IMPLEMENTATION OF NETWORK APPLICATION LAYER PARSER FOR MULTIPLE TCP/IP FLOWS IN RECONFIGURABLE DEVICES

James Moscola, Young H. Cho, John W. Lockwood *

Dept. of Computer Science and Engineering
Washington University, St. Louis, MO
{jmm5, young, lockwood}@arl.wustl.edu

ABSTRACT

This paper presents an implementation of a high-performance network application layer parser in FPGAs. At the core of the architecture resides a pattern matcher and a parser. The pattern matcher scans for patterns in high-speed streaming TCP data streams. The parser core augments each pattern found with semantic information determined from the patterns location within the data stream. The packet payload parser can provide a higher level of understanding of a data stream for many network applications. Such applications include high performance XML parsers, content-based/aware routers, and others. Additionally, a TCP processor allows stateful packet payload parsing of up to 8 million simultaneous TCP flows. The payload parser has been implemented in a Xilinx Virtex E 2000 FPGA on the Field-Programmable Port Extender platform. The parsing module runs at 200 MHz and parse raw data at 6.4 Gbps. The payload parser, integrated with the TCP processor, runs at 100 MHz for a throughput of 3.2 Gbps.

1. INTRODUCTION

Over the past several years packet content inspection has become a crucial part of network processing. Increasing numbers of mainstream networks are integrating packet payload processors to perform tasks such as data filtering, virus scanning, and content-based and aware routing. These applications require efficient, hardware-based pattern matchers. Many such pattern matching architectures have been proposed and developed [1, 2, 3, 4, 5]. However, all of these pattern matchers simply determine if a given pattern is present in a packet without considering the context of the pattern in the data stream. Without contextual information, the identified information has limited use. Additionally, many pattern matchers scan data at a packet granularity. This prevents such pattern matchers from finding patterns that may span across multiple packets.

In this paper we describe the implementation of a hardware based packet payload parser that addresses the problems described above. The architecture takes the next evolutionary step in network packet processing by implementing a complete payload parser for high-speed network traffic. The parser makes it possible to augment each pattern that is detected with semantic information based on the pattern's location in the data stream. Pattern semantics are defined by one or more grammars that accompany a pattern rule set. Additionally, a TCP protocol processor [6] is employed which allows the architecture to process entire TCP flows as a single continuous message as opposed to as fragmented pieces of data. The architecture has been implemented on the Field-Programmable Port Extender (FPX) platform [7].

2. RECONFIGURABLE HARDWARE PARSER

The architecture for the payload parser consists of a number of major components including a TCP deserializer which reconstructs packets received from the TCP protocol processor, an input/context controller, an SDRAM controller, a pattern matcher, and a parser core. The layout of these components is shown in figure 1. This section will describe each of these components with the exception of the TCP deserializer. For a description of the TCP deserializer see previous work by Schuehler [6].

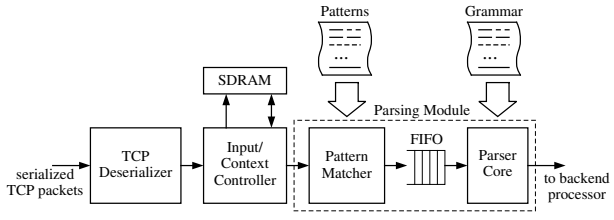

Fig. 1. Hardware parser architecture

*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.

2.1. Input/Context Controller

After passing through the TCP processor, data enters the payload parser as serialized TCP packets. The serialized packets consist of a TCP packet prepended with essential flow information such as a flow ID number, a start-of-flow flag, an end-of-flow flag, and the length of the TCP data. The packets are passed through the TCP deserializer to decode the packets and reconstruct the necessary control signals.

