THE OPTIMIZATION OF HTTP PACKETS REASSEMBLY BASED ON MULTI-CORE PLATFORM

Fan Yang[1], Yinan Dou[1], Zhenming Lei[1], Huanhao Zou[2], Kun Zhang[1]

1 School of Information and Communication Engineering,
Beijing University of Posts and Telecommunications, Beijing, China
2 IBM China Research Lab
yangfangood@gmail.com, {dyn, lzm, zhangkun}@kuanguang.com.cn, zouhh@cn.ibm.com

Abstract

HTTP is the most popular application-layer protocol in the Internet. Improving the performance of HTTP packets reassembly is the key point of HTTP traffic analysis. In this paper, we first parallel the traditional serial HTTP packets reassembly system according to the HTTP protocol features. This optimization improves the throughput of packets reassembly system by 45% on the generic multi-core platform. Second, by considering the parallelization extra cost on the multi-core platform, we propose a new method to calculate system speedup based on the Amdahl's law. Furthermore, we make a theoretical analysis to discuss the relationship between system speedup and throughput improvement and then we can estimate the upper bound on throughput improvement.

Keywords: packets reassembly; multi-core; http; performance

1 Introduction

Presently, HTTP protocol is not only used for WEB service, but also carries a lot of useful information. Lots of studies focus on the HTTP packets deep inspection. DPI technology has been widely used in monitoring and analysis for application layer information [1]. However, a complete application layer message is often divided into several HTTP packets, so that packets can pass through a link with a proper maximum transmission unit (MTU). In order to get the meaningful application layer message, traffic monitoring system has to reassemble the HTTP pieces into one HTTP packet so as to make DPI analysis more effective. For example, when a client sends a HTTP GET to require for a big jpg image, server will reply a 200 OK message, which contains the target image. Because of the big size of image, one 200 OK message has to be cut into several HTTP packets to send to client. When the traffic monitoring system catches these packets, it must reassemble the separated packets into a complete message so as to further analysis.

Traffic monitoring system is often located between two networks (such as Metropolitan Area Networks) and receives the packets which mirrored from the backbone link. In this location, monitoring system must deal with the whole link HTTP traffic rather than several HTTP sessions. Besides this, HTTP packets reassembly is a computing intensive work [2]. To meet the requirement of computing, we use the generic multi-core platform which has a strong computational capability. Using multi-core platform, the performance of HTTP packets reassembly can be improved significantly. We do the HTTP packets reassembly optimization based on an open Source Programming Interface called Libnids [3]. This program can complete network packets capture, IP defragmentation, TCP stream reassembly, attacks detection, etc. The implementation of IP defragmentation and TCP stream reassembly are the same as Linux kernel protocol stack, so they are stable and reliable. Using Libnids to do our optimization on multi-core platform, we can obtain a good performance of HTTP packets reassembly.

The rest of this paper is organized as follows: section 2 states the background and motivation. Section 3 discusses the optimization methods and parallelization strategies. Section 4 describes our experimental environment and results. Section 5 makes a theoretical analysis for the optimization. Section 6 presents the final conclusion and future work.

2 Background and motivation

2.1 HTTP packets reassembly proportion

Table 1. HTTP packets reassembly proportion

Applications		Proportion
Web Services	200 OK	70%-80%
	POST	80%-90%
Http Streaming Video		60%-70%

978-1-4244-6853-9/10/$26.00 ©2010 IEEE

Nowadays, IP fragments are quite few, while a large number of HTTP pieces to be reassembled exist in network. A lot of sensitive information is cut into several HTTP packets, and it's difficult to get valuable information without reassembly. By our statistics, we find that HTTP packets reassembly proportion is very high when we use the Internet, such as WEB services and HTTP streaming videos. We choose these two applications and estimate the reassembly proportion in Table 1.The large proportion in Table 1 implies great potential for optimization.

2.2 Serial HTTP packets reassembly process

In traffic monitoring system, HTTP packets reassembly commonly consist of network packet capture, IP layer defragmentation, TCP layer reassembly, and then submit reassembled data to HTTP protocol stack [4]. Because HTTP protocol stack does not provide any reassembly mechanism, this work is finished in IP layer and TCP layer. However, If we can make some effective optimization according to HTTP protocol characteristic, it will be helpful for improving packets reassembly performance.

