Context-Free-Grammar based Token Tagger in Reconfigurable Devices *

Young H. Cho and James Moscola and John W. Lockwood
Washington University in St. Louis
Department of Computer Science and Engineering, Campus Box 1045
St. Louis, Missouri 63130–4899
Email: {young,jmm5,lockwood}@arl.wustl.edu

Abstract

In this paper, we present reconfigurable hardware architecture for detecting semantics of streaming data on 1+ Gbps networks. The design leverages on the characteristics of context-free-grammar (CFG) that allows the computers to understand the semantics of data. Although our parser is not a true CFG parser, we use the linguistic structure defined in the grammars to explore a new way of parsing data using Field Programmable Gate Array (FPGA) hardware. Our system consists of pattern matchers and a syntax detector. The pattern matchers are automatically generated using the grammar token list while the syntax detector is generated based on the aspects of the grammar that define the order of all possible token sequences. Since all the rules are mapped onto the hardware as parallel processing engines, the meaning of each token can be determined by monitoring where it is being processed. Our highly parallel and fine grain pipelined engines can operate at a frequency above 500 MHz. Our initial implementation is XML content-based router for XML remote procedure calls (RPC). The implementation can process the data at 1.57 Gbps on Xilinx VirtexE FPGA and 4.26 Gbps on the Virtex 4 FPGA.

1. Introduction

By performing high-level analysis of content, the accuracy of network traffic analyzers can be improved. Recent research results in the area of network intrusion detection show that a single field programmable gate array (FPGA) device can search for several thousands of worm signature strings on 1+ Gbps networks [11, 14, 9, 34]. However, the naive pattern searches used in these implementations do not consider the context of the text in the data. Therefore, they are susceptible to false positive identifications.

In this paper, we present a novel hardware architecture that uses context-free grammar (CFG) to enable a higher level of understanding of the streaming data. For a given grammar description, the automatic hardware generator builds high performance pattern detection engines. Then, the syntactical structure is formed out of the pattern detection engines using the First and Follow set algorithms of most predictive parser generators.

We begin in section 2 with a brief overview of related works. Then we describe the architecture and algorithms for our research work in section 3. In section 4, we implement and integrate our CFG parser based data tagging engine in XML content-based router. We conclude in section 5 with the direction of our future research and suggestions of other applications.

2. Related Works

The role of a content-based packet classifier is to associate packets with strings of interest [17, 36, 22]. These associations are then used by network applications to perform features such as content-aware traffic processing [4], content-based routing [39], web switching, and packet filtering. There are a several commercial products such as Broadcom StrataSwitch [18] and PMC Sierra ClassiPI [19] that are earlier examples of high performance packet classifiers. Since most of these processors are designed to run sequential programs, their performance decrease dramatically as number of rules increase.

Due to significant economic damage caused by computer network attacks in recent years, there is a demand for fast pattern matching technologies in network security systems. An effective method of preventing an outbreak of network malware is to filter traffic based on the results from deep packet inspection. Deep packet inspection searches network

*This research was sponsored by the Air Force Research Laboratory, Air Force Materiel Command, USAF, under Contract number MDA972-03-9-0001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL or the U.S. Government.

packet contents for common regular expressions found in worms or viruses [28, 15, 37, 25].

Deep packet inspection algorithms are computationally intensive. They do, however, allow for a high degree of parallelism. Several research groups have developed parallel pattern matching hardware on FPGA hardware. Such implementations can protect networks by filtering packets via high performance deep packet inspection engines [8, 34] working in conjunction with automatic signature generators [21, 32, 20, 26]. Although such pattern matching technologies provide the ability to scan high speed networks at line speed, they lack the intelligence to interpret the patterns based on their context.

Computer programming languages can be represented with Context-Free-Grammars (CFG) to assign contextual meanings and structure to tokens in programs [2]. Grammars can represent application level communication languages. One of the most popular languages on the Internet is called the Extensible Markup Language (XML). Increasing usage of XML prompted high performance grammar parsers [16]. Many scalable software implementations of XML parser have been developed by commercial companies including Cisco, Intel, IBM, Microsoft, and Sun. As with the pattern matching engines, a few researchers have explored the use of a specialized hardware parsers to increase the parsing bandwidth [38, 10].

Our research augments the hardware implementation of the pattern search engine with the generic algorithm used in CFG parsers to produce a high-performance contextual token tagger for streaming data.

3. Design Architecture

CFGs have been used in computer science to describe the structure of computer languages. They are powerful enough to express complex programming language structures such as C++ or Java, and yet simple enough to use that computer programs can easily process them. With a few exceptions (Dutch, Swiss German, and Bambara [31, 13]), it is also shown that most natural languages such as English and Chinese can be expressed using CFGs [33].

Leveraging on the established study of CFG parsers, we describe an effective way to augment fast pattern recognition hardware engines with the ability to parse data.

