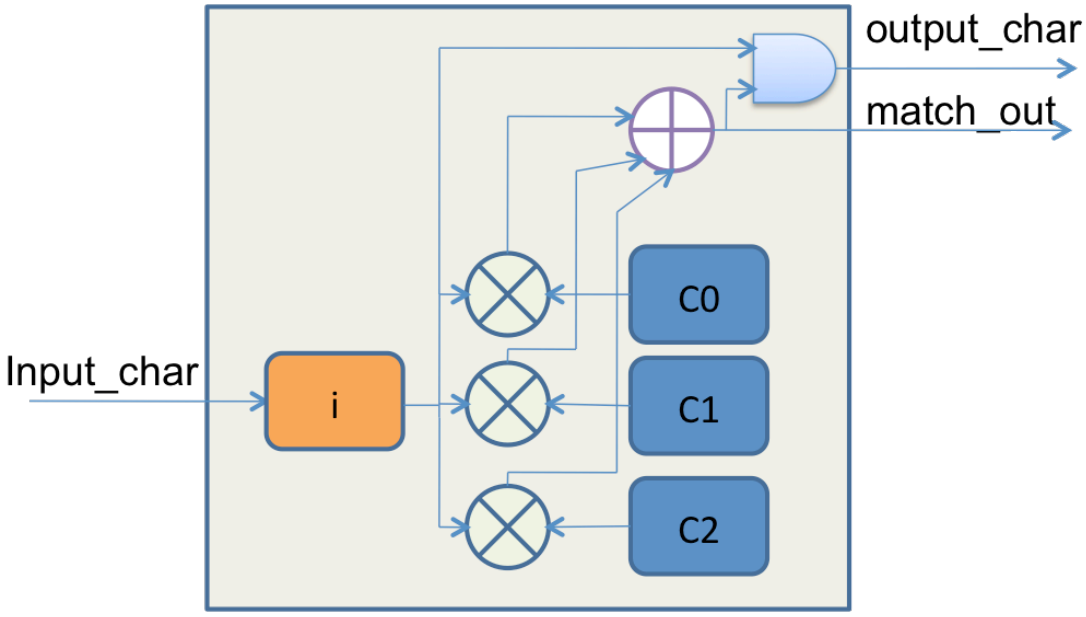

Figure 3.8: The Character Class Match Opcode.

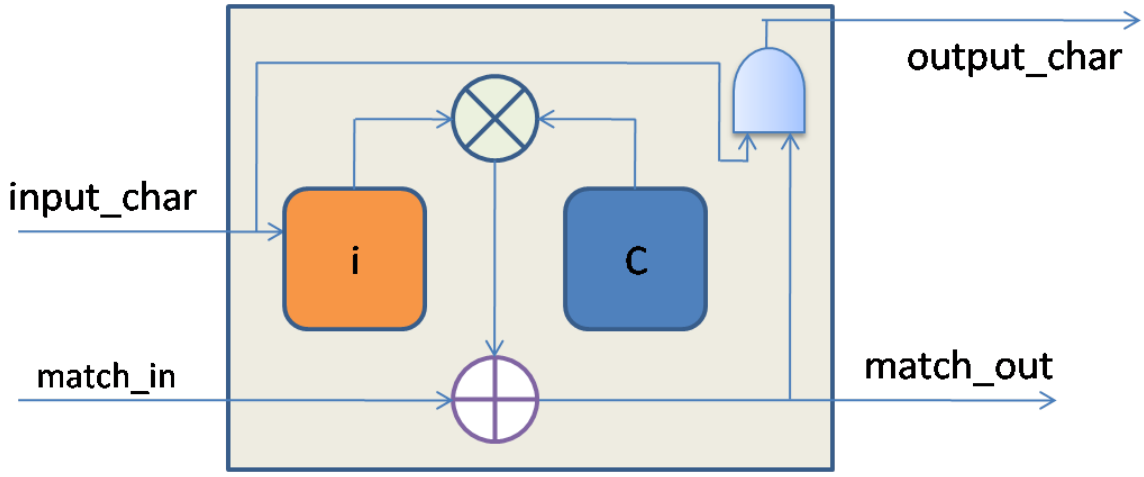

Figure 3.9: The * repetition Match Opcode.

ter the Implementation Flow (i.e. hardware mapping and place and route). We have utilized the IPCORE based comparator blocks in our character-match opcodes.

The character class match opcode, shown in Figure 3.8, essentially consists of multiple comparator blocks the output of which is enabled high when either one of the comparator matches.

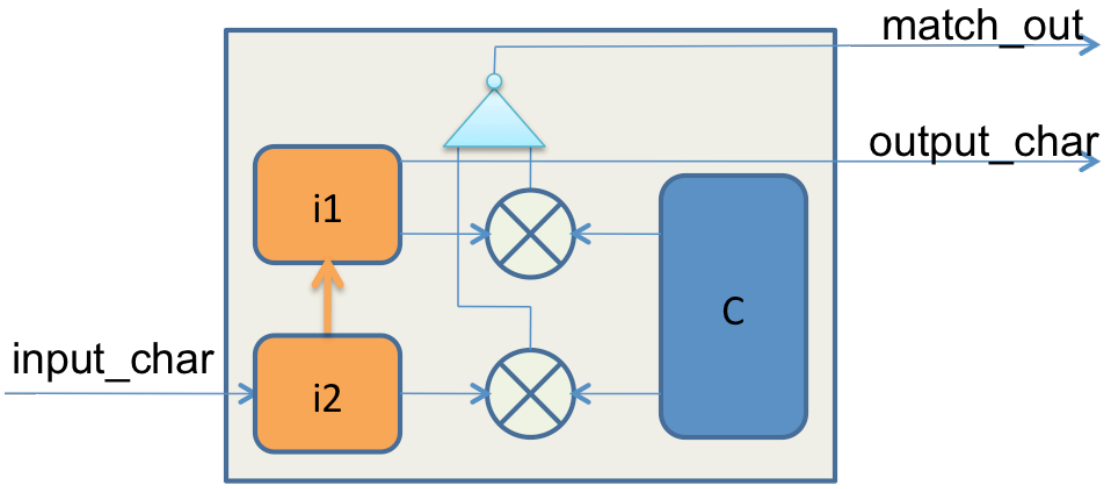

Figure 3.10: The + repetition Match Opcode.

Repetition Opcodes

The star repetition opcode, depicted in Figure 3.9, matches 0 to any number of the given character(s) from its input. The plus opcode, shown in Figure 3.10 matches one or more number of the given character(s) from its input. Thus the plus opcode enables the succeeding opcode after matching a repeating series of the given character / string. These opcodes are essentially built from the character-match opcodes with additional logic to implement the repetition. An important feature of PERL and thus PCRE is that, given a input and the regular expression, only the first match (left most) is reported. This feature simplifies the generated hardware with the star operator.

Quantifier Opcodes

Quantifiers are used in the PCRE language semantics to indicate quantifiable repetitions. An example quantifier opcode block is depicted in Figure 3.11. The quantifier opcode may be

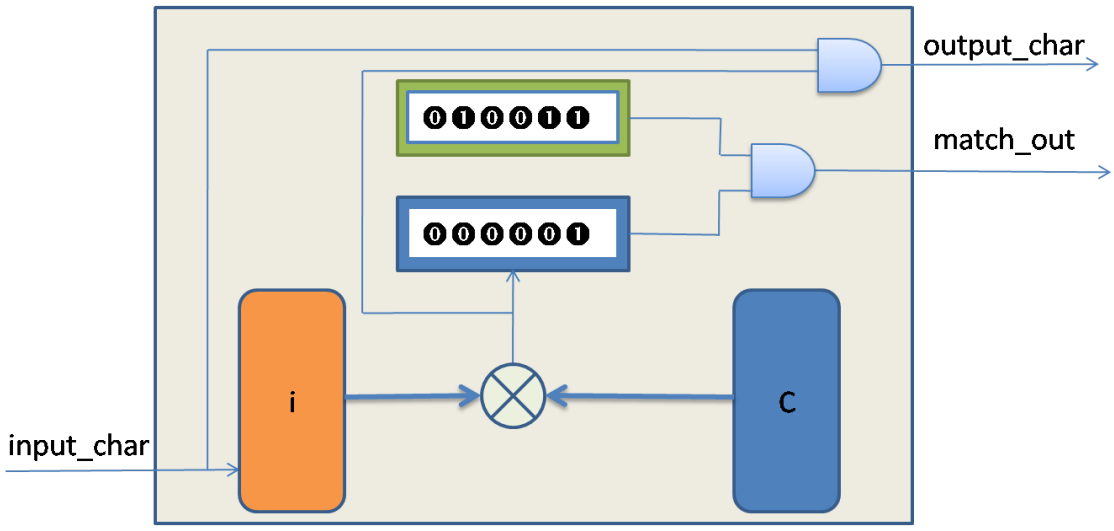

Figure 3.11: The quantifier Match Opcode.

Table 3.5: Simple Quantifiers Occurrence Table

Count Interval	Occurrence	Min	Max
0 - 10	7	{1}	{9}
10 - 99	49	{10}	{69}
100 - 999	109	{100}	{512}
1000 +	5	{1024}	{1024}

Table 3.6: Ranged Quantifiers Occurrence Table

Count Interval	Occurrence	Maximum Range
0 - 9	2	{0,1}
10 - 99	0	N/A
100 - 999	2	{1,221}

classified into three categories viz. simple quantifiers, ranged quantifiers and unbounded quantifiers. Using quantifiers we may specify either exact repetitions as in {n} ('n' is a constant which indicates the number of repetition) or repetitions of characters with at least one bound (lower and / or upper) as in {lb, ub}. As an example, the simple regular expression "a{10}" indicates matching exactly 10 occurrences of the character 'a'. We may also quantify a regular expression token like, "[ab]{10}". This regex would match a string of 10 characters made up of any combination of 'a' and 'b' in the string. We follow the PCRE greedy match quantification rules while matching using quantifiers i.e. we return the first possible match. A detailed discussion of various semantics of regular expression quantifiers, other than the one used by PERL would exceed the scope of this chapter and the interested reader may refer to [162]. Simple quantifiers compute the repetition of the given character, string or a token, a predefined/exact number of times. The occurrences of simple quantification in our test SNORT database is tabulated in Table 3.5 in decimal ranges. The majority of the cases are the counts between (100 - 199).

Ranged quantification involves the repetition of the given token in a pre-determined, bounded range of occurrences. Table 3.6 shows that there are only four occurrences of ranged quantifiers in the test SNORT database.

The most frequently occurring quantification is that of the unbounded quantification

Table 3.7: Unbounded Quantifiers Occurrence Table

Count Interval	Occurrence	Min	Max
0 - 9	1	{3,}	{3,}
10 - 99	12	{14,}	{71,}
100 - 999	43	{100,}	{512,}
1000 +	1857	{1000,}	{1075,}

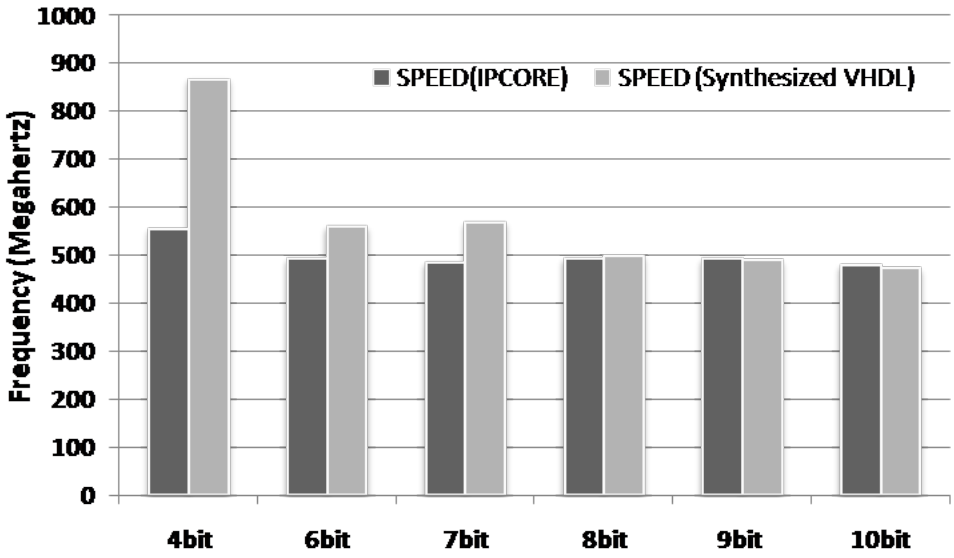

Figure 3.12: Speed of unrestricted counters in (MHz) with increasing count size and the two implementation types. For counts until 8 bit the synthesized counters are faster, while for 9 and 10 bit counters, the IP Core is slightly faster.

which involves repetition of 'n' or more number of times, where 'n' is a constant related to the given instance of an unbounded quantification. As an example the regular expression "\x5C{1023 , }" would match a series of 1023 or more of the character Hex '5C' viz. the '\', back-slash character. As seen in Table 3.7, the counting range of unbounded quantifiers start from (3 and above) and ends at (1075 and above). The count interval of (1000 and above) is the most frequently occurring case in the test SNORT database.

The essential hardware unit for the quantifier opcode is the binary counter. An n-bit

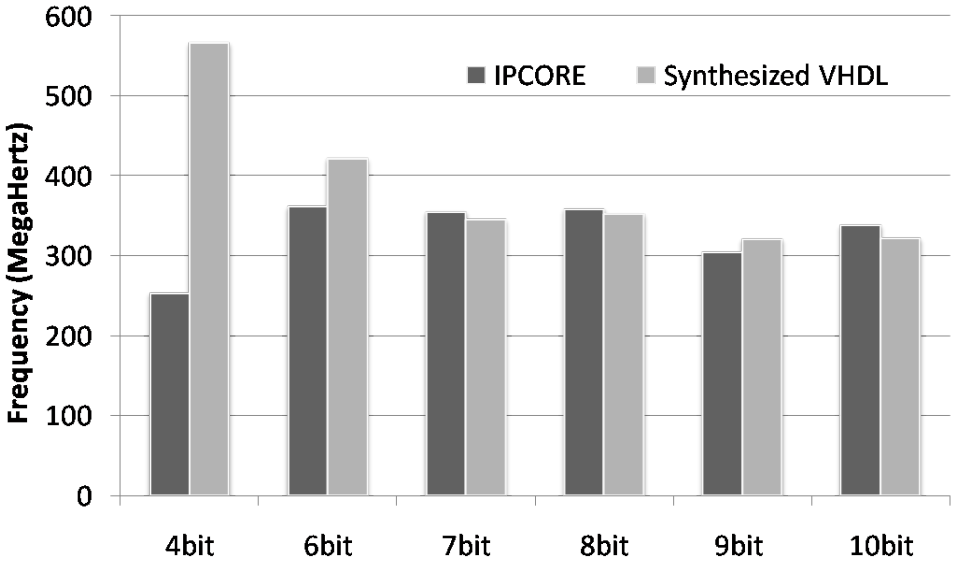

Figure 3.13: Speed of restricted counters in (MHz) with increasing sizes and the two implementation types. The 4 bit and 6 bit restricted counters are faster when synthesized while the performance of the synthesized and IP core is comparable for higher count sizes

binary counter can count from 0 to 2^n - 1. Since the counting range of many opcodes may be less than 2^n - 1, we need to add additional combinational logic to restrict the count range. Therefore the size of such counters is greater than unrestricted binary counters.

We implemented unrestricted as well as restricted versions of 4, 6, 7, 8, 9 and 10-bit counters using both IPCORES and synthesized VHDL. The clock speed of the 4 bit unrestricted as well as restricted counter generated from synthesized VHDL was considerably higher than the IPCORE based one, as depicted in Figure 3.12 and Figure 3.13. The other counters viz. 6, 7, 8, 9 and 10-bit showed marginal difference in performance based on the implementation method. The area of the synthesized counters was higher than the IPCORE counters, for the case of unrestricted counters as shown in Figure 3.14. In case of the unrestricted counters, the 10 bit synthesized counter occupies relatively larger area when compared to the IPCORE ba-

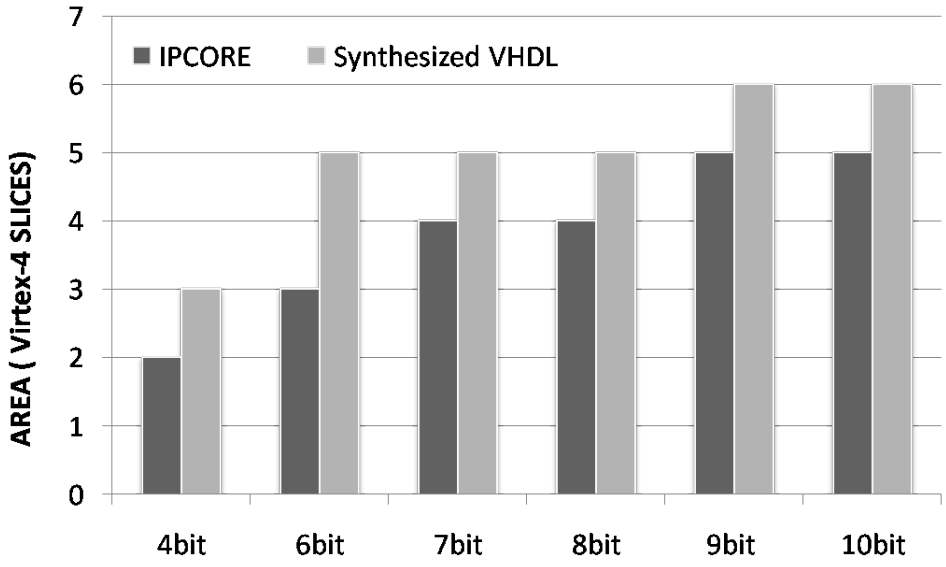

Figure 3.14: Variation of Area of un-restricted counters on FPGA with increasing count size and the two implementation types. The area of the counters implemented using IP Core is lesser when compared to the area of synthesized counters.

sed counters, as shown in Figure 3.15. The 10 bit unrestricted counter which is also the most frequently occurring (Table 3.7) is a good candidate for an IPCORE based implementation due to the compact area and similar performance vis-a-vis synthesized counter.

Based on the performance data we utilize the synthesized counter for the case of 4 bit counters (both restricted and unrestricted) while we utilized the IPCORE based ones for all other ones.

Back References Opcode

The PCRE engine allows for back-references to be used primarily for the convenience of recollecting the marshalling / enclosing meta-characters which are useful for payloads containing programming language specific constructs. As an example, some programming lan-

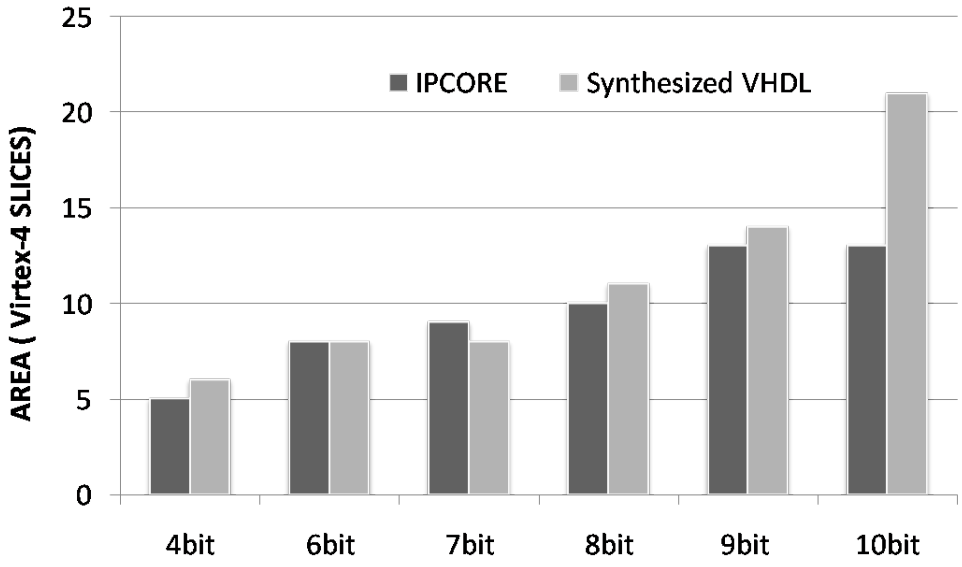

Figure 3.15: Variation of Area of counters with restricted count on FPGA with increasing count size and the two implementation types. Restricting the count i.e. counting before the maximum range of the counter, imposes substantial penalty on the size of the counter which results in more than doubling the area of counts up to 8 bit and almost triples the area for the 9 and 10 bit counters.

guages allow strings to be enclosed either in single quotes ‘ ’ or double quotes “ ”. A back-reference on the enclosing quotes would store the opening quote in the memory, and would recollect it while encountering the closing quote. The contents of the matched back reference can be stored in a RAM, to be referred later on in the payload.

Two rulesets in the SNORT rules database, namely the oracle and the web-client rules consist of 219 rules that include 262 back-reference opcodes. Such a long list of back-reference opcodes is unsuitable for storage in the distributed logic on the FPGA, and due to its large size, would cause over-mapping of logic resources. Our design uses the Block RAM for storing back reference data for the aforementioned rulesets, thus mitigating the problem of overmapping. While generating VHDL code of the regular expression engine from the

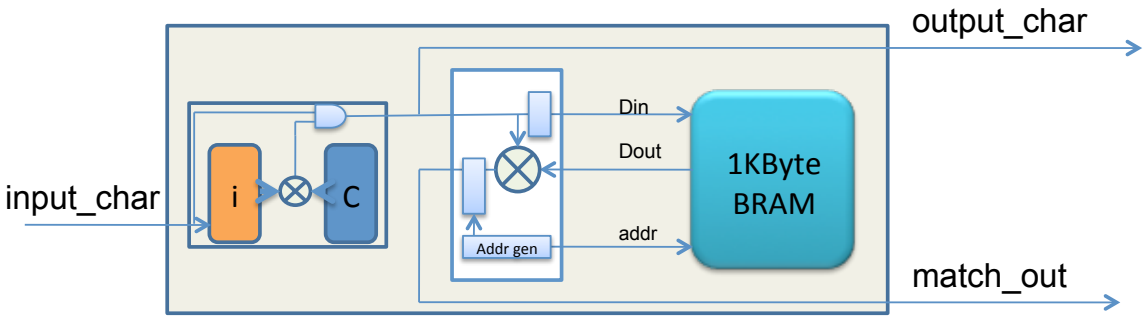

Figure 3.16: The backreference Opcode.

PCRE opcodes, our compiler utilizes Block RAM for local storage of back reference. An example back reference opcode block containing the comparator opcode and the associated BRAM shown in Figure 3.16. The area cost of the back-reference opcode is 8 slices, plus any additional area required by the comparator opcode which generates the back reference data.

3.4.4 NFA Implementation on FPGA

A software implementation of regular expression engines process a single instance at any one time on a CPU core. The FPGA implementation allows for multiple engines processing regular expression matching to operate concurrently as shown in Figure 3.17, thus increasing the throughput. Various optimizations to the engines are possible, including sharing of prefix, common subexpressions and constant matches in the hardware [101] along with choice of engines implemented as a DFA or as a NFA. Moreover, since the logic design on an FPGA can be updated as and when required, it makes them an ideal platform for supporting newer or updated regular expressions, and thus any modification to the SNORT IDS ruleset would

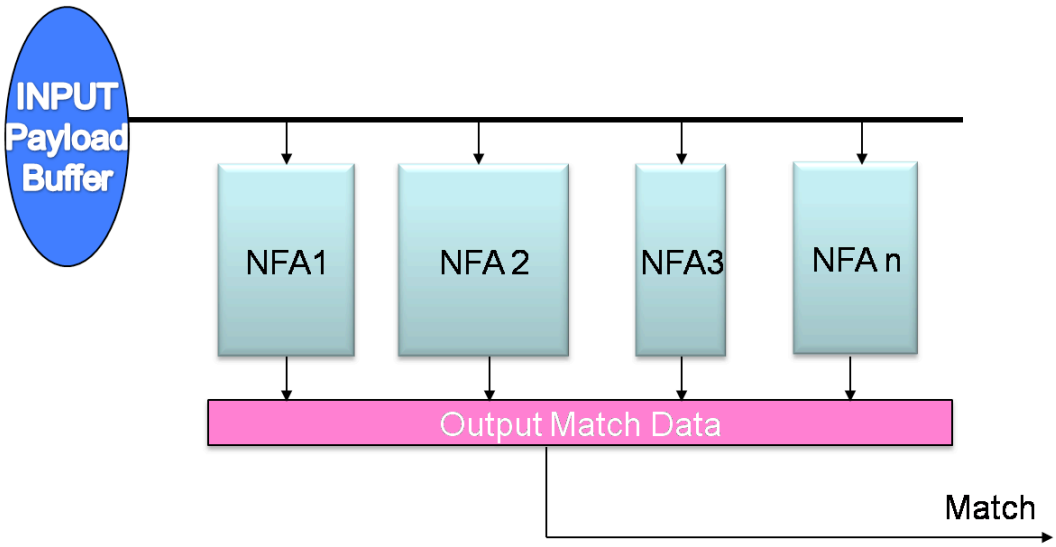

Figure 3.17: Multiple NFA engines executing in parallel on a FPGA.

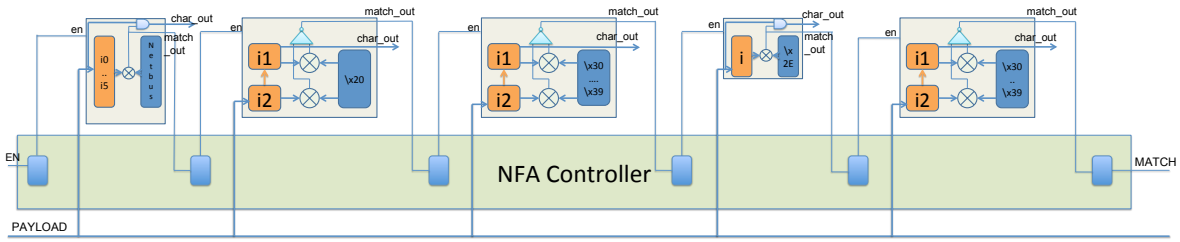

Figure 3.18: The NFA derived from the SNORT Rule 'NetBus\s+\d+\x2E\d+'. This NFA occupies 71 slices and can run at 331MHz. The NFA controller implements flip-flops to enable subsequent stages in the NFA and generates the match output at the final flop. The NFA controller occupies 2 slices in this regular expression.

result in the re-compilation of the new or modified rule only. Due to the inherent parallelism of hardware implementations, NFA based regular expression makes the most practical sense on FPGA. Figure 3.18 depicts the internal organization of an NFA generated from a SNORT rule implemented on the FPGA that takes one character as input every clock cycle.