We study the traditional serial packets reassembly process from Libnids 1.23. According to the system functions and modules, we divide it into five phases and draw the architecture in Figure 1.

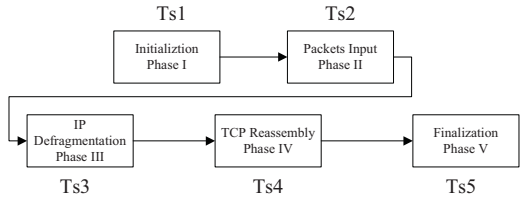

Figure 1. Five modules of serial HTTP packets reassembly architecture

2.3 CPU cycles proportion analysis for serial HTTP packet reassembly

We use Intel VTune [5] to analyze CPU cycles proportion of the serial reassembly system [6, 7]. The results are showed in Table 2.

Table 2. CPU cycles proportion which consumed by five phases separately

phase	Proportion	
I	0.27	T_s1
II	10.10	T_s2
III	14.83	T_s3
IV	73.72	T_s4
V	1.08	T_s5

In Table 2, we know that the proportion of phase I and phase V are very small, while phase IV has about 3/4 proportion. It means that phase IV consumes most of the time in packets reassembly system. So we must parallel this part in order to make a better task balance. This optimization will improve the reassembly performance.

3 Optimization and parallelization strategies

3.1 Reassembly optimizations according to HTTP packets characteristics

Because there is no specific optimization for HTTP in Libnids reassembly mechanism, we carry out following methods to increase reassembly throughput based on HTTP packets features.

1) The beginning of HTTP packets reassembly

The old reassembly mechanism needs whole session traffic to finish reassembly. It requires that a session must start from "three-way handshake"; otherwise all the packets in the session will be dropped. As we known, in high-speed link, small packets collecting may give great pressure to the traffic mirror equipment. So we propose a method that reassembly start from the first packet which containing HTTP data rather than "three-way handshake". Specifically, for HTTP request, the content of "GET", "POST", "HEAD" are the beginning of reassembly; for HTTP response, the content of "HTTP" is the beginning of reassembly. Thus, on the one hand, we will not drop the whole session because of losing "three-way handshake" packets; on the other hand, we will not wait for the sessions without actual data transfer.

2) The end of HTTP packets reassembly

First, for HTTP request, if the request packets contain the domain of "Content-Length", we will reassemble the packets based on this length. On the contrary, we will end the packets reassembly with two "CRLF" if there is no "Content-Length" domain.

Second, for HTTP response, if the response packets contain domain of "Content-Length", we will reassemble the packets based on this length; conversely we can determine the reassembly end with a FIN packet.

Third, in order to avoid a long wait for some packets and reduce unnecessary resources consumption, we add the timeout checking function for HTTP packets reassembly.

3) Unidirectional HTTP packets reassembly

In old reassembly mechanism, a HTTP session is defined as a bidirectional connection between a pair of hosts. In the process of reassembly, if one packet missed in either direction, bidirectional HTTP reassembled data can not be submitted to application layer, even if reassembly in one direction is completed. This shortcoming not only costs system memory, but also causes an application data analysis delay.

Therefore, we divide one bidirectional HTTP session into two unidirectional sessions, so the reassembly

can be carried out separately. In this way, we not only decouple the two directional HTTP packets, but also make a more explicit analysis for two respective directional HTTP traffic.

3.2 Parallelization strategies

After analyzing the old serial reassembly system, we think that it is easy and efficient to parallel the serial system depend on the five phases described in section 2.1. Phase I and phase V can not be paralleled, they are the fixed part in the system. From Table 2, we can divide phase III and IV into two separate parallel threads but the load imbalance will influence the performance. So phase II can be arranged with phase III in one thread. However, the proportion of phase II + III is 24.93%, which still cannot match phase IV (73.72%). This is showed in Table 3. After further analysis, we find the function "mkhash" which in phase IV has a proportion of 23.18%. This function is used to calculate hash values and we make a load balance by moving function "mkhash" from phase IV to III. Table 4 shows the proportion of each phase.