3.1. Conceptual Overview

Most computer programming languages are defined with a CFG. A CFG is a formal way to specify a class of languages. It consists of tokens, non-terminals, a start symbol, and productions. There are two stages to CFG parsing. The first stage performs lexical analysis and the second stage performs syntactical analysis. For lexical analysis, the input data is scanned to determine the sequence of regular expressions separated by delimiters. These regular expressions are called the tokens. The token list is often defined separately from the production list. The symbols in the token list are then used as terminals in the production list [2].

No.	Production
1	$E \rightarrow (E)$
2	$E \rightarrow 0$

Figure 1. CFG Production list describing "0" with balanced parenthesis

The grammar in figure 1 expresses the syntax for the text "0" with balanced parenthesis on left and right. This example consists of two productions rules, each consisting of a non-terminal followed by an arrow and a combination of nonterminals, tokens, and *or* symbols which are expressed with a vertical bar. The left side of the arrow can be seen as a resulting variable whereas the right side is used to express the language syntax.

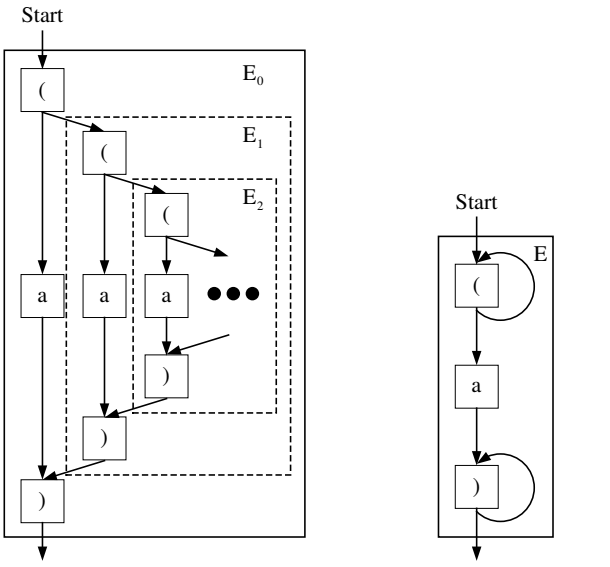
In every set of productions there must be one start symbol to indicate the beginning of the language. Normally, the nonterminal result of the first production is the start symbol of the language. For the example, the nonterminal symbol E is the start symbol.

This formal representation is more powerful than a regular expression. In addition to describing all forms of a regular expression, it is able to represent more advanced programming language structures such as balanced parenthesis and recursive statements like "if-then-else".

In programming language compilers, language parsers based on CFGs can verify syntax as well as construct a parse tree of input source code. The parse tree reveals contextual meaning of the words in input program. Discovering word semantics allows a back-end process to generate platform specific executable code.

Unlike the traditional table look-up [10] or recursive descent methods used in most CFG parsers, we attempt to directly map the grammar structure into logic. For instance, the grammar in figure 1 can be translated as the logic structure in figure 2a. A major difficulty in mapping the grammar in such way is the amount of duplicated hardware caused by infinite number of possible recursion in the grammar. Even if the recursiveness is restricted to a finite number, the hardware area would grow exponentially according to the number of recursive non-terminals.

Traditional software implementations of parsers rely on a built-in context switch function in language to handle recursive executions [27]. Such behavior can be duplicated in hardware using stacks. However, we find that the main purpose of saving recursive state information is for error detection in the input data. For our application, we assume that



(a) Push-down Automaton (b) Finite-State Automaton

Figure 2. Hardware structure of balanced parenthesis example

the data already conforms to the grammar at hand. Therefore, we choose not to implement hardware that saves recursive states. This design decision allows us to effectively convert the push-down automaton design into the finite state automaton as seen in figure 2b.

Without implementing stacks, the parser is not a true CFG parser. On the other hand, our design can parse a language that is a superset of the grammar. Therefore, in most case, it is able to parse the data given the input data represents a correct sentence of the grammar. By collapsing the recursive states into a single state, certain grammars may introduce additional state transitions. In such a case, it may not be possible for the hardware to identify the tokens immediately. However, due to our parallel processing engines, most token sequences can be correctly identified given that only one of the state transition will eventually continue its transition.

Figure 3 is the block diagram of the CFG parser architecture. The architecture consists of character decoders and tokenizers generated based on the token list. The control flow of token detection is determined from the production list and directly mapped onto the hardware as paths between the tokenizers. The output signals of the tokenizers are encoded to indicate the index of the token. The back-end receives the token index along with the pattern for application level processing (i.e. text filter and HTML router).

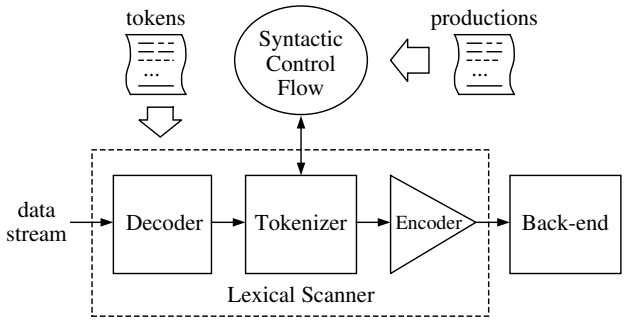


Figure 3. Architecture of hardware grammar based tokenizer

3.2. Lexical Scanner

Before one can build a grammar structure, the input stream must be correctly tokenized. In this section, we describe our hardware lexical scanner.

