Building High-performance Application Protocol Parsers on Multi-core Architectures

Kai Zhang[1], Junchang Wang[2], Bei Hua[3], Xinan Tang[4]
School of Computer Science and Technology
University of Science and Technology of China (USTC)
Hefei, Anhui, 230027, China
Multi-core Computing and Communication Lab
Suzhou Institute for Advanced Study, USTC
Suzhou, Jiangsu, 215123, China
{[1]kay21s, [2]wangjc}@mail.ustc.edu.cn
[3]bhua@ustc.edu.cn
[4]xinan.tang@sbcglobal.net

***Abstract*—Parsing packet payloads according to the syntax and semantics of an application protocol is a key step in analyzing network traffic. However, it is still a challenge to fulfill this task with high speed(10Gbps+) because parsing packets through deep-content analysis to build a corresponding syntax tree requires tremendous computing resources. Multi-core architectures provide a viable solution for building high-performance parsers for application protocols.**

Existing sequential application protocol parsers are hard to be reused, and building a new protocol parser from scratch is error-prone and time-consuming. This paper proposes a general and efficient approach to building high-performance parallel application protocol parsers on multi-core platforms. First, the open-source lexical analyzer FLEX is used to describe a protocol and generate a sequential parser. Then a source-to-source translation is performed to transform the sequential parser into a parallel one. Finally, an efficient parallel run-time system is built by employing lock-free design principles from top to bottom to support multi-threaded execution on multi-core processors. Experimental results show that our parsers achieve nearly 20Gbps for average HTTP packets and 5Gbps for the challenging smaller FIX packets.

***Keywords*-Multi-core; Protocol Parser; High-performance Network Processing;**

I. Introduction

Application protocol parsers, which translate raw packet streams into high-level representations of the traffic, form important components of network measurements, network monitoring tools(Tcpdump [4], Ethereal [1]), real-time network intrusion detection systems (Snort [3]), smart firewalls, and application proxies. Unfortunately, existing application protocol parsers are usually tightly coupled with their specific application environments which usually have different interfaces, data structures and hard-coded implementations, and thus are difficult to be reused. On the other hand, building application protocol parsers from scratch is a tedious, error-prone and sometimes prohibitively time-consuming task due to the complexity.

Recent works mainly focus on designing new declarative languages and compilers to simplify the construction of application protocol analyzers. For instance, binpac [15] proposes a declarative protocol language and implements a compiler to translate the declaration into a C++ parser. GAPA [5] proposes a self-contained system that handles both protocol parsing and traffic analysis, in which GAPAL, a protocol specification language, is designed to describe both ASCII and binary protocols. However, both systems run at very low speeds, e.g., tens to hundreds of Mbps for HTTP workload, which are far from the 1Gbps line rate of a typical enterprise network, let alone the upcoming 10Gbps speed.

Internet traffic approximately doubles every year, which is faster than what Moore's Law predicts for the semi-conduct industry. In addition, avalanche of new application protocols has emerged with increasing complexity. All of these factors make network equipment become the bottleneck of the Internet. As multi-core architecture becomes mainstream in the computer industry, more and more CPU cores are built into a general-purpose CPU, and therefore powerful PC becomes a promising candidate for high-performance network equipments. Recent progress in this area is very inspiring. For example, by exploiting the massively parallel processing power of GPU and highly optimized packet I/O engine, PacketShader [17] implements a PC-based software router that achieves nearly 40Gbps IPv4 forwarding on an 8-core PC. RouteBricks [14] implements a software router architecture that parallelizes router functionality across both multiple servers and multiple cores within a single server, demonstrating a

[4]This work was done when he was a honorary professor in USTC.

This paper is supported by the Fundamental Research Funds for the Central Universities under Grant No.WK0110000007.

35Gbps parallel router prototype whose capacity can be linearly scaled through the use of additional servers.

Both PacketShader and RouteBricks run on network layer to do *stateless* IP forwarding, where packets are treated independently and thus massive parallelism can be easily exploited on the individual packet level. On the contrary, application protocol parsers work on top of the TCP layer and deal with *stateful* packet processing where individual packets must be grouped into flows and pass through more processing steps, such as IP defragmentation, TCP reassembly and stateful analysis, message reconstruction, and message parsing. Furthermore, they must handle incremental input and maintain partial information across packets, messages, and two directions of a connection. Consequently, application protocol parsers require intensive computing and memory accesses, and thus are very difficult to achieve high performance.

