SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2014; **7**:265–278
Published online 1 March 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.727

RESEARCH ARTICLE

SRC: a multicore NPU-based TCP stream reassembly card for deep packet inspection

Shuhui Chen[1]*, Rongxing Lu[2†] and Xuemin (Sherman) Shen[2†]

[1] College of Computer Science, National University of Defense Technology, Changsha 410073, China
[2] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

ABSTRACT

Stream reassembly is the premise of deep packet inspection, regarded as the core function of network intrusion detection system and network forensic system. As moving packet payload from one block of memory to another is essential for the reason of packet disorder, throughput performance is very vital in stream reassembly design. In this paper, a stream reassembly card (SRC) is designed to improve the stream reassembly throughput performance. The designed SRC adjusts the sequence of packets on the basis of the multicore network processing unit by managing and reassembling streams through an additional level of buffer. Specifically, three optimistic techniques, namely stream table dispatching, no-locking timeout, and multichannel virtual queue, are introduced to further improve the throughput. To address the critical role of memory size in SRC, the relationship between the system throughput and memory size is analyzed. Extensive experiments demonstrate that the proposed SRC achieves more than 3 Gbps in terms of reassembly and submission throughput and triply outperforms the traditional server-based architecture with a lower cost. Copyright © 2013 John Wiley & Sons, Ltd.

KEYWORDS

network security; deep packet inspection; multicore NPU; stream reassembly

***Correspondence**

Shuhui Chen, College of Computer Science, National University of Defense Technology, Changsha 410073, China.
E-mail: shchen@nudt.edu.cn

†Please ensure that you use the most up to date class file, available from the SEC home page at http://www3.interscience.wiley.com/journal/114299116/home

1. INTRODUCTION

Deep packet inspection (DPI) is considered as a crucial technique for network intrusion detection system (NIDS) [1] and network forensic system (NFS) [2], where packet payloads need to be matched against predefined patterns to identify viruses, attacks, criminal evidences, and so on. Generally, the case of a low-speed network, server-based approach can satisfy the throughput requirement. However, with the exponential increase of bandwidth, multi-gigabits per second links are widely applied and placed in the campus network, and gradually, the traditional server-based approach (even for a server with high performance) no longer meets the critical performance requirement. Therefore, plentiful research efforts have been put to improve the overall throughput by achieving efficient content matching.

To minimize the time cost for content matching, different rule matching algorithms using field-programmable gate array [1,3], graphic processing unit [4–7], or ternary content-addressable memory [8] are proposed. However, decreasing only the content-matching time is not sufficient to achieve the desired system performance because stream reassembly, as an important preprocessing plug-in, takes a major part of the whole workload. Experiments conducted in [9] demonstrates that reassembly takes 80% of the load of NIDS when the matching time decreases to 1% of the overall load. In addition, when we utilized a Dell server with two Xeon5405 CPUs, 2G DRAM, and Intel 82599 network interface card (NIC) to take Snort tests, we obtained 2.5 Gbps throughput without dropping any packet when turning off the Stream5 (which is the stream reassembly component in the current 2.9.2 version of Snort) but observed that the throughput will abruptly decrease to 1.2 Gbps once the Stream5 is turned on.

In general, when packets are transmitted through network, they might be dropped because of various reasons, that is, the processing ability of routers, out of order caused by the balance of multipath and others. Therefore, to trace the information stream between the two ends, NIDS and NFS exert to fetch each packet belonging to a stream (also

known as session or connection) and reconstruct what have been sent from the transport layer of the end systems.

Nevertheless, many attacks cannot be detected if the stream reassembly is not conducted completely, as either signatures may cross the packets or stick attack could not be identified [10,11]. Although many researches focusing on flow measure and analysis have appeared in the past years, most of them only investigate the stream attributes but not stream reassembly [12,13]. In addition, although there are some open source softwares fulfilling the stream reassembly task, for example, Stream5 (Stream4) and Libnids, there still exists a big gap between their throughput and the network link bandwidth.

Memory access time is often considered as one of the major throughput constrain of network security device. Generally, the immanent fluidness of network packets results in the very low hit ratio of cache. Because the number of memory access is predefined and cannot be changed subsequently, the crux of enhancing system performance is to improve the efficiency of each memory access in terms of access time.