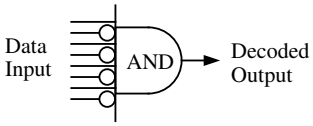


Figure 4. Decoder for 8-bit hexadecimal number 0xAA

In order to design a compact pattern matching engine, our design decodes the input [6, 12, 3, 35]. As shown in figure 4, the decoder logic is an 8-bit input AND gate with inversion of the bits needed to identify each character. All the letters used in the tokens are decoded uniquely. Each decoded character is assigned a wire to provide succinct inputs to the tokenizers.

In regular expressions, there are a few special characters that get used often. As shown in figure 5, combinations of decoded characters may be used to express more complex symbols such as case insensitive characters and alphanumeric characters. In addition to these decoders, delimiters are also defined for the tokens.

String detectors are chains of pipelined AND gates as shown in figure 6a. At each stage of the pipeline, the output is asserted on when decoded character bit and output of the previous stage are both asserted. Such a chain is capable of detecting specified strings in data streams [11].

As a stream of data enters the hardware, token delimiters effectively hold the detection of the next pattern. Therefore, the delimiter detection output is inverted and connected to

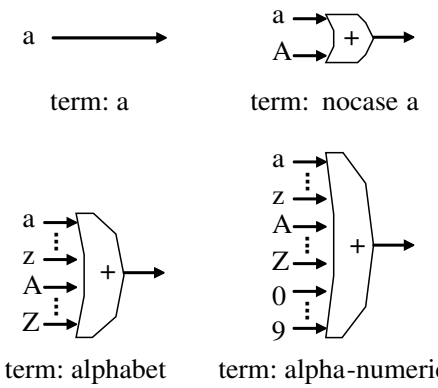


Figure 5. Sample of pre-decoded circuit for special characters

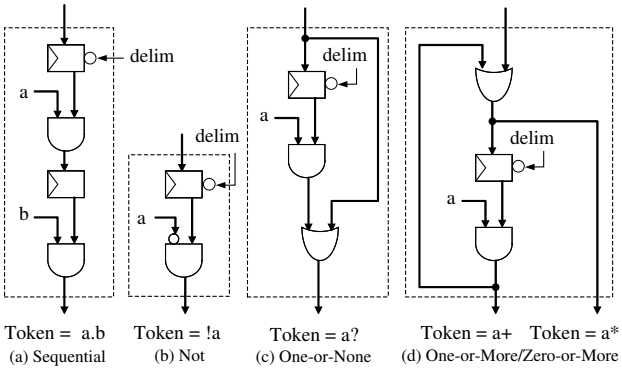


Figure 6. Constructing regular expression with delimiter

the enable signals of the first registers in the token detection chains. It is necessary that only the first register of each token is stalled because otherwise, two partial tokens separated by a delimiter could be recognized as a single token.

In addition to string recognition, regular expressions use functions such as Not, One-or-None, One-or-More, and Zero-or-More to represent a rich set of patterns. We represent these functions in the hardware logic as shown in figure 6. Our hardware generator instantiates combinations these elementary logic templates to build regular expression detectors.

Regular expressions recognize a class of patterns using different combinations of functions. For instance, the expressions a+ or a* would translate into the hardware logic shown in figure 6d. The logic recognizes all of the patterns that are a multiple sequence of the "a" character. If the input stream consisted of a stream of contiguous "a"s, the logic would indicate detection at every cycle. Although such detection is correct, we want to indicate the detection of the longest pattern for the purpose of tokenization.

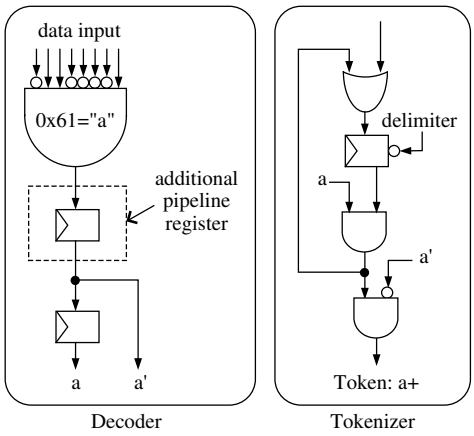


Figure 7. Modified one-or-more instance of "a" character detection logic

This can done in hardware by delaying the input to the tokenizer logic by adding pipeline registers. By using the decoded bits in the earlier stages of the pipeline, we can effectively look at the future characters to find the longest pattern. Figure 7 shows modified logic of a+ to indicate a detection only for the longest sequence of consecutive "a" characters. Notice that with the addition of an extra AND gate, the tokenizer in figure 7 only asserts the detection bit when the current character is "a" and the next character is not "a".