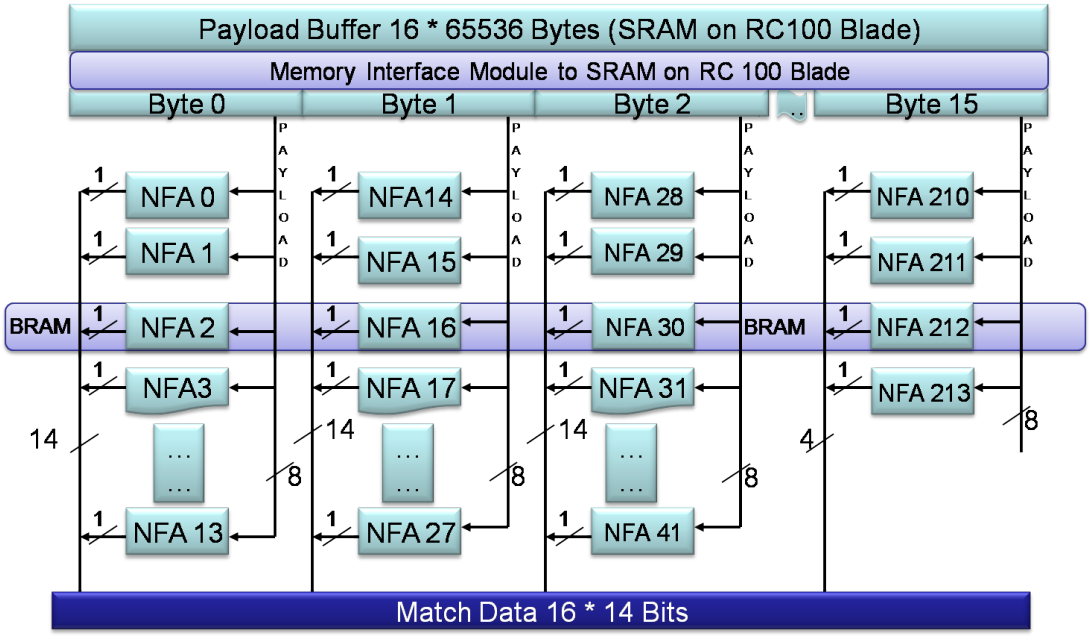

Figure 3.19: Architecture of parallel PCRE Engines on Virtex-4 LX 200 FPGA. Each of the sixteen byte-wide bank obtains a character from the Memory Interface Module and sends them to the 14 NFA engines on that bank. The BRAM is utilized by NFAs implementing the back reference opcode.

Implementation Architecture of PCRE engines on FPGA

PCRE engines from the SNORT rulesets are aggregated together in a single FPGA design, to be implemented and run in parallel. As depicted in Figure 3.19, 214 NFA engines can fit on a single FPGA chip. The NFA engines receive a character every clock cycle on the 8 bit payload line. The RASC RC 100 Blade allows for 128 bits to be retrieved from the Payload buffer each clock cycle, thus a total of sixteen separate payload lines are generated, that operate in parallel. As depicted in the architecture, the FPGA processes sixteen parallel payload banks, and each bank except bank 16 has fourteen parallel NFAs. The banked regular expression architecture is extendible to multiple FPGAs. It is possible to duplicate regular expression banks on multiple FPGAs to improve the throughput by executing parallel regular

expression scans on multiple payloads. The memory interface module is an SRAM memory controller that interfaces to Memory 1 on the RC 100 Blade. Memory 1 serves as a Payload Buffer that receives the data worth sixteen payloads from the host CPU on the SGI Altix 4700. The payload buffer needs to store 16 * 65,536 Bytes or 1 MBytes, since the maximum size of each payload is 64 KBytes, and 16 of them are processed in parallel on the FPGA. The FPGA based PCRE engines containing regular expressions with back-references are allocated memory space on the on-chip Block RAM. Each PCRE engine generates a 1 bit output data that represents whether the payload matches the regular expression. A total of 224 bits are transferred back to the host upon completion of streaming of the packet through the PCRE engines on the FPGA. The 224 bit match data is then decoded at the host to obtain the individual match status of each regular expression engine.

Using the RASC hardware, it is also possible to send priority encoded match data in 16 groups, each consisting of 4 bit encoded data, as output every clock cycle. Such a method can help capture multiple regular expression matches on the given payload data. In order to process this data, we need to allocate additional software processing routines that would decode the output data every clock cycle. A set of API known as RASC Abstraction Layer manages the data transfer between the host processor on SGI Altix system and the RASC RC 100 Blade. The block diagram in Figure 3.20 depicts the top level integration of the SGI RASClib (RASC Abstraction Layer) APIs and SNORT IDS for executing PCRE matches in hardware.

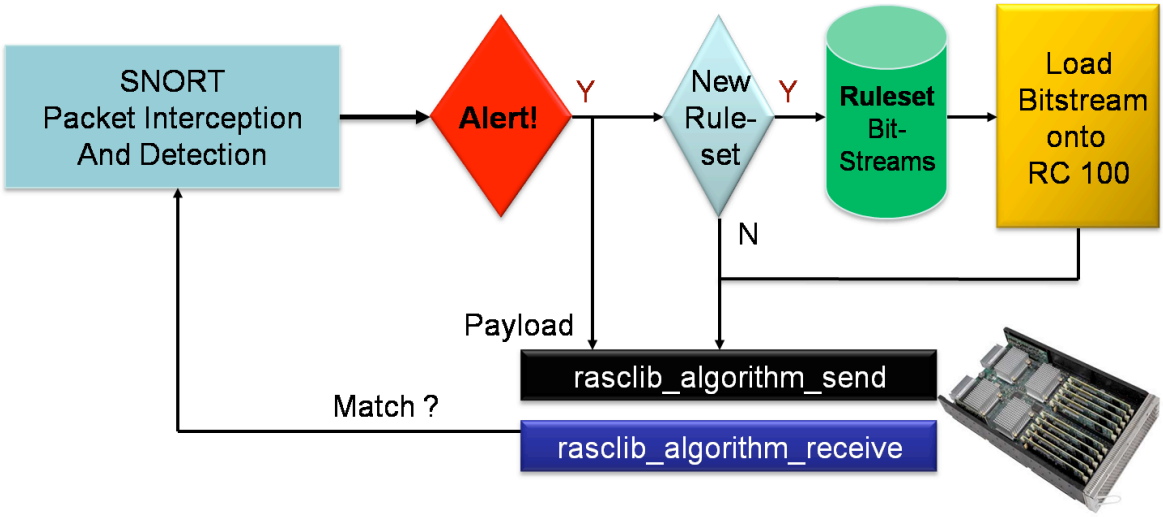

Figure 3.20: Overall system using SNORT IDS and PCRE Engines on FPGA

3.5 Experimental Results

We perform comprehensive performance analysis for both the software only case on various computing platforms, as well as the case when a RASC blade is available on the SGI Altix 4700 system for accelerating regular expression engines on hardware. We run our experiment on a set of network dump of size 10 GBytes. Our network dump file has been generated by running *wireshark v1.0* on the SGI Altix 4700 system, over a period of time, resulting in a collection of TCP payload data. The payload data is split with tcp-split and then exported as C-Arrays file using *wireshark*. Each TCP payload is parsed from the C-Arrays source file and stored into a final payload dump file. We load the payload file on the main memory of the Altix machine before running our experiments. The payload data consists of, in decreasing order of frequency, web / html data (∼50%), ftp data (∼30%), ssh payload (∼15%), and other miscellaneous network data.

3.5.1 Software only performance with multi-cpu load balancing

We have used three different 64-bit computers for testing the software execution of PCRE engine with different regular expressions. The first system is a dual core X86-64 desktop with an Intel Core2 Extreme X6800 processor (Conroe) clocked at 3.5GHz with a 1066MHz FSB and 4.0GBytes RAM. The second system is a X86-64 SGI Workstation consisting of two dual core Intel XEON 5160 processors (Woodcrest) clocked at 3.0GHz with 1333MHz FSB and 16.0GBytes RAM. The third system is the DSM IA-64 SGI Altix 4700 system with 16 Itanium 2 (Montecito) processors clocked at 1.6GHz with 64GBytes RAM. We test the software throughput by evaluating the payload data with a total of twelve regular expressions, obtained from the SNORT database across three different performance scenarios. We have selected regular expressions from backdoor, ftp, web-client and web-cgi rule-sets. Table 3.1 exemplifies two PCRE rules, that are among the rules we use in our benchmark. The performance of the software bases PCRE execution engine is highly sensitive to the actual payload. When the payload contains data that cannot match the regular expression, or is irrelevant to a particular regular expression, the throughput delivered by the software is excellent, and vice versa.

We have invoked PCRE engines on all the available CPU cores, except for the SGI Altix 4700, where we have invoked 30 threads, since 2 threads are used by the RASCLib APIs to transfer data to/from the two FPGAs. We compiled the PCRE v6.7 library using GCC v4.0.2 on the X86-64 systems and using ICC v9.1.042 on the SGI Altix 4700.

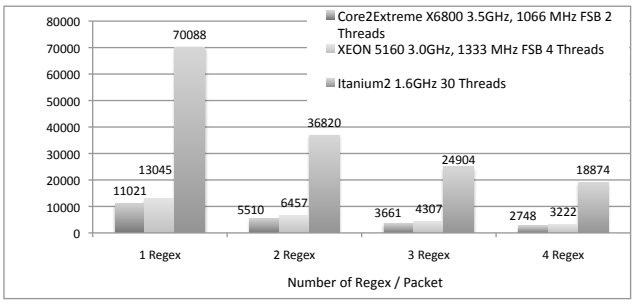

Figure 3.21: Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of minimal malicious activity in the network payload dump.

Software Benchmark with Minimal Malicious Activity

In this benchmark less than $\sim$5% packets trigger the IDS to execute regular expression matches. The four regular expression rules used for this test are from backdoor ruleset. Since there is no backdoor activities on the network, a single threaded regular expression engine running backdoor rules can easily offer throughput of the order of a few Gbps. Thus, with a multi processor load balanced system such as the SGI Altix, it is possible to keep up and even exceed link speed of 10 Gbps. Figure 3.21 depicts the total throughput of the system when running each payload through rules from backdoor rule-set. The throughput decreased linearly with increasing number of rules executed per network payload. The overall throughput drops from 70,088 Mbps to 18,874 Mbps on the SGI Altix 4700, when the number of regex per packet increased from one to four. From Figure 3.22 it can be seen that the throughput is also sensitive to the CPU clock frequency as well as the overall memory subsystem of a given computing system. The best per-thread performance is obtained by a dual core desktop with a high CPU clock frequency.

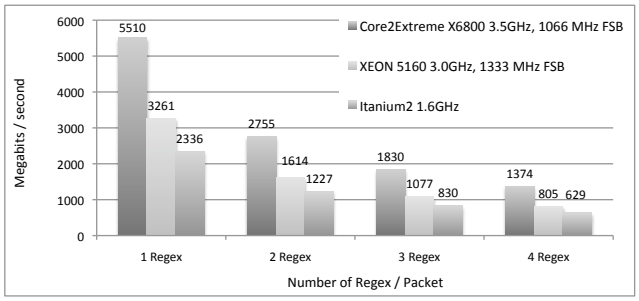

Figure 3.22: Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of minimal malicious activity in the tcp payload dump.

Software Benchmark with Moderate Malicious Activity

In our benchmark, moderate network activity scenario consists of the cases when $\sim$30% packets trigger the IDS to execute regular expression matches. The four regular expression rules used for this test are from ftp ruleset running and Figure 3.23 details the system throughput running this test. The throughput of the software based IDS decreases because the PCRE engine now needs to further inspect the payload for potential malicious signature. In this example, with moderate ftp activity on the network,the software based engine matching regular expressions from the ftp rule-set, exhibits slowdown when compared to the aforementioned minimal activity scenario.

Once more, the throughput decreased linearly with increasing number of rules per network payload. The overall throughput drops from 2058 Mbps to 515 Mbps on the SGI Altix 4700, when the number of regex per packet increased from one to four. This kind of scenario makes it impossible to run regular expression engines at 10 Gbps, though it is marginally sufficient for a 1 Gbps network interface. Interestingly, from Figure 3.24 the per core throughput

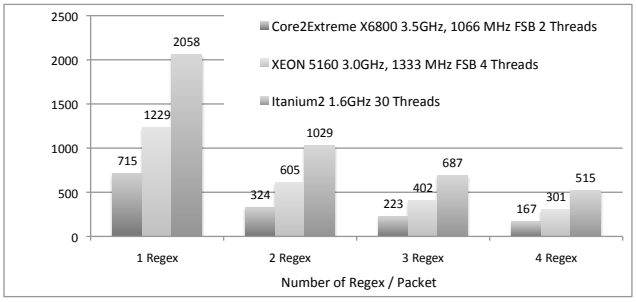

Figure 3.23: Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of moderate amounts of malicious activity in the tcp payload dump.

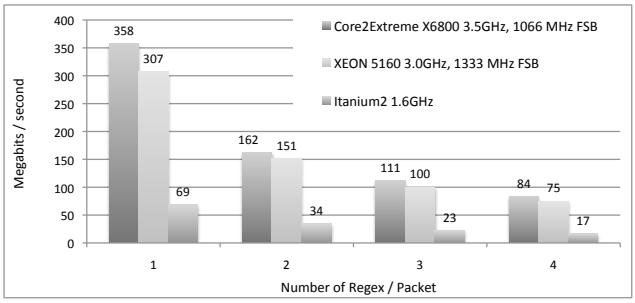

Figure 3.24: Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of moderate amounts of malicious activity in the tcp payload dump.

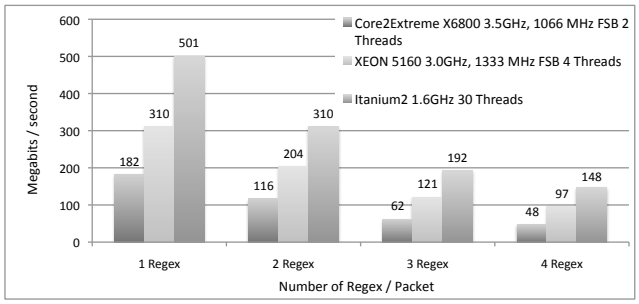

Figure 3.25: Comparison of system throughput with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of high amounts of malicious activity in the tcp payload dump.

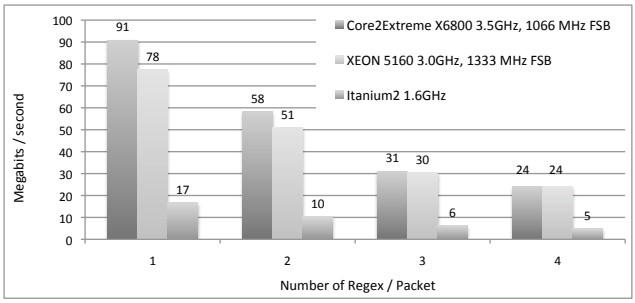

Figure 3.26: Comparison of system throughput per processor thread with varying number of regular expression rules executing per TCP payload. The results are shown for software based execution in presence of high amounts of malicious activity in the tcp payload dump.

variation decreases with increasing number of regular expression per packet, when compared across the same architecture viz. the C2E X6800 and the XEON 5160. The X6800 performance is 16.6% higher for one regex per packet, but only 12% higher for four regex per packet.

Software Benchmark with High Malicious Activity

High activity scenario entails the case when $\sim$50% of the network payload trigger the IDS to execute regular expression match. The throughput of the software based PCRE engines dramatically decreases in this scenarios, since a substantial number of packets are parsed through regular expression rules. We have selected four regular expressions from the web-client, web-cgi rule-sets. During this test, the rules trigger multiple times, indicating payload containing potential malicious activity vis-a-vis the web based rules. By running thirty threads in parallel on the SGI Altix 4700 system we were able to achieve 501 Mbps with one regex per packet and 148 Mbps with four regex per packet as seen from Figure 3.25. From Figure 3.26, we see that the per-thread performance of the C2E X6800 and the XEON 5160

have almost equalized for both three regex per packet and four regex per packet. The per thread performance of Itanium 2 has also dramatically decreased, with only 5 Mbps achievable per core for executing four regex per packet. This is the kind of scenario that can be expected during a concerted attack, i.e. when streams of packets arrive at the IDS and most of them trigger the SNORT IDS to execute multiple regular expression tests. As a result, this kind of situation can slow down the IDS to a crawl.

3.5.2 Hardware Benchmark and Comparison with Single Threaded Software Execution

Our compilation and hardware implementation of PCRE engines are compared against PCRE engines executed on software on actual SNORT rules. We run the experiment on a set of network dump of size 2.0 GB. Our network dump file has been generated by running *tcpdump* on the SGI Altix 4700 supercomputer, over a period of time, and collecting the payload data in a single file. We load the payload file on the main memory of the Altix machine before sending it onto the RASC Blade. When evaluating the baseline software implementation of PCRE, we store the file on a ramdisk on an SGI workstation. We have created six project directories with 25, 50, 75, 100, 200 and 400 regular expression derived from the backdoor, web-client, and spyware-put rulesets to allow us to obtain speedup data with varying number of regular expressions. More than 214 regular expressions entailed extremely long place and route (PAR) effort requirements. The primary reason for PAR failure with such large designs, that encompass the whole FPGA is the lack of available system RAM, which in our case was

14 GBytes.

Experimental Setup

For our software baseline testing, we utilize the aforementioned rulesets and create five project locations each with the same set of rules that were implemented on hardware. Each of the project directory is accessed by a shell script that invokes *pcre_compile* on the regular expression and thereafter executes by invoking *pcre_execute*. Each iteration of the experiment was executed 100 times to determine the average execution time, and hence the average throughput. We run the software baseline benchmarks on an Intel XEON 5160 (Woodcrest 3.0GHz 1333MHz Front Side Bus) based SGI Workstation, with 16 GBytes RAM.

We utilize the three aforementioned rulesets for compilation into VHDL and thereafter implementation onto the FPGA hardware on the SGI RASC RC 100 Blade. The hardware NFA blocks utilize a modular connection and are individually synthesized. The synthesized regular expression engines from each of the ruleset are collated together in a project location, along with the payload buffer and the memory interface module. We utilize the Xilinx ISE 9.2i synthesis tool namely 'XST' to synthesize this project. Thereafter the netlist containing the regular expressions and the payload buffer is mapped onto the Virtex-4 LX 200 FPGA, along with the SGI Core services using the 'MAP' tool. The next stage involves running the PAR and bitgen tools to finally generate a bitstream file that would configure the hardware resources on the FPGA. The bitstream file is copied to the bitstream repository, that would be accessible by the SGI Altix RASC daemon. All the FPGA compilation tools are run

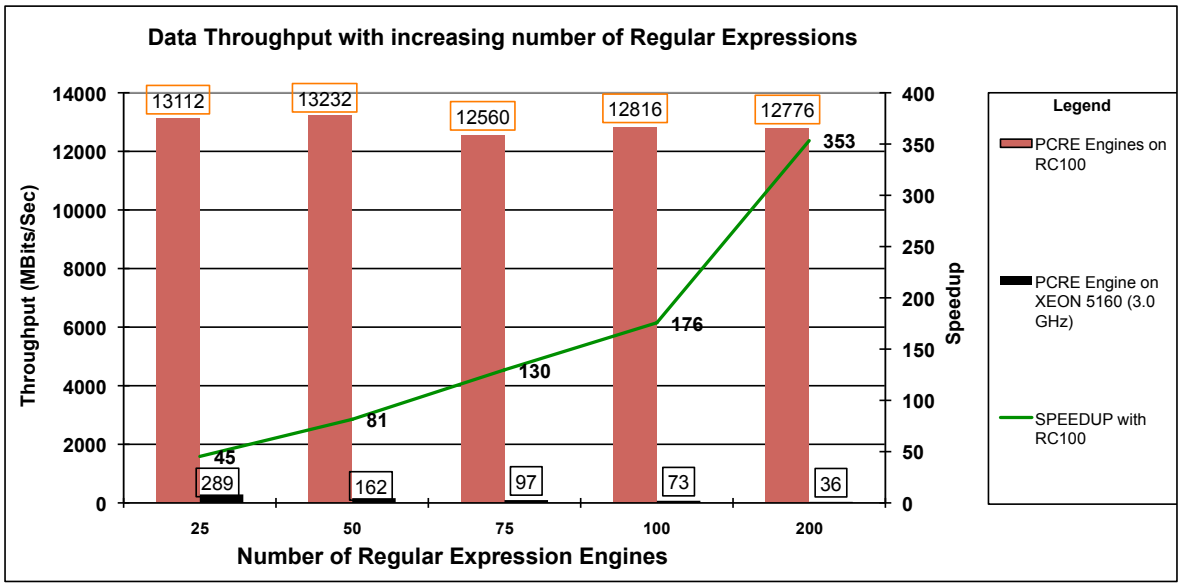

Figure 3.27: Throughput of the PCRE engines on the SGI RASC RC100 Blade as function of the number of regular expressions. The speedup is in comparison to software execution on a 3.0 GHz Xeon. The throuput improvement is 353x using 200 regular expressions.

on the baseline SGI XEON 5160 workstation. The host code that reads the payload file, calls the RASClib APIs and receives the match data in a 'C' program. Sixteen bytes each corresponding to one of 16 TCP payloads, are sent to the RASC Blade each clock cycle. The SGI Altix 4700 at our site, utilizes two partitions, each with 32 Intel Itanium2 (Montecito 1.6GHz) processor cores, and 642 GByte main memory. The host code has been compiled using Intel C Compiler version 9.1.042, for optimum performance on the Itanium2. In order to benchmark the performance of the hardware, we program the FPGA first with the bitstream containing 25 regular expression engines. Next we call the RASClib API's to send the 2.0 GB worth of payload, to the RASC Blade, and obtain the match data for the TCP packets. We then repeat the experiment with each set of regular expressions, each time sending 2.0 GB payload to the RASC Blade, and receiving the match data. Each iteration of the experiment is run

100 times and the average time of execution is recorded to obtain the average throughput. We include the bit-stream loading time overhead for each run of the benchmark. Since the RASC RC 100 Blade is connected to a NUMA Link interconnect on the Altix 4700 system, that is a shared memory system, the throughput data shows slight variability, which is dependent of a number of factors including the location of the memory that stores the payload, the actual system load at the time, etc.

Performance Results

The speedup and throughput charted out in Figure 3.27, corresponds to both the baseline case (black) bars and on the RASC RC 100 blade viz. the (red) bars. The hardware design after the synthesis, place and route flow, clocked at a little more than 150MHz for all the five regular expression rulesets. Even though individual components can run at 300MHz, the primary reason for the 150 MHz clock rate is that back reference opcode using on-chip BRAM on Virtex-4 LX 200 can operate at 155 MHz and thus is the critical path for timing. Hence we programmed the RC 100 algorithm clock to operate at a fixed frequency of 150 MHz. [2] As the number of regular expressions on the hardware increased each packet was processed through many more regular expressions in parallel. Since the baseline software entails serial execution of the same payload over multiple regular expressions, the effective throughput declines with increasing number of regular expression engines. Therefore the speedup provided by the hardware increased linearly with increasing number of regular expressions when com-

[2]The design runs at 150 MHz by utilizing a fractional (n/m) multiplier (DCM Block) with value (3/4) that multiplies the base frequency (200MHz) of the clock generator on the FPGA.

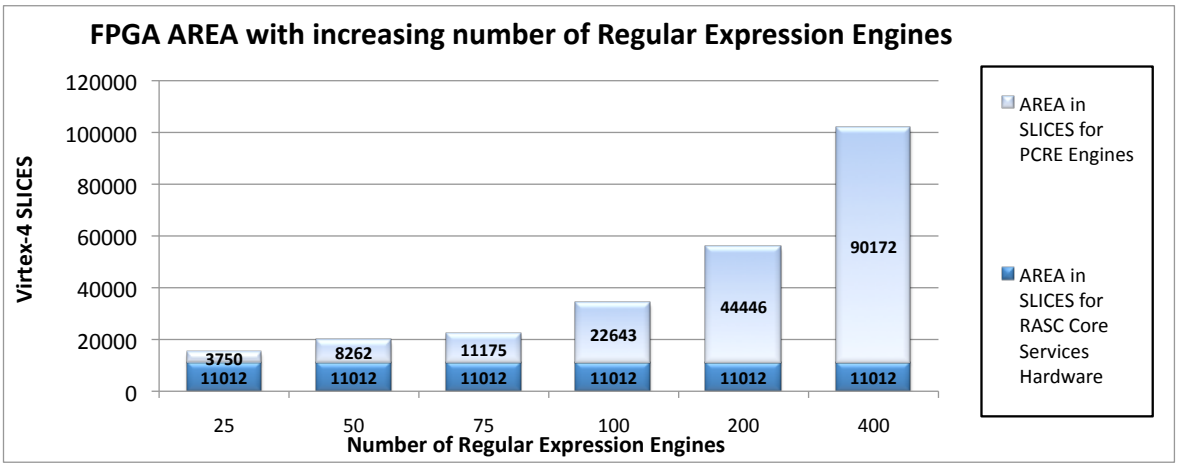

Figure 3.28: Area (in slices) occupied by PCRE engines on the Virtex-4 FPGA on SGI RASC RC100 Blade. The lower dark section is the fixed area cost dedicated to the RASC Core services on the FPGA (11,012 slices).

pared to the baseline software execution scenario. The throughput obtained with 25, 50, 75, 100 and 200 regular expressions implemented on the FPGA with the test network dump data set are 13.1, 13.2, 12.6, 12.8 and 12.7 GBits/s respectively, with an average throughput of (**12.9 GBits/s**). We can modify the host code driver, to broadcast the Payload buffer from the host to the second FPGA on the RASC RC 100 Blade as well and would double the number of implemented SNORT rules. The fact that the SGI Altix4700 supports seventy FPGAs in total can allow a potential implementation of the complete SNORT ruleset in hardware with room to spare for future expansion.