Traditional network security devices such as NIDS and NFS adopt high-performance CPU platforms such as x86 and MIPS. These typical CPUs focus on how to improve the calculation performance, so they make as perfect or effective as possible on cache coherence, branch prediction, out-of-order execution, multicore parallelism, and so on. Their improvements on memory access concentrate on how to increase the hit rate of the cache; therefore, stream reassembly using legacy CPU could not achieve very high performance.

Currently, advanced progresses have been made in the network electron component area. For example, Raza Microelectronics has developed XLR, XLS and XLP network processing units (NPUs), whereas Cavium has launched series of OCTEON NPUs. The emergence of these multicore NPUs can largely improve the processing ability of the network devices and network security devices. On the other hand, multicore NPUs have many hardware improvements on multithread operations, which decrease the thread switching overhead, hide the memory access latencies, and employ the memory access cycle efficiently. As a result, their memory access performance can be improved remarkably through these techniques. However, there are several issues that need to be tackled regarding using NPU to implement stream reassembly; for example, how to distribute packets to different NPU cores, how to accelerate stream timeout processing, and how much memory should be used for the NPU.

To address the aforementioned issues, in this paper, we specially exploit the stream reassembly issue and study on how the new multicore NPU can be used to improve the stream reassembly performance. We present a new stream reassembly card (SRC) design, which enables to manage and reassemble streams through adding a level of buffer to adjust the sequence of packets by using the multicore NPU. Specifically, the contributions of this paper are threefold.

- First, a multicore NPU-based stream reassembly architecture is introduced. To the best of our knowledge, this is the first work on employing multicore NPU-based stream reassembly technology specifically for NIDS and NFS.
- Second, several improvements have been introduced to increase the throughput of the stream reassembly architecture. It has been found that the implementation of the stream reassembly architecture may be restricted by the DRAM size, so the relationship among memory requirement, timeout limit, and link bandwidth is systematically researched.
- Finally, an SRC is implemented and evaluated to demonstrate that the SRC can achieve more than 3 Gbps in terms of capturing and processing, triply outperforming over the traditional server-based architecture.

The remainder of this paper is organized as follows. The related work is provided in Section 2, and the motivations on selection of multicore NPU are presented in Section 3. Then, Section 4 depicts the system architecture and framework. The three improvements are introduced in Section 5. The relationship among memory size, timeout policy, and link bandwidth are analyzed in Section 6, followed by the experimentation and result analysis in Section 7. Finally, we conclude our work in Section 8.

2. RELATED WORK

There are two open source programs: Libnids [14] and Tcpflow [15] that fulfill transmission control protocol (TCP) stream reassembly. Libnids is an application programming interface (API) component of NIDS. Libnids offers Internet protocol (IP) defragmentation, TCP stream reassembly, and TCP port scan detection. It can obtain the data carried by the TCP streams with reliability and is widely used in the NIDS and forensic systems. On the other hand, Tcpflow is a program that captures data transmitted as part of TCP connections (flows) and stores the data in two files that are convenient for protocol analysis and debugging. Tcpflow understands sequence numbers and correctly reconstructs data streams regardless of retransmissions [16] or out-of-order delivery. But, it cannot process IP fragments properly, and its performance is not also suitable for network links with more than 1 Gbps bandwidth.

Previous researches related to TCP streams are often focus on network measurements. For example, in [13], the authors have used data recorded from two different operational networks and study the flows in size, duration, rate, and burst, to examine how they are correlated. In [17], the authors concerned the problem of counting the distinct flows on a high-speed network link. They proposed a new timestamp-vector algorithm that retains the fast estimation and small memory requirement for the bitmap-based algorithms, while

reducing the possibility of underestimating the number of active flows.

In [18], a TCP reassembly model and a stream verification methodology have been introduced for deriving and computing reassembly errors. In [19], it has presented an algorithm to solve the problem of TCP stream reassembling and matching performance problem for NFS and NIDS. Instead of caching the total fragments, their method stores each fragment with a two-element tuple that is constant size data structure; thus, the memory requirement involved in caching fragments is largely reduced.