In a software parser implementation, the semantics of a token is determined based on its production. When the same token is used in two different contexts, the semantics can be distinguished by referring to the parse tree. However, for streaming applications, one would want to determine the context of the tokens during the detection process. We facilitate this process by automatically duplicating the tokens used in multiple contexts and defining them as different tokens.

3.3. Syntactic Structure

The syntax of a CFG based language is defined in its production list. A production list defines all the possible token transitions. During the parsing process a CFG parser maintains state information which includes the current production, current token, as well as all recursively instantiated productions. Given any state, a parser can determine which of the token transitions are allowed. Then one of the allowed transitions can take place based on the detected token.

For each terminal symbol Z
$FIRST[Z] \leftarrow \{Z\}$
repeat
  For each production $X \rightarrow Y_1 \ldots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if k=0)
      then $nullable[X] \leftarrow true$
    For each i from 1 to k, each j from i+1 *to* k
      if $Y_1 \ldots Y_{i-1}$ are all *nullable* (or if i=1)
        then $FIRST[X] \leftarrow FIRST[X] \cup FIRST[Y_i]$
      if $Y_{i+1} \ldots Y_k$ are all *nullable* (or if i=k)
        then $FOLLOW[Y_i] \leftarrow FOLLOW[Y_i] \cup FOLLOW[X]$
      if $Y_{i+1} \ldots Y_{j-1}$ are all *nullable* (or if i+1=j)
        then $FOLLOW[Y_i] \leftarrow FOLLOW[Y_i] \cup FOLLOW[Y_j]$
until *FIRST*, *FOLLOW* and *nullable* no longer change

Figure 8. Algorithm for finding First() and Follow() sets from a production list

No.		Production
1	E $\rightarrow$	if C then E else E | go | stop
2	C $\rightarrow$	true | false

Figure 9. CFG Production list describing if-then-else statement

For our architecture, we find all token transitions defined in the production list using a well known algorithm defined in figure 8. As the name suggests, the Follow set algorithm traverses through the production list to find a set of tokens that can follow any given symbol. Then the Follow set can be used as a guide to connect the tokenizers in hardware.

Since all the tokenizers are enabled by their input pin, the input of the a starting symbols must initiate the detection process. The First set of the first production contains all possible starting terminal tokens. If the beginning of the text is known, then the starting tokenizers can be enabled once at the beginning of the data. Otherwise, starting tokenizers can be enabled at all times. Since our hardware consists of parallel detection engines, such a configuration will look for all sequences of tokens starting at every byte alignment of the data.

To clearly illustrate our design method, we use an example grammar defined in figure 9 to generate the corresponding parser hardware. First, we need to build the decoder logic and the tokenizers for all the terminals in the grammar. As we can see in the grammar definition, the terminal tokens are 'if', 'then', 'else', 'go', 'stop', 'true', and 'false'.

Non-terminals	Follow Set
if	$\{true, false\}$
then	$\{if, go, stop\}$
else	$\{if, go, stop\}$
go	$\{else, \epsilon\}$
stop	$\{else, \epsilon\}$
true	$\{then\}$
false	$\{then\}$

Figure 10. Follow set for each of the terminal tokens

Once the decoders and tokenizers are generated, we find the Follow set for each of the terminal tokens. Figure 10 shows a table of the Follow set for all the tokens in the grammar. In addition to the terminal tokens, we need to find all the possible start terminals. Since production E is the starting production, we can use the First set of E as the starting symbols. Then, we forward the output of each token to the inputs of the tokens listed in its Follow set. When there is more than one connection to the input of the tokenizer, an OR gate is used to combine the signals into a single bit input. Figure 11 shows our hardware parser for the if-then-else grammar.

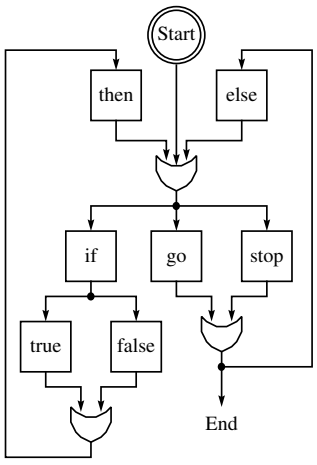


Figure 11. Tokenizer logic for if-then-else grammar

Since we do not maintain the recursive state of the parser, our parser can accept all of the tokens in the Follow sets regardless of the current state. Therefore, our parser behaves differently from other true parsers in two general cases. One type of incorrect behavior can be seen when the data input does not conform to the parser grammar, thus causing illegal transition to occur in a given state. We can, however, avoid

such problem by assuming that the input conforms to the grammar. This assumption would be a constraint for our application. Another type of incorrect behavior occurs when the transition list for a given token contains two or more tokenizers that can accept the input data which would have been mutually exclusive in a true parser. For most cases, this issue can be resolved in the hardware itself. Unlike a software implementation, our hardware structure can run in parallel. Therefore, if multiple transitions takes place, all of them can be executed in parallel. In most cases, due to the context of the data, only the correct transition path will be allowed to continue. Otherwise, all detections may be passed on to the back-end of the processor to select the preferred path pre-determined by the application.