FPGA Area Analysis

In this subsection we evaluate the area occupied by our design on the FPGA as function of the number of regular expressions. The area metrics used for a design on a Xilinx FPGA is known as slice (as discussed in section 3.1.4). The regular expressions in the SNORT

DB range from small 12 character long rules, up to 602 character long rules. The hardware complexity is related to the size of the regular expression (in text form), as well as the opcodes involved. This is because two regular expressions with the same size (in text form) may lead to slightly different sized hardware. The smallest regular expression that we implement is a ten character long rule and it occupies 13 slices, while the largest regular expression, is a 545 character long rule which occupies 1102 slices. In Figure 3.28, the area occupied on the FPGA for increasing number of regular expressions is charted out. Each occurrence bar is divided into two parts, the lower dark colored section is the fixed area cost dedicated to the RASC Core services on the FPGA. It occupies 11,012 slices. The light colored section of the bar on top is the actual area for the given number of regular expressions. The total area of 200 regular expressions is 55,012 slices for a 62% occupancy of the FPGA. The last bar corresponding to 400 regular expressions is over-mapped onto the Virtex-4 LX 200 FPGA. It may be possible to implement 400 regular expressions in a future device with more slices than the Virtex-4 LX 200. From the area data we calculate the variable area cost per regular expression on an average to be $\sim$190 slices without accounting for the core services. While accounting for the fixed area of core services the area cost per regular expression turns out to be $\sim$339 slices.

3.5.3 Single Processor Power Consumption Analysis

FPGAs are very low power devices with power consumption on the order of 30 to 40 Watts, mainly due to their low clock speeds. On the other hand microprocessors consume a lot

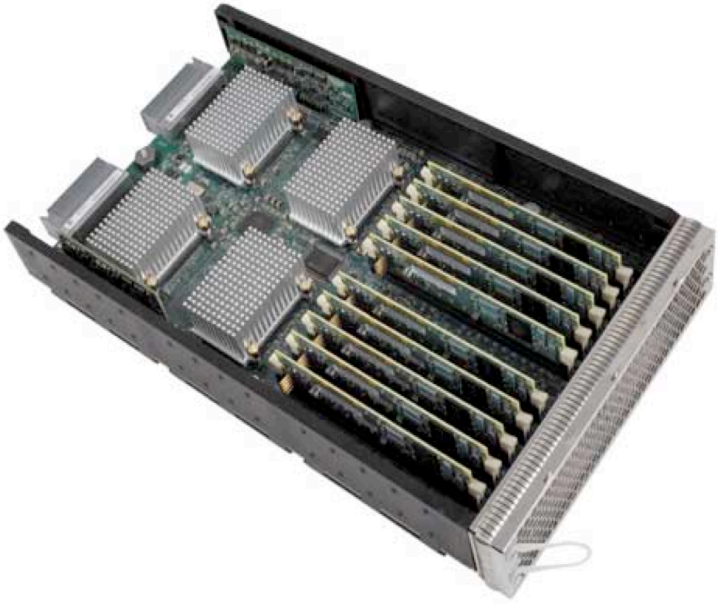

Figure 3.29: Picture of the RASC RC100 Blade usable on the SGI Altix 4700

of power and generate high amounts of heat, with additional power required for cooling purposes. As can be seen in Figure 3.29 the FPGAs on the RASC blade have short heat sinks and passive cooling. On the other hand, high performance processors and support chips consume a huge amount of power and require active cooling. The TDP (Thermal Design Power) [86] of a 3.0 GHz dual-core Intel XEON 5160 (65nm) processor is 80W [85]. Thus by utilizing an FPGA it is easily possible to scale down the computing power requirements without affecting application performance.

The maximum power consumed in Watts by the Virtex-4 LX 200 (90nm) FPGA is $\sim$ 42W [167] and an additional $\sim$ 80 watts is used by the host processor for data transfer to and from the FPGA. Therefore by utilizing an FPGA that offers $\sim$ 150 times the throughput of a dual core processor, we have reduced the power foot print of the system by [80 * 150 / (42

+ 80)] i.e. $\sim$100 times; a two orders decrease in the power consumption of the computing silicon devices.

3.6 Conclusion

Previous approaches towards compiling PCRE to FPGA work did not utilize the PCRE compilation framework. The earlier compilation approaches were mainly either hand coding of the regular expressions to hardware or used a custom conversion process. As an example, [33] used their custom conversion flow for generating hardware from a subset of the SNORT IDS PCRE rules.

We describe a novel compilation method in this chapter, one that allows direct compilation of PCRE opcodes generated from SNORT rules to VHDL. Our work allows compilation of PCRE to FPGA by utilizing the PCRE library compilation front end. We have developed an opcode dump module that processes the opcode in the format required by our opcode to VHDL compiler. Using the generated dumped opcodes we generate VHDL hardware modules. We simulate the hardware and test its compatibility with the software based PCRE execution, using the same generated opcodes and thus maintain compatibility with the software flow.

We have presented an extensive analysis of the performance of regular expression based SNORT IDS with actual network payload and concluded that the throughput of multiprocessor software based regular expression engines decline drastically during periods of

malicious activity

We also demonstrate the actual implementation of SNORT regular expression rules on FPGA using a 16 bank architecture. We test our proof-of-concept design on a Virtex 4 LX 200 FPGA on a SGI RASC RC 100 blade. The resulting hardware system provides an average throughput of 12.9 Gbps on network payload data on a number of SNORT rules ranging from 25 to 200, by maximizing bus bandwidth available on the RASC blade. Our design performs between 45X up until 353X when compared to a baseline implementation on a Intel XEON 5160 CPU at 3.0 GHz.

We have also shown a two order decrease in power consumption by utilizing an FPGA for implementing regular expression engines vis-a-vis implementing the engines in parallel on a general purpose processor.

Chapter 4

Partial Reconfiguration on FPGA

FPGAs are ideal platforms for implementing reconfigurable co-processor and accelerator systems. FPGAs provide versatility and adaptability in the hardware by means of reconfigurability. The FPGA can be programmed with a new bitstream to provide new functionality in hardware. Xilinx Virtex series FPGA also provide modular and partial re-configurability, which allows a portion of the FPGA to be reconfigured, while the rest of the hardware functions as before. In this chapter we demonstrate the utility of partial reprogramming for providing multiple co-processor cores on a CSoC. We provide details on how we automate the system interface between the CPU and the coprocessor on the CSoC. We also demonstrate how partial reconfiguration can help provide quick adaptability for FPGA systems that implement regular expression matching hardware for accelerating SNORT IDS.

4.1 Dynamic Co-Processor Interface Automation

By integrating one or more (hard or soft) CPU core on the chip, new generation platform FPGAs have become configurable systems on a chip (CSoC) that support a combined software and hardware execution model. More recently, FPGAs, using new design tools, have also provided support for partial reconfiguration. The CSoC system designer is left with the task of interfacing IP Cores to the CPU and also for realizing partial reconfiguration across the cores. In this chapter, we describe a software tool to automate the interface between the CPU and the reconfigurable fabric. Our tool generates hardware wrappers for the IP Cores that makes them look like a C function invocation in the source code. We also use our tool to support partial reconfiguration: the same wrapper is used for a multitude of IP Cores and the user selects the core to be invoked in the program.

Modern FPGAs integrate a (hard or soft) processor core, with the reconfigurable fabric. These CSoC use the CPU to support the software execution and rely on one or more hardware cores for accelerating frequently used code segments (loop nests). These hardware cores can be either custom designed or can be selected from a library of pre-existing IP Cores. The hardware cores are tightly coupled with the CPU using very high speed, point to point links for fast data transfer in the Virtex-4 FX CSoC. CPUs such as the PowerPC 405 also support custom instructions for communicating with these co-processors. The co-processors act as accelerators for compute intensive portions of the applications such as floating point intensive calculations [178], discrete transformation algorithms (FFT, DWT, DCT, etc) [152][46]

[200], and also for custom applications such as Smith-Waterman string matching, encryption/decryption engines, etc. A multitude of co-processing functionality can be realized using IP Cores that are highly configurable and performance optimized. Interfacing the available library of IP Cores to the on-chip processor is a time-consuming and tedious task which almost always, needs to be taken care of manually. The system designer is left with the task of interfacing each and every core to the co-processor interface [98]. Partial reconfiguration offers the system designer the possibility to leverage the same basic hardware structure for accelerating multiple tasks (programs) on the CSoC. Partial reconfiguration on the FPGA makes it possible to create a system that can enable re-configuration of pre-assigned parts of the FPGA without affecting the static parts, or inducing a system-wide reset. The high overhead of reconfiguration, at this point in its development (msec), precludes using it dynamically within the same task. It is however a very powerful tool to overcome the area limitation of a single FPGA platform across multiple applications since re-configuration can be combined together with a CPU context switch. The system designer is also left with the task of generating the interface between static and dynamic regions of the FPGA as required for partial reconfiguration [65] [204] [60] [34] [205] [93]. In this chapter we describe a software tool for automatically generating the communication interface between the software running on the CPU and a tightly coupled IP Core based co-processing system on the Virtex-4 FX FPGA. It generates hardware wrappers for the core that makes it look like a C function invocation in the source code. We extend this tool to support partial reconfiguration: the same static wrapper is used for multiple cores and the user selects the core to be invoked in the program.

Our compiler for FPGA-based reconfigurable systems, ROCCC [72] [71] [73]leverages the huge wealth of IP Cores by allowing the user to import these cores into the source code. The compiler automatically generates a wrapper structure that would hide the timing and stateful nature of the IP Cores and makes each available to the C language compiler, as an un-timed side-effect free function call. We also use the ROCCC compiler approach to support run-time reconfiguration by automating the generation of the interface between static and dynamic regions of the FPGA. The user can switch between multiple functional units by calling the appropriate C function in the code, thus entailing the use of the same hardware wrapper for multiple IP Cores. Utilizing our software tool along with the ROCCC infrastructure we have been able to automatically configure multiple IP Cores on the fabric viz. FP (Floating point) Adder, FP Multiplier, Integer divider CORDIC engine and an FFT engine. Moreover using partial reconfiguration we have been able to overcome the area limitation of a single FPGA platform (Virtex-4 FX12), using five different IP Cores. We have allocated a region of 1800 slices for the co-processor, thus resulting in a reduction in the floor area by 2656 slices due to partial reconfiguration. Moreover the area dedicated for the hardware wrapper is no more than 14 slices, quite miniscule, when compared to the actual IP Core area.

4.2 System Overview for IP Core Wrapper Generation and Partial Reconfiguration

The target of our research emphasizes automatic wrapper generation and reconfiguration for IP Cores configured on CSoC systems. These systems are self contained embedded processing solutions often targeted for reconfigurable computing applications. The major ingredients in our system are the CSoC system, IP Core libraries and the Compiler infrastructure (ROCCC).

4.2.1 The CSoC platform

Our CSoC system consists of a Platform FPGA, which in turn are field programmable gate array logic along with one or more (soft/hard) processors all on a single chip. The CPU on the CSoC runs an Operating System as well as application software. With the advent of higher performance FPGA fabrics it is now possible to instantiate software code accelerators on the FPGA and use it for speeding up execution on the processor. In the past, the limiting factor for speedup of these FPGA based accelerators had been the on-chip bus used for data communication between the host-code and the accelerator, since the same bus is used for various other peripherals too. Software developed around the PowerPC core on the Virtex-4 FX FPGA can communicate with fabric co-processors using point to point buffered links (also known as Fast Simplex Links) [197] hence alleviating performance based issues, present on a bus based architecture [194] [13] [197]. The Virtex-4 FX also provides a high performance

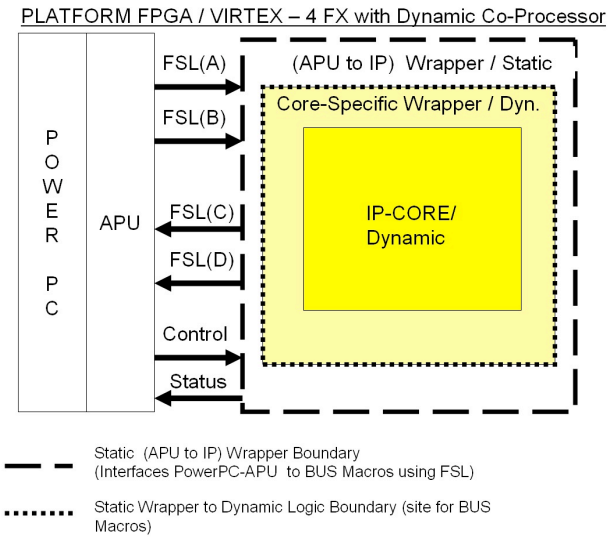

Figure 4.1: System Architecture of the dynamic co-processor system on FPGA

bus architecture (PLB and OPB) for connection with various on-fabric peripheral controllers such as memory (DDR/SRAM) controller, Ethernet, UART, keyboard and mouse controller, Peripheral controllers are synthesized as soft cores on the FPGA fabric, thus user defined peripherals may also use this bus for communication with the CPU or other on-chip peripherals.

4.2.2 APU (Auxiliary Processing Unit) on Virtex-4 FX

The PowerPC 405 core on the Virtex-4 FX FPGA is a 32-bit architecture with on-chip instruction and data cache memory. An Auxiliary Processor Unit (APU) [136] controller accompanies the core to interface it to hardware accelerators on the fabric. The APU supports 32-bit custom instructions and 64-bit data. The co-processors instantiated on the Virtex-4 FX FPGA use the APU on the PowerPC for seamless communication with the FPGA fabric. Additionally there is also an option to use a bus based architecture, FCB (Fabric Co-Processor

Bus) for sharing the APU with more than one co-processor. As depicted in Figure 4.1, the system architecture used for our dynamic co-processor system involves a Xilinx Virtex-4 FX, the APU interface and two FSL channels. Data is sent/received over the FSL link from the Power PC to the compiler generated (APU to IP) wrapper. The wrapper parses input/output data according to the current IP Core instantiated on the dynamic fabric and maps them onto the slice macro interface. The slice macros interface the static wrapper to the dynamic wrapper and through it to the IP Core. Handshaking/control signals are mapped onto the Control bus and status/acknowledgment signals from the IP Core to the wrapper are mapped onto the Status bus.

4.2.3 IP Cores

Intellectual Property cores have been available for a while for FPGA based systems. These IP Cores are highly optimized replacement for sequential software used for time-critical systems such as real time audio-video encoders/decoders, FIR filters, DSP blocks and also for highly specialized applications such as string matching based on Smith-Waterman algorithms. These IP Cores have also been used in various FPGA based applications for rapid prototyping of system accelerators and co-processors. Most IP Cores are relationally placed macros for FPGA which are already mapped to the target architecture and many of them are relationally placed and routed as well. IP Cores result in higher performance designs along with lowering of the design effort for the system. Most IP Cores share a similar input/output architecture which consist of two input bus and one to two output bus, along with certain

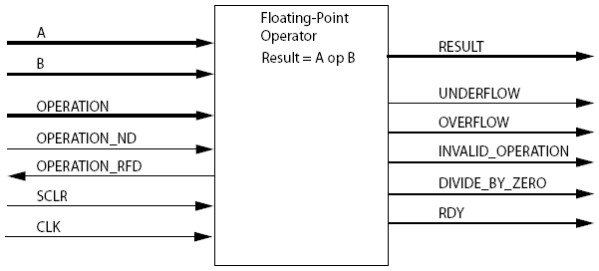

Figure 4.2: An example Floating Point IP Core, demonstrating the I/O interface

control/acknowledgement signals. Thus it is possible to encompass these interfaces into a standard I/O wrapper architecture [203] [201] which would serve as a superset for I/O interface to all possible IP Cores targeted at a particular system. Our system currently targets such compatible IP cores with future extensions planned for IP cores with arbitrary number of inputs or outputs.

Depicted in Figure 4.2 is an example of a compatible IP Core viz. a Logicore series Floating Point unit from Xilinx and Qinetiq [199]. The input bus (A, B) and output bus (RESULT) can be configured either as 32-bit single precision or 64-bit double precision, conformant to the IEEE 754 specifications. This core can be configured for a floating point operation such as adder, subtractor, multiplier, divider, square-root, and comparator. Adder and subtractor can be combined in a single unit. Various status signals originating from the IP Core are Underflow, Overflow, Invalid operation, Divide by Zero. The OPERATION signal selects either Add / Subtract, or from a multitude of compare operations if a Comparator is configured. The computational latency of the floating point unit is 5 clock cycles.

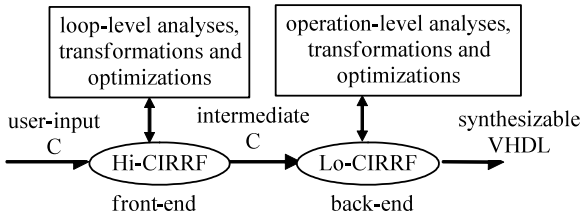

Figure 4.3: ROCCC system overview

4.2.4 ROCCC Overview

An overview of the ROCCC framework is depicted in Figure 4.3. We have separated the front and back ends to achieve modularity and eventually allow the use of a variety of front end and back end tools.

ROCCC is built on the SUIF2 [7] and Machine-SUIF [157] platforms. It compiles C code into VHDL code for mapping onto the FPGA fabric of a CSoC device. Information about loops and memory accesses is visible in our front-end IR (intermediate representation) viz. Hi-CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics). Accordingly, most loop level analysis and optimizations are done at this level. ROCCC performs a very extensive set of loop analysis and transformations, aiming at maximizing parallelism and minimizing area. The compiler also minimizes accesses to memory by effecting smart re-use of data. The compiler also performs scalar replacement at front-end. All memory loads are moved to the top of the loop body and all memory stores are moved to the bottom of the loop body. Machine-SUIF is an infrastructure for constructing the back end of a compiler. Machine-SUIF's existing passes, like the Control Flow Graph (CFG) library [30], Data Flow Analysis library [158] and Static Single Assignment library[159] provide useful optimization

and analysis tools for our compilation system. We build the back-end using Machine-SUIF. The compilers back-end Lo-CIRRF, converts the input from control flow graph (CFG) into data flow graph (DFG), and generates synthesizable VHDL codes. We rely on commercial tools viz. Xilinx XST to synthesize the generated VHDL codes for Virtex-4 FX.

4.2.5 Interface Synthesis

As introduced in the system overview section, the ROCCC compiler generates synthesizable VHDL code for applications written in un-timed C. In this section, we present our approach using the ROCCC system to wrap IP Cores. The compiler takes in a C-function intended for co-processing operation and automatically generates the corresponding IP Core, along with high-level abstractions. Taking the high-level wrapper abstractions as input, ROCCC generates synthesizable wrappers in VHDL separately as well as C language driver code for communication across the FSL channels. The wrappers are instantiated as components in the outer circuit and enable a seamless connectivity between the on chip CPU and the IP Cores instantiated on the fabric.

C language function calls

ROCCC recognizes co-processing function calls by checking a certain pragma and records this pragma into an Intermediate Representation field for further use. It inserts Assembly code required to access the FSL channels. The putfsl assembly call is used to write 32-bit data to the FSL, while getfsl call reads back 32-bit data from FSL. The software function

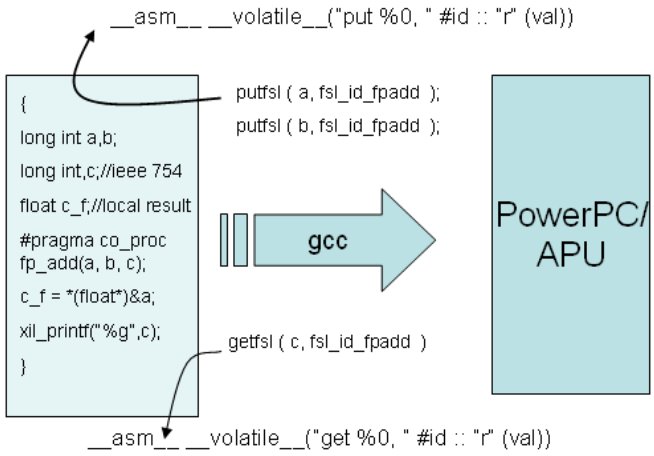

Figure 4.4: The C function call to the co-processor and the #pragma directive

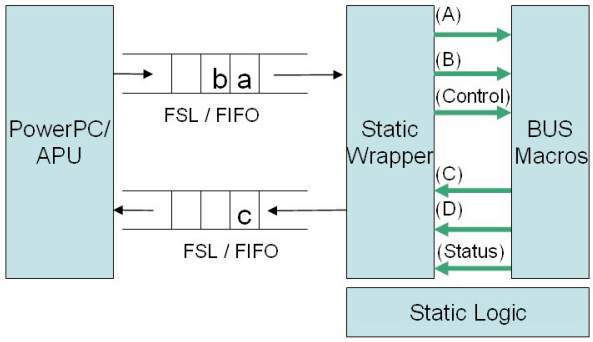

Figure 4.5: Data flow using FSL from the Virtex-4 APU to the static wrapper

call to the co-processor sends/receives 32-bit data through the putfsl/getfsl assembly calls as depicted in Figure 4.4. The APU copies the data into/from the FSL and therefore to/from the static wrapper i.e. the (APU to IP) wrapper.

The Internal Configuration Access Port (ICAP) is used by the function call to load in a partial bitstream file in order to re-program the co-processor region with a new IP Core by making use of the OPB-HWICAP hardware.

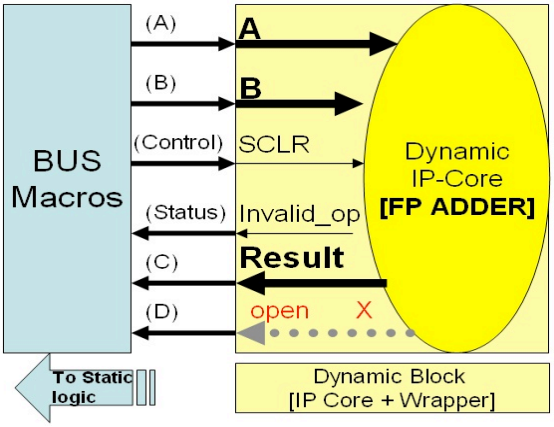

Figure 4.6: Data flow using FSL from the Virtex-4 APU to the static wrapper

Generation of the static (APU to IP) wrapper

The static wrapper provides an interface between the PowerPC APU and the first stage into the fabric, as depicted in Figure 4.5. The static wrapper uses the standard FSL interface, to provide for data input/ output and clock signals for synchronization. The static wrapper buffers the input data and presents them to the slice macros and also buffers output data to be communicated back using the FSL channel and into the Power PC APU.

Dynamic wrapper

The dynamic wrapper is a second wrapper which is generated in the partial reconfigurable region of the FPGA. It is a VHDL entity which connects the 32-bit input/output signals, the control signal, and the status signal from the slice macros onto the corresponding ports of the IP Core. We would like to emphasize that the connectivity from / to slice macros for each IP Core is specified in its respective dynamic wrapper. Thus the dynamic wrappers present a standard interface for connectivity between slice macros and the IP Core as shown

```
entity rmodule is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
        B : in  STD_LOGIC_VECTOR (31 downto 0);
        C : out  STD_LOGIC_VECTOR (31 downto 0);
        D : out  STD_LOGIC_VECTOR (31 downto 0);
        clk : in  STD_LOGIC);
end rmodule;

architecture Behavioral of rmodule is
component cordic_module –From Logicore
                port (
                x_in: IN std_logic_VECTOR(15 downto 0);
                y_in: IN std_logic_VECTOR(15 downto 0);
                phase_in: IN std_logic_VECTOR(15 downto 0);
                x_out: OUT std_logic_VECTOR(15 downto 0);
                y_out: OUT std_logic_VECTOR(15 downto 0);
                clk: IN std_logic);
end component;

u_cordic: cordic_module
                port map (
                x_in => A(31 downto 16),
                y_in => A(15 downto 0),
                phase_in => B(15 downto 0),
                x_out => C(31 downto 16),
                y_out => C(15 downto 0),
                clk => clk);
end Behavioral;
```

Figure 4.7: A compiler generated dynamic wrapper for CORDIC engine

in Figure 4.6. A compiler generated dynamic wrapper is depicted in Figure 4.7, which maps the slice macro interface to the ports of the IP Core. The input signals A, B, and output signals C, D are connected to the slice macros during synthesis and so are the control/status signals.

Dynamic Co-Processor Instantiation

We also use our tool to support dynamic partial reconfiguration. Dynamic partial reconfiguration at runtime allows re-use of FPGA resources to obtain a plurality of functionality, from the same hardware block, but at different times, and also without affecting the static parts of the device. The compiler generates the wrappers for each IP Cores that need to be dynami-

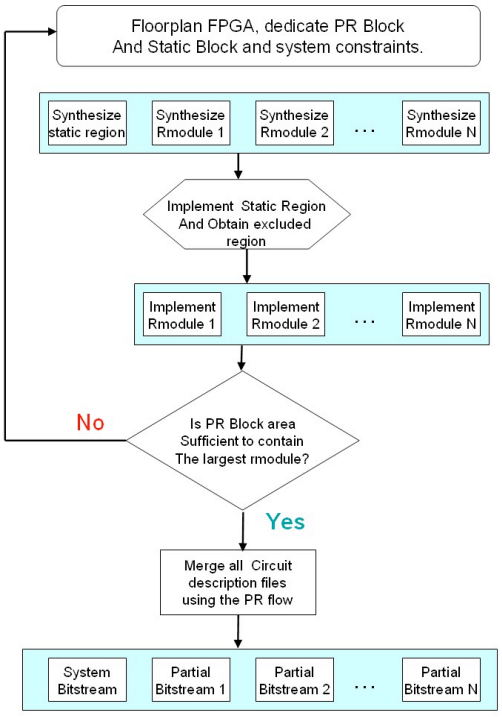

Figure 4.8: The Partial Reconfiguration Module Generation Flowchart for FPGA

cally reconfigured. The design flow in Figure 4.8 involves the generation of the static logic along with the various partial reconfigurable logic (wrapped IP Cores). Thereafter the FPGA is floor planned to allocate a pre-determined area for the dynamic logic and the rest of the floor area is dedicated to static logic. The area dedicated to the dynamic logic, also known as the PR-Block (Partial Reconfigurable Block), is such that it may allow for the largest IP block to be placed and routed within it. I/O and communication of the static logic with the PR-block takes place using certain pre-configured hard macro blocks known as slice macros [204], as shown in Figure 4.9.