Dharmapurikar *et al*. [20] have introduced a hardware-based reassembly system to solve both the efficiency and robust performance problems in the face of the adversaries to subvert it. They characterized the behavior of out-of-sequence packets seen in benign TCP traffic and designed a system that addresses the most commonly observed packet-reordering cases in which connections have at most a single sequence hole in only one direction of the stream.

3. THE NECESSITY OF MULTICORE NETWORK PROCESSING UNIT

The NIDS or NFS takes advantage of a "promiscuous mode" component or "sniffer", to obtain copies of packets directly from the network media, regardless of their destinations. Raw packets captured by the NIDS or NFS are confused and disordered messes, whereas DPI needs these packets to be fabricated as an integrated block according to their affiliated TCP stream before they are sent to the matching engine.

Figure 1 shows an example that a stream composed of six packets is obtained in a monitor point where packets 2 and 3 are in disorder and packet 4 is repeated. The stream reassembly process needs to swap disordered packets 2 and 3 and delete the second unwanted repeated packet 4. This process incurs three times packet movement: packet 2 moving ahead, packet 3 moving backwards, and packet 5 moving ahead as well. This is just an example of a single stream. But in a real network environment, one backbone link may contain a large number of streams. In other words, there may be a large number of packet movements in the reassembling process. In addition, modern servers always use dynamic random-access memory (DRAM), e.g., DDR2 or DDR3 as their main memory; one memory access may take a number of cycles to obtain a result as DRAM has a relatively long startup time.

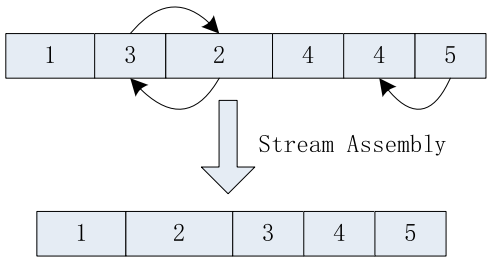

Figure 1. An example of stream reassembly.

However, using multicore NPU can improve the throughput of the system for the following reasons:

(1) There are many hardware threads in one core and many cores in one NPU, which makes the total threads in an NPU be more than a dozen. Threads of this kind are hardware based instead of software based, so the switching overhead is very low. The large number of hardware contexts enables software to more effectively leverage the inherent parallelism exhibited by packet-assembling applications. When one hardware thread is waiting for the result of the memory accessing, another hardware thread could switch in and make another memory accessing request without much overhead. If many threads can take advantage of the pipeline mechanism, the latency of the DRAM will be hidden, and the effective bandwidths of the DRAM access would increase.

(2) A multicore NPU often has a low electric power consumption, so it is easy to be manufactured as a card. By utilizing a card, a server can also conduct the task of DPI, attack warning, and so on. When an NPU-based card is used, an extra buffer is introduced to the flow processing, so the packets can be sorted as they are being transferred from the memory of the card to the memory of the host (server). It is a form of trading space for performance. In this way, when the packets have been received into the memory of the card, they are stored in the memory as per their reaching order, but their order is maintained by the software running on the NPU.

(3) The architecture of NPU often has favorable input/output (I/O) features, and the packets could be imported from the interface to the memory with high throughput. Moreover, its dispatching mechanism (distributing packets to different threads or different cores) is perfectly designed, so that dispatching component could pipeline with the corresponding processing threads (or cores). As the dispatching component generally dispatches packets according to the selected bits from the packet head, stream reserved would not be a considerable problem. As many researches [22,23] focus on how to accelerate the packets capturing performance, an approach to jointly consider packet capture and stream reassembly is cost effective.

(4) An NPU often has a well-designed message-passing mechanism among different threads, which employs cross-bar structure or fast shared static random-access memory (SRAM) as its transferring medium, and makes the cooperation and synchronization between threads facile.

4. SYSTEM ARCHITECTURE

In our proposed scheme, NPU is used as a cooperated stream managing component, which captures raw packets and submits ordered as well as nonrepeated packets to

the host to conduct further inspection. Instead of moving packets in its memory, NPU just keeps the order of the packets and submits them to the CPU according to the order, which wipes off the cost of packet moving, and the performance of the system is mostly dependant on the consecutive packet copying from NPU to CPU. As depicted in Figure 2, the packets arrival sequence is like the example of Figure 1, while the NPU removes the redundant packets and keeps their order in its memory. In the end, the NPU submits the TCP control block (TCB) and the packets according to the original order at the appropriate time.