3.4. Token Index Encoder

Each tokenizer produces the 1-bit output to indicate a token match. In some applications it maybe sufficient to simply indicate the match, with identification accomplished in software. However, it is often more desirable to produce the corresponding index number.

A small index encoder module can be written in VHDL as a chain of CASE statements. However, an encoder with CASE statements does not translate efficiently in most synthesis tools. In a naive implementation of an encoder for a large set of rules, the index encoder is almost always the critical path for of the entire system since rest of the design is highly pipelined.

If we assume only one token is detected at any given clock cycle, the address encoder can be built using the combination of outputs from the binary tree of OR gates [7].

Given binary tree outputs, each index bit should be asserted if any of the odd nodes on the corresponding level of the tree are asserted. For example, a four bit index encoder for a 15-input encoder is written as equations 1 through 4.

$$Index_3 = a_1 \tag{1}$$

$$Index_2 = b_1 + b_3 \tag{2}$$

$$Index_1 = c_1 + c_3 + c_5 + c_7 \tag{3}$$

$$Index_0 = d_1 + d_3 + d_5 + d_7 + d_9 + d_{11} + d_{13} + d_{15} \tag{4}$$

In most cases, we may assume that only one tokenizer output will be asserted at any given clock cycle. Then we can construct a compact encoder using the combinations of outputs from the binary tree of OR gates.

As mentioned in the above sections, there is still a possibility that a search engine will detect more than one pattern at any instance. In this situation, the previously described index encoder may not be sufficient. One solution to the conflict is to divide the set into multiple sets; where each subset contains all of the tokens that can possibly be asserted at any one time. Then each subset of the tokenizers can have its own index encoder. However, in some cases, the back-end logic may be expecting only one index at any given time. In such a case, one can assign priority to the tokens.

For all the tokenizers that can assert encoder inputs at the same time, one can assign index numbers to satisfy equation 5; which applies a bit-wise OR to all the indices where I_n is an index number where a higher value of n indicates a higher priority.

$$I_n|I_{n-1}|...|I_0 = I_n \tag{5}$$

Once all the indices are assigned for the contending tokens according to the desired priorities, the indices can be assigned to the rest of the patterns. In this method, there is a limitation placed on the size of prioritized tokens. The maximum number of indices for each set is equal to the number of index output pins.

With registers at the output encoded address bits, the critical path has maximum of (log n)-1 gate delays where n is the number of the input pins. Since the tokenizers are pipelined after every gate, the longest chain of gates in the index encoder becomes the critical path.

When implementing a time critical design in an FPGA, it is important to consider its architecture. The elementary logic unit of our target FPGA consists of a four input look-up-table (LUT) followed by a one bit register. Therefore, we can structure the index encoder to insert a register at the output of each LUT. Such pipelining efficiently utilize the hardware resources while obtaining low latency. The logic for index equations is also further pipelined to maintain one level of logic between pipelined registers.

3.5. Back-end Processor

The back-end processor is customizable logic where many different data processing functions can be implemented. Contextual information of the tokens can be used to process the data more accurately to reduce the number of false positive. Some of the most obvious applications would be in data filtering, data-mining, content-based routing and packet classification.

4. Implementation

Since its introduction in 1996, XML has become one of the most popular languages on the Internet. It can be used to describe a wide range of structured data ranging from financial transactions to sheet music. One such use of XML is for Remote Procedure Calls (XML-RPC). XML-RPC allows remote procedure calls to be made between systems over the Internet regardless of operating system or environment. XML-RPC is commonly used by companies offering

web services as a means for clients to remotely access these online services. As more web services become available over the Internet it is desirable to have a system that can route XML-RPC messages based on the service requested in the content of the message. This example will illustrate how the CFG parsing module presented in this paper can help to create such a content-based routed. An example of an XML-RPC message router is shown in figure 12. The example shows how XML-RPC messages destined for either a bank server or a shopping server can be routed using the contextual information from within the XML-RPC message. As messages pass through the system, the CFG parser tagger asserts a signal associated with a service when that service is found in a message. This signal is then used to control a switch which routes the message to the appropriate destination. The remainder of this section will discuss this example in more detail.

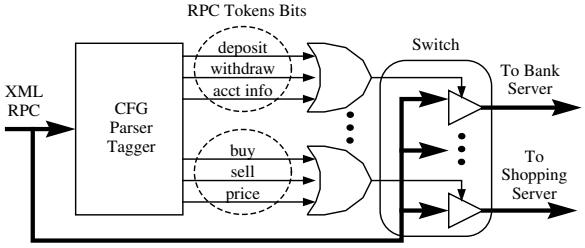


Figure 12. Example of an XML-RPC message router

4.1. Grammar for XML-RPC

As with most commonly used XML formats, there is a Document Type Definition (DTD) that defines the syntax of an XML-RPC message. A DTD is used to validate XML documents as well as a guide to ensure that all implementations for a given XML format are compatible. In essence, a DTD is a grammar for a given XML format. The DTD for XML-RPC is shown in figure 13.