Table 3. Proportion of phase I – V before balance

phase	Proportion
I	0.27
II + III	24.93
IV	73.72
V	1.08

Table 4. Proportion of phase I – V after balance

Phase	Proportion	
I	0.27	T_p1
II + III + mkhash	48.11	T_p2
IV - mkhash	50.54	
V	1.08	T_p3

From these analyses, we describe the parallelization strategy as Figure 2.

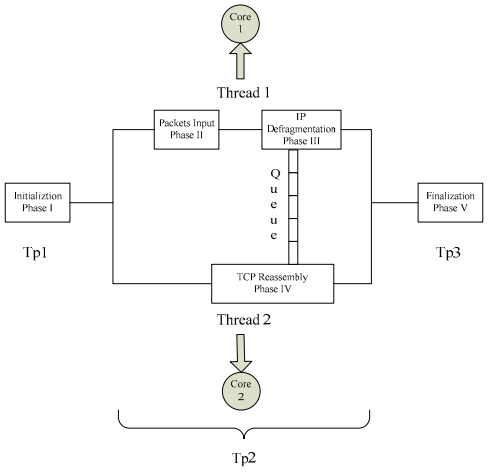

Figure 2. Parallelization strategy

Summarizing parallelization strategies as follows:

1) Taking into account load balance, we distribute the most resource-consuming part to two separate threads so as to implement parallelism.
2) Using the affinity-based scheduling for a multi-core server and distributing different tasks to each core.
3) Because IP fragments are quite few in current network, we put IP layer defragmentation module and packets input module in one thread.
4) IP layer reassembly and TCP layer reassembly communicate with a single-producer single-consumer ring queue so as to avoid synchronization with locks.
5) We replace a malloc() by pre-allocating a bulk of memory space to solve the problem caused by frequently allocating and releasing memory

4 Experimental results

4.1 Experiment platform

We carry out experiment on DELL PowerEdge R710 multi-core server with dual Intel Xeon 5520 2.26GHz processors. We choose CentOS 5.3 (kernel 2.6.18) and gcc 4.1.2 as software experiment.
The trace is collected from one backbone link in CMNET, 2009-4-4. This trace totally consists of HTTP traffic. Table 5 describes the details

Table 5. Trace conditions

trace	
packet number	200000
total length of data	108809584 bytes
average packet length	544.05 bytes
flow number	18033

4.2 Experimental results

For the trace mentioned above, we achieved approximately 45% throughput improvement.
Table 6 compares the maximum throughput between serial and parallel reassembly system.

Table 6. Comparison of maximum throughput

	throughput (Mbps)
serial	1459.14
parallel	2121.15
improvement	45.37

In serial reassembly system, for this HTTP trace, maximum throughput can reach 1.4 Gbps. After optimizing, the parallel HTTP packets reassembly system throughput can reach 2.1Gbps. This means that after optimization, packets reassembly system obtains an improvement of 45.37% in throughput.

4.3 Parallel extra cost

Comparing serial reassembly mechanism, Parallelization brings up extra cost to HTTP packets reassembly system. The main overhead is caused due to queue operations which between IP and TCP reassembly. By measuring parallel system, we get the CPU cycles proportion of each extra-cost function as Table 7.

Table 7. Extra cost proportion

	function	percentage
Thread 1	Rate_control	8.70
	putRecvQueue	3.57
	BuffDataPoolInit	3.24
	getBufferDataMem	2.53
Thread 2	putBufferDataMem	3.05
	getRecvQueue	1.69
Total		22.78

We will discuss the extra cost in next section.

5 Theoretical analysis and discussion

5.1 System speedup

Consider a system consisting of three parts: initialization, computation and finalization [8]. For single-core CPU, the total time of serial execution is

$$T_{total}(1) = T_{initialization} + T_{compute} + T_{finalization} \quad (5\text{-}1)$$

When we run the system on a multi-core platform, some parts must be paralleled. Suppose that initialization and finalization parts cannot be executed concurrently with other parts, but that computation part could be divided into multiple tasks that can run independently on different cores. Thus the total execution time on a n-core platform could be calculated by the following formula:

$$T_{total}(n) = T_{initialization} + \frac{T_{compute}}{n} + T_{finalization} \quad (5\text{-}2)$$

This formula includes serial computation part and parallel computation part.