The key concepts relative to the stream management will be discussed in the following section.

4.1. Stream in transmission control protocol transferring level

The TCP specification defines several "states" that any given connection can be in. The states observable by an NIDS and NFS (those involving the actual exchange of data between two hosts) are not the same as TCP connection states. Only two states ("CLOSED" and "ESTABLISHED") would be taken into account.

Transmission control protocol is a reliable, sequenced stream protocol that employs IP addresses and ports to determine whether a packet belongs within a stream. To reassemble the information flowing through a TCP connection, NIDS and NFS must figure out what sequence numbers are being used. TCB is a data structure used by NIDS and NFS to describe the stream that is in the "ESTABLISHED" state. NIDS or NFS should have a mechanism by which TCBs can be created for newly detected connections and destroyed for connections that are no longer alive.

In the following discussion, we focus on three different critical actions that the NIDS may perform during the processing for a connection. They are TCB creation (the action that an NIDS decides to instantiate a new TCB for a detected connection), packet reordering (the process an NIDS uses to reconstruct a stream associated with an open TCB), and TCB termination (the action that the NIDS decides to close a TCB). Simple discussions of these actions are as follows:

(1) TCB creation

The NIDS has different approaches to employ for when to create the TCBs. It may attempt to determine the sequence numbers being used simply by looking at the sequence numbers appearing in TCP data packets (referred as synchronization on data), or it may rely entirely on the three-way handshake (3WH). In our proposed method, we use synchronization on data as the signal of TCB creation for the purpose of simplicity.

(2) Packet reordering

For a packet that belongs to an existing stream is received, NIDS needs to decide its position in the designated stream. If the NIDS does not use sequence numbers (simply inserts data into the "reassembled" stream in the order it is received), it will not work properly because an attacker can blind such a system simply by adjusting the order of the sent packets, whereas the actual data received by the application level of the destination will not be the same as the data obtained by the NIDS.

(3) TCB termination

The TCB termination is crucial because the maintenance of connections in NPU is resource consuming. When a connection is terminated, it does not need to assign resources to it as well. There are two kinds of approaches that respectively use RST (a flag in TCP head used to reset a connection) or FIN (a flag in TCP head used to normally close a connection) to terminate a connection. Note that a connection can be alive infinitely without any data exchanging. Thus, it is inadequate to manage the per-connection resource problem because TCP connections do not implicitly time out. The lack of a method to determine if a connection is idle or closed forces the NIDS and NFS to terminate the connection and delete its TCB when no packet appears in the connection for a long time. The problem with TCB termination is that an NIDS can be tricked into closing a connection that is still active; thereby, it forces the system to lose state and increase the probability of detection evasion. On the other hand, a NIDS that fails to terminate a TCB for a really closed connection is also vulnerable because the memory will be exhausted rapidly. In our proposed method, both FIN and RST messages are utilized as the basic judgment to indicate the termination of the connections. In addition, we also rely on timeouts as an

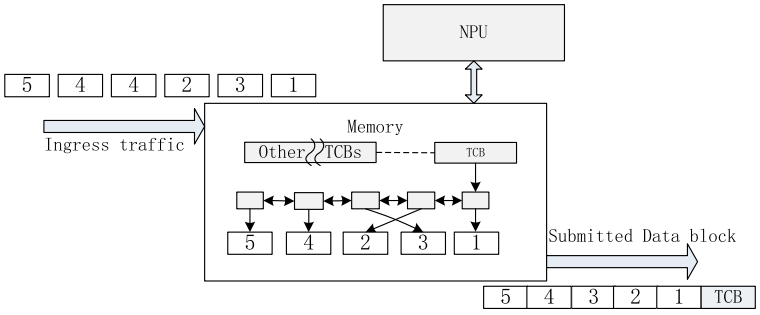

Figure 2. Stream reassembly using network processing unit.

auxiliary instrument. When a connection has been terminated, its remainder data on both directions will be submitted to the host, and then, its corresponding TCB will be deleted. Each TCP connection can be expressed as a four-element tuple including source IP, source port, destination IP, and destination port. Once a packet is captured, its corresponding stream needs to be localized, and the TCB data structure needs to be updated. The minimum information of a TCB should be composed of the aforementioned four-element tuple, client's expected sequence number, server's expected sequence number, pointer to the next TCB for resolving hash collision, the time that the last packet was received, and the pointer point to the buffered packets. Generally, TCB is attached to a hash table indexed by hash algorithm by using some bits from the four-element tuple as parameters. Collisions lead to several TCBs being attached to one table entry.