Before we can automatically generate VHDL to parse XML-RPC messages, the DTD in figure 13 is first converted into a grammar in Bachus Naur Form (BNF) which is compatible with our code generator implementation. Specifically, we've chosen the input format that is used with the Lex and Yacc tools. The Lex and Yacc tools have been around for many years and have even spawned improvements in the form of Flex and Bison. Because of the popularity of the Lex and Yacc tools, and their offspring, we can take advantage of the numerous grammars already available and use them as input to our parser. The Yacc style grammar of the XML-RPC DTD is shown in figure 14.

```
<!ELEMENT methodCall        (methodName, params)>
<!ELEMENT methodName        (#PCDATA)>
<!ELEMENT params            (param*)>
<!ELEMENT param             (value)>
<!ELEMENT value             (i4|int|string|
   dataTime.iso8601|double|base64|struct|array)>
<!ELEMENT i4                (#PCDATA)>
<!ELEMENT int               (#PCDATA)>
<!ELEMENT string            (#PCDATA)>
<!ELEMENT dateTime.iso8601  (#PCDATA)>
<!ELEMENT double            (#PCDATA)>
<!ELEMENT base64            (#PCDATA)>
<!ELEMENT array             (data)>
<!ELEMENT data              (value*)>
<!ELEMENT struct            (member+)>
<!ELEMENT member            (name, value)>
<!ELEMENT name              (#PCDATA)>
```

Figure 13. DTD for XML-RPC

```
STRING            [a-zA-Z0-9]+
INT               [+-]?[0-9]+
DOUBLE            [+-]?[0-9]+.[0-9]+
YEAR              [0-9][0-9][0-9][0-9]
MONTH, DAY        [0-9][0-9]
HOUR, MIN, SEC    [0-9][0-9]
BASE64            [+/A-Za-z0-9]
%%
methodCall: "<methodCall>" methodName params "</methodCall>";
methodName: "<methodName>" STRING "</methodName>";
params:     "<params>" param "</params>";
param:      | "<param>" value "</param>" param;
value:      i4 | int | string | dateTime | double
            | base64 | struct | array;
i4:         "<i4>" INT "</i4>";
int:        "<int>" INT "</int>";
string:     "<string>" STRING "</string>";
dateTime:   "<dateTime.iso8601>" YEAR MONTH DAY
            `T' HOUR `:' MIN `:' SEC "</dateTime.iso8601>";
double:     "<double>" DOUBLE "</double>";
base64:     "<base64>" BASE64 "</base64>";
struct:     "<struct>" member_list "</struct>";
member:     "<member>" name value "</member>";
name:       "<name>" STRING "</name>";
array:      "<array>" data "</array>";
data:       | "<data>" value "</data>";
%%
```

Figure 14. Yacc style Grammar for XML-RPC

4.2. Routing XML-RPC Messages

To route XML-RPC messages based on the requested service, a router needs to know which service has been requested by the remote client. In an XML-RPC message, the service name is represented as a `STRING` value enclosed in the `<methodName>` and `</methodName>` tags. This `STRING` value is the information needed by the router to route the message successfully. Using the CFG in figure 14, a parser module can be generated that can notify a routing module of the presence the requested services in a message.

To generate a parser module that parses XML-RPC messages, the grammar in figure 14 is loaded into the VHDL code generator which completely generates all the code required for the parser. The generated code for the parser module consists of a character decoder, an alphanumeric decoder, a whitespace decoder, VHDL for each token in the grammar, and structural code to connect all the com-

ponents together. Each token has an index value associated with it which is output from the parser each time the token matches. Each service may also have a unique index value.

To route messages based on their XML-RPC content, token indices can be sent synchronously with the message data to a routing module. When the routing module detects an index associated with any of the known services, it can use that index value to select the output port for the XML-RPC message.

4.3. Area and Performance

The XML-RPC token tagger was synthesized and placed and routed on both the Xilinx VirtexE 2000 and the Xilinx Virtex 4 LX200 chips. For synthesis, we used Synplicity's Synplify Pro v8.1. Placing and routing was completed using version 7.1 of the Xilinx back-end tools. The faster of the two chips, the Virtex 4, places and routes the design at 533MHz. Such a high clock frequency is a result of the highly pipelined nature of the architecture. Processing only 1 byte per clock cycle, the design is able to achieve a throughput of 4.26 Gbps. Including all of the decoder logic, the design uses just over one LUT per byte of the input grammar. As the size of the grammar increases, and the decoders become a smaller percentage of the overall logic, the number of LUTs per byte decrease even further. The complete results are presented in table 1.