These slice macros need to be manually placed around the boundary of the PR-block. We have employed the Xilinx PlanAhead 8.1 visual floorplanning tool for iterative design and

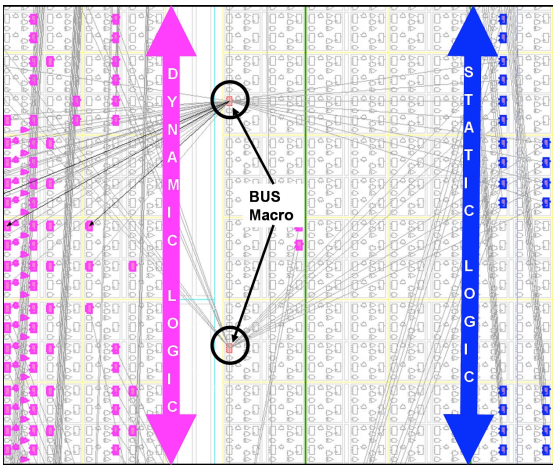

Figure 4.9: SLICE macros placed on the Dynamic / Static logic boundary

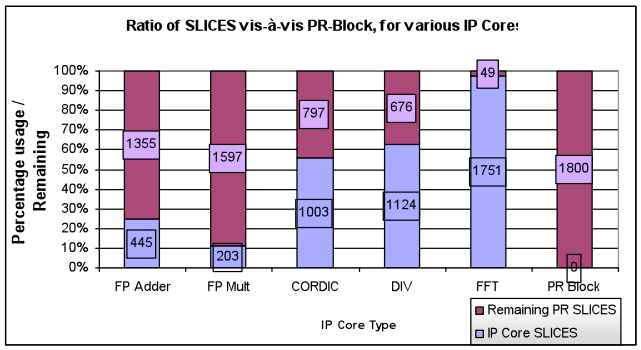

Figure 4.10: SLICE usage for various IP Cores, and PR Block occupancy

placement. The final stages of the partial reconfigurable flow generates N static bitstreams and N partial bitstreams, where N is the number of different IP Cores which are to be configured in the PR-Block. Each of the N static bitstream contains the static design with the IP Core numbered N already programmed into the stream, while each of the N partial bitstreams contains the logic to re-program the PR-Block with the functionality of the Nth IP Core. Thus the system may choose to start with one of the static bitstreams during power-up and thereafter reprogram the PR-Block with the desired functionality.

4.2.6 Experimental Results

We have incorporated four Xilinx Logicore IP Cores in our compiler infrastructure for the purpose of conducting experiments. These cores are enumerated in, Table 4.1. The floating point adder, on the Xilinx Logicore IP Core [199] takes in two 32-bit single precision values conformant to the IEEE 754 standard (A, B) and outputs their sum in single precision (result). The floating point multiplier takes in two 32-bit single precision values (A, B) and outputs their 32-bit product (result). The FP multiplier [199] has been configured to utilize four DSP48 blocks for fast multiplication of the significand values from the floating point inputs. The CORDIC (Coordinate Rotational Digital Computer) IP Core [196] performs a rectangular-to-polar vector translation. The IP Cores takes in as input the angle and magnitude in a polar coordinate and generates the equivalent vector (X, Y) in Cartesian coordinate. The CORDIC module has been configured to utilize eight DSP48 blocks for fast multiplication for calculating the new coordinates and to enable scaling. The IP Core for a pipelined Integer divider [202] does arithmetic division on a 32bit dividend and a 32bit divisor thus resulting in a 32bit quotient and a 32-bit fraction value. For calculation of FFT, we have configured the Logicore FFT IP Core [198] for 256 points, operating on 16-bit data. The core is configured for Burst I/O for non simultaneous processing and data loading/unloading. Nine DSP48 blocks have been utilized for fast multiplication operations. The static wrapper contains logic for timing and burst data loading/unloading from the FFT unit. We have targeted the Xilinx Virtex-4 FX12 FPGA containing 5472 slices, on the ML403 Evaluation board. The design tools that we used are the Xilinx EDK 8.1 (PR-5) for generation of the Imple-

Table 4.1: Area Covered by the Dynamically instantiated IP Cores

IP Core	SLICES	DSP 48 Blocks	Clock Speed MHz	Bitstream KBytes	Reconfig Time	
					JTAG	Sel.MAP
Floating Point Adder 32-bit	431	0	250	79	0.2 sec	5ms
Floating Point Multiplier 32-bit	431	4	218	76	0.19 sec	4.8ms
CORDIC Rotate 16-bit	989	8	220	99	0.24 sec	6ms
Divider Fixed Point 32-bit	1111	0	228	112	0.28 sec	7ms
FFT 16-bit 256 Point	1736	9	250	142	0.29 sec	7.1ms

mentation files for the static subsystem and various wrappers for peripherals. We used Xilinx ISE 8.1i XST for synthesizing, and Xilinx PlanAhead 8.1 for floorplanning, implementing and testing the partial reconfiguration designs.

These five examples illustrate how a multitude of IP Cores can be effectively configured as co-processors on a FPGA using C based function calls. The execution time overhead at both the input side and output side for these four examples is one clock cycle except for the static wrapper for FFT engine. The configuration units (slices) dedicated to the reconfigurable block is 1800 slices as shown in Figure 4.10 and slice macros and wrappers account for less than 1% of slices dedicated to the PR Block (Table 4.2).

Table 4.2: The Area Covered by IP Wrappers and Wrapped IP Cores.

Entity	**Property**	**Floating Point Adder 32-bit**	**Floating Point Multiplier 32-bit**	**CORDIC Rotate 16-bit**	**Divider Fixed Point 32-bit**	**FFT 16-bit 256-Point**
Static Wrapper	SLICES	12	12	12	12	12
Static Wrapper	SLICES % w.r.t. IP Core	2.7	6.4	1.2	1.06	0.74
Dynamic Wrapper	SLICES	2	2	2	2	2
Dynamic Wrapper	SLICES % w.r.t. IP Core	0.45	0.98	0.2	0.17	0.11
Wrapped IP Core	SLICES	445	203	1003	1124	1751
PR Block	**1800 SLICES** dedicated for the PR Block					

4.3 Adaptive Hardware/Software Regular Expression Based IDS

FPGAs can be reprogrammed to change it's functionality. This allows reconfiguration of the FPGA with the type of the intrusion detection engines required at the moment i.e. adaptability to the current network conditions. Additionally Xilinx FPGAs also support partial reconfiguration flow, so that a part of the FPGA could be reconfigured. This reduces the hardware re-programming time, when only a part of the FPGA needs to be modified. IDS that employ these FPGA can maintain execution through software threads while the FPGA is reconfigured during the brief moment. Moreover an IDS with two or more FPGAs can maintain execution on the other FPGAs while one of them is either partially or fully reconfigured with a different set of regular expression engines. Partial reconfiguration also reduces the

synthesis and place and route time of a PR (Partial Reconfigurable) block from a few hours to a few minutes, since only the hardware module under question is processed. An FPGA system supporting partial reconfiguration can respond to new types of network attack much faster than FPGA systems that only support full reconfiguration.

With our proof of concept hardware system, we demonstrate an adaptable regular expression based IDS using Virtex-4 LX 200 FPGAs that have been floor-planned for partial reconfiguration. Our novel design allows partial reprogramming across 16 banks of regular expression rule-sets. We implement 448 different regular expressions on two FPGAs and perform multiple partial and full reconfigurations. We measure the throughout of the integrated Field Programmable Gate Array (FPGA) and multiprocessor SGI Altix system with varying number of reconfigurations per minute. The maximum sustainable throughput of our design is 19.84 Gbps per FPGA. Our adaptive IDS can provide better than 10 Gbps throughput even with 32 partial reconfigurations per minute. Our system can also sustain 10 Gbps throughput with four full-reconfigurations per minute. Our IDS design can be extended to similar FPGA accelerated multi-processor system.

We describe a novel architecture on an integrated hardware/software regular expression IDS that can successfully maintain throughput at 10 Gbps scale even under a range of partial and full reconfiguration scenarios running on a proof-of-concept platform. The synthesis, place and route time for implementing a set of new regular expression signatures on the FPGA has been brought down from a few hours to minutes, for the hardware to be configured with new signatures by using the partial reconfiguration flow on the FPGA.

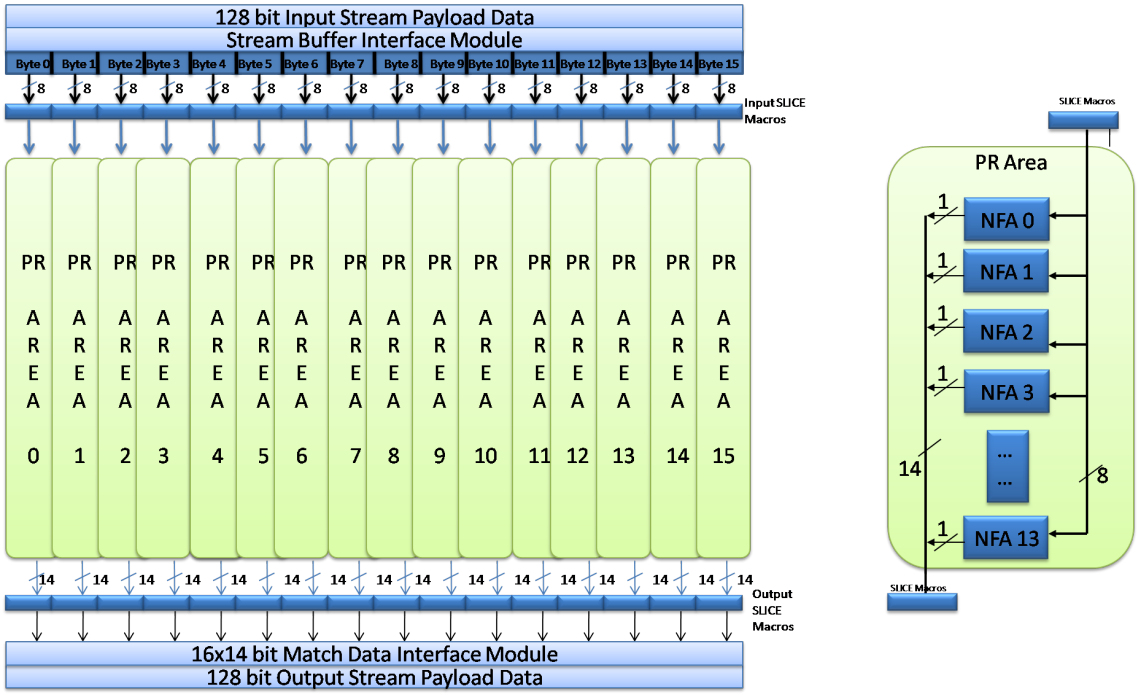

Figure 4.11: Architecture of the Sixteen Partial Reconfigurable Area blocks on the Virtex-4 LX 200 FPGA. Each PR block consists of fourteen NFA engines. A PR Block, expanded on the right hand side of the figure, obtains one byte payload data through the SLICE Macro each clock cycle and outputs 14-bit match data on completion of a match.

To the best of our knowledge, such an architecture is a first of a kind demonstration of an adaptable hardware/software regular expression based IDS. We have also benchmarked our proof of concept FPGA regular expression based IDS test-bed using a thirty-two core SGI Altix 4700 supercomputer with a RASC Blade consisting of two FPGAs. We show that by utilizing our architecture, it is possible to avert concerted attacks and also to adapt towards changing network activities, up-to thirty two times a minute, while still maintaining 10 Gbps throughput.

4.3.1 The FPGA Architecture

The Xilinx Virtex-4 LX FPGA on the RASC Blade is the largest FPGA in the Virtex-4 series. As depicted in Figure 4.11, we implement 16 banks, each consisting of 14 NFA engines on one FPGA. We have divided the 80,000 algorithm SLICES and 600KByte on-chip BRAM into 16 partial reconfigurable banks, each consisting of 5000 SLICES and 40KByte BRAM. We utilize the Virtex-4 synchronous SLICE macros[208] to transfer data, clock and enable signal between the static and the PR region on the FPGA. Two sets of SLICE macros have been utilized; the first to transfer data from the stream buffer interface module and the latter to transfer the match data from a the banks of NFAs to the output streaming data interface. All the regular expression engines on the FPGA share a similar hardware interface that is the signals for clock, reset, enable, character in and match out as shown in Figure 4.11. Thus it is possible to pre-allocate the physical area block for all the sixteen banks of regexp engines on the FPGA. Each bank of NFAs receives a character every clock cycle from the payload line and outputs the match data, on completion of streaming of the payload.

4.3.2 Xilinx Partial Reconfiguration Flow

We utilize the Xilinx ISE 9.2.04i PR7 toolkit partial reconfiguration flow to support partial reconfiguration of the FPGA on the RASC blade at runtime[208]. Partial reconfiguration allows a system designer to obtain a plurality of functionality, from the same hardware block, but at different times, and also without affecting the static parts of the device. Reprogramming partial-reconfigurable banks on the FPGA allows the IDS to quickly re-use the FPGA

when the parameters of network activity changes i.e. a different rule-set is required to be loaded. The static logic on the FPGA includes the RASC core services hardware that enables streaming data interface to the algorithm block. We have employed the Xilinx PlanAhead 10.1 visual floorplanning tool[87]. Four SLICE macros have been used per bank; two for input data and two for output match data. These SLICE macros have been manually placed around the boundary of each PR block. The final stages of the partial reconfigurable flow generates the partial bitstreams for each regular expression bank as well as the full bitstream that includes all the 16 banks as well as the core services. We have created scripts that sequentially compile a given PR block of 14 regular expressions across each of the 16 banks available on the FPGA. This results in future flexibility in loading a required bit-stream of a rule-set in any of the 16 banks. Using floor-planning, each NFA bank can be synthesized and placed and routed independently of each other. The time required to generate one partial bitstream is 20 minutes.

4.3.3 The Hardware/Software Integrated Test System

Including an FPGA based accelerator for implementing parallel regular expression engines can result in a tremendous boost in the overall performance on an IDS. We have completely utilized the FPGA as well as general purpose processing resources to obtain maximum flexibility, as well as performance. Our integrated system utilizes both the FPGAs on the RASC blades. We dedicate two Itanium 2 processors for data transfer to / from the two FPGAs. The rest of the thirty Itanium 2 processors are utilized when: a) A given regular expression

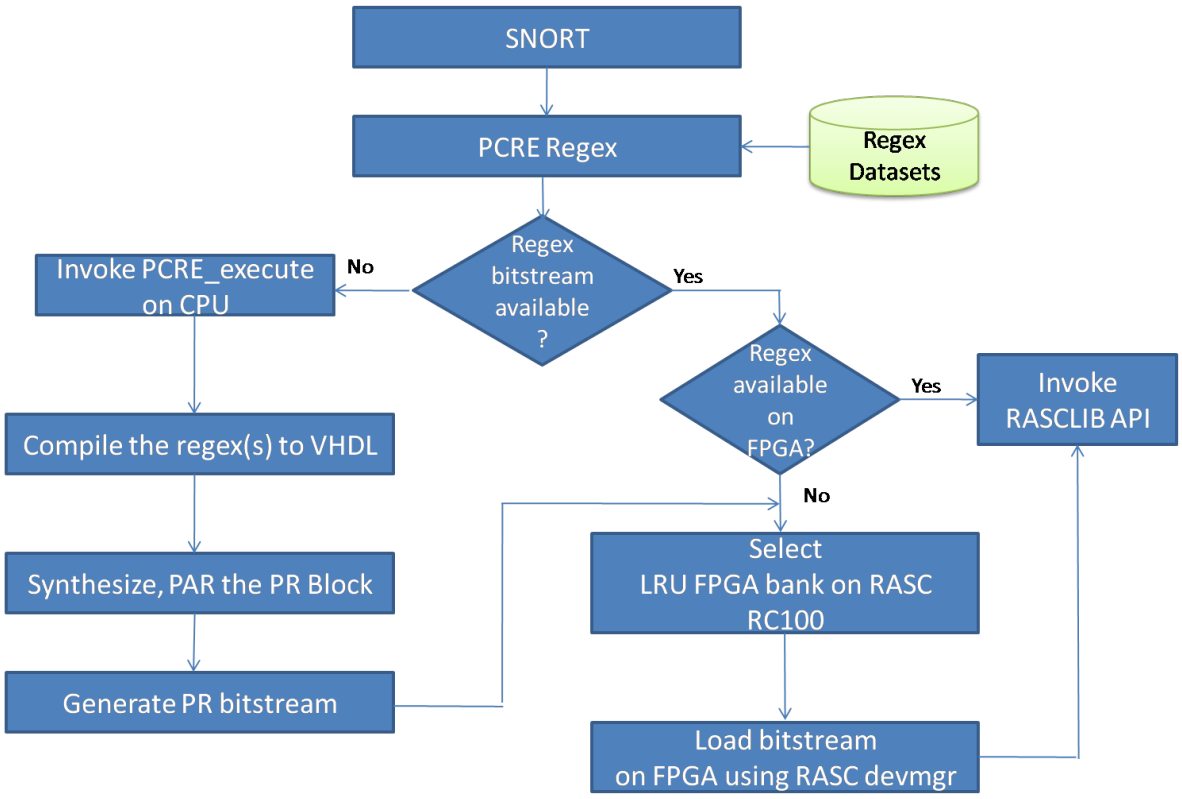

Figure 4.12: Using regular expression engines with an integrated FPGA hardware and multiprocessor software flow

required by SNORT is unavailable in any of the FPGA banks but exists as a partial bit-stream b) A given regular expression is updated in the SNORT rule-set and is in the process of being compiled, synthesized and implemented. When all the thirty two banks have been programmed with bit-streams, we swap the least recently used rule-set with the partial-bitstream of the new rule-set. The overall flow of operation of the integrated hw/sw system is depicted in Figure 4.12. The regular expression rule-sets are selected by SNORT depending on current network activity. If the rule-sets had been already compiled and are loaded on the FPGA, the RASCLIB APIs are invoked to transfer the payload data onto the FPGA on RASC Blade and achieve maximum throughput in hardware. If the bitstream is available, but is not loaded on any of the FPGA, then two processes occur simultaneously. Firstly the software threads are invoked to start executing the regular expressions on the thirty Itanium 2 processors. Simultaneously the least recently used FPGA bank is re-programmed with the new bitstream. And thereafter hardware acceleration is resumed. Only when the rule-set has not been compiled earlier (for e.g. when a rule is updated by the security community) then the complete bank of rule-set needs to be synthesized, placed and routed, and the partial bitstream needs to be generated before hardware acceleration is possible.

4.3.4 Hardware Performance

The hardware design after the synthesis, place and route flow, clocked at 155 MHz with 224 different regular expressions constituted from the four rule-sets viz. web-client, web-cgi, ftp and backdoor including the same rules that were used for the software execution. Thus

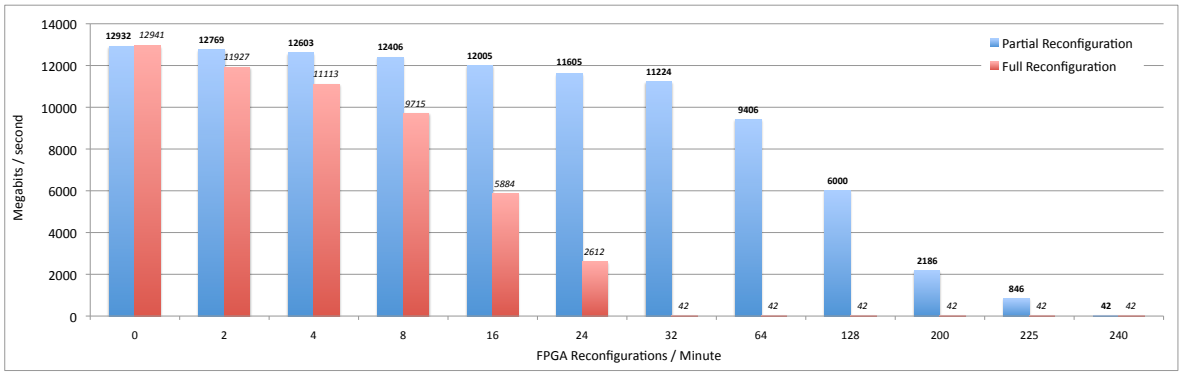

Figure 4.13: System Throughput with increasing number of FPGA reconfigurations / minute. During the FPGA reconfiguration, the software based PCRE engines are utilized. Data is plotted for both partial one bank reconfiguration and complete FPGA reconfiguration.

our design offers a theoretical peak throughput of 19.84 Gbps per FPGA, as each FPGA processes 16 Bytes Input and 16 Bytes Output per clock cycle.

We programmed the RC 100 algorithm clock to operate at a fixed frequency of 150 MHz. Each packet was processed through 14 regular expressions in parallel on each of the 16 banks on the FPGA. The average throughput obtained with 224 regular expressions implemented on any one of the FPGA with the test network dump data set on RASC RC 100 is (**12.9 GBits/s**). The actual throughput (12.9 Gbps) is less than the theoretical sustained throughput (19.84 Gbps) due to limitations, i.e. a shared crossbar, on the Altix system. Thorough details on the hardware design, performance and experimental results on RC 100 are discussed in Chapter 2 and [116]. When we include the second FPGA, we add another 224 regular expression to the hardware while still maintaining the 12.9 GBits/s aggregate throughput. It is also possible to achieve twice the throughput viz. 25.8 Gbps on RC 100, by programming both the FPGAs with the same set of regular expressions and by streaming in 32 different payloads, 16 to each FPGA on the RASC Blade. We do not use the second configuration at this time, since

our goal is to achieve 10 Gbps level performance for as many different regular expressions as possible.

4.3.5 Hardware/Software Performance with Reconfiguration

Section 4.3.3 describes the flow of operation while using the SGI Altix system with the FPGAs on RASC blade for implementing a regular expression based IDS. We have run experiments and obtained throughput results for two scenarios viz. with partial reconfiguration of one NFA bank at a time, and also with full reconfiguration of the FPGA. Since the algorithm clock is suspended during any kind of reconfiguration, the RASC blade cannot function during that time frame. Therefore the software based PCRE engines are utilized while reconfiguration.

We run 30 parallel threads, each thread running 14 regular expressions per packet on the SGI Altix when the RASC blade undergoes reconfiguration. The reason for running 14 regular expressions per packet in software is that, in hardware, each payload data is scanned by 14 regular expressions. The payload data is the aforementioned 10 GByte network dump. We have analyzed the system throughput with regular expression rules from web-client and web-cgi rule-sets. Since the hardware performance is agnostic to the extent of malicious activity on the network payload dump, we run the software subsystem on high malicious activity scenario, in order to demonstrate the worst case performance scenario of the hardware/software IDS.

As seen in Figure 4.13 we have varied the number of reconfigurations per minute from

zero to 240. We repeated the experiments for both the partial and full reconfiguration scenarios. Since the time required for a full reconfiguration is around two seconds while that for a partial reconfiguration is 0.25 seconds, the effective throughput decreases much more rapidly while under going multiple full reconfigurations per minute. Even at 32 partial reconfigurations per minute, the FPGA hardware supports a throughput of more than 10 Gbps. On the contrary, while executing the same number of full reconfigurations per minute, the FPGA hardware is essentially spending all the time reconfiguring, thus rendering itself unusable. The system then runs entirely on software thus performing at only 42 Mbps. The 10 Gbps cutoff points on the SGI Altix system for partial and full reconfiguration are thirty two reconfigurations / minute and four reconfigurations / minute respectively.

Projected Throughput with DPR

Virtex-4 FPGAs support Dynamic Partial Reconfiguration (DPR), where the rest of the FPGA may keep on operating while the partial bitstream is being loaded on one of the NFA banks. Such a scenario offers two kind of advantages, the first being that only the regular expressions corresponding to the NFA bank being reprogrammed, need to be run on software while the hardware is being reconfigured. Secondly the hardware acceleration throughput is reduced by only a fraction, since fifteen out of sixteen banks can continue operating. We have calculated the projected throughout with DPR on RASC Blade, and that is plotted in Figure 4.14. Such a hardware can be easily designed for running an adaptable IDS. This system has a worst-case performance of around 12.1 Gbps, i.e. the case when partial reprogramming the FPGA two

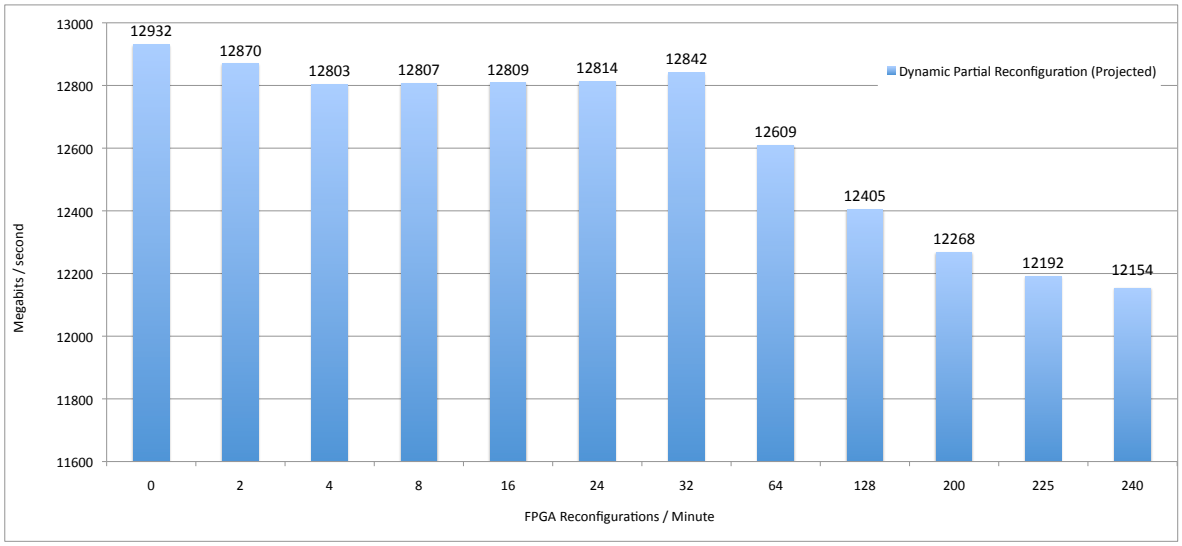

Figure 4.14: Projected System throughput with increasing numbers of DPR per minute. With DPR, only the regexes corresponding to the NFA bank being reprogrammed need to be run in software.

hundred and forty times a minute (since the time taken for a single partial reconfiguration is about 0.25 second).