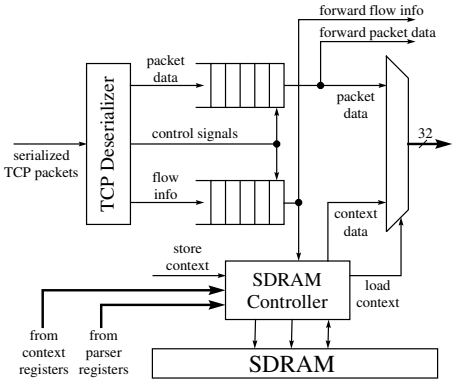

Fig. 2. Detailed view of the input/context controller

As shown in figure 2, the packet data and flow information are buffered in FIFOs until the system is ready to process the packet. Since our design processes entire TCP flows as a single continuous message, the system must first determine if the packet to be processed is the first packet of a TCP flow. If so, the packet data is simply forwarded to the parsing module for processing. If the packet is the second packet of a TCP flow, or any subsequent packet, the state of the parser may need to undergo a context switch. If this is the case, the context of the flow must be read from off-chip SDRAM and loaded into both the pattern matcher and the parser core. Once a packet has been completely processed, the state of the parsing module is written back to SDRAM.

2.2. Pattern Matcher

Our pattern matcher is a modified decoded character pipeline [1, 8] which has been scaled to accept a four character wide input. Scaling is achieved by replicating the pipeline until there is one pipeline for each character in the input width as shown in figure 3. The scaled pipeline receives four characters (32-bits) per clock cycle from the input/context controller. As a character enters the pipeline two copies of the character are created. The first copy of the character is decoded from an 8-bit to a 256-bit representation. This allows each character to be represented as a single bit, thus decreasing the routing resource required for string detectors. A second copy of the character remains as the smaller 8-bit representation which decreases the amount of data that must be written to SDRAM during a context switch. Each incoming 8-bit character is also passed through a delimiter decoder that outputs a single bit indicating whether the 8-bit value is a member of the delimiter set.

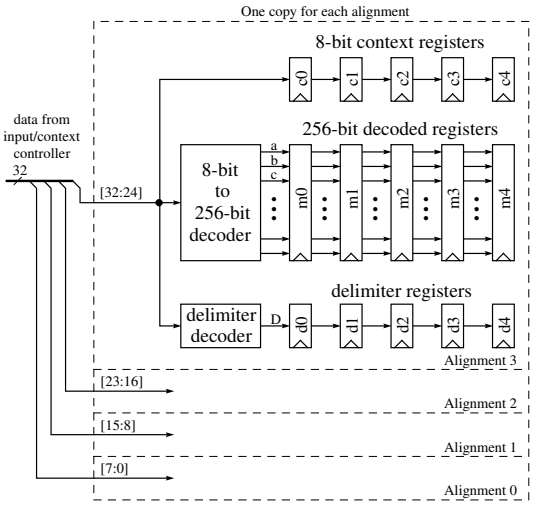

Fig. 3. Detailed view of the pattern matcher pipeline

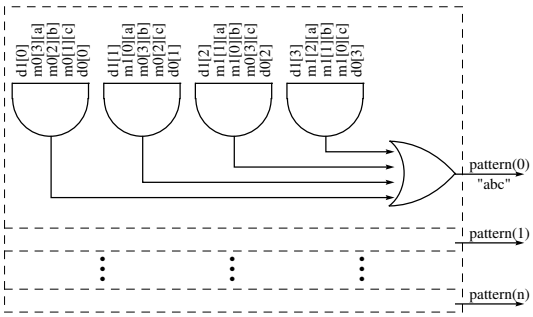

Fig. 4. Detailed view of a string detector

Patterns are detected by ANDing together bits from the decoded character pipeline. Since we use a scaled pipeline, we need to check for the presence of a pattern at each possible starting alignment. A pattern is detected if it is found at any one of the four possible starting alignments. Figure 4 illustrates the logic required to match the patten “abc”. The notation in figure 4 is *Register[Alignment][Character]*. For example, *m0[3][a]* represents the ‘a’ character bit of register *m0* in alignment 3. The outputs of the string detectors are passed to the parser core.

2.3. Parser Core

The parser core defines the semantics of each pattern that is detected by the pattern matcher. The hardware logic for

the parser core is determined from grammars that accompany the pattern rule set. While processing data, the parser maintains the state of the grammar allowing it to determine which patterns can occur next.