4.2. Frameworks of stream reassembly card

The framework of SRC is depicted in Figure 3. In SRC, packets are captured from interface into memory; for maintaining the TCP connection data, a hash table known as stream table is used. Hash is calculated on the <SrcIP, SrcPort, DesIP, DesPort>. When the packets enter the memory, their locations are stored in the packet descriptions. Besides the pointers that point to packets, packet descriptions also contain the packet length and the fields used to dispatch the packets to the threads.

Threads running on the NPU process a received packet and then wait for another packet circularly. Once the data need to be submitted, every thread itself is responsible for the task of submitting the packets from the memory of the NPU to the memory of the host. Both the softwares running on the NPU and CPU share a little memory space in the double data rate synchronous dynamic random-access memory (DDR SDRAM, abbreviated as DDR) of the NPU for message communication, which is utilized by the NPU to gain the address of the direct memory access (DMA), the timeout limit of the host setting, the BlockSize, and the consuming states. CPU can also employ the memory space to obtain the running states of the NPU. As the packets are DMAed to the host memory, the transferring is conducted one packet after another, which is because the packets are not stored consecutively in the memory of the NPU while we need them to be consecutive when they reach the memory of the CPU.

Software running on the NPU mainly executes the three actions mentioned in Section 4.1: TCB creation, packet reordering, and TCB termination. When a packet reaches one core, the corresponding thread looks up in the stream tables to determine if a corresponding TCB exists. If not, the corresponding TCB is created, and the packet is appended to the TCB. Otherwise, the packet is appended to an existent TCB, and its link position is determined; meanwhile, a judgment is made on whether the total packet size of the stream is equal or larger than BlockSize (submitting block size). If the answer is positive, all the packets are submitted in the light of their sequence to the host. In fact, there are three situations that trigger data to be submitted to the software running on the host:

(1) When the size of the received packets attains or exceeds the submitting block size (BlockSize is called), the data block that is made up of these packets has to be submitted. We need to submit the data when the buffer possessive for one stream is too large, as the memory is limited. The larger the BlockSize is, the larger the whole DRAM space will be. But if we adjust the BlockSize to be very small, the data that the host obtains will be small as well, which may degrade the performance of the NIDS and NFS. Therefore, the selection of the Blocksize causes a tradeoff between memory space and overall performance.
(2) In the situation of a packet with a finishing tag (RST or FIN is set in the head of the TCP packet) has been received, it indicates that the corresponding stream will be terminated by the server or the client. In this case, the data block also needs to be submitted to the host, and the corresponding TCB needs to be deleted.
(3) In the situation of no packet for a certain stream has been received for a very long time (referred as

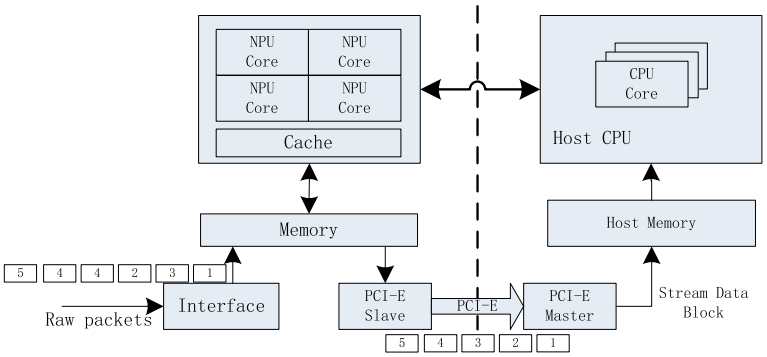

Figure 3. Frameworks of stream reassembly card.

stream timeout), it also indicates that data submission is obligatory. This is because either the communication on the stream may be terminated accidentally, or the stream is idle. The TCBs of such streams and their corresponding packets cannot be maintained forever, because of the memory limitation. Obviously, the memory space is likely dependent on the stream timeout. The larger the stream timeout is set to, the larger the memory space will be required. On the other side, the shorter the stream timeout is, the less accuracy of the stream reassembly will be.