4.4 Conclusion

We have extended the ROCCC compiler infrastructure and have been able to effectively synthesize co-processors selected form Xilinx Logicore IP Core library. These IP Core libraries have been instantiated on the Virtex-4 CSoc. Using the generated IP Core wrappers, we have been able to connect the IP Cores to the Virtex-4 FX APU. The wrapped IP cores are synthesizable accelerator modules which are instantiated using a C function call in software. By incorporating the partial reconfiguration flow for FPGAs with the IP Core, we have effectively shared the reconfigurable fabric among various IP Cores, to overcome area limitation

on CSoCs. The dynamic IP Cores are instantiated from a C function call by means of programming a partial bitstream in the FPGA.

We have also demonstrated the use of partial reconfiguration on FPGAs for implementing an adaptable regular expression based IDS. We have proposed a novel hardware/software integrated system using CPUs and FPGAs, that can quickly adapt the hardware IDS to changing network payload types and still maintain an overall throughput greater that 10 Gbps. We have implemented 448 regular expression engines across two Virtex-4 LX 200 FPGAs, offering a sustained throughput of 19.84 Gbps, and 12.9 Gbps on a RASC blade. We have integrated SGI Altix 4700 and RASC blade into an IDS that maintain the functionality during partial and full reprogramming of the FPGA, by executing the regular expression on multiple load balanced CPUs. This proof-of-concept system also demonstrates quickly adaptability by virtue of partial re-configuration. Our proof-of-concept system provides greater or equal to 10 Gbps throughput reconfiguring itself upto 32 times a minute.

Chapter 5

Boosting XML filtering with a scalable FPGA-based architecture

In this chapter we document XML filtering and how we implement multiple XPATH expressions on FPGA, by converting them to PCRE and then into VHDL hardware blocks. We discuss the implementation of the ancestor descendant and parent-child axis in XPath. We also describe how we implement the parent-child axis by incorporating a hardware stack into the regular expression engines. We document the character pre-decoding block, and its impact on area/clock speed. Finally we include details on matching twig queries on FPGA.

5.1 XML Pub-sub

The growing amount of XML encoded data exchanged over the Internet increases the importance of XML based publish/subscribe (pub-sub) and content based routing systems. The

input in such systems typically consists of a stream of XML documents and a set of user subscriptions expressed as XML queries. The pub/sub system then filters the published documents and passes them to the subscribers. Pub/sub systems are characterized by very high input ratios, therefore the processing time is critical. In this chapter we propose a "pure hardware" based solution, which utilizes XPath query blocks on FPGA to solve the filtering problem. By utilizing the high throughput that an FPGA provides for parallel processing, our approach achieves drastically better throughput than the existing software or mixed (hardware/software) architectures. The XPath queries (subscriptions) are translated to regular expressions which are then mapped to FPGA devices. By introducing stacks within the FPGA we are able to express and process a wide range of path queries very efficiently, on a scalable environment. Our experimental evaluation reveals more than one order of magnitude improvement compared to traditional pub/sub systems.

Publish/subscribe applications (or simply pub-sub) (Figure 5.1) are an important class of content-based dissemination systems where the message transmission is guided by the message content, rather than its destination IP address. System architectures may vary (centralized within a server or distributed over a network of brokers) but they all follow the same asynchronous event-based communication paradigm. The input is a stream of messages, generated outside of the system by a set of *publishers*. These messages are then selectively delivered to interested *subscribers* that have declared their interest by submitting *profiles* to the pub-sub system. This process is also known as *message filtering*. Examples of pub-sub systems include notification websites (e.g. *hotwire.com* and *ticketmaster.com*), where a user

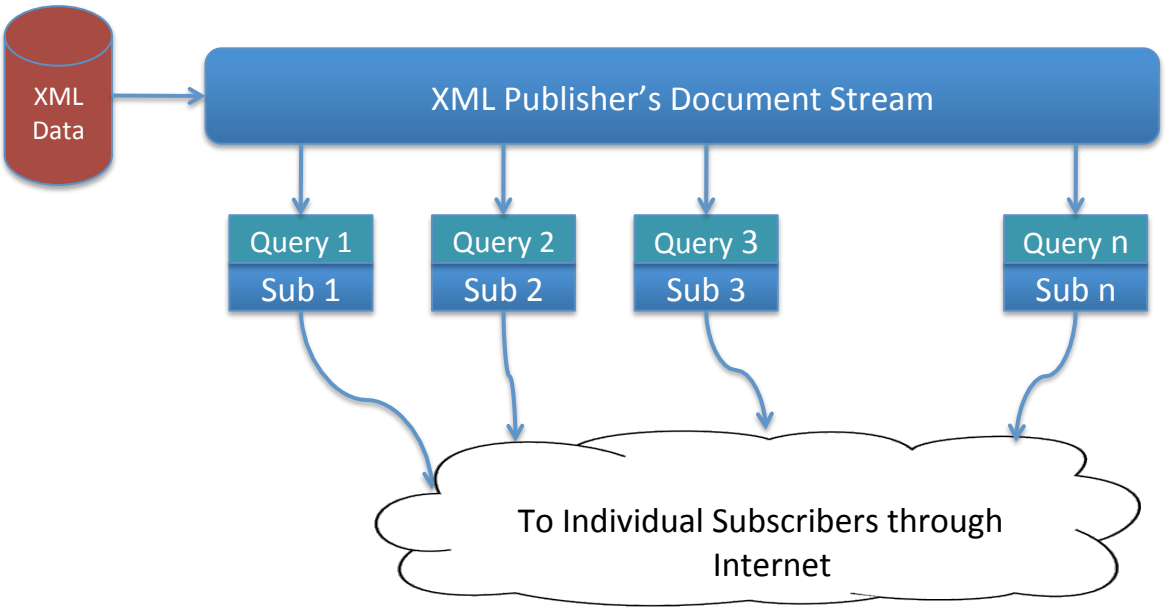

Figure 5.1: An XML Publish Subscribe System. A published XML document stream is parsed and filtered through multiple subscriber profiles.

can subscribe for specific events ("Rock concerts in Chicago") and get automatic notifications when the event occurs. Increasingly such environments are becoming XML-based, i.e., the messages are exchanged in XML while the users express their subscriptions using XML query languages like XPath.

Given the high volumes of messages and profiles, the filtering process becomes a critical performance requirement for pub-sub systems. The predominant solutions to this problem perform clustering of the user profiles based on their similarity in order to narrow down the search in the profile space. This is done by the use of Finite State Machines (FSM). In particular, elements of the user profiles are mapped to states in the state machine. The clustering is then performed by combining multiple profiles in a single FSM by analyzing and discovering the common profile paths. Since user profiles are typically known in advance

(i.e., they play the role of data, while documents play the role of traditional queries) it is possible to be analyzed and clustered as needed before the filtering process starts.

When a document arrives in a pub-sub system, it is parsed by an event-driven parser like SAX [3] that reports low level parsing events such as: "start document", "start element", etc. As events are produced by the SAX parser, they are processed by the filtering system which uses them to drive transitions between the FSM states. For example, a transition is taken from the current FSM state if there is an outgoing edge labeled with the tag currently being processed. If during this process an "accept" FSM state is reached the document satisfies the corresponding profile(s) associated with that state.

5.1.1 Using FPGA for XML Filtering

The above described approach however is highly oriented towards the traditional von Neumann architecture model which requires multiple clock cycles per instruction to analyze a single XML tag. This issue is known as the *von Neumann bottleneck* and can limit the filtering speed to few hundreds of clock cycles per single XML tag. While parallelism can be achieved with multi-core machines (as a software-hardware solution), FPGAs offer a viable alternative due to their power efficiency (less power consumption and cooling costs) [167, 85] as well as higher throughput. The work in [74], quantitatively demonstrates the benefits of using FPGAs over general purpose CPUs for streaming applications. While multi-core systems come with 2 and 4 CPUs it is not always feasible to achieve proportional speed-up due to the bottleneck in shared cache memory and the front side bus.

The way to resolve this limitation is to use a non-traditional highly parallel architecture. In this chapter we present a novel filtering approach which is based on the use of Field-Programmable Gate Arrays (FPGA) to find the matching profiles for given document.

FPGAs are increasingly being made available as co-processors on high-performance computation systems. They are packaged in modules, which are dropped in CPU sockets on server motherboards with bridges to the FSB on Intel platforms and Hypertransport on AMD. High density FPGAs such as Xilinx Virtex-4LX 200 and Altera Stratix EPS20 have millions of logic gates, abundant high speed dual port memory and ALU blocks on the silicon fabric. These high density FPGAs can be used to implement in hardware the computationally intensive portions of the software code. Multithreaded software components with streaming data input and output like the pub-sub applications are ideal candidates for acceleration on FPGA co-processing systems since a huge amount of data can be processed in parallel on the FPGA.

Since pub-sub XML filtering involves multiple queries processed over a single document data-stream, it is possible to utilize FPGAs for parallelising the filtering performance. Each query can be implemented on the FPGA unit as a hardware datapath circuit and with appropriate optimizations it is possible to fit thousands of queries on a single FPGA chip. Moreover, having the parallel processing modules implemented on the same chip eliminates the need for expensive communications between them. This in turn allows for full pipelining of the parsing and filtering processes: as an event is produced by the parser it is immediatelly forwarded to the filtering module. This results in accelerated query processing and further-

more leads to substantial savings in a general purpose computation infrastructure by reducing the amount of power required by the CPUs.

In this chapter we present a 'proof of concept' for the use of FPGAs in boosting XML filtering performance. We first describe the case where a user profile query is expressed as an XPath query. We utilize a four step approach that converts such query into hardware description, suitable for implementation on FPGA. The first step involves conversion of an XPath query to PERL compatible regular expressions (PCREs). The regular expressions are clustered by their common prefixes in order to produce more compact representation on the board and are then translated to VHDL using our "regex to VHDL" compiler [116]. Moreover, in order to support parent-child relationships, we introduce the use of stacks and modify the regular expression hardware to use them. The highly optimized VHDL code is then deployed on the FPGA board. The stream of documents is forwarded to the board where it is processed with high degree of parallelism. Our experimental evaluation reveals that this architecture achieves orders of magnitude improvement in the terms of running time compared to the state of the art software based XML filtering systems.

Furthermore, our system can also consider profile queries expressed as twig patterns. A straightforward solution to implementing twigs would be to decompose them into individual path queries and process them individually; this however requires an extra post-processing step that combines the results and eliminates false positives. Instead, we transform the document and the queries into sequences and perform filtering as subsequence matching. The document and twig query structures are captured using Prfer sequence encoding [141, 97].

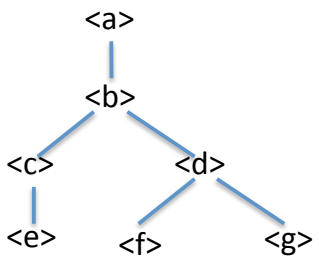

Figure 5.2: An example XML tree

Here we take advantage of the fact that subsequence matching (like parsing and filtering) is also a "linear" kind of processing that can be performed very fast on FPGAs.

5.2 Compilation System Overview

We describe a general overview (Figure 5.3) of our compilation workflow, which loads the filtering logic on the FPGA chip. Detailed description of the individual steps in the workflow follows later in this section.

5.2.1 XPath Expressions

XPath profiles / expressions or simply XPaths are UNIX directory like expressions which contains a given sequence of tags and relationship between them. This makes XPath profiles simple to use and understand. Several XPath expressions can be combined efficiently into optimized query trees. The XPath profiles are used to verify a certain subtree structure in the document. Upon identification of the subtree, data is retrieved from that subtree, of the XML document. The two important relationship operators used in XPath are the ancestor

descendant "//" and parent child "/". As shown in Figure 5.2, tag <a> is the global ancestor of all the other tags in the tree. Descendants are all the tags which are below a given tag, for example both <c> and <e> are descendants of tag <b> and also tag <a>. Children of a parents are the tags immediately succeeding a parent tag, for e.g. <b> is a child of parent <a>, <f> is a child of parent <d> and so on. Our hardware implementation with a tag stack allows us to support parent child axis from the XPath.

5.2.2 XPath on FPGA

FPGA devices allow implementation of multiple datapaths operating in parallel which makes them suitable for streaming applications like XML filtering. Moreover, because the datapath is implemented in the hardware, the load and store operations from the von Neumann model are eliminated resulting in more efficient processing.

As it can be seen from Figure 5.3 in the first step of the compilation workflow the tag elements in the XPath expressions, representing the user profiles, are replaced with fixed length string encodings. This is done to simplify the processing and to ensure that each tag element occupies the minimum amount of area possible on the FPGA device. Reducing the footprint of the individual XML tags results in higher query density on the chip and thus better usage of the hardware.

After this step the XPath expressions are translated to their equivalent PCRE form. During this translation process the navigation directions inside the XPath expression (parent-child "/" and ancestor-descendant "//") are replaced with their PCRE counterparts. We de-

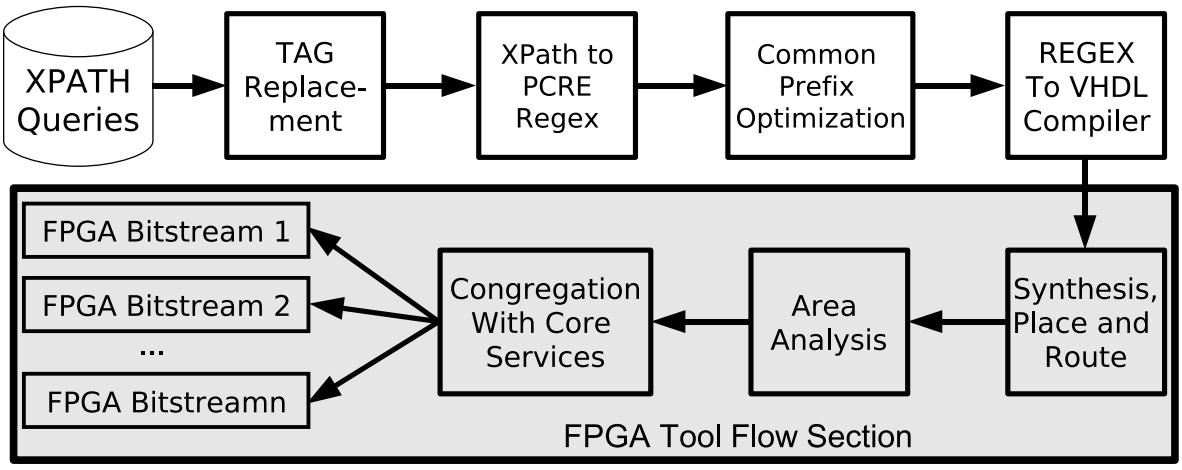

Figure 5.3: Compilation Flow of XPath expressions to FPGAs. The XPATH profiles go through a four step compilation process to generate the HDL. The lower gray section denotes the hardware flow for converting HDL to a bitstream for the FPGA.

scribe this process in detail later in this section. In order to further reduce the query footprint on the FPGA device we cluster the regular expressions by their common prefixes. Those common prefixes are implemented as a single block on the FPGA unit. The result from the clustering step is a forest of "common prefix" trees. Each tree is compiled to generate a set of VHDL hardware blocks. The rest of the workflow involves FPGA specific compilation steps which will be discussed later as well.

5.2.3 Dictionary Replacement

The area of the FPGA chip is a limited resource. In order to get better usage of it we minimize the tag footprint on the chip through a dictionary replacement process which replaces the XML tags in the input documents and the user profiles with a fixed length strings. In our implementation the length of the strings is set to 2 symbols which means that the size of all open tags is limited to 32 bits (2 symbols plus 2 tags markers of length 8 bits) and close tags

Table 5.1: PCRE operators used for implementing XPath profiles on FPGA

Operator	Meaning
\w	Matches A to Z, a to z, 0-9, _
\s	Matches a blank space
\c	Matches A to Z, a to z
\d	Matches a Decimal digit
+	Repeat 1 or more number of times
*	Repeat 0 or more number of times

- to 40 bits. As an example, <test.document> tag is mapped to <a1>, while the closing tag </test.document> would map to </a1>.

5.2.4 XPath to Stack-enhanced Regular Expressions

If the XPath expression contains only the ancestor-descendant axis the translation to regular expression is straightforward. While the YFilter approach, maps an XPath profile to a sequence of NFA states connected with transitions, our approach maps an XPath profile to a regular expression. As an example the XPath profile "$a0//b0$" will be translated to the regex " <a0> [\w\s]$^{+}$ [<\c\d> — </\c\d>]* <b0> ". The various regular expression operators are explained in Table 5.1.

The regular expression in the above example accepts a sequence of XML tags which starts with <a0> and includes <b0>. It first matches the tag <a0>. Once this is matched, it will look for one (or more) characters (the [\w\s]$^{+}$ part) corresponding to text between tags and then will check for any number (0 or more) of open OR closed tags (the [<\c\d> — </\c\d>]* part) before it matches <b0>.

Moreover, in order for <b0> to be a descendant of <a0> in the document, the regular

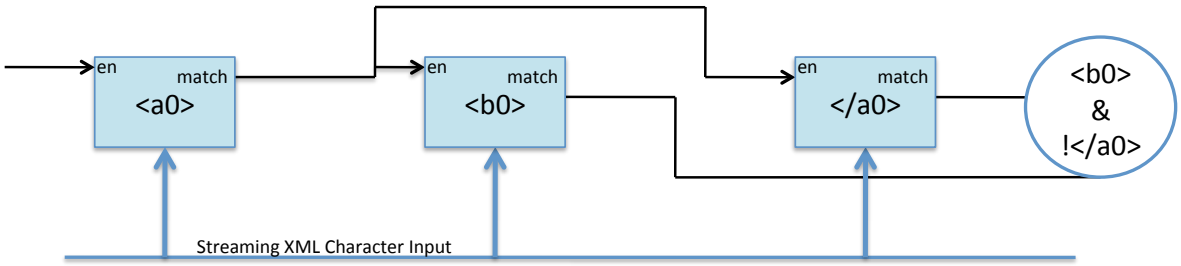

Figure 5.4: The block diagram for XPath <a0>//<b0>, showing the implementation of the ancestor-descendant axis

expression should match before the closing of <a0>. To implement this, during the hardware generation step for this regular expression, our compiler automatically adds a negation block on </a0> so that <b0> is matched before </a0> appears in the stream. The block diagram of the regular expression on the FPGA is shown in Figure 5.4. Each block represents a tag parser that searches for the given tag in the document stream. The right most hardware block (depicted as a circle), provides the final result from the matching process of the regular expression. Each block receives input from the 8 bit streaming XML interface and works in parallel with the other blocks.

The translation of the parent-child axis to a regular expression requires special treatment. This is due to the fact that the regular expressions are memoryless structures and one needs to ensure that the matched XML tags occur on consecutive levels in the document. For example, the level on which the parent is matched should be remembered so as to ensure that the child is matched on a consecutive level (e.g. it is immediately below the parent). The regular expression hardware is thus modified to include the notion of memory. In our implementation this is accomplished through the use of a tag stack which keeps the current path in the XML document. When an open tag is encountered it is pushed into the stack.

Similarly when a close tag is reached it is popped from the top of the stack (TOS).

An example of a XPath expression that includes parent-child axis is shown in Figure 5.5. The XPath expression "a0/b0" is translated to a modified regular expression with a stack control directive. The modified regular expression is: " <a0> [\w\s]$^{+}$ [<\c\d> — </\c\d>]* [Stack1] <b0> ".

When testing a parent-child relationship, in addition to checking for the ancestor-descendant property we have to ensure that the level difference between the respective tags is one. Hence we use an extra hardware block - the TOS matching -, which continuously monitors the top of the stack and ascertains that the matched element <b0> is indeed a child of the previously matched element <a0>.

Figure 5.5 describes how we monitor the current level. The XML tag stack block, works in parallel with the ancestor-descendant block on the FPGA. The additional Tag Filter block extracts XML tags from the document stream. When an open XML tag is extracted, it triggers the push function and this tag gets pushed into the stack. In a similar way closing tags trigger the pop function and remove the head of the stack. A difference with the previous ancestor-descendant match is that finding <b0> after <a0> is not enough; we need also that the top of the stack is <a0> (when <b0> is found).

Since many regular expressions are using the same XML input data stream, we need only one stack block per data stream.

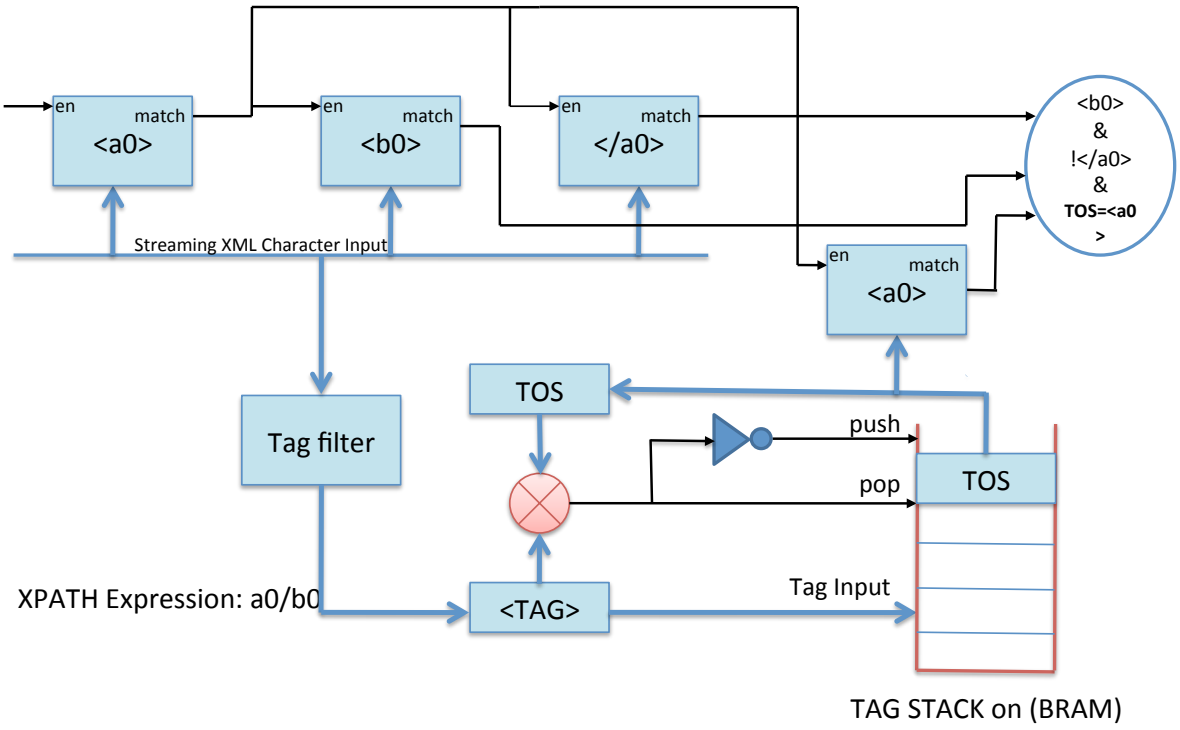

Figure 5.5: The block diagram for XPath <a0>/<b0>, showing the implementation of the parent-child axis. The additional hardware includes the tag filter, stack and TOS match blocks

5.2.5 Common Prefix Optimization

The regular expressions derived from the XPath profiles typically depict large commonality in their prefixes. For example "a0//b0// c0//d0" and "a0//b0//c0//e0" share the common prefix "a0//b0//c0", with corresponding suffixes "d0" and "e0". The hardware cost of implementing the regular expressions is measured in terms of the FPGA area used to implement the logic. It is thus advantageous to combine multiple regular expressions into a common prefix tree. Such a tree can help reduce the area cost of the hardware by implementing the common prefix as a single block on the chip. In the above example, instead of implementing two regular expression hardware blocks and duplicating the "a0//b0//c0" logic, we can have a single implementation for the common path. As a result, more profiles can fit in a given

FPGA area.

Given a set of XPath profiles, we first create their regular expressions and then sort them in alphabetical order. We then run a common prefix discovery algorithm on the sorted list of the regular expressions. The algorithm recursively grows the common prefix one tag at a time. The result is a forest of common prefix trees, each representing a set of profiles. From these trees we then create the FPGA hardware.

5.2.6 Area Efficient Character Decoder Hardware

Implementing XPath profiles on FPGAs mainly involves implementing character matching blocks to identify XML tags in the input document stream. The character matching hardware block compares sequences of characters from the input stream to a given sequence that defines an XML tag. Figure 5.6 exemplifies the comparator hardware that matches an XML tag. Each character requires an 8-bit comparator block. The implemented character matching blocks for the XML tags consist of many redundant blocks, the prime examples being the open '<', close '>', and end tag '/' characters.

It is possible to simplify the character match hardware with a 8-bit ASCII character pre-decoder. The character pre-decoding hardware decodes the incoming ASCII data stream at the input. An 8-bit input is decoded into one of 256 possible 1-bit character signal every clock cycle. As an example, if the input was HEX '0x60', the output line for the character 'a' would be high on that clock cycle and the rest of the other 255 outputs would be all zeros. The character decoder hardware block simplifies character matching by replacing 8-bit

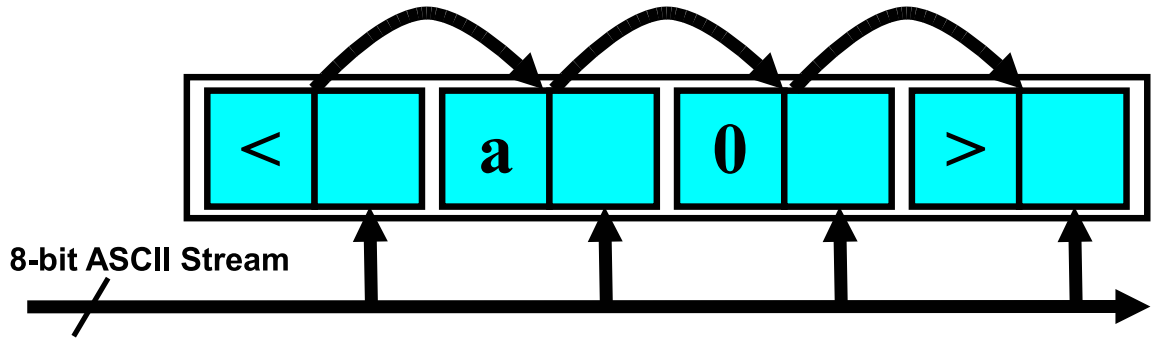

Figure 5.6: Block diagram of the Character Match Hardware Block for a tag <a0>. The hardware is a 8-bit x 4 comparator block.

character match hardware blocks with a 1-bit comparator and results in area efficient hardware. Figure 5.7 depicts the character pre-decoder block, and the simplified 1-bit comparator blocks for matching an XML tag. Moreover since 1-bit data lines are routed on the FPGA for each character in the XML tag, the FPGA routing overhead is reduced, which in turn leads to a design which offers faster clock speed.