In our parsing structure, each pattern is represented using a simple primitive that consists of a single register and a single AND gate. The inputs to the AND gates are the outputs of the pattern matcher. The output of each AND gate represents a transition in the state of the grammar and is routed to the input of other pattern registers. Transitions are determined from the grammar using the *FOLLOW* set algorithm which traverses through the production list to find sets of patterns that can follow any other pattern in the grammar. These sets can then be used to determine the transitions necessary for mapping the grammar onto the hardware.

As the parser processes a data stream, the parser core receives a signal from the pattern matcher for each pattern that is found. This allows the parser core to traverse through the grammar and maintain the semantics of the data stream. These signals are also sent downstream to a backend processor accompanied by the state of the parser core. The state of the parser core indicates where in the grammar each pattern is found. This allows the backend processor to make more intelligent decisions about the meaning of patterns found.

To illustrate the structure of the parser core, we use the example grammar shown in figure 5. To create the parser core, we find the *FIRST* and *FOLLOW* sets (figure 6) for each of the patterns in the grammar. The output of each pattern primitive is forwarded to the inputs of the patterns listed in its *FOLLOW* set. When there is more than one connection to the input of a pattern register, an OR gate is used to combine the signals into a single bit input. Additionally, we define the starting point of the grammar as the *FIRST* set of the starting production. Figure 7 shows the hardware parser for the if-then-else grammar.

2.3.1. Context Switching in the Parser Core

In order to support millions of simultaneous TCP flows, our parser core needs to have an efficient context switching mechanism. To minimize the amount of data that is required to save the state of the parser, our hardware parser does not use stacks to support push-down automata. Maintaining millions of stacks on-chip is not currently possible, and swapping a stack in and out of off-chip memory between packets would dramatically decrease throughput. Thus, our parser does not behave exactly like a context-free grammar parser. Without stacks, the state of our parser core is wholly represented by the values in the parsers pattern registers. At the end of each packet, these values are written to SDRAM to save the state of the parser. When reloading the state of the parser core, the state information read from SDRAM and used to set each of the pattern registers in the parser.

No.		Production
1	E →	if C then E else E | go | stop
2	C →	true | false

Fig. 5. Production list for if-then-else grammar

Terminals	*FOLLOW* Set
if	$\{true, false\}$
then, else	$\{if, go, stop\}$
go, stop	$\{else, \epsilon\}$
true, false	$\{then\}$

Fig. 6. *FOLLOW* set for each of the terminal tokens

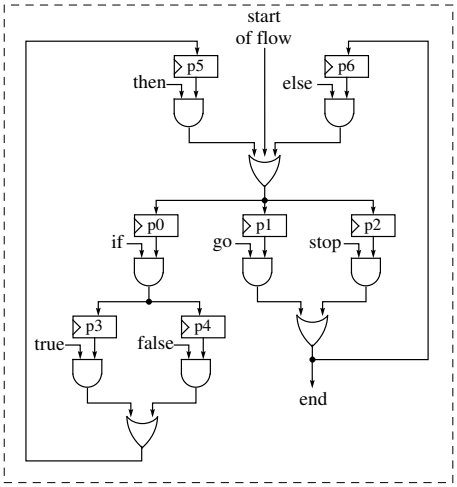

Fig. 7. Detailed view of the parser core

3. CASE STUDY: XML-RPC ROUTER

This section describes how our architecture was used to implement a parser that can be used to route XML Remote Procedure Calls (XML-RPC). The example XML-RPC router shown in figure 8 illustrates how the outputs of our parsing architecture can be used to control a simple switch.

To properly route XML-RPC messages based on the requested service, the hardware parser identifies which service has been requested by the remote client. In an XML-RPC message, the service name is represented as a `STRING` value in the `<methodName>` and `</methodName>` tags. Our parser detects these values and forwards them to a backend switch which can then route packets to the appropriate destination. The complete XML-RPC grammar that is used to generate the parser is shown in figure 9. The grammar contains 45 patterns with approximately 300 bytes of pattern data.