In this paper, we investigate a viable approach to building high-performance application protocol parsers, which is (1) general enough to be applied to any application protocol, (2) easy to be constructed, and (3) efficient enough to meet 10Gbps throughput. The main idea is to use open source lexical analyzer FLEX to specify protocols and generate a sequential protocol parser, then parallelize the sequential code on multi-core architectures to achieve high speed. This solution is general since FLEX can be used to describe any application protocols, and easy to be developed since it avoids introducing a new language.

We develop high-efficient parallel protocol parser in two parts. First, we build a parallel run-time system on Intel multi-core architectures by exploiting lock-free design principle from top to bottom to eliminate unnecessary synchronization among CPU cores, as well as locks implicitly used in libc calls. Second, we perform a source-to-source translation to transform a sequential parser into a parallel one.

To evaluate the effectiveness of this parallelizing method, we implement a HTTP parser and a FIX (Financial Information eXchange protocol) parser on Intel multi-core processors. Experimental results show that the HTTP parser achieves almost 20Gbps on average HTTP packets and more than 5Gbps for the FIX parser with very small packet size. To the best of our knowledge, no literature has ever reported such a high speed for HTTP and FIX parser.

The rest of the paper is organized as follows. In section II, we present FLEX as a protocol description language and parser generation tool. In section III we discuss general design principles and implementation strategies of protocol parsers. To evaluate the protocol parsers, a system which parallelizes a complete L2 to L7 network processing system is presented in section IV. In section V, we report experimental results and analyze system performance. Section VI discusses related work, and section VII concludes.

II. FLEX as a Parser Generation Tool

In terms of syntax and grammar, application protocols are roughly classified into binary protocols and text-based protocols. Our application protocol parsers focus on the more complex text-based protocols which use ASCII text to encode both the structure and the content of messages.

```
 1: ^"GET"   | ^"PUT"     | ^"COPY"    | ^"HEAD"    |
 2: ^"POST"  | ^"MOVE"    | ^"TRACE"   | ^"MKCOL"   |
 3: ^"UNLOCK"| ^"CONNECT"| ^"OPTIONS" | ^"PROPFIND"|
 4: ^"LOCK"  | ^"DELETE" | ^"PROPPATCH"
 4:                     {BEGIN(ML); return REQUEST;}
 5:
 6: ^"HTTP"\/(0\.9|1\.0|1\.1){space}[1-5][0-1][0-9]
 7:                    {BEGIN(ML); return RESPONSE;}
 8: .                              {return NON_HTTP;}
 9:
10: <ML>"Transfer-Encoding:␣"           {BEGIN(TE);}
11: <ML>"Content-Length:␣"             {BEGIN(LEN);}
12: <ML>({alphanum}|-|:)+      {return SKIP_METHOD;}
13: <ML>{CRLF}    {BEGIN(INITIAL); return COMPLETE;}
14:
15: <TE>"chunked"       {BEGIN(ML); return CHUNKED;}
16: <TE>"identity"     {BEGIN(ML); return IDENTITY;}
17: <LEN>{digit}+        {BEGIN(ML); return LENGTH;}
18: <<EOF>>                              {return -1;}
```

Figure 1. HTTP Parser Skeleton

Application protocol parsers involve parsing protocol message headers to extract information such as message type, message length and semantically meaningful data fields. Text-based protocols usually have a character string called header field in each line, therefore searching predefined strings (also called patterns) is an essential task of a protocol parser. FLEX is an open source tool for generating scanners that recognize lexical patterns in text. A scanner to be generated is described in the form of pairs of regular expressions and actions, which are called rules.

FLEX also provides a mechanism for conditionally activating rules. For example, Any pattern with the prefix <sc>is active only when the scanner is in the start condition named <sc>. Rules activated by a start condition keep active until the next BEGIN action is executed.