Device	Freq (MHz)	BW (Gbps)	# of Bytes	# of LUTs	LUTs/ Byte
VirtexE 2000	196	1.57	300	310	1.03
Virtex4 LX200	533	4.26	300	302	1.01
Virtex4 LX200	497	3.97	600	526	.88
Virtex4 LX200	445	3.56	1200	975	.81
Virtex4 LX200	318	2.54	2100	1652	.79
Virtex4 LX200	316	2.53	3000	2316	.77

Table 1. Device utilization for XML token taggers of varying sizes

The grammar for XML-RPC is relatively small with only 45 tokens and approximately 300 bytes of pattern data. In order to test the scalability of the architecture, larger XML grammars were created by repeatedly duplicating the 300 byte grammar. The larger grammars contained up to 400 tokens and up to 3000 bytes of pattern data as shown in figure 15. On the Virtex 4, the largest design places and routes at 316MHz. This is a difference of 217 MHz from the XML-RPC grammar. A timing analysis reveals the critical

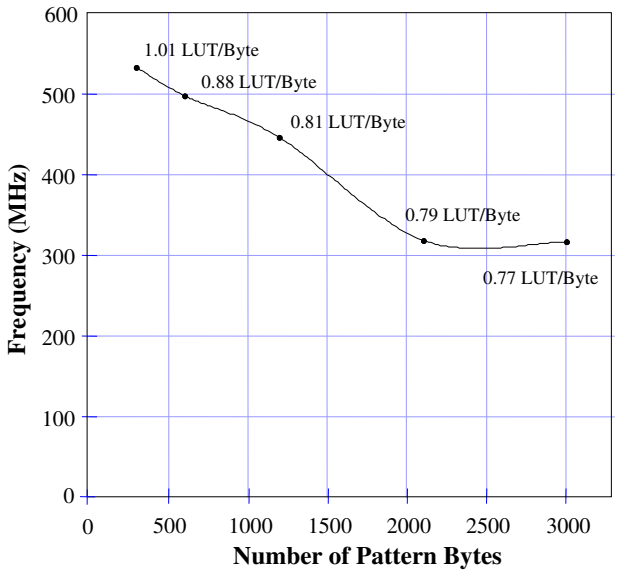


Figure 15. Frequency versus the number of bytes in the grammar on the Virtex 4 LX200

paths for both the XML-RPC grammar and the larger XML grammar are entirely routing delay associated with the large fanout of the decoded character bits as they are routed to each of the tokens. For the largest XML grammar, the routing delay alone for several of the decoded character bits is just under 2 nanoseconds. These results indicate that the we need a more efficient method of routing decoded character bits to the token logic. Possibilities for improving the routing delay include a register tree to pipeline the fanout, or replicating decoders and balancing the fanout across them.

5. Conclusions

In this paper we presented a new way of using CFGs to generate a hardware-based pattern matching engine that also provides contextual meaning of the tokens. Unlike the traditional method of sequential processing of a parser, we directly map the grammar structure into logic. Based on the location of the pattern detection, one can detect the pattern as well as its context in the grammar. By using a parallel and highly pipelined design, our engine is capable of processing data at rates over 4 Gbps on the Xilinx Virtex 4 FPGA.

5.1. Applications

Uses for a hardware based CFG parser vary from everyday applications to niche applications. We've already shown how the parser can be used to create a content-based

router with the XML-RPC example in section 4. Other applications for the networking community include more powerful network intrusion detection and prevention systems as well as several network content aware applications such as Semantic Web.

The architecture can also be used for high-speed processing of natural languages. If provided with a grammar for a natural language a parser can be used as a front end to a high-speed semantic processing system. By identifying words within their context, a semantic processing system could more accurately define the meaning of each word. Another application for natural language processing could be using two grammars in different languages to more accurately translate documents from one language to another since word ordering is not always the same.

A more advanced application of a CFG parser in hardware might be a full source code compiler. In this type of application, the parser could identify tokens to create a parse tree. This parse tree could then be used for high-speed generation of executable code.

5.2. Future Work

The architecture presented in this paper is merely a starting point for a more robust and powerful hardware-based CFG parser. Future changes to the architecture will include improvements in speed and accuracy. To help improve the speed, we first need to incorporate changes to decrease the fanout delay of the decoder output. Other improvements in speed can be gained by scaling the design to process 32-bits or 64-bits per clock cycle.

To improve the robustness of the system, methods for error detection and correction can be implemented. With error detection and correction, the hardware based parser will be able to gracefully recover from errors when the input data doesn't match the grammar. After recovering from an error, the parser will continue processing from the point of the error.

Additionally, a stack can be added to the architecture to give the hardware parser all the power of a software parser.

We also plan to incorporate this work into the Field-programmable Port Extender (FPX) [23, 24, 1]. The FPX is a general purpose, reprogrammable platform that performs data processing in FPGA hardware. It can be used to augment networks with high-speed hardware-based packet processing. Modules have already been developed for the FPX that aid in the processing of common protocols such as IP [5] and TCP [29, 30]. By using the available infrastructure, we can quickly port our parsing hardware to process network packets.