To create the hardware parser in figure 8, we use a series of scripts to generate the required VHDL code for the parsing module. The input to our scripts is a Yacc style grammar. A string detector is generated for each terminal symbol

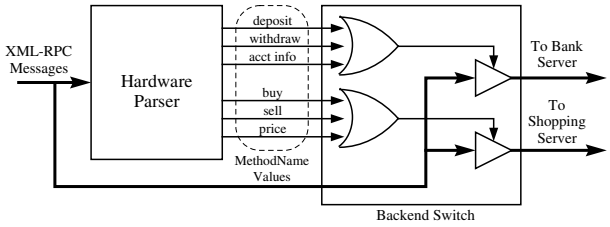

Fig. 8. Example of an XML-RPC message router

```
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>";
%%
```

Fig. 9. Grammar for XML-RPC

in the grammar. The logic for the parser core is determined from the productions of the grammar. The generated parsing module is combined with the infrastructure discussed in this paper. To test larger packet payload parsers, we replicated the XML-RPC grammar to create grammars with up to 400 patterns and 3000 bytes of pattern data.

3.1. Area and Performance

The hardware for five different size XML parsers was synthesized and placed and routed on the Xilinx Virtex E 2000 -8. Table 1 shows the results for the parsers, where the first number represents the parsing module, and the second number represents the parsing module integrated into the FPX.

When parsing the base XML-RPC grammar the parsing module is able to run at 201 MHz with a bandwidth of 6.44 Gbps. As the size of the grammar increases, the frequency of the design decreases slightly due to larger fanouts and extra routing resources required by the pattern matcher. Our results also show that as the size of the grammar increases the number of lookup-tables (LUTs) and D-flip-flops (DFFs) increases linearly.

The integrated design consists of the TCP deserializer, an input/context controller, an SDRAM controller, and the parsing module and runs at 100 MHz with a bandwidth of 3.2 Gbps. Since the size of the infrastructure does not change, the number of LUTs and DFFs maintain the linear growth displayed in the parsing module.

# of Bytes	Freq (MHz)	BW (Gbps)	LUTs	LUTs/ Byte	DFFs
300	201 / 94	6.44 / 2.99	650 / 2032	2.17 / 6.77	1674 / 2978
600	188 / 101	6.02 / 3.23	1039 / 2459	1.73 / 4.10	2439 / 3666
1200	190 / 96	6.08 / 3.07	1790 / 3233	1.49 / 2.69	3781 / 4917
2100	173 / 95	5.54 / 3.03	2968 / 4325	1.41 / 2.06	5323 / 6684
3000	174 / 86	5.57 / 2.74	4133 / 5439	1.38 / 1.81	6905 / 8345

Table 1. Device utilization for parser modules

4. CONCLUSION

This paper presented the implementation of a network application layer parser. The parsing module of the architecture maintains the state of the provided grammar. Integrating the parsing module with a TCP processor enables the parser to process and maintain state information for up to 8 million simultaneous TCP flows. The parser was implemented in the Xilinx Virtex E -8 FPGA on the FPX platform. The highly pipelined parsing module was placed and routed at over 200 MHz and can process data at over 6.4 Gbps. Integrating the parsing module into the FPX limits the frequency of the parser to 100 MHz with a bandwidth of 3.2 Gbps.

5. REFERENCES

[1] 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: IEEE, April 2004.

[2] 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*. Montpellier, France: Springer-Verlag, September 2002, pp. 452–461.

[3] C. R. Clark and D. E. Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," in *International Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, 2003, pp. 956–959.

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

[5] 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: IEEE, April 2004.

[6] D. Schuehler and J. Lockwood, "A Modular System for FPGA-based TCP Flow Processing in High-Speed Networks," in *International Conference on Field-Programmable Logic and Applications (FPL)*, Antwerp, Belgium, Aug. 2004, pp. 301–310.

[7] J. W. Lockwood, "An open platform for development of network processing modules in reprogrammable hardware," in *IEC DesignCon'01*, Santa Clara, CA, Jan. 2001, pp. WB–19.

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