Figure 1 is a simplified HTTP parser skeleton described with FLEX-style rules. It identifies the overall structure and essential fields of HTTP messages from packet streams. The first rule (line 1-5) identifies HTTP requests, the second rule (line 6-7) identifies HTTP responses, and the third rule (line 8) ignores other unrecognizable packets. The transfer-length of an HTTP message can be determined by one of the following two methods depending on how the message-body is encoded. If a Transfer-Encoding header field is present and has the value ”identity”, the message length is indicated in the Content-Length header field. If the Transfer-Encoding header field has the value ”chunked”, the message body is made up of a series of chunks whose size is given at the first line of each chunk. After a request/response message is detected, scanner activates the start condition ML which activates four rules in line 10-13 for searching the lines of Transfer-Encoding and Content-Length, skipping any other header lines (line 12), and identifying the end of message header(line 13).

This example shows that FLEX has the capability to describe any text-based application protocols by combining rules with start conditions. Furthermore, parsers generated by FLEX can be easily extended. For example, if $Cookie$ field needs to be analyzed, we can just add a rule as follows and write its processing code in main program.

```
<ML>"Cookie:␣"              { return ML_Cookie; }
```

Since what we require is (1) a streaming-based parser and (2) a parallel parser, two issues must be addressed before the code generated by FLEX can be used in network traffic analysis.

First, FLEX-generated parser processes input in a ”pull” mode, i.e., when an input buffer is exhausted the parser blocks to wait for further input. However, the network input is a steam of packets, and a protocol parser has to cope with potentially a large number of concurrent connections, which makes it impossible to spawn a thread for each connection. To solve this problem, a control block is allocated for each flow to save its parsing states, and the parsing process resumes whenever a new packet of the same flow arrives. Using one thread to handle a number of streams simplifies the streaming-based processing and can retain the control flow of the original code for further parallelization.

Second, protocol parsers generated by FLEX are sequential programs and cannot take advantage of multi-core architectures to achieve high performance. The following sections solve the problem by parallelizing the sequential parsers on multi-core processors.

III. Design Space Exploration

Two aspects of work should be done to parallelize sequential protocol parsers on multi-core processors, one is a parallel run-time system, and the other is a source-to-source transformation tool. This section presents design space explorations related to the two aspects.

A. Run-time System Built with Lock-free Design Principle

Previous experiences [10] [9] [6] have shown that lock-based schemes are not suitable for fine-grained network applications, and domain specific connection-affinity must be exploited to utilize the inherent parallelism in network applications. In general, connection-affinity has two properties: 1)packets belonging to the same connection must be processed in order; 2)packets belonging to different connections can be processed in parallel.

To enforce the first property and exploit the second one, we adopt a pipelined Run-To-Completion (RTC) parallel model to completely eliminate locks. In addition, we apply lock-free principle to entire parallelization framework to take advantage of the multi-core architecture.

We will review the basic and most important design principles for parallelizing L7 network applications on multi-core platforms, based on which we build a high-efficient parallel run-time system for the protocol parser.

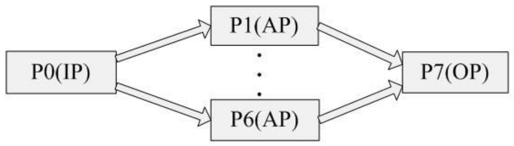

Figure 2. Pipeline Organization

1) Pipelined RTC Model : We build the run-time system with a pipelined approach. Each pipeline is divided into three stages named Input (IP), Application (AP), and Output (OP), respectively. Fig. 2 shows a 6-pipeline organization where IP stage and OP stage each uses one core, and AP stage uses six cores. IP core is in charge of packet receiving and dispatching, AP core runs in RTC model to process a packet from beginning to end without serving other packets, and OP core collects and records analysis results and sends them out. To guarantee the connection-affinity property, IP core uses a *symmetric* packet-classifying-hash function to dispatch packets so that packets belonging to the same connection are distributed to the same AP core.

The RTC model greatly simplifies the parallelizing work and the corresponding source-to-source code transformation. With only one packet processed on each AP core at a time, the sequential control flow of an application remains the same, and only global tables need to be split among all AP cores so that each core just accesses its own private sub-tables. In this case, the focus of the source-to-source transformation is simply to replace all global accesses with local ones in each AP core.

2) Lock-free FIFO: We use a single-producer/single-consumer FIFO to connect two neighboring cores in a pipelined fashion. To prevent core-to-core communication from being a bottleneck, we develop a cachefriendly concurrent lock-free FIFO by aggregating read-/write operations based on cache line access [10] [13]. The main idea is to separate the head and tail pointers of a FIFO in different cache lines so that cache line thrashing can be avoided as much as possible. When the FIFO is nearly empty or full, a spin-loop is used on both sides to synchronize the enqueue/dequeue operation.