For multi-core computation, speedup is used to measure the improvement of parallelization. Speedup is defined as:

$$S(n) = \frac{T_{total}(1)}{T_{total}(n)} \quad (5\text{-}3)$$

Ideally, we want to the speedup to be equal to n, the number of cores. This is sometimes called linear speedup. Unfortunately, this is an ideal that can rarely be achieved because time consumed in system initialization and finalization cannot be improved by adding more cores. Moreover, parallelization brings extra cost like locks and atomic operations for synchronization, limiting the speedup. The terms that cannot be run concurrently are called the serial terms. There running times represent some fraction of the total, called the serial fraction, denoted f.

$$f = \frac{T_{initialization} + T_{finalization}}{T_{total}(1)} \quad (5\text{-}4)$$

The fraction of time spent in the parallelizable part of the system is then (1-f). We can thus rewrite the expression for total computation time with n cores as:

$$T_{total}(n) = fT_{total}(1) + \frac{(1-f)T_{total}(1)}{n} \quad (5\text{-}5)$$

Now, rewriting S(n) in terms of the new expression for $T_{total(n)}$

$$S(n) = \frac{T_{total}(1)}{(f + \frac{1-f}{n})T_{total}(1)} = \frac{1}{f + \frac{1-f}{n}}$$

We obtain the famous Amdahl's law [9]:

$$S(n) = \frac{1}{f + \frac{1-f}{n}} \quad (5\text{-}6)$$

Amdahl's law requires that the proportion of serial part is fixed, which is appropriate for our system.

5.2 System speedup with parallel extra cost

In practice, if considering extra cost brought by parallelization, Amdahl's law is no longer suitable for the actual situation. Usually, the proportion of parallelization extra cost is unchanged. Therefore, we introduce a new fraction denoted f_r for describing system extra cost and get the extra cost time:

$$T_r = f_r T_{total}(n) \quad (5\text{-}7)$$

Considering the extra cost, we rewrite the S(n) (5-3) as $S_r(n)$ to describe speedup :

$$\begin{aligned} S_r(n) &= \frac{T_{total}(1)}{T_{total}(n) + T_r} \\ &= \frac{T_{total}(1)}{T_{total}(n) + f_r T_{total}(n)} \\ &= \frac{T_{total}(1)}{(1+f_r)T_{total}(n)} \\ &= \frac{S(n)}{1+f_r} \\ &= \frac{1}{(f + \frac{1-f}{n})(1+f_r)} \end{aligned}$$

So we get

$$S_r(n) = \frac{1}{(f + \frac{1-f}{n})(1+f_r)} \quad (5\text{-}8)$$

From Table 2, we get the serial fraction $f=f_{Ts1}+f_{Ts5}=0.27\%+1.08\%=1.35\%$. From Table 7, we get the extra cost fraction $f_r=22.78\%$. We use two cores to reassemble HTTP packets, so n=2. We use formula (5-8) to calculate speedup $S_r(n)$, which the parallelization extra cost has been taken into account.

$S_r(n)$=1.6072. It means that the theoretical upper bound on speedup is 60.72%.

5.3 Relationship between speedup and throughput improvement

We know that system speedup is a ratio between serial and parallel running time, from Figure 1 and 2, we know:

$$S(n) = \frac{\sum_{i=1}^{5} T_{si}}{\sum_{j=1}^{3} T_{pj}} = \frac{T_{s1} + T_{s2} + T_{s3} + T_{s4} + T_{s5}}{T_{p1} + T_{p2} + T_{p3}}$$

So we get

$$S(n) = \frac{T_{s1} + T_{s2} + T_{s3} + T_{s4} + T_{s5}}{T_{p1} + T_{p2} + T_{p3}} \quad (5\text{-}9)$$