References

[1] Field Programmable Port Extender Homepage. Online: `http://www.arl.wustl.edu/projects/fpx-/reconfig.htm`, Aug. 2000.

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

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

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *RFC 2475: An Architecture for Differentiated Services*, December 1998.

[5] F. Braun, J. W. Lockwood, and M. Waldvogel. Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware. *IEEE Micro*, Volume 22(Number 3):66–74, Feb. 2002.

[6] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[7] Y. H. Cho and W. H. Mangione-Smith. Programmable Hardware for Deep Packet Filtering on a Large Signature Set. In *First IBM Watson P=ac2 Conference*, Yorktown, NY, October 2004. IBM.

[8] Y. H. Cho and W. H. Mangione-Smith. A Pattern Matching Co-processor for Network Security. In *IEEE/ACM 42nd Design Automation Conference*, Anaheim, CA, June 2005. IEEE/ACM.

[9] Y. H. Cho and W. H. Mangione-Smith. Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2005. IEEE.

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

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

[12] C. R. Clark and D. E. Schimmel. Scalable Parallel Pattern-Matching on High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[13] C. Culy. The Complexity of the Vocabulary of Bambara. *Linguistics and Philosophy*, pages 345–351, 1985.

[14] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, Stanford, CA, August 2003. IEEE Computer Society Press.

[15] I. Dubrawsky. Firewall Evolution - Deep Packet Inspection. *Infocus*, July 2003.

[16] J. Fontana. XML Vendors Set to Unveil Gigabit Speeds. *Network World*, Apr 2004.

[17] P. Gupta and N. McKeown. Algorithms for Packet Classification. In *IEEE Network Special Issue on Fast IP Packet Forwarding and Classification for Next Generation Internet Services*. IEEE, March 2001.

[18] B. Inc. Strada Switch II BCM5616 Datasheet. In *http://www.broadcom.com*. Broadcom Inc., 2001.

[19] S. Iyer, R. Kompella, and A. Shelat. ClassiPI: An Architecture for Fast and Flexible Packet Classification. In *IEEE Network Special Issue on Fast IP Packet Forwarding and Classification for Next Generation Internet Services*. IEEE, March 2001.

[20] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *13th USENIX Security Symposium*, San Diego, CA, August 2004. USENIX.

[21] C. Kreibich and J. Crowcroft. Honeycomb . Creating Intrusion Detection Signatures Using Honeypots. In *Second Workshop on Hot Topics in Networks*, Cambridge, MA, November 2003. ACM.

[22] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *ACM SIGCOMM*, Philadelphia, PA, August 2005. ACM.

[23] J. W. Lockwood. An Open Platform for Development of Network Processing Modules in Reprogrammable Hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, CA, Jan. 2001.

[24] J. W. Lockwood, N. Naufel, J. S. Turner, and D. 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, Feb. 2001.

[25] J. McHugh, A. Christie, and J. Allen. Defending Yourself: The Role of Intrusion Detection Systems. *IEEE Software Magazine*, 17(5):42–51, September 2000.

[26] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic Signature Generation for Polymorphic Worms. In *IEEE Security and Privacy Symposium*, Oakland, CA, May 2005. IEEE.

[27] T. Parr and R. Quong. ANTLR: A predicated LL(k) parser generator, July 1995.

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

[29] D. V. Schuehler and J. Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. In *Proceedings of Hot Interconnects 10 (HotI-10)*, Stanford, CA, USA, Aug. 2002.

[30] D. V. Schuehler, J. Moscola, and J. W. Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System. In *Proceedings of Hot Interconnects 11 (HotI-11)*, pages 89–94, Stanford, CA, USA, Aug. 2003. IEEE.

[31] S. M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, pages 333–343, 1985.

[32] G. V. S. Singh, C. Estan, and S. Savage. Automated Worm Fingerprinting. In *ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, December 2004. ACM.

[33] Z. Solan, D. Horn, E. Ruppin, and S. Edelman. Unsupervised Learning of Natural Languages. *Proceedings of National Academy of Sciences of the United States of America*, 102:11629–11634, 2005.

[34] H. Song and J. W. Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In *ACM International Symposium on FPGAs*, Monterey, CA, February 2005. ACM.

[35] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[36] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification using Extended TCAMs. In *IEEE International Conference on Network Protocols*. IEEE, 2003.

[37] R. Stiennon. Deep Packet Inspection: Next Phase of Firewall Evolution. *Gartner Report*, December 2002.

[38] J. van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML Accelerator Engine. In *First International Workshop on High Performance XML Processing*, New York, NY, May 2004. ACM.

[39] E. Yoneki and J. Bacon. Content-Based Routing with On-Demand Multicast. In *24th International Conference on Distributed Computing Systems - Workshop on Wireless Ad Hoc Networking*, Tokyo, Japan, March 2004. IEEE.