3) Pre-allocated and Lock-free Control Blocks: Protocol implementations usually use some fixed-sized data structures called control blocks to record states. For example, TCP control blocks are used to track the states of TCP connections. In existing implementations, control blocks are dynamically allocated and de-allocated by calling libc functions – malloc() and free(). However, hidden locks are found in the two functions which seriously degrade application's performance in multi-core environments when they are frequently invoked.

To remove the hidden-locks, memory pre-allocation is adopted in our design. A bulk of memory space is pre-allocated for each type of control blocks on each CPU core. The allocation of a control block includes locating the corresponding free list via CPU ID and List ID, and getting a free block from the list; the de-allocation of a control block simply returns the block to its corresponding free list. Since each CPU core operates on its own lists, no lock/unlock operation is needed as it would be in malloc() and free().

Table I
CPU CYCLES USED FOR MEMORY ALLOCATION

	Allocated Block Size(Bytes)				
(CPU Cycles)	32	64	128	256	512
Our malloc	52	53	54	58	75
Libc malloc	170	202	271	410	695
Improv.(%)	69	74	80	86	89

Table I compares the CPU cycles taken to allocate different size of blocks on a single core by our lock-free malloc() and the libc malloc(). As shown in the table, lock-free malloc() outperforms libc malloc() by nearly 90% for 512-byte blocks. As the number of CPU cores increases, higher improvements are expected since hidden-locks are completely eliminated in our lock-free malloc().

B. Source-to-Source Transformation

FLEX-generated C code cannot run in parallel due to the use of global variables. A set of global variables are used by FLEX to track the parsing states, e.g., variable $yytext$ is used to point to the current pattern identified.

Since protocol parser works on the basis of connection, and all packets belonging to the same connection are distributed to the same AP core in our parallel run-time system, global variables can be converted to local ones by performing the following two passes:

```
                                  struct yy_buffer_state
                                  {
char * yytext;                      char * yytext;
int    yyleng;             =>       int    yyleng;
char * yy_c_buf_p;                  char * yy_c_buf_p;
char   yy_hold_char;                char   yy_hold_char;
...                                 ...
                                  }
```

Figure 3. Mapping from Global Variables to Local Variables

1) In the first pass, all functions that use those global variables are found out. This can be done in a bottom-up fashion. If a global is referenced by a pointer, we require an annotation to that variable; otherwise an accurate alias analysis must be used [19].
2) In the second pass, for each function found out in 1) we add an extra pointer parameter such as $*yy\_ptr$ that will receive a pointer to the struct yy_buffer_state; for the function body, a variable reference is replaced by a pointer dereference as shown below:

```
yy_hold_char   -->   yy_ptr->yy_hold_char
```

At runtime, whenever a connection is established, a yy_buffer_state instance is allocated from heap and its pointer is recorded in the connection control block. During the life time of a message, this structure is associated with the connection and the parser only visits the assigned structure so that it can process millions of connections in parallel. With the above techniques, any FLEX-generated application protocol parsers can be easily and effectively parallelized.

IV. RUN-TIME SYSTEM SETUP

To build a parallel run-time system customized for network processing, we parallelize a TCP/IP stack and a port independent protocol identification engine from Libnids [2] on Intel multi-core processors. In this section, we introduce the hardware platform and discuss design decisions in pipeline mapping and run-time system implementation.

A. Hardware Platform

We build our run-time system on commercial multi-core architectures. Our server is equipped with an Intel Xeon L5640 Westmere CPU(Hexa-core processor, 2.26Ghz) and 8GB memory. The L1 and L2 cache are built within each core, which are 64KB and 256KB, respectively. The L3 cache is 12MB and is shared among all cores. It also has a built-in memory controller which offers lower memory access latency. Our system runs on a 64-bit Linux 2.6.36 kernel and is compiled by GCC 4.1.2 with -O2 option.