Among the above formula, $\sum_{i=1}^{5} T_{si}$ represents serial running time, $\sum_{j=1}^{3} T_{pj}$ represents parallel running time. Supposing total traffic of the trace is F, from Figure 1, serial system throughput should be:

$$R_s = \frac{F}{T_{s2} + T_{s3} + T_{s4}} \quad (5\text{-}10)$$

Parallel system throughput should be:

$$R_p = \frac{F}{T_{p2}} \quad (5\text{-}11)$$

So the system throughput improvement can be calculated as follow:

$$V = \frac{R_p}{R_s} = \frac{\frac{F}{T_{p2}}}{\frac{F}{T_{s2} + T_{s3} + T_{s4}}} = \frac{T_{s2} + T_{s3} + T_{s4}}{T_{p2}}$$

So we get

$$V = \frac{T_{s2} + T_{s3} + T_{s4}}{T_{p2}} \quad (5\text{-}12)$$

R_p represents parallel system maximum throughput, and R_s represents serial system maximum throughput

Comparing formula (5-9) and (5-12), we know that when the proportion of T_s1, T_s5, T_p1 and T_p3 are small, we get S(n)≈V. From Table 2 and 4, we get $f_{Ts1}=f_{Tp1}$=0.27%, $f_{Ts5}=f_{Tp3}$=1.08%, so we regard V≈ S(n).

Thus we obtain an important conclusion: When paralleling a serial system, we can use system speedup to estimate throughput improvement if serial proportion which can not be paralleled is relatively low. In this way, we can get a theoretical upper bound on throughput improvement by calculate system speedup.

5.4 Error analysis

From section 5.2, we get the value 60.72%, which is the theoretical upper bound of speedup considered extra parallel cost. While in our experiment, from section 4.2, we know the parallel HTTP packets reassembly system improves 45.37% in throughput. In section 5.3, we introduce a conclusion that we can use speedup to estimate throughput improvement in our system. By analyzing the deference between 60.72% and 45.37%, the reason is that when calculate the theoretical speedup, we cannot get the Linux kernel cost while the experimental result includes the kernel cost. So the test result is less than theoretical value.

6 Conclusions and future works

We briefly summarize our works as follows:

1) Analyzing the CPU cycles consumption of the serial packets reassembly system which based on Libnids.
2) According to the HTTP protocol features, make a specific optimization for HTTP packets reassembly.
3) Declaring a parallelization strategy based on a generic multi-core platform and improving the throughput by 45%.
4) Deriving a new formula to calculate system speedup with parallel extra cost based on the Amdahl's law.
5) Discussing the relationship between system speedup and throughput improvement and proposed a method to estimate system throughput improvement.
6) Using the trace which collected from a backbone link to do our experiment and obtain some valuable results.

In the future, we will continue the research on following aspects: 1) analyzing the influence of different traffic on packets reassembly performance; 2) the scalability of packets reassembly system. It is helpful for improving performance if we can solve these issues.

Acknowledgements

This work was supported by key project in the National Science & Technology Pillar Program (2008BAH37B04) and consolidated project of IBM. Institute of China and Beijing University of Posts and Telecommunications

References

[1] Christopher L. Hayes, Yan Luo. DPICO: A High Speed Deep Packet Inspection Engine Using Compact Finite Automata. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and

[2] communications systems. Dec. 2007
[3] Erich M. Nahum, David J. Yates, PJames F. Kurose, PDon Towsley. Performance Issues in Parallelized Network Protocols. Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation. Nov. 1994
[4] Libnids. http://libnids.sourceforge.net/
[5] Koufopavlou, O.G., Tantawy, A.N., Zitterbart, M. Analysis of TCP/IP for High Performance Parallel Implementations. 17th IEEE Conference on Local Computer Networks, Minneapolis, Minnesota. Sep. 1992
[6] Intel Software Network. http://software.intel.com/en-us/intel-vtune/
[7] Hassan Shojania, Hardware-based performance monitoring with VTune Performance Analyzer under Linux, http://hassan.shojania.com.
[8] Intel white paper: Using Intel VTune Performance Analyzer to Optimize Software on Intel Core i7 Processors
[9] Patterns for Parallel Programming, Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, Addison-Wesley Professional Publishing (2004)
[10] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In AFIPS Conference Proceedings, pages 483–485, 1967.