5.2.7 Regular Expression to VHDL compilation

A regular expression syntax could be defined using various syntaxes such as PERL, UNIX, etc. Our implementation uses the PERL semantics. The compiler uses a modified version of the PCRE library v6.7 compilation flow. It simulates the behavior of a regular expression in VHDL, suitable for implementation on FPGA. We modified the compiler to take into account the stack directives and generate the hardware blocks to support parent-child axes.

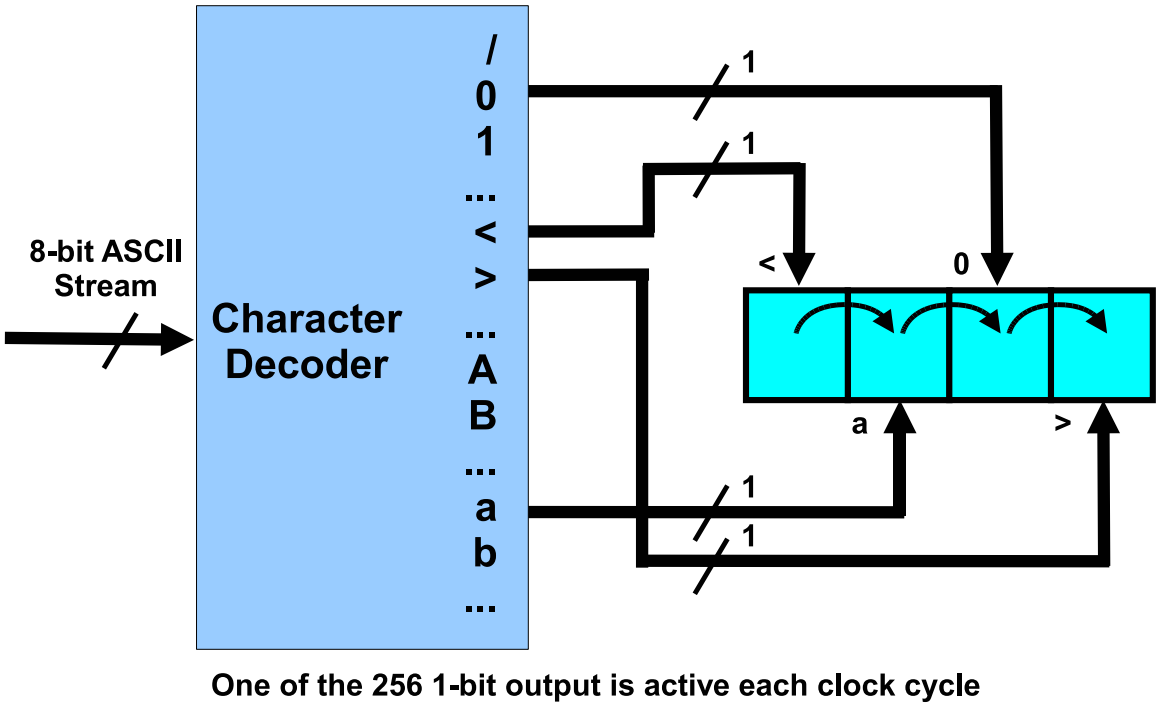

Figure 5.7: Block diagram of the Character Pre-Decoder Hardware Block for a tag <a0>. The hardware is a 1-bit x 4 comparator block.

5.2.8 FPGA Implementation

After obtaining the VHDL sources for the user profiles, we generate additional hardware blocks including an input ASCII decoder, two output priority encoders (one each for queries with or without parent-child axes) and the tag stack. We group the VHDL sources into two sets, i.e. profiles without parent-child axes and profiles with parent-child axes. The organization of XPath expressions on the FPGA is depicted with an example in Figure 5.8. The four XPath profiles on the left correspond to expressions that contain parent-child axes and thus use the on-chip FPGA stack. When the streaming document matches a given profile, the output priority encoder is set to that profile.

We synthesize the generated VHDL code, using the XILINX synthesis tool to obtain the

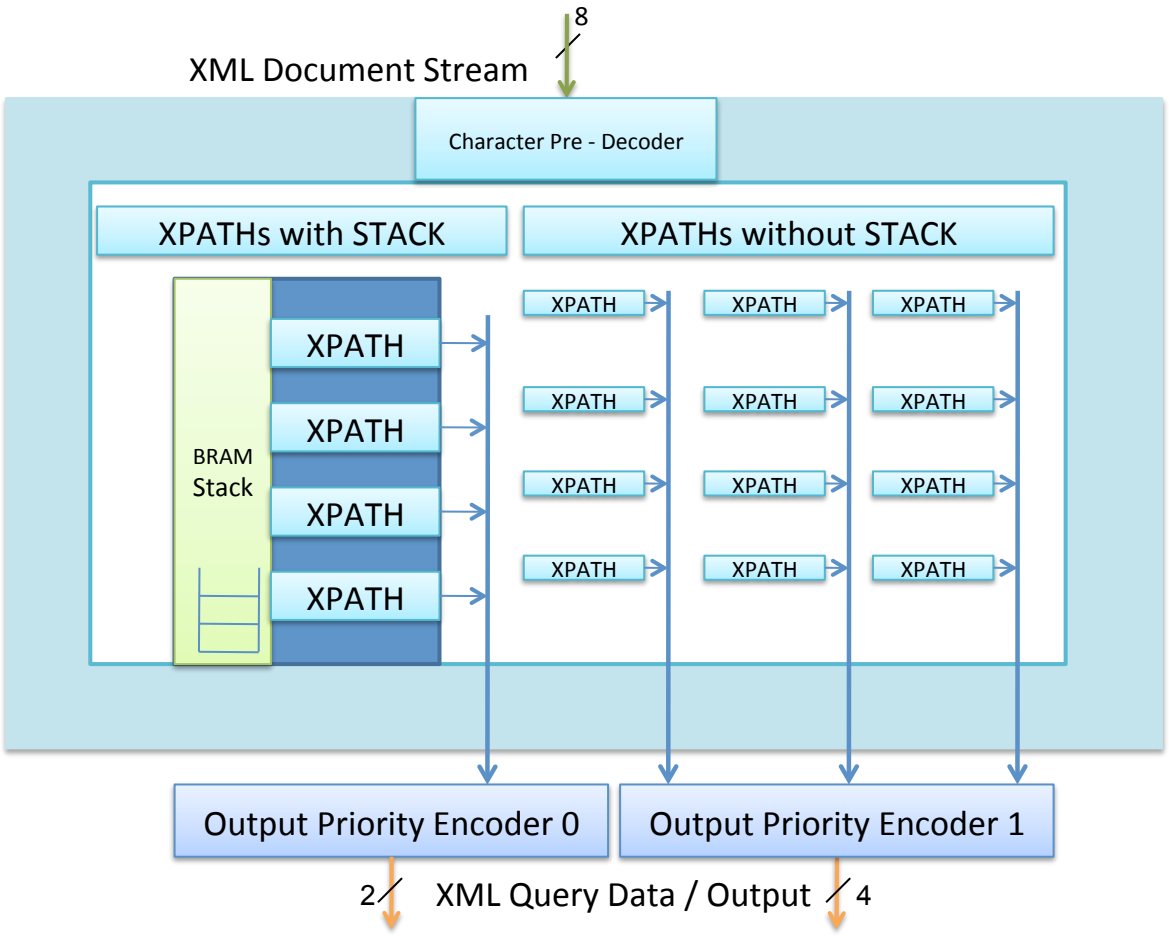

Figure 5.8: An example FPGA organization denoting the input / output data path with sixteen XPath expressions.

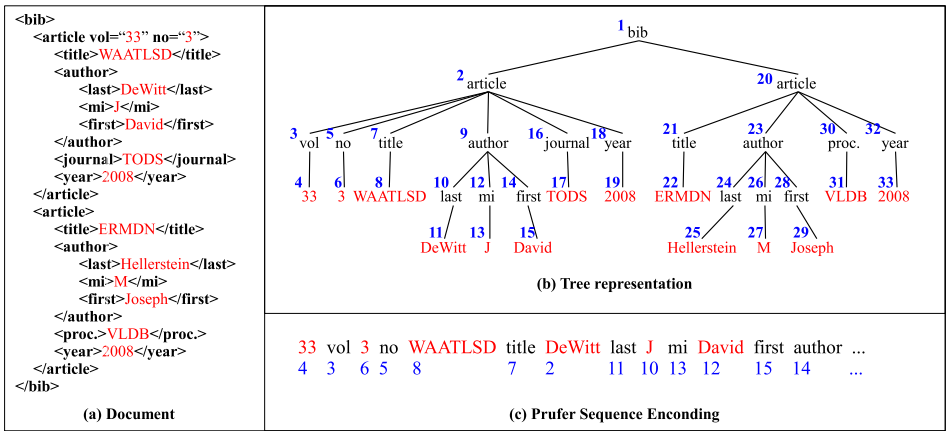

Figure 5.9: XML document, Tree and Prufer sequence representation.

hardware netlist. The next step involves running the Place and Route tool, which report the clock frequency of the hardware design.

Our target FPGA is a Virtex-4 LX 200 device, and the target hardware is the Silicon Graphics RASC RC 100 board. In order for our FPGAs to run on this board we had to add a hardware module (RASC Core Services) which allows us to send and receive data and control the FPGA from the host system. Finally we generate the bitstreams that are loaded on the FPGA.

5.3 Twig Profiles on FPGAs

In this section we will describe an extension of the basic system architecture, described in 5.2, designed to handle XML twig pattern queries. Unlike XPath queries, which can only look for the presence of a given path inside the structure of the XML document, the twig pattern queries can be used to locate more complex structures like subtrees.

One possible solution for the twig pattern matching problem is to decompose the twig

query into individual paths and process them separately. The results from the individual paths are then combined in a post processing step to produce the final outcome of the query. This approach however requires extra processing time in the post filtering step. Moreover the common sections of the individual paths will be processed multiple times which is redundant.

Because of all these disadvantages we choose to employ more holistic approach based on Prfer sequence encoding of the XML document. This approach have been used in the past in software based XML filtering systems like PRIX and FiST. In the next subsection we provide a brief overview of the Prfer sequence encoding and the filtering systems based on it. Then we proceed with the description of our hardware based approach.

5.3.1 Overview of Prüfer Sequences

The term Prüfer sequence (or Prüfer code) is originaly used in the graph theory to describe a unique sequential encoding of a labeled tree. It can be generated by a simple iterative algorithm. The algorithm takes as an input a tree with n nodes and produces as output a sequence of length n - 2 which encodes the structure of the tree, To do the algorithm iteratively removes nodes from the tree until all nodes but the last two have been removed. At each iteration the algorithm finds and removes the leaf node with the smallest label and adds to the the Prüfer sequence the label of this leaf's parent.

In the rescent years Prüfer sequences have been successfully applied in the XML domain in combination with a tree numbering schema. This is possiblke due to the fact that any XML document can be viewed as a tree structure where every tag element coresponds to

a node in the tree (see figure 5.9). Before proccesing the document with the Prüfer code generation algorithm a numbering schema is applied to it. The numbering schema (typically a preorder one) associates unique labels with the XML tags (the tree nodes) which are later used to determine the sequence in which the tags will be removed from the document. There are however two minor differences between the Prüfer code generated for XML documents in systems like PRIX and the original definition used in the graph theory: (i) In the XML domain typically the deletion of the tags from the tree continues till only one node is left. (ii) In the original definition of the Prfer encoding the leaf nodes are not encoded in the sequence. Only nodes which have at least one child form the Prfer sequence for the tree. To overcome this problem the document tree is expanded and grows in height with one level by adding artificial a child node to each leaf in the tree.

The reason which makes the Prüfer encoding of XML documents so apealing for the procces of twig pattern matching is given by the matching theorem, which states that if a tree Q is a subgraph of another tree T then the Prüfer encoding of Q is a subsequence of the Prüfer encoding of T. The reverse however is not true. It is possible to have subsequence matching between two Prüfer encoding of trees T and Q without Q being a subgraph of T. Theorem 1 guarantees having no false dismissals but it is possible to have false positives.

In software, using the existing XML parser SAX, the generation of the Prüfer sequence for XML documents is relatively simple by implementing the startElement and the endElement methods provided by the SAX default handler. The parser traverses the document in a preorder. Every time when the method startElement is invoked we place the tag element in a

Algorithm 1 Prüfer code generation

Require: document D
Ensure: The Prüfer V sequence encoding of D

```
1: Set V = 0                                          ▷ the resultant Prüfer sequence
2: Set S ← ∅                                          ▷ auxiliary stack
3: saxParser.parse(D)
4: procedure STARTELEMENT                             ▷ SAX parameters omitted for clarity
5:     S.push(node)                                   ▷ node is the element read
6: procedure ENDELEMENT                               ▷ SAX parameters omitted for clarity
7:     node = S.pop()
8:     V.append(node)
9:     if the current element is leaf then
10:        node = S.pop()
11:        V.append(node)
```

stack. Every time when the method endElement is invoked we pop an element from the stack and append it to the document encoding. Systems like PRIX and FiST use this encoding to match it to the Prüfer sequence encoding of the queries to determine if there is a match.

5.3.2 FPGA implementation of Prüfer subsequence matching

In this subsection we continue with describtion of our hardware implementation of streaming Prüfer sequence conversion of an XML document. We also describe how we can execute twig pattern matching using the generated Prfer sequence.

Figure5.10 shows the actual hardware block required for simultaneous Prfer sequence generation from the XML document stream and twig pattern matching. As the sequence is being generated, the hardware simultaneously matches the resulting sequence with twig patterns.

The hardware works by extracting XML tags from a document stream and pushing open tags on a hardware stack. This hardware stack can output both the TOS and TOS - 1 (the

element below the TOS) every clock cycle. Since each tag name consists of two symbols, the TOS and TOS - 1 blocks are sub-divided into TOS_0 TOS_1 and $TOS\text{-}1_0$ and $TOS\text{-}1_1$ blocks. and each of these sub-blocks are connected to character pre-decoder hardware. They are decoded into the corresponding character for subsequent use in sequence matching. A leaf detection hardware block, implemented as a state machine, detects XML leaf tags when a stack pop operation immediately follows a push operation. When a leaf tag is detected, the TOS and TOS - 1 are copied to the respective output register blocks. If a non-leaf XML tag closes, only the TOS - 1 i.e. the parent of the current closing tag is connected to the output block. Thus we are able to dynamically generate the PS of a streaming XML document. The dynamically generated PS of the document is matched with multiple twig patterns on the FPGA hardware by matching it with with pre-generated PS of twig patterns.

To implement the twig match in hardware, we convert the twig patterns to their PS before implementing them on the FPGA. While converting a twig pattern to its PS, we add a specification to identify the leaf nodes in the sequence. Once a twig pattern is converted to its PS sequence i.e. PS (Q) the hardware is generated as follows:

1. A leaf node and its parent obtained from the PS(Q) is converted to character matching blocks, connected to the TOS and TOS-1 registers using the appropriate character match line at the decoders. As mentioned above, The TOS and TOS - 1 blocks are connected to two character decoder blocks which decode the name of the twig node. As an example lets consider the sequence b0, a0. Since b0 is the leaf node, it is connected to the ‘b’ output of TOS_0 and ‘0’ output of TOS_1. The parent node a0 is connected to

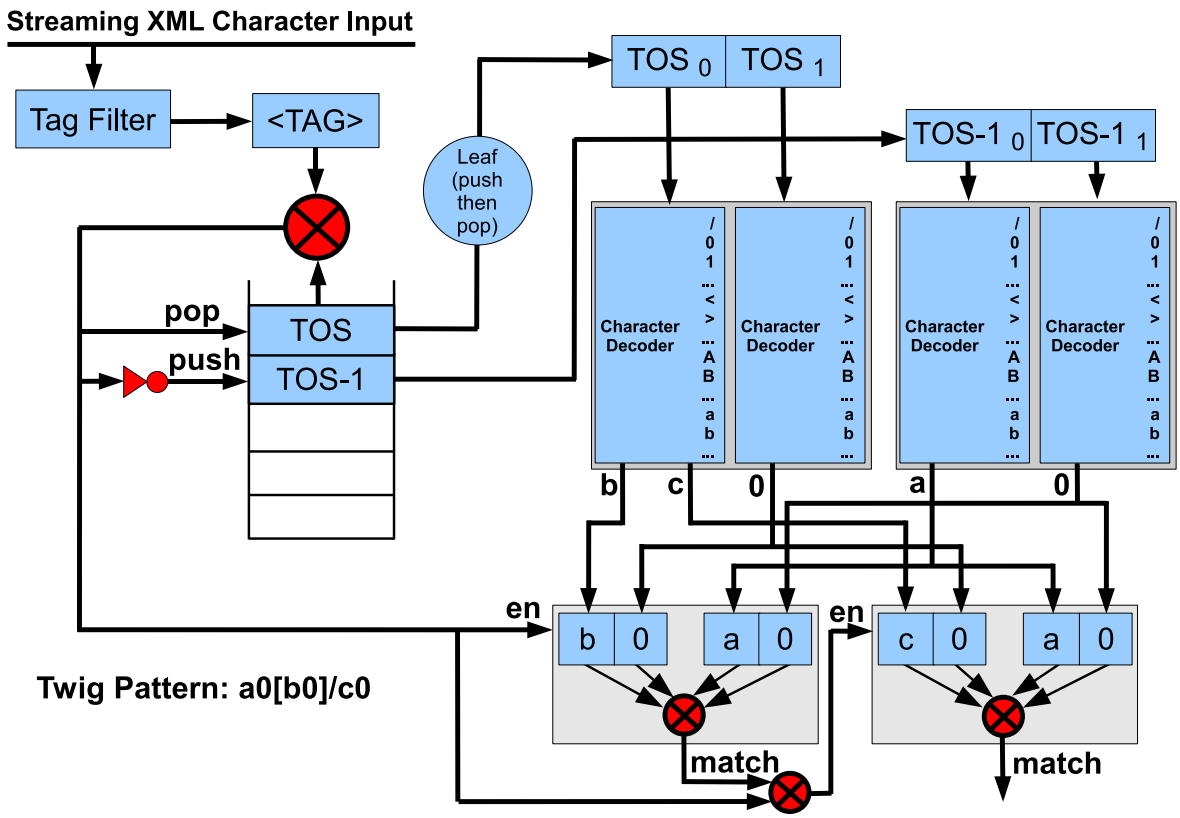

Figure 5.10: The block diagram for twig matching hardware. This block generates Prüfer sequence of the XML tags from a streaming document and matches it with Prüfer sequences of the queries in twig form. This figure is an example of the query a0[b0]/c0.

the ‘a’ output of TOS-1 $_0$ and ‘0’ output of TOS-1 $_1$.

2. Any non leaf node obtained from the PS(Q) is connected to the TOS - 1 block. As an example lets consider a sequence e0, f0. Since both are non-leaf nodes, they are both connected to the TOS-1 block using the ‘e’ ‘0’ and ‘f’ ‘0’ outputs of the corresponding character decoders.

3. In order to enable progressive sequence matching we establish a control mechanism. The first node executes a match when a tag is popped from the hardware stack. If the current node matched, it would enable the next hardware matching block. The block would then wait for the stack to pop again and execute a match. A succesful match would enable the next node to match , if any or else control is returned to the first node. When the final node executes a successful match it would imply a successful twig pattern match.

Our hardware can accurately match parent-child relationship in the twig patterns by utilizing the hardware stack. We can implement multiple twig patterns and they can all run in parallel on the FPGA.

5.4 Experimental Evaluation

We performed a preliminary comparison to assess the behavior of the proposed FPGA based solution and compare it with the traditional software-based filtering approach. For this purpose we use the YFilter version 1.0, considered to be the state of the art for software-based

XML filtering, running on a Core 2 Quad 2.66 GHz with 8GB of RAM available.

During the evaluation we measure the throughput of the system, e.g. the size of the set of documents (in megabytes) provided as input divided by the time (in seconds) between the moment when the set enters the system to the moment those documents are filtered by the matching process.

We used the *ToXGene* XML document generator [24] to generate the input profiles dataset. Using the same DTD structure we have generated different sets of profiles with varying path length, i.e. 2 Tags, 4 Tags and 6 Tags using the PathGenerator class in YFilter. The number of queries in each set varies from 16 to 1024. The streaming documents vary in size from one to eight MBs.

For the hardware implementation we use the Silicon Graphics Altix 4700 [151] computation system along with the RASC RC 100 [176] blade. We stream XML data stored in the memory (RAM) of the Altix system to the FPGA on the RASC blade. We also stream the output of the priority encoders from the FPGA back to the Altix system. The output of the priority encoders is also continuously decoded by the host system, to ascertain the XPath expressions that matched. When a match occurs, we output the profile that matched as well as the current location of the match in the document stream. The experiments are run by programming the FPGA with a bitstream containing the same queries generated for the software version.

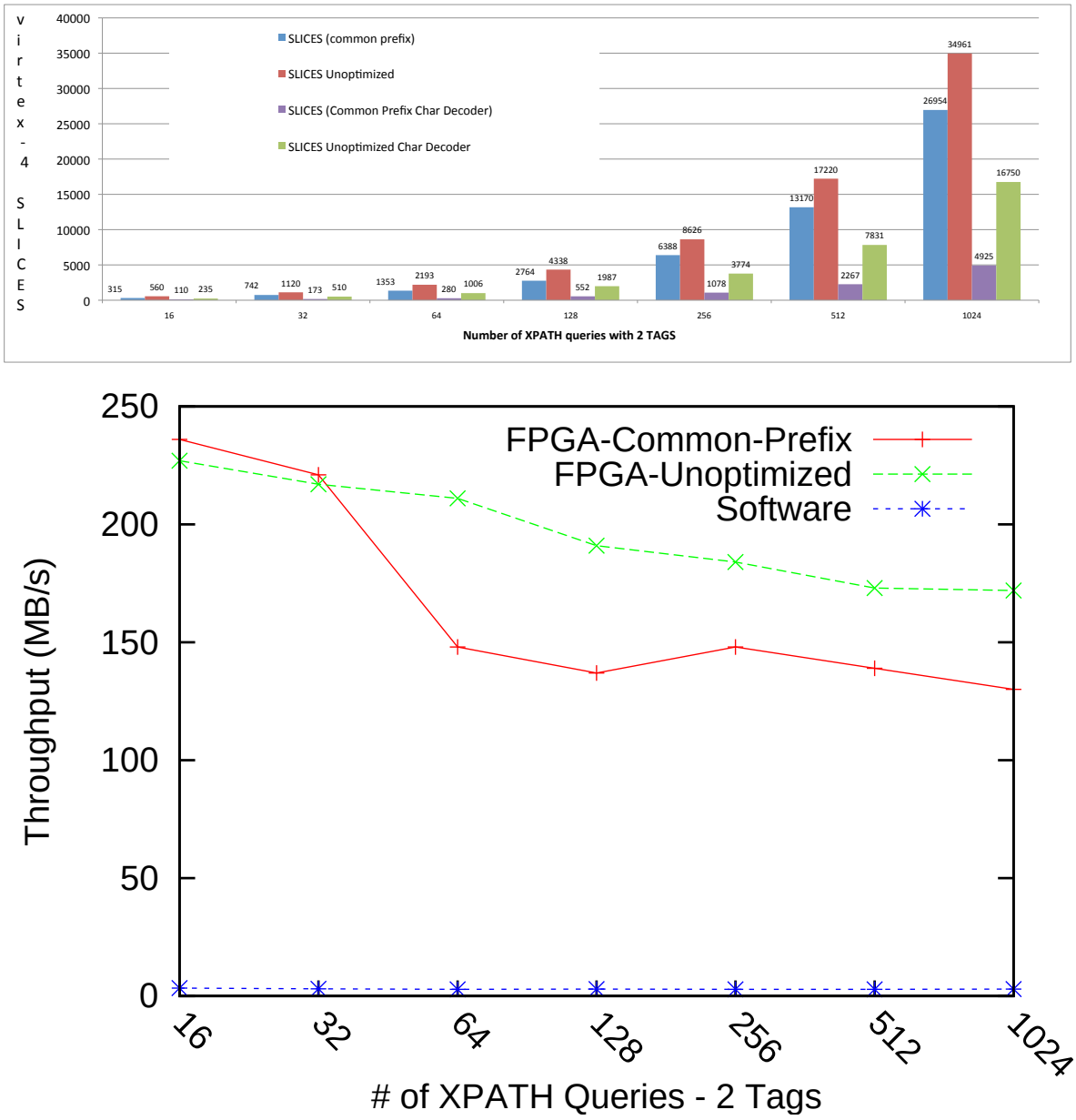

Figure 5.11: The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of two tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of two tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware.