B. Pipeline Mapping

Our parallel run-time system is built on a pipelined RTC model illustrated in Fig. 2 where pipeline partition is the first decision to make. Roughly a packet processing pipeline can be divided into 6 stages: (1) packet input and checksum verification; (2) IP defragmentation; (3) TCP processing; (4) port independent protocol identification; (5) L7 processing such as HTTP protocol parsing; (6) post-processing on the analyzed results. Since no hardware FIFO support is present in commodity multi-core processor so far, we implement efficient software lock-free FIFO to connect neighboring stages.

Table II
EXECUTION TIME OF PIPELINE STAGES INCLUDING FIFO AND LB

Pcap	LB	FIFO	L3	TCP	Iden.	HTTP
160	120	150	170	1300	80	400

To make a sensible decision on pipeline partitioning, the execution time of each pipeline needs to be measured. Table II shows the average execution time of each stage including FIFO and Load Balance(LB) when running the HTTP parser with trace File-1(See Section V-A) as the input. The trace file is read into memory in advance, and then packets are fed into the system for processing. It's worth noting that execution time of TCP stage varies significantly with the number of concurrent TCP connections and the number of cores serving in this stage, therefore its cost cannot be taken as a fixed one.

Assume T_{IP}, T_{FIFO}, and T_{AP} are the execution time of IP stage, $FIFO$ operation and AP stage, respectively. In a balanced pipeline execution model with N pipelines, the following equation is satisfied.

$$T_{IP} + T_{FIFO} = (T_{AP} + T_{FIFO})/N \qquad (1)$$

Using the data in Table II, T_{IP} and T_{AP} can be calculated as follows:

$$\begin{aligned} T_{IP} &= T_{Pcap} + T_{LB} = 280(cycles) \\ T_{AP} &= T_{L3} + T_{TCP} + T_{Iden.} + T_{HTTP} \\ &= 1950(cycles) \end{aligned}$$

The optimal value of N is 5 (pipelines), since $(280 + 150 = 430) \approx ((1950 + 150)/5 = 420)$.

Considering the total number of available cores is 6 in our server, we use 4 pipelines to build the HTTP parser(one core is reserved for OP and OS scheduler).

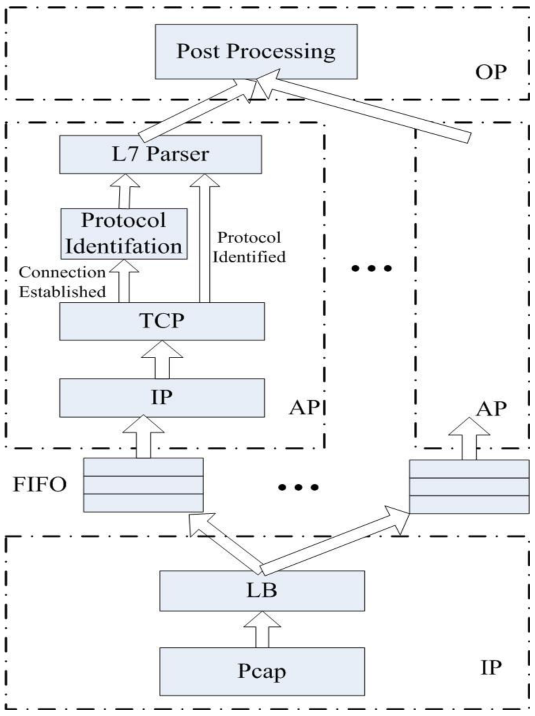

Figure 4. Pipeline Mapping

V. EXPERIMENTS AND ANALYSIS

This section evaluates the performance of our system and analyzes the factors that may have impacts on system performance. Experimental environment is described in Section IV-A. Although 10Gbps I/O engine has been reported in [7] [17] [14] [16], we choose to read the trace file from memory instead of real NIC so that higher input speed is allowed and the maximum throughput of our protocol parsers can be tested.

A. Trace File Characteristics

Three trace files are used in the experiments, whose characteristics are described in Table III. Column $Packets$ counts the number of packets in each trace file, and $Pkt.\ Len.$ is the average size of these packets. $TCP\ Conn.$ is the average number of concurrent TCP connections observed in the trace files, and $Conn.\ Rate$ indicates the average arrival rate of new TCP connections measured in the number of new connections per 1000 packets. When a header line strides two packets, a new buffer is allocated and the constituent pieces of the header line are copied from their original buffers to the new buffer for concatenation. We call this the A/B buffer problem. Column A/B is the percentage of packets that arouse A/B buffer problem. A chunked-encoding message takes more CPU cycles than a transfer-encoding message, as the length of each chunk is unknown in advance and the parser has to inspect the packet payload to skip them. Column *Chunk* denotes the percentage of packets that are encoded in chunks.

Table III
CHARACTERISTICS OF TRACE FILES

File	Packets	Pkt. Len.	TCP. Conn.	Conn. Rate	Per.(%)	
					A/B	Chunk
1	2,472,221	764	31,768	49.8	0.2	3.0
2	5,697,000	319	8,721	84.5	21.9	32.4
3	1,000,000	92	77,709	94.8	0.0	0.0

File-1 was collected from a university gateway in January 2011, and features large packet size and large number of concurrent TCP connections. File-2 was generated by an HTTP protocol generator, and features high TCP connection rate, high percentage of A/B buffer packets and chunked-encoding packets. The version of HTTP in the two trace files is HTTP 1.1. To apply the method to other application protocols, we build a FIX protocol parser and use File-3 as its input. FIX is a TCP-based application protocol for real-time information exchange in securities transaction and market. File-3 was generated by a FIX protocol generator, and features small packet size, large number of concurrent TCP connections and high TCP connection rate. In addition to that, FIX parser has to inspect each byte of the FIX packet, and therefore the workload is very heavy.

B. Performance Evaluation and Workload Analysis

This section evaluates the throughput of the two parallel protocol parsers. The two parsers run on the hardware platform described in IV-A. Of the six cores, one core is reserved for OP stage and OS scheduler, and therefore at most five cores, i.e., four pipelines, are used in the experiments. The throughput of the two parsers tested with different number of cores are reported in Table IV, where the first two rows correspond to the HTTP parser with File-1 and File-2 as its input, and the last row corresponds to the FIX parser with File-3 as its input. To analyze the cause of performance variance, execution time of each pipeline stage is measured and listed in Table V.

Table IV
PERFORMANCE ON HEXA-CORE SERVER

(Gbps)	Number of Cores Used				
File	1	2	3	4	5
1	7.4	7.3	12.0	15.3	19.0
2	2.1	2.1	4.1	5.9	7.8
3	1.4	1.4	2.6	3.9	5.3

Table V lists the average cost of each pipeline stage of the three trace files, and the last column shows the total costs of AP stage. Since the cost of Pcap stage and FIFO stay the same, they are not listed in the table. The identification stage takes less cycles because its time is averaged among multiple packets per connection.

Table V
PIPELINE STAGE COSTS PER PACKET

(Cycles)	Pipeline Stage					AP
File	LB	L3	TCP	Iden.	L7	Total
1	120	170	1300	80	400	1950
2	90	167	500	77	2200	2944
3	65	172	2040	50	3100	5362

Since File-1 has the largest average packet size and the least percentage of A/B buffer and chunked-encoding packets, it is the best-case input for HTTP parser and the throughput reaches almost 20Gbps. The large number of concurrent TCP connections leads to a heavy load on the TCP stage.

File-2 has larger percentage of A/B buffer and chunked-encoding packets and thus it requires more processing resources. In this trace file, each generated packet has a long header field in the HTTP payload, which requires byte-by-byte skipping in parsing(1800+ cycles more than File-1).

File-3 is the worst case input and gets the lowest throughput of 5.3Gbps. It contains very small packets (less than 100 bytes), huge number of concurrent TCP connections and high TCP connection rate which increases the workload of TCP stage(2040 cycles). Moreover, since each field of a FIX packet needs to be parsed, almost each byte has to be loaded into cache.

When one core is used, the parser degenerates to a sequential one, therefore no FIFO is needed and the communication cost is zero. When two cores are used, FIFO is used to connect IP core with AP core, which introduced a communication cost of 150 cycles, and that's why performance drops from column 1 to column 2 in Table IV. From the last column, we can deduce that the huge cost in AP stage results in an imbalanced pipeline, which is the root cause for the low performance of File-2 and File-3. However, as the number of cores increases, the overhead of AP stage can be amortized and the overall performance can be improved continuously. As shown that our HTTP parser can run much faster than 10Gbps in average cases and approach 80% of 10Gbps in extreme cases, we believe our system can handle real HTTP traffic at 10Gbps in practice.