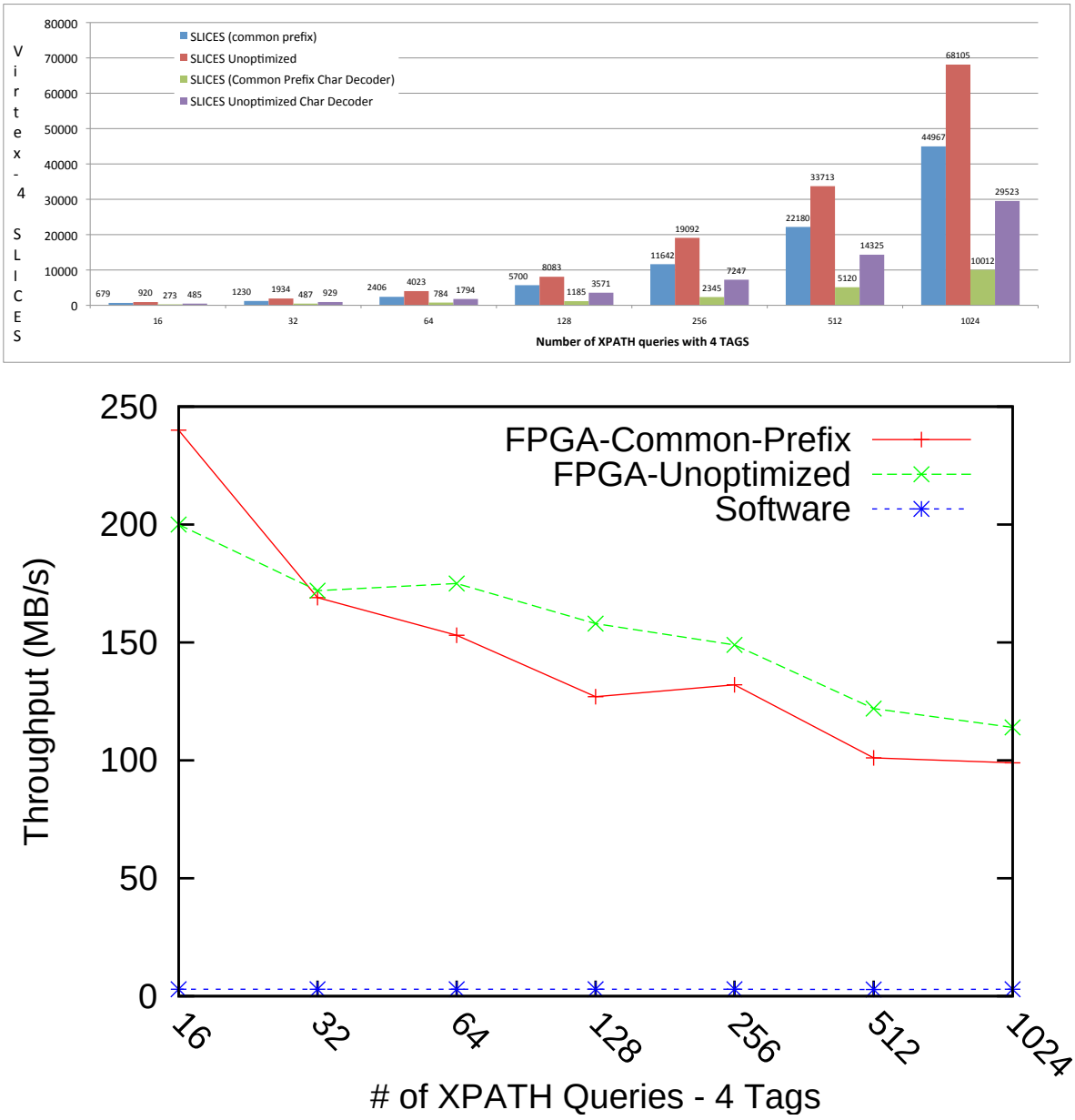

Figure 5.12: The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of four tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware.

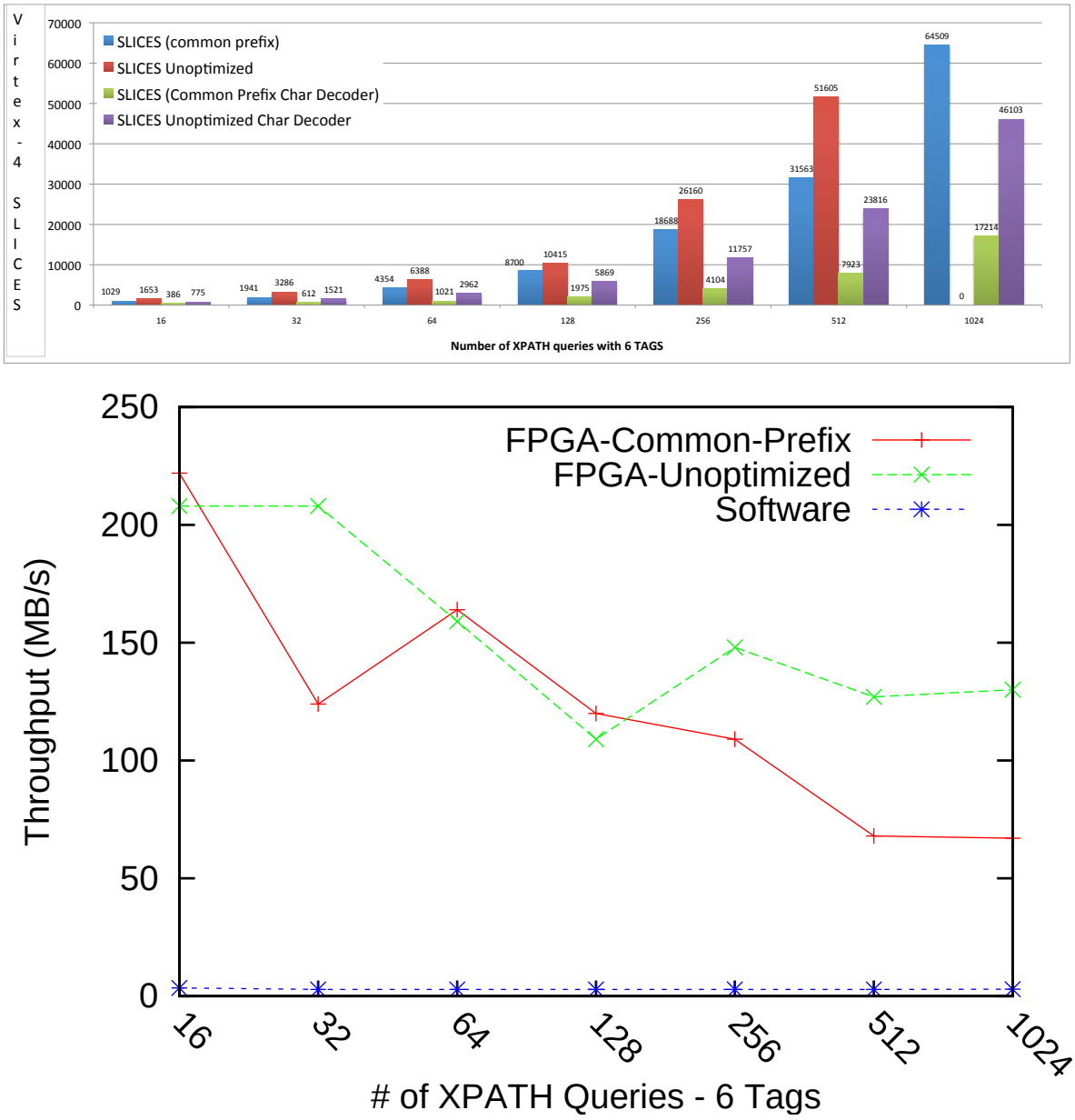

Figure 5.13: The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags long XPaths. The graph at the bottom demonstrates the variation of throughput of the FPGA hardware in MHz with increasing number of four tags long XPaths. The four cases considered are common prefix optimized and unoptimized XPaths with character match blocks and character decoding hardware.

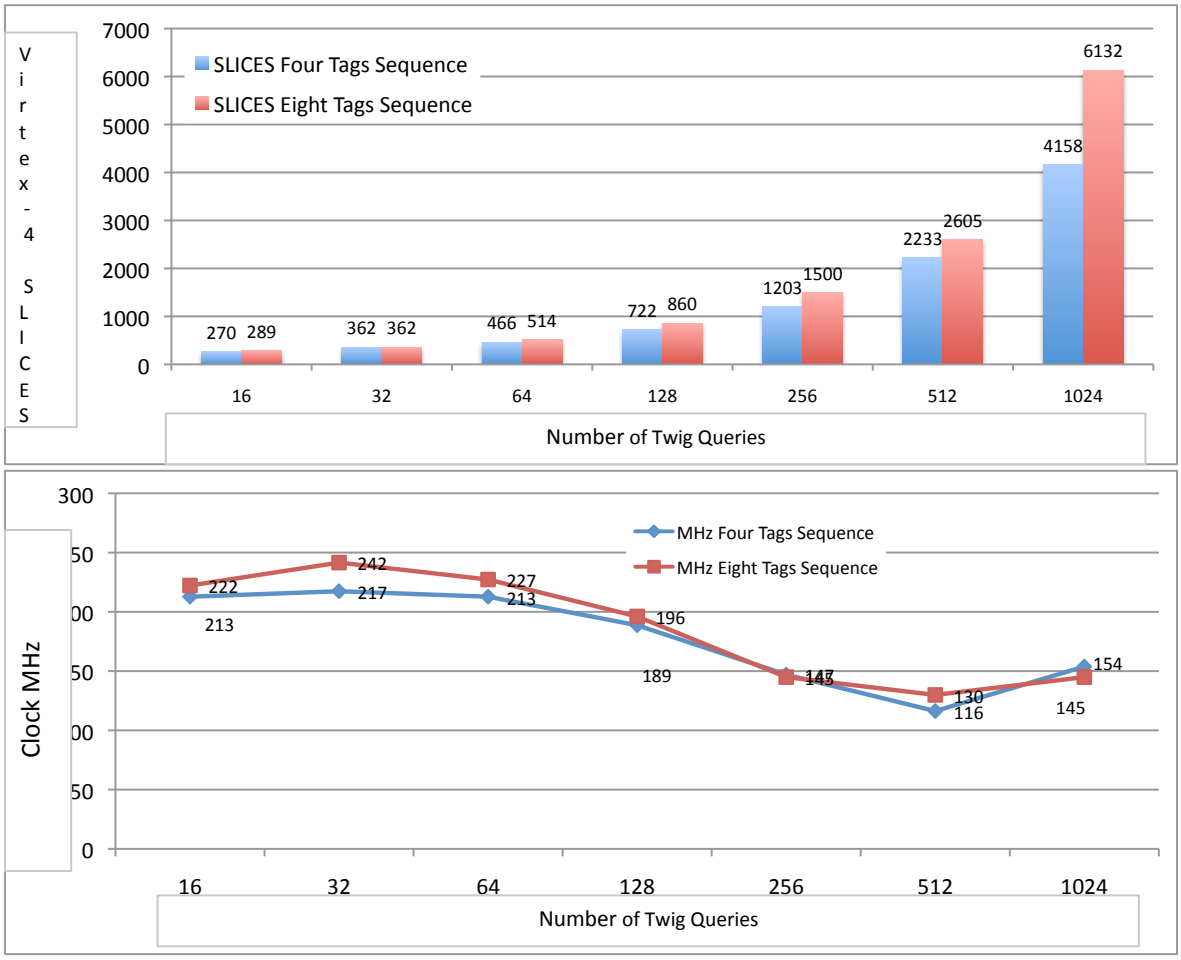

Figure 5.14: The graph on the top demonstrates variation of FPGA area (in SLICES) with increasing number of four tags and eight tags long sequence matching paths for twig queries. The graph at the bottom demonstrates the variation of clock speed of the FPGA hardware in MHz with increasing number four tags and eight tags long sequence matching paths for twig queries.

5.4.1 Performance and Speedup

We first examine the effect of the common prefix optimization. For each experiment the number of profiles implemented in the FPGA varies from 16 to 1024. We measured the areas on the FPGA that was occupied with and without the prefix optimization. The results appear in the subfigures on the left side (depicted as bars) in Figures 5.11 5.12 5.13. For all the approaches the area increases almost linearly with the number of XPath profiles implemented on the FPGA. However, the optimized approach uses less space, which implies that the number of profiles served by the FPGA can increase further.

The second set of experiments examines the throughput of both the hardware and software approaches. The results are charted out in in the bottom subfigures of Figures 5.11 5.12 5.13 and we compare the throughput with increasing number of XPath profiles. Clearly the FPGA approach provides orders of magnitude throughput improvement (around 30 times for some datasets). It can be seen that increasing XPath lengths lead to decreased speedup offered by FPGA. The same is nearly true about the increasing number of XPath profiles implemented on FPGA. The reason is that, adding hardware complexity leads to lower clock rates on the FPGA.

The third set of experiments have been used to obtain throughput and area on FPGA for implementing twig queries. Figure 5.14 demonstrates the variation in area for implementing increasing numbers of four and eight tags Prüfer sequence matching hardware on FPGA. Also included is the speed trends in MHz clock speed of the resulting hardware. An increase in speed is observed for the 1024 four and eight tags sequence matching hardware and it is

due to the change in optimization strategy of the hardware synthesis tool.

5.5 Conclusion

In this chapter we have provided a systematic flow that enables implementation of XPath profiles on FPGA for acceleration of XML filtering. We demonstrate how we can convert XPath to PCRE and thenceforth use our PCRE to FPGA compiler to generate the hardware.

Since regular expressions have limited memory as proved by the 'Pumping Lemma', we enhance the XPath profiles on the FPGA with a hardware tag stack. Our stack enhanced architecture allows implementation of parent-child XPath axis on hardware.

We have implemented common prefix optimization before generating the hardware to minimize the FPGA area and make good use of the commonality among XPath profiles. We have implemented upto one thousand XPath profiles, on the FPGA.

The effective throughput of FPGA design with character match and common prefix optimization while implementing 1024 six tag wide XPaths is 68 MBytes/s and it is 101 MBytes/s when we implement 1024 four tag wide XPaths, and it is 139 MBytes/s when implementing 1024 two tag wide XPaths. The percentage area usage on FPGA is 30.2% for 1024 two tags long XPaths, 50% for 1024 four tags long XPaths and it is 72% for six tags long XPaths.

Using the character decoding hardware we obtain throughput of 128 MBytes/s on a design that implements 1024 six tag wide XPaths, and 133 MBytes/s on a design that implements 1024 four tag wide XPaths, and 139 MBytes/s on a design that implements 1024 two tag

wide XPaths. The percentage area usage on FPGA with common prefix optimized design along with character decoding is 5.5% for 1024 two tags long XPaths, 11.2% for 1024 four tags long XPaths and it is 19.3% for six tags long XPaths.

The software based filtering provides an average throughput of 3 MBytes/s on the same XPath profiles. Thus we have been able to accelerate XPath filtering atleast by 20X using FPGAs for implementation of the filtering hardware.

With respect to the twig queries, the utilization is 5% of the FPGA area for implementing 1024 four tags long sequences, and 7% area for implementing eight tags long sequences. The throughput is 154 MBytes/s for 1024 four tags long sequence and 145 MBytes/s for 1024 eight tags long sequences on FPGA.

Chapter 6

Conclusions

This chapter summarizes the conclusions of this dissertation, and the research outcomes. We mention the contributions which include our PCRE to FPGA compiler, accelerated regular expression system on SNORT IDS, tool support for partial reconfiguration and dynamic co-processors on FPGA, adaptive hardware-software regular expression based IDS, and finally the scalable XML filtering architecture.

6.1 PCRE to FPGA compiler

This dissertation presents the details of our compilation tool which converts a PCRE to VHDL via PCRE opcodes. Previous approaches towards implementation of regular expressions on FPGA relied on manual conversion or a limited conversion process without conversion and optimizations provided by the PCRE compiler. Our tool solves a very important limitation towards implementation of PCRE on FPGA, and allows direct conversion of PCRE

(constrained by hardware limitations) to VHDL for implementation on FPGA.

Our tools uses the front end parser of the original PCRE compiler and extends it towards generation of VHDL. Our compiler makes it possible to simulate the hardware and test its compatibility with the software based PCRE execution, sine the opcodes can be used by software based PCRE execution engine. We have provided implementation details of various PCRE opcodes in hardware using two paradigms, which are IP core based and synthesis based on VHDL code. We have also provided the speed and area variation of the opcodes in hardware with different parameters.

6.2 Accelerating regular expression of SNORT IDS

In this dissertation we have using several detailed benchmarks demonstrated that the performance of software based regular expression matching cannot keep up with network throughputs, especially multi Gbps links.

We have developed a framework for acceleration of SNORT IDS rules by compiling them to VHDL. We have also studied SNORT IDS rules and have developed our PCRE compilation system to specifically support the most frequently occurring PCRE opcodes obtained from the SNORT rules. Our hardware architecture for accelerating regular expression rules from SNORT IDS uses 16 parallel rule-set banks. Each bank implements fourteen regular expression rules on the FPGA. We have also provided benchmark data of the dwindling throughput of regular expression system of SNORT IDS under certain network scenarios

when the amount of malicious activity increases.

Using our hardware architecture on the SGI RASC Blade, we obtain more than 350X speedup when compared to a baseline state of the art CPU viz. the Intel Xeon 5160 and our design can sustain a throughput of 12.9 Gbps. FPGAs consume a lot less power than conventional CPUs. Using our architecture and benchmark results we have demonstrated two orders of savings in power consumption when an FPGA is used to accelerate regular expression sub-system of SNORT IDS.

6.3 Dynamic Co-Processors on FPGA

FPGAs are used for implementing both fixed and configurable hardware circuits. FGPA based co-processing hardware are available in the form of IP cores and share similar I/O and control interface. The hardware IP cores need to obtain data and send results back to the software. In this dissertation, we have demonstrated our framework for automated wrapper generation for IP cores that allow them to interface to a CPU. Our framework for wrapped IP core allows a C function call to instantiate an IP core on the configurable fabric. CSoCs, which include a processor and configurable fabric on the same chip, also use the configurable logic to implement fixed hardware peripherals. We have enhanced our wrapper generation framework to include another layer of interface in order to allow partial reconfiguration on the FPGA. We have extended the C function call to program a partial bitstream on the FPGA containing the IP Core and thus instantiate a dynamic co-processor. By calling the different C

functions for different co-processors, it is possible to re-program the configurable hardware without upsetting the state of operation of the CSoC.

6.4 Adaptive Hardware-Software Regular Expressions based IDS

An adaptive hardware based IDS can respond to a network attack by reconfiguring itself. Since often times, only certain sets of regular expressions need to be modified, a full reconfiguration would be inefficient. Thus we have extended our regular expression hardware to support partial reconfiguration, by virtue of modularization, to allow re-programming of individual regular expression banks, rather than the whole FPGA. Our novel design allows partial reprogramming across 16 banks of regular expression rule-sets on an FPGA/

Our benchmarks is executed on a proof of concept test-bed using a thirty-two core SGI Altix 4700 supercomputer which also has a RASC Blade consisting of two FPGAs. The IDS can maintain execution through software threads during that brief moment when the FPGA is being reconfigured.

We have implement 448 different regular expressions in 32 modular rule-sets, on the two FPGAs. Our adaptive IDS can provide better than 10 Gbps throughput even with 32 partial recongurations per minute. Our system can also sustain 10 Gbps throughput with four full-recongurations per minute. Our IDS design can be extended to similar FPGA accelerated multi-processor system.

6.5 Scalable Architecture for XML Filtering on FPGA

XPath profiles are used for describing subscriber preference for data and messages over XML for pub-sub applications. Since XPath involves a state machine based XML tree traversal, and pub-sub involves multiple XPath profiles executing on a streaming XML document, making them ideal for hardware acceleration. Moreover it is straightforward to transform XPath expression to a PCRE.

We have been able to convert multiple XPath profiles to PCRE and run common prefix optimization on the set of profiles. The optimized XPath trees have been converted to hardware via a modified version of PCRE to VHDL compiler, which adds hardware stack for enabling 'parent-child' matching in the XPaths.

Using a centralized character decoding hardware, we have been able to further optimize the area and performance of the implemented XPaths, as compared to and implementation using character match hardware.

Our benchmarks show a throughput gain of more than 20X on FPGA when compared to software based XPath filtering.

We implement a streaming Prüfer sequence generation hardware for XML document and match it against hardware based sequence match blocks. The sequence match blocks are the Prüfer sequences of twig queries and thus allow us to match multiple twig queries on the FPGA.

Bibliography

[1] Cray XD1. http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf.

[2] *CRAY XD1 FPGA Datasheet*. http://www.cray.com/downloads/FPGADatasheet.pdf.

[3] SAX: Simple API for XML. http://www.saxproject.org/.

[4] SGI RASC technology, http://www.sgi.com/products/rasc.

[5] Xtremedata inc. http://www.xtremedatainc.com/.

[6] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[7] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. *An Overview of the SUIF2 Compiler Infrastructure*. Computer Systems Laboratory, Stanford University.

[8] Altera. *NIOS CPU Data Sheet*. Altera, November 2004. http://www.altera.com/literature/ds/ds_nios_cpu.pdf.

[9] Altera. *Stratix II Device Family Data Sheet*. Altera, 2007. http://www.altera.com/literature/hb/stx2/stx2_sii5v1_01.pdf.

[10] Altera. Stratix ii gx transceiver FPGAs overview, 2008. http://www.altera.com/products/devices/stratix-fpgas/stratix-ii/stratix-ii-gx/features/transceiver/s2gx-mgt-transceiver.html.

[11] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. pages 53–64, 2000.

[12] AMD. Amd hypertransport™ technology. http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_2353,00.html.

[13] Ahmad Ansari, Peter Ryser, and Dan Isaacs. Accelerated system performance with apu-enhanced processing. *Xilinx Xcell Journal*, January 2005. `http://china.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_v4acu52.pdf`.

[14] Armando Astarloa, Aitzol Zuloaga, Unai Bidarte, José Luis Martín, Jesús Lázaro, and Jaime Jiménez. Tornado: A self-reconfiguration control system for core-based multiprocessor csopcs. *J. Syst. Archit.*, 53(9):629–643, 2007.

[15] Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 135–144, Napa, California, USA, 2004. IEEE Computer Society.

[16] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 223–232, Monterey, California, USA, 2004. ACM.

[17] Z.K. Baker, Hong-Jip Jung, and V.K. Prasanna. Regular expression software deceleration for intrusion detection systems. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–8, Madrid, Spain, August 2006.

[18] Z.K. Baker and V.K. Prasanna. A computationally efficient engine for flexible intrusion detection. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(10):1179–1189, Oct. 2005.

[19] Sudarshan Banerjee. *Application mapping for platform FPGAs with partial dynamic reconfiguration*. PhD thesis, Long Beach, CA, USA, 2007. Adviser-Nikil Dutt.

[20] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Considering run-time reconfiguration overhead in task graph transformations for dynamically reconfigurable architectures. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 273–274, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 335–340, New York, NY, USA, 2005. ACM.

[22] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Parlgran: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 491–496, Piscataway, NJ, USA, 2006. IEEE Press.

[23] Sudarshan Banerjee, Elaheh Bozorgzadeh, Nikil Dutt, and Juanjo Noguera. Selective bandwidth and resource management in scheduling for dynamically reconfigurable architectures. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 771–776, New York, NY, USA, 2007. ACM.

[24] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. Toxgene: a template-based data generator for XML. pages 616—616, 2002.

[25] Michael Barr. A reconfigurable computing primer. *Multimedia Systems Design*, pages 44–47, September 1998.

[26] J. Becker and M. Hübner. Run-time reconfigurabilility and other future trends. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 9–11, New York, NY, USA, 2006. ACM.

[27] Jürgen Becker. Adaptive reliable chips - reconfigurable computing in the nano era. In *ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 1–2, Washington, DC, USA, 2008. IEEE Computer Society.

[28] R. Beckert, T. Fuchs, St. Ruelke, and W. Hardt. A run-time scheduling framework for a reconfigurable hardware emulator. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 147–150, Washington, DC, USA, 2007. IEEE Computer Society.

[29] R. Beckert, T. Fuchs, St. Ruelke, and W. Hardt. A tailored design partitioning method for hardware emulation. In *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 99–105, Washington, DC, USA, 2007. IEEE Computer Society.

[30] F. Berthelot, F. Nouvel, and D. Houzet. Design methodology for dynamically reconfigurable systems. *JFAAA*, pages 47–52, January 2005.

[31] Florent Berthelot and Fabienne Nouvel. Partial and dynamic reconfiguration of FPGAs: a top down design methodology for an automatic implementation. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 436, Washington, DC, USA, 2006. IEEE Computer Society.

[32] J. C. Bispo, I. Sourdis, J. M.P. Cardoso, and S. Vassiliadis. Synthesis of regular expressions targeting FPGAs: Current status and open issues. In *Int. Workshop on Applied Reconfigurable Computing (ARC 2007)*, pages 179–190, Mangaratiba, Rio de Janerio, Brazil, March 2007.

[33] Joao Bispo, Ioannis Sourdis, Joao M.P.Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126, Dec. 2006.

[34] Brandon Blodget, Philip James-roxby, Eric Keller, Scott Mcmillan, and Prasanna Sundararajan. A self-reconfiguring platform. In *FPL '03: Proceedings of the 2003 Conference on Field Programmable Logic and Applications*, 2003.

[35] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[36] Maik Boden, Thomas Fiebig, Markus Reiband, Peter Reichel, and Steffen Rülke. Gepard - a high-level generation flow for partially reconfigurable designs. In *ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 298–303, Washington, DC, USA, 2008. IEEE Computer Society.

[37] Cristiana Bolchini and Antonio Miele. Design space exploration for the design of reliable sram-based FPGA systems. In *DFT '08: Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 332–340, Washington, DC, USA, 2008. IEEE Computer Society.

[38] Cristiana Bolchini, Antonio Miele, and Marco D. Santambrogio. Tmr and partial dynamic reconfiguration to mitigate seu faults in FPGAs. In *DFT '07: Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 87–95, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Cristiana Bolchini, Davide Quarta, and Marco D. Santambrogio. Seu mitigation for sram-based FPGAs through dynamic partial reconfiguration. In *GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 55–60, New York, NY, USA, 2007. ACM.

[40] Pierre Bomel, Guy Gogniat, and Jean-Philippe Diguet. A networked, lightweight and partially reconfigurable platform. In *ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, pages 318–323, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Michael Bonetta and Cam Petriw. Bro intrusion detection system. `http://www.bro-ids.org/`.

[42] Alisson V. Brito, Matthias Kuhnle, Michael Hubner, Jurgen Becker, and Elmar U. K. Melcher. Modelling and simulation of dynamic and partially reconfigurable systems using systemc. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 35–40, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.

[44] B. Buyukkurt and W.A. Najj. Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs. *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 655–658, Sept. 2008.

[45] Weinan Chen, Ying Wang, Xiaowei Wang, and Chenglian Peng. A new placement approach to minimizing FPGA reconfiguration data. In *ICESS '08: Proceedings of the 2008 International Conference on Embedded Software and Systems*, pages 169–174, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Jeff Child. Signal processing ip cores, cots journal, 09/2003. `http://www.cotsjournalonline.com/pdfs/2003/09/cots09_techfocus1.pdf`.

[47] Young H. Cho and William H. Mangione-Smith. A pattern matching coprocessor for network security. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 234–239, Anaheim, California, USA, 2005.

[48] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 452–461, Montpellier, France, September 2002.

[49] Christopher R. Clark, Craig D. Ulmer, and David E. Schimmel. An FPGA-based network intrusion detection system with on-chip network interfaces. *Intl. Journal of Electronics*, 93(6):403–420, 2006.

[50] Christopher Claus, Johannes Zeppenfeld, Florian Müller, and Walter Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 498–503, San Jose, CA, USA, 2007. EDA Consortium.

[51] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, Graz, Austria, 1979.