When one core is used, the parser degenerates to a sequential parser, where no FIFO is needed and the communication cost is zero. When two cores are used, FIFO is used to connect IP core with AP core, and a 150 cycles of communication cost is introduced as well. Thats why performance drops from column 1 to column 2 in Table IV. Although the AP stages workload imposed by File-2 and File-3 is amortized among four pipelines, it is still much higher than the workload of IP stage, and thats why these two cases have lower performance. However, if more cores are available, system performance will improve.

C. System Speedup

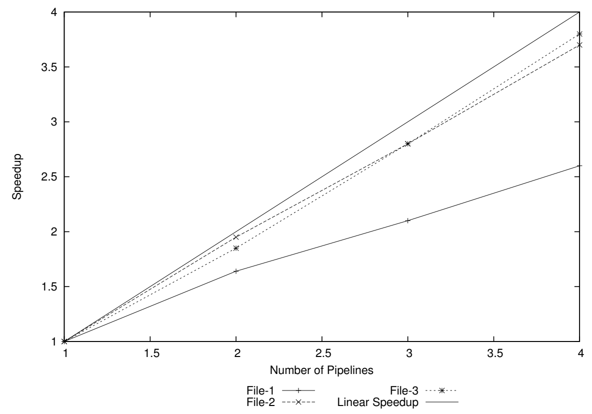

Figure 5. System Speedup

Figure 5 plots the speedup curves of the three cases in Table V. The two curves corresponding to File-2 and File-3 are close to linear speedup, whereas the curve corresponding to File-1 increases a little slower.

According to Amdahls law, speedup of a parallel system is limited by the sequential portion of the workload. In the three cases, the sequential portion of the workload (i.e., the workload of IP stage) is the same, 430 cycles; but the parallel portion of the workload (i.e., the workload of AP stage) varies greatly, from 1950 cycles in case 1 to 5362 cycles in case 3. Case 1 has the highest percentage of sequential workload, so the system speedup is the lowest. When four pipelines are used in case 1, the workload of IP stage and AP stage on each pipeline almost balance, reaching the optimum configuration. On the contrary, case 3 has the lowest percentage of sequential workload (only 8%), so the system speedup is remarkable. Even four pipelines are used in case 3, the workload of AP stage on each pipeline is still much higher than that of IP stage; thats why case 3 has the lowest throughput. To improve the performance of FIX parser, either more pipelines are employed when extra cores are available, or try to reduce the workload of AP stage by further optimization.

D. Memory Pre-allocation

This section evaluates the impact of memory management strategy on system performance. Two memory management strategies are compared, dynamic memory allocation with libc malloc() and free(), and memory pre-allocation with our lock-free malloc() and free(), and the throughput of HTTP parser running on different number of cores with File-1 as its input are reported in Table VI. In the table, row *Libc* lists the system performance when libc malloc() and free() are used to allocate memory dynamically, and row *Ours* is the system performance when memory pre-allocation policy is used with lock-free malloc() and free().

Table VI
PERFORMANCE WITH DIFFERENT MEMORY MANAGEMENT STRATEGIES

(Gbps)	Number of Cores Used				
	1	2	3	4	5
Libc	7.4	7.1	11.7	14.7	15.9
Ours	7.4	7.3	12.0	15.3	19.0
Imp.(%)	0	2.8	2.7	4.0	19.5

By replacing dynamic memory allocation with memory pre-allocation, the overhead of kernel context switching in allocating and freeing memory is eliminated. Moreover, by avoiding the use of the locks embedded in libc malloc() and free(), the contention in multi-core environments is eliminated. Therefore, the system performance improves. In Table VI, nearly 20% improvement is achieved with four pipelines. As more cores are used, even higher improvement can be expected.

E. Scalability to Number of Patterns

So far, our HTTP parser skeleton only includes basic rules that are necessary to recognize HTTP request and response messages from network traffic. In a real application, more header lines need to be inspected. We add more rules to the parser skeleton to allow more header lines to be parsed, and Table VII lists the system performance of HTTP parser on File-1 when 10 and 20 more rules are added to the parser skeleton. Each column also gives the performance degradation compared with the basic one.

Table VII
PERFORMANCE WITH DIFFERENT NUMBER OF PATTERNS

	Number of Patterns		
	Basic	+10	+20
Speed(Gbps)	19.0	18.9	18.6
Deg.(%)	–	-0.5	-1.5

As shown in the table, when twenty more rules are added into the parser, performance degradation is only 1.5%. This demonstrates that the scalability of our parser to support more complicated DPI-based applications.

VI. Related Work

Existing work on application protocol parsers mainly focuses on designing new declarative languages and compilers to facilitate the construction of application protocol analyzers [15] [5], however the speed of these parsers is very low. We rely on open source lexical analyzer FLEX to specify protocols and generate sequential protocol parsers without introducing new languages. And this work mainly focuses on parallelizing sequential protocol parsers on multi-cores to achieve high speed.

Snort [3] is an open source NIDS that has tract great interests from both industry and the academia to port it on the multi-core architectures [9] [18] [12]. Although Snort has embedded protocol parsers, they cannot be reused as general application protocol parsers to build other other DPI (Deep Packet Inspection)-based traffic monitoring systems.

RouterBricks [14] and PacketShader [17] implement high-speed software routers on multi-cores to do stateless IP forwarding. Our work deals with stateful packet processing that usually requires intensive computation and memory access, and therefore high performance is very difficult to achieve. The work of parallelization of port-independent protocol identifier is reported in [8] [10], where [8] proposes a highly scalable parallelized L7-filter system architecture with affinity-based scheduling on a multi-core server. However, its maximum performance is below 1.5Gbps even with eight cores.

VII. Conclusion

This paper proposes a general approach to building high-performance parallel application protocol parsers on multi-core platforms. The use of FLEX facilitates the protocol description and parser generation, and source-to-source translation transforms the FLEX-generated code into a parallel one. By exploiting lock-free design principles in building parallel run-time system, unnecessary data sharing is eliminated and high-performance is achieved.

Experimental results show that the approach is competent and efficient in building high-performance application protocol parsers. Our parallelized HTTP parser achieves almost 20Gbps line rate on average HTTP packets, and the FIX parser achieves more than 5Gbps for small-sized packets.

References

[1] Ethereal. "http://www.ethereal.com/".

[2] Libnids. "http://libnids.sourceforge.net/".

[3] Snort. "http://www.snort.org/".

[4] Tcpdump. "http://www.tcpdump.org/".

[5] N. Borisov, D. Brumley, H. J. Wang, and C. Guo. Generic application-level protocol analyzer and its language. In *Network and Distributed System Security Symposium*, 2007.

[6] L. Foschini, A. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A parallel architecture for stateful, high-speed intrusion detection. In *Proc. of ICISS*, 2008.

[7] G. Liao and X. Zhu and L. N. Bhuyan. A new server i/o architecture for high speed networks. In *HPCA'11*, 2011.

[8] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding. A scalable multithreaded l7-filter design for multi-core servers. In *ANCS'08*.

[9] Intel Corporation. Supra-linear packet processing performance with intel multi-core processors white paper, 2006. "http://www.intel.com/technology/advanced_comm/311566.htm".

[10] J. Wang and H. Cheng and B. Hua and X. Tang. Practice of parallelizing network applications on multi-core architectures. In *ICS'09*, 2009.

[11] M. Kulkarni, P. Carribault, and E. K. Pingali. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA'08, 2008.*

[12] A. Kunze, S. Goglin, and E. Johnson. Symerton - using virtualization to accelerate packet processing. In *ANCS'06*, 2006.

[13] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *IPDPS*, 2010.

[14] M. Dobrescu and N. Egi and K. Argyraki and B. Chun and K. Fall and G. Iannaccone and A. Knies and M. Manesh and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *SOSP'09*, 2009.

[15] R. Pang, V. Paxson, R. Sommer, and L. Peterson. A yacc for writing application protocol parsers. In *IMC'06*, 2006.

[16] QUALCOMM. UIO-IXGBE. "https://opensource.qualcomm.com/wiki/UIO-IXGBE".

[17] S. Han and K. Jang and K. Park and S. Moon. Packetshader: A gpu-accelerated software router. In *SIGCOMM'10*, 2010.

[18] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.

[19] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proc. of PACT*, 1997.

[20] J. Verdu, M. Nemirovsky, and M. Valero. Multilayer processing - an execution model for parallel stetful packet procesing. In *ANCS'08*.