[52] S. Commuri, V. Tadigotla, and L. Sliger. Efficient controller implementations for robot control. In *CSECS'06: Proceedings of the 5th WSEAS International Conference on Circuits, Systems, Electronics, Control & Signal Processing*, pages 48–53, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[53] Sesh Commuri, V. Tadigotla, and L. Sliger. Task-based hardware reconfiguration in mobile robots using FPGAs. *J. Intell. Robotics Syst.*, 49(2):111–134, 2007.

[54] Kerry J. Cox and Christopher Gerg. *Managing Security with Snort and IDS Tools Managing Security with Snort and IDS Tools*. O'Reilly, August 2004.

[55] O. Cret, S. Mathe, B. Szente, Z. Mathe, C. Vancea, F. Rusu, and A. Darabant. FPGA-based scalable implementation of the general smith-waterman algorithm. pages 410–415, 2004.

[56] Srinivasan Dasasathyan, Rajesh Radhakrishnan, and Ranga Vemuri. Framework for synthesis of virtual pipelines. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 326, Washington, DC, USA, 2002. IEEE Computer Society.

[57] Yan-Xiang Deng, Chao-Jang Hwang, and Jiang-Lung Liu. An object-oriented cryptosystem based on two-level reconfigurable computing architecture. *J. Syst. Softw.*, 79(4):466–479, 2006.

[58] Sarang Dharmapurikar, Praveen Krishnamurthy, T.S. Sproull, and J.W. Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52–61, Jan.-Feb. 2004.

[59] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems*, 28(4):467–516, 2003.

[60] Nij Dorairaj, Eric Shiflet, and Mark Goosman. Planahead software as a platform for partial reconfiguration. *Xilinx Xcell Journal*, Fourth Quarter 2005.

[61] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Performance bounds of partial run-time reconfiguration in high-performance reconfigurable computing. In *HPRCTA '07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, pages 11–20, New York, NY, USA, 2007. ACM.

[62] John Emmert, Charles Stroud, Brandon Skaggs, and Miron Abramovici. Dynamic fault tolerance in FPGAs via partial reconfiguration. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 165, Washington, DC, USA, 2000. IEEE Computer Society.

[63] Endace. Multi gigabit intrusion detection. `www.endace.com`.

[64] Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. A design methodology for dynamic reconfiguration: The caronte architecture. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 163.2, Washington, DC, USA, 2005. IEEE Computer Society.

[65] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual property re-use in embedded system codesign: an industrial case study. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 37–42, Washington, DC, USA, 1998. IEEE Computer Society.

[66] Robert W. Floyd and Jeffrey D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3):603–622, 1982.

[67] Benjamin Glas, Alexander Klimm, Oliver Sander, Klaus Müller-Glaser, and Jürgen Becker. A system architecture for reconfigurable trusted platforms. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 541–544, New York, NY, USA, 2008. ACM.

[68] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 404–413, Montpellier, France, September 2002.

[69] I. Gonzalez, S. Lopez-Buedo, and F. J. Gomez-Arribas. Implementation of secure applications in self-reconfigurable systems. *Microprocess. Microsyst.*, 32(1):23–32, 2008.

[70] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[71] Zhi Guo. *Automatic generation of vhdl from c for code acceleration on reconfigurable devices*. PhD thesis, USA, 2006. Adviser-Walid Najjar.

[72] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from C codes for FPGAs. In *ACM/IEEE Conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, February 2005. IEEE Computer Society.

[73] Zhi Guo, Walid Najjar, and Betul Buyukkurt. Efficient hardware code generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):1–26, 2008.

[74] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170, New York, NY, USA, February 2004. ACM Press.

[75] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. pages 419–430, 2003.

[76] Darrin M. Hanna and Michael DuChene. Executing large algorithms on low-capacity FPGAs using flowpath partitioning and runtime reconfiguration. *Microprocess. Microsyst.*, 31(5):302–312, 2007.

[77] Edson L. Horta and John W. Lockwood. Parbit: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs. Technical report, Washington University in St. Louis, 2001.

[78] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 343–348, New York, NY, USA, 2002. ACM.

[79] Randy Ren-Fu Huang. *Hardware-assisted fast routing for runtime reconfigurable computing*. PhD thesis, Berkeley, CA, USA, 2004. Chair-John Wawrzynek.

[80] M. Hübner and J. Becker. Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 1–4, New York, NY, USA, 2006. ACM.

[81] Michael Hubner, Lars Braun, Jurgen Becker, Christopher Claus, and Walter Stechele. Physical configuration on-line visualization of xilinx virtex-ii FPGAs. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 41–46, Washington, DC, USA, 2007. IEEE Computer Society.

[82] Michael Hubner, Christian Schuck, Matthias Kuhnle, and Jurgen Becker. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive micro-electronic circuits. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 97, Washington, DC, USA, 2006. IEEE Computer Society.

[83] Michael Huebner, Tobias Becker, and Juergen Becker. Real-time lut-based network topologies for dynamic and partial FPGA self-reconfiguration. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 28–32, New York, NY, USA, 2004. ACM.

[84] B.L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 111–120, Napa, California, USA, April 2002.

[85] Intel. Intel xeon 5160 tdp. `ftp://download.intel.com/design/network/papers/30117401.pdf`.

[86] Intel. *Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor*. Intel, March 2004. `ftp://download.intel.com/design/network/papers/30117401.pdf`.

[87] B. Jackson. *Partial Reconfiguration Design with PlanAhead*. Xilinx, xilinx 2.1 edition edition, March 2008.

[88] H-J Jung, Z. K. Baker, and V. K. Prasanna. Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems. In *Proceedings of the Reconfigurable Architectures Workshop at IPDPS (RAW '06)*, Rhodes Island, Greece, 2006.

[89] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 151.2, Washington, DC, USA, 2005. IEEE Computer Society.

[90] Heiko Kalte and Mario Porrmann. Replica2pro: task relocation by bitstream manipulation in virtex-ii/pro FPGAs. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 403–412, New York, NY, USA, 2006. ACM.

[91] Toshihiro Katashita, Atusi Maeda, Kenji Toda, and Yoshinori Yamaguchi. Highly efficient string matching circuit for ids with FPGA. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 285–286, Napa, California, USA, April 2006.

[92] Markus Koester, Mario Porrmann, and Ulrich Ruckert. Placement-oriented modeling of partially reconfigurable architectures. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 164.2, Washington, DC, USA, 2005. IEEE Computer Society.

[93] Ralf Krueger. Dynamic reconfiguration of functional blocks. *Xcell Journal*, January 2005. `http://www.xilinx.com/publications/xcellonline/xcell_52/xc_v4config52.htm`.

[94] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 155–164, Orlando, Florida, USA, 2007. ACM.

[95] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM Comput. Commun. Rev.*, volume 36, pages 339–350, Pisa, Italy, 2006.

[96] Kevin Kwiat, Warren Debany, and Salim Hariri. Software fault tolerance using dynamically reconfigurable FPGAs. *Great Lakes Symposium on VLSI*, 0:0039, 1996.

[97] J. Kwon, P. Rao, B. Moon, and S. Lee. Fist: Scalable XML document filtering by sequencing twig patterns. 2005.

[98] Tien-Lung Lee and Neil W. Bergmann. Interfacing methodologies for ip re-use in reconfigurable system on-chip. In *SPIE International Symposium on Microelectronics, MEMS and Nanotechnology*, Perth, Australia, December 12 / 2003.

[99] S. Letz, M. Zedler, T. Thierer, M. Schutz, J. Roth, and R. Seiffert. XML offload and acceleration with cell broadband engine. In *XTech: Building Web 2.0*, 2006.

[100] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195, New York, NY, USA, 2002. ACM.

[101] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of regular expression pattern matching circuits on FPGA. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, pages 12–17, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[102] R. W. Linderman, C. S. Lin, and M. H. Linderman. FPGA acceleration of information management services. In *High Performance Embedded Computing (HPEC)*, 2004.

[103] J. Lockwood. An open platform for development of network processing modules in reprogrammable hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, California, USA, January 2001.

[104] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (fpx). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.

[105] Michael G. Lorenz, Luis Mengibar, Enrique SanMillan, and Luis Entrena. Low power data processing system with self-reconfigurable architecture. *J. Syst. Archit.*, 53(9):568–576, 2007.

[106] W. Lu, K. Chiu, and Y. Pan. A parallel approach to XML parsing. In *IEEE/ACM Int'l Workshop on Grid Computing*, pages 223–230, 2006.

[107] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. pages 227–238, 2002.

[108] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML accelerator engine. In *1st Int. Workshop on High Performance XML Processing*, 2004.

[109] R. Lysecky and F. Vahid. Prefetching for improved bus wrapper performance in cores. *ACM Transactions on Design Automation of Electronic Systems*, January 2002.

[110] Mateusz Majer, Jürgen Teich, Ali Ahmadinia, and Christophe Bobda. The erlangen slot machine: A dynamically reconfigurable FPGA-based computer. *J. VLSI Signal Process. Syst.*, 47(1):15–31, 2007.

[111] James Martin. Benefits of partial reconfigurability in circuit-switched wdm networks. *J. High Speed Netw.*, 14(3):201–213, 2005.

[112] Torsten Mehlan, Jochen Strunk, Torsten Hoefler, Frank Mietke, and Wolfgang Rehm. Irs - a portable interface for reconfigurable systems. In *PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, pages 187–191, Washington, DC, USA, 2006. IEEE Computer Society.

[113] Andre Meisel, Alexander Draeger, Sven Schneider, and Wolfram Hardt. Design flow for reconfiguration based on the overlaying concept. In *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 89–95, Washington, DC, USA, 2008. IEEE Computer Society.

[114] Nele Mentens, Benedikt Gierlichs, and Ingrid Verbauwhede. Power and fault analysis resistance in hardware through dynamic reconfiguration. In *CHES '08: Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, pages 346–362, Berlin, Heidelberg, 2008. Springer-Verlag.

[115] A. Mitra, Z. Guo, A. Banerjee, and W. Najjar. Dynamic co-processor architecture for software acceleration on csocs. In *IEEE Int. Conf. on Computer Design (ICCD)*, 2006.

[116] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating snort ids. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, Orlando, Florida, USA, 2007. ACM.

[117] Abhishek Mitra, Ge Yao, and Walid Najjar. Performance analysis of sgi rasc rc100 blade on 1-d dwt. In *Reconfigurable Systems Summer Institute*, Urbana, Illinois, USA, July 2007.

[118] M. Moro, P. Bakalov, and V. Tsotras. Early profile pruning on XML-aware publish-subscribe systems. pages 866–877, 2007.

[119] J. Moscola, Y. H. Cho, and J. W. Lockwood. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(2), 2008.

[120] 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, California, USA, April 2006.

[121] James Moscola, Young H. Cho, and John W. Lockwood. Reconfigurable Content-based Router Using Hardware-Accelerated Language Parser. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(2), 2008.

[122] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 31, Napa, California, USA, 2003. IEEE Computer Society.

[123] Nallatech and EDA Geek. Nallatech showcases fsb, pci express fpga accelerator products at sc08. *EDA Geek*, 2008. `http://edageek.com/2008/11/18/supercomputing-front-side-bus/`.

[124] T. Ngai, S. Singh, B.K. Britton, W.-B. Leung, H. Nguyen, G.P. Powell, R. Albu, W.B. Andrews, J. He, and C.W. Spivak. A new generation of orca FPGA with enhanced features and performance. *Custom Integrated Circuits Conference, 1996., Proceedings of the IEEE 1996*, pages 247–250, May 1996.

[125] Jia Ni, Chuang Lin, Zhen Chen, and Peter Ungsunan. A fast multi-pattern matching algorithm for deep packet inspection on a network processor. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, pages 16–16, XiAn, China, September 2007.

[126] Jose Luis Nunez-Yanez, Xiaolin Chen, Nishan Canagarajah, and Raffaele Vitulli. Statistical lossless compression of space imagery and general data in a reconfigurable architecture. In *AHS '08: Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 172–177, Washington, DC, USA, 2008. IEEE Computer Society.

[127] NVIDIA. Nvidia cuda™ technology. `http://www.nvidia.com/object/cuda_learn.html`.

[128] Makoto Onizuka. Light-weight xpath processing of XML stream with deterministic automata. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 342–349, New York, NY, USA, 2003. ACM.

[129] Björn Osterloh, Harald Michalik, Björn Fiethe, and Karel Kotarowski. Socwire: A network-on-chip approach for reconfigurable system-on-chip designs in space applications. In *AHS '08: Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 51–56, Washington, DC, USA, 2008. IEEE Computer Society.

[130] Kyprianos Papademetriou and Apostolos Dollas. A task graph approach for efficient exploitation of reconfiguration in dynamically reconfigurable systems. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 307–308, Washington, DC, USA, 2006. IEEE Computer Society.

[131] K. Paulsson, M. Hübner, and J. Becker. On-line optimization of FPGA power-dissipation by exploiting run-time adaption of communication primitives. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 173–178, New York, NY, USA, 2006. ACM.

[132] Katarina Paulsson, Michael Hubner, and Jurgen Becker. Strategies to on- line failure recovery in self- adaptive systems based on dynamic and partial reconfiguration. In

AHS '06: Proceedings of the first NASA/ESA conference on Adaptive Hardware and Systems, pages 288–291, Washington, DC, USA, 2006. IEEE Computer Society.

[133] Katarina Paulsson, Michael Hübner, and Jürgen Becker. Cost-and power optimized FPGA based system integration: methodologies and integration of a low-power capacity-based measurement application on xilinx FPGAs. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 50–55, New York, NY, USA, 2008. ACM.

[134] Katarina Paulsson, Michael Hubner, Markus Jung, and Jurgen Becker. Methods for run-time failure recognition and recovery in dynamic and partial reconfigurable systems based on xilinx virtex-ii pro FPGAs. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 159, Washington, DC, USA, 2006. IEEE Computer Society.

[135] Katarina Paulsson, Ulrich Viereck, Michael Hübner, and Jürgen Becker. Exploitation of the external jtag interface for internally controlled configuration readback and self-reconfiguration of spartan 3 FPGAs. In *ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 304–309, Washington, DC, USA, 2008. IEEE Computer Society.

[136] David Pellerin, Greg Edvenson, Kunal Shenoy, and Dan Isaacs. Accelerating powerpc software applications. *Xilinx Xcell Journal*, 9/1/2005.

[137] Rodolfo Pellizzoni and Marco Caccamo. Hybrid hardware-software architecture for reconfigurable real-time systems. In *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 273–284, Washington, DC, USA, 2008. IEEE Computer Society.

[138] F. Peng and S. S. Chawathe. Xpath queries on streaming data. pages 431–442, 2003.

[139] Feng Peng and Sudarshan S. Chawathe. Xsq: A streaming xpath engine. *ACM Trans. Database Syst.*, 30(2):577–623, 2005.

[140] Conrado Pilotto, José Rodrigo Azambuja, and Fernanda Lima Kastensmidt. Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 199–204, New York, NY, USA, 2008. ACM.

[141] H. Prüfer. Neuer beweis eines satzes über permutationen. *Archiv Für Mathematik und Physik*, (27):142–144, 1918.

[142] P. Rao and B. Moon. Sequencing XML data and query twigs for fast pattern matching. *ACM Trans. Database Syst.*, 31(1):299–345, 2006.

[143] Ian Robertson and James Irvine. A design flow for partially reconfigurable hardware. *Trans. on Embedded Computing Sys.*, 3(2):257–283, 2004.

[144] Martin Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[145] Martin Roesch. Snort & regular expressions, 2006. `http://seclists.org/focus-ids/2006/Jan/0095.html`.

[146] Martin Roesch. Snort v2.4 - the de facto standard for intrusion detection/prevention, 2006. `http://www.snort.org/`.

[147] B. Rousseau, Ph. Manet, D. Galerin, D. Merkenbreack, J.-D. Legat, F. Dedeken, and Y. Gabriel. Enabling certification for dynamic partial reconfiguration using a minimal flow. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 983–988, San Jose, CA, USA, 2007. EDA Consortium.

[148] Patrick Schaumont and Ingrid Verbauwhede. A quick safari through the reconfiguration jungle. In *In Design Automation Conference*, pages 172–177. ACM Press, 2001.

[149] Jordana Seixas, Edson Barbosa, Stelita Silva, Paulo Sergio B. Nascimento, Vinícius Kursancew, Remy Eskinazi, Edna Barros, and Manoel Eusebio. Aquarius: a dynamically reconfigurable computing platform. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 171–176, New York, NY, USA, 2007. ACM.

[150] Lukas Sekanina. Evolvable hardware. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 3627–3644, New York, NY, USA, 2007. ACM.

[151] SGI. SGI Altix family. `http://www.sgi.com/products/servers/altix/`.

[152] Lesley Shannon. Impact of intellectual property cores on field programmable gate array designs. Master's thesis, University of Toronto, 2001.

[153] Hooman Shayani, Peter Bentley, and Andy M. Tyrrell. A cellular structure for online routing of digital spiking neuron axons and dendrites on FPGAs. In *ICES '08: Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware*, pages 273–284, Berlin, Heidelberg, 2008. Springer-Verlag.

[154] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using FPGAs. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Rohnert Park, California, USA, 2001. IEEE Computer Society.

[155] Miguel L. Silva and Joao Canas Ferreira. Using a tightly-coupled pipeline in dynamically reconfigurable platform FPGAs. In *DSD '05: Proceedings of the 8th Euromicro Conference on Digital System Design*, pages 383–387, Washington, DC, USA, 2005. IEEE Computer Society.

[156] K. Siozios, G. Koutroumpezis, K. Tatas, D. Soudris, and A. Thanailakis. Dagger: A novel generic methodology for FPGA bitstream generation and its software tool implementation. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 165.2, Washington, DC, USA, 2005. IEEE Computer Society.

[157] M. D. Smith and G. Holloway. *An introduction to Machine SUIF and its portable libraries for analysis and optimization*. Division of Engineering and Applied Sciences, Harvard University.

[158] M. D. Smith and G. Holloway. *Machine SUIF Bit-Vector Data-Flow-Analysis Library*. Division of Engineering and Applied Sciences, Harvard University.

[159] M. D. Smith and G. Holloway. *Machine SUIF Static Single Assignment Library*. Division of Engineering and Applied Sciences, Harvard University.

[160] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: A reconfigurable hardware NIDS filter. In *Proceedings of 15th International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, August 2005.

[161] D. Soudris, S. Nikolaidis, S. Siskos, K. Tatas, K. Siozios, G. Koutroumpezis, N. Vasiliadis, V. Kalenteridis, H. Pournara, I. Pappas, and A. Thanailakis. Amdrel: a novel low-energy FPGA architecture and supporting cad tool design flow. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 3–4, New York, NY, USA, 2005. ACM.

[162] I. Sourdis, J. C. Bispo, J. M.P. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Int. Journal of Signal Processing Systems for Signal, Image, and Video Technology*, October 2007.

[163] I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis. Scalable multigigabit pattern matching for packet inspection. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):156–166, Feb. 2008.

[164] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In *Proceedings of Int. Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, 2003.

[165] S. Spetka, S. Tucker, G. Ramseyer, and R. Linderman. Imagery pattern recognition and pub/sub information management. In *36th IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, pages 37–41, 2007.

[166] SPIRIT. The spirit consortium. `www.spiritconsortium.org`.

[167] Dave Strenski. FPGA floating point performance – a pencil and paper evaluation. *HPC Wire*, January January 2007.

[168] Dinesh C. Suresh, Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Automatic compilation framework for bloom filter based intrusion detection. In *ARC*, volume 3985, pages 413–418, Delft, The Netherlands, March 2006.

[169] P. Sutton. Partial character decoding for improved regular expression matching in FPGAs. In *IEEE International Conference on Field-Programmable Technology*, pages 25–32, The University of Queensland, St Lucia Campus, Brisbane, Australia, December 2004.

[170] Dimitris Syrivelis and Spyros Lalis. System- and application-level support for runtime hardware reconfiguration on soc platforms. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 29–29, Berkeley, CA, USA, 2006. USENIX Association.

[171] Jesús Tabero, Julio Septién, Hortensia Mecha, and Daniel Mozos. Allocation heuristics and defragmentation measures for reconfigurable systems management. *Integr. VLSI J.*, 41(2):281–296, 2008.

[172] V. Tadigotla and S. Commuri. Dynamic image filter selection using partially reconfigurable FPGAs for imaging operations. In *CSECS'06: Proceedings of the 5th WSEAS International Conference on Circuits, Systems, Electronics, Control & Signal Processing*, pages 60–65, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[173] Heng Tan, Ronald F. DeMara, Anuja J. Thakkar, Abdel Ejnioui, and Jason D. Sattler. Complexity and performance tradeoffs with FPGA partial reconfiguration interfaces. In *Proceedings of the Reconfigurable Architectures Workshop at IPDPS (RAW '06)*, Rhodes Island, Greece, 2006.

[174] Lin Tan, Brett Brotherton, and Timothy Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Trans. Archit. Code Optim.*, 3(1):3–34, 2006.

[175] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. *SIGARCH Comput. Archit. News*, 33(2):112–122, 2005.

[176] RASC Development Team. *Reconfigurable Application-Specific Computing User's Guide*. SGI, 007-4718-007 edition, February 2008.

[177] Anurag Tiwari and Karen A. Tomko. Saving power by mapping finite-state machines into embedded memory blocks in FPGAs. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20916, Paris, France, February 2004.

[178] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale. Trident: An FPGA compiler framework for floating-point algorithms. In *FPL '05: Proceedings of the 2005 Conference on Field Programmable Logic and Applications*, 2005.

[179] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 4:2628–2639 vol.4, 7-11 March 2004.

[180] Antonino Tumeo, Matteo Monchiero, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. An internal partial dynamic reconfiguration implementation of the jpeg encoder for low-cost FPGAs. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 449–450, Washington, DC, USA, 2007. IEEE Computer Society.

[181] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 135–, April 2004.

[182] Theo Valich. GPGPU drastically accelerates anti-virus software, September September 12 2007. `http://www.theinquirer.net/en/inquirer/news/2007/09/12/gpgpu-drastically-accelerates-anti-virus-software`.

[183] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *RAID*, pages 107–126, 2007.

[184] Craig Leres Van Jacobson and Steven McCanne. The libpcap project. `http://sourceforge.net/projects/libpcap/`.

[185] J. Villarreal and W.A. Najjar. Compiled hardware acceleration of molecular dynamics code. *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 667–670, Sept. 2008.

[186] Jason Villarreal, John Cortes, and Walid A. Najjar. Compiled code acceleration of namd on FPGAs. In *Reconfigurable Systems Summer Institute*, Urbana, Illinois, USA, July 2007.

[187] VSIA. Virtual socket interface association. `http://vsi.org`.

[188] M. Wala and D. Bouldin. Integrating and verifying intellectual property blocks using platform express and modelsim. *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 758–761 Vol. 1, Aug. 2005.

[189] K. Watanabe, N. Tsuruoka, and R. Himeno. Performance of network intrusion detection cluser system. *ISHPC*, pages 278–287, 2003.

[190] K. B. Wheeler. Load balancing for high speed parallel network intrusion detection. Master's thesis, University of Notre Dame, April 2005.

[191] Business Wire and Nallatech. Nallatech to support and deliver product for intel(r) quickpath interconnect. *Business Wire*, 2008. http://www.allbusiness.com/electronics/computer-equipment-computer/5298096-1.html.

[192] M. J. Wirthlin. A dynamic instruction set computer. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 99, Washington, DC, USA, 1995. IEEE Computer Society.

[193] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[194] Xilinx. *EDK, PowerPC 405 Processor Block Reference Guide, UG018*. Xilinx. http://www.xilinx.com/bvdocs/userguides/ug018.pdf.

[195] Xilinx. Virtex-4 multi platform FPGA. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/.

[196] Xilinx. *Xilinx CORDIC 3.0, DS 239*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/cordic.pdf.

[197] Xilinx. *Xilinx Fast Simplex Link v2.00a*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/FSL_V20.pdf.

[198] Xilinx. *Xilinx FFT v3.2, DS 260*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/xfft.pdf.

[199] Xilinx. *Xilinx Floating point Operator v2.0, Logicore*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/floating_point_ds3315.pdf.

[200] Xilinx. Xilinx intellectual property library, logicore. http://www.xilinx.com/ipcenter/.

[201] Xilinx. *Xilinx OPB IPIF specifications DS414*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_ipif.pdf.

[202] Xilinx. *Xilinx Pipelined Divider v 3.0, DS305*. Xilinx. http://www.xilinx.com/ipcenter/catalog/logicore/docs/sdivider.pdf.

[203] Xilinx. *Xilinx PLB IPIF specifications DS414*. Xilinx. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/plb_ipif.pdf.

[204] Xilinx. *Two Flows for Partial Reconfiguration: Module Based or Difference Based*. Xilinx, 2003. china.xilinx.com/support/documentation/application_notes/xapp290.pdf.

[205] Xilinx. *Xilinx ISE 8.1i Development System Reference Guide*. Xilinx, 2006. http://toolbox.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf.

[206] Xilinx. *Virtex-4 RocketIO Multi-Gigabit Transceiver, UG076 (v4.1)*. Xilinx, November 2008. http://www.xilinx.com/support/documentation/user_guides/ug076.pdf.

[207] Xilinx. *Xilinx UG190 Virtex-5 FPGA User Guide*. Xilinx, 2008. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[208] Xilinx. *Early Access Partial Reconfiguration User Guide, ug208 edition*. Xilinx, March 2006.

[209] Xtremedata. VAA: VLDB analytics appliance, December 2007.

[210] Xtremedata. *XD1000^{TM} FPGA Coprocessor Module for Socket 940*. Xtremedata, October 2007.

[211] Xtremedata. FPGA acceleration in HPC: A case study in financial analytics. Technical report, Xtremedata, November 2006. http://www.xtremedatainc.com/pdf/FPGA_Acceleration_in_HPC.pdf.

[212] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, San Jose, California, USA, 2006. ACM.

[213] Sherif Yusuf, Wayne Luk, M. K. N. Szeto, and W. Osborne. Unite: Uniform hardware-based network intrusion detection engine. In *Int. Workshop on Applied Reconfigurable Computing*, pages 389–400, Delft, The Netherlands, March 2006.

[214] Xun ZHANG, Hassan RABAH, and Serge WEBER. Auto-adaptive reconfigurable architecture for scalable multimedia applications. In *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 139–145, Washington, DC, USA, 2007. IEEE Computer Society.

[215] Xun Zhang, Hassan Rabah, and Serge Weber. An auto-adaptation method for dynamically reconfigurable system-on-chip. In *ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 499–502, Washington, DC, USA, 2008. IEEE Computer Society.