The total connection records are maintained in a hash table called stream table for efficient access. Note that the hash needs to be independent of the permutation of source and destination pairs, which could be achieved by comparing the source IP address with the destination IP address, and the less one is always made to be the first parameter, or some hash algorithms that are not sensitive to the sequence of parameters are used. By using the hash values as the indexes to the stream table, the corresponding connection can be located. Hash collisions can be resolved by chaining the colliding TCBs in a linked list.

A data block submitted to the host consists of a TCB and several subblocks; each of which represents a data transmission in the TCP/IP transport level from one peer to another. In addition, adjoining subblocks are from two directions—one from client to server and the other from server to client. Some subblocks may consist of only one packets, whereas other subblocks may have several packets, which are determined by the application level protocol, and the data amount needs to be transferred. For example, according to the pop3 protocol, before the mail body is transferred, there will be several interactions ahead to make the server authenticate the client and the client to check if it has mails on the server. After the stream reassembly, the data block will likely be in the form shown in Figure 4. Except subblock 7, all the other subblocks consist of only one packet, as they are very short and need not to be divided into multiple packets. Subblock 7 consists of an "+OK" replay and the mail body. So even if the mail body is not very large, subblock 7 should contain at least two packets.

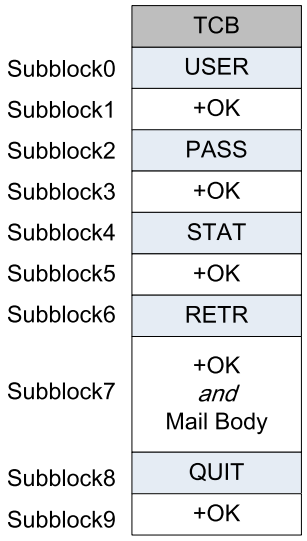

Figure 4. Data structure example after reassembly.

The data submission procedure handled by the packet-processing threads needs to work cooperatively with the programs running on the host CPU. A consecutive memory chunk needs to be allocated by the CPU to store the packets uploaded, and for the convenience of the packet organization, the chunk is divided into fix-sized buffers that are organized as a ring. Consumer software (NIDS or NFS) running on the host continually processes the data block received. When the speed of the consumer software is higher than that of the threads running on the NPU, ringed buffers will be full; finally, the NPU cannot upload data and can only check if there is any vacant buffer. Once a vacant buffer emerges, submission continues. In the situation of the ringed buffers are full, packets arrive continually, but there is no thread that can process, as all the threads are checking the state of the memory, packet dropping cannot be avoided. So, the processing speed of the consumer software running on host must match the data submission speed.

Different security applications running on the host have different operations on the data block submitted. For example, if we want to run applications on the host to take evidence for the forensics by recovering the e-mail body, it needs to scan the data on the stream level; after the subblocks of "USER", "PASS", "LIST", and "RETR", the subblock from pop3 server to pop3 client is the content of the mail.

4.3. The procedures of stream reassembly

The two significant data structures in stream reassembly are stream table and TCB. Stream table consists of many entries; each of which points to a list of TCBs with the same hash value.

Two types of threads are used to fulfill the stream reassembly: the packet-processing thread and the timeout thread. The packet-processing threads are responsible for packet receiving, stream reconstruction, and data submission; moreover, stream reconstruction is divided into TCB creation, packet reordering, and TCB termination, as depicted in Algorithm 4.3. The timeout thread is a simple cycle procedure; it accesses TCBs one by one ceaselessly, comparing the current time with the time of the last coming packet in every stream. If the gap between the two times is larger than the appointed value, timeout thread deems that the corresponding stream may be idle

DOI: 10.1002/sec

or closed, so it submits the remaining data and deletes the TCB to leave space to other streams.

Algorithm 1. Reconstruction (pkt).

Input: pkt : new arrived packet,
Output: $stream$: packet corresponding stream
$stream \leftarrow SearchStream(pkt)$;
if *(stream=NULL)* **then**
  $stream \leftarrow TCB\_Creation(packet)$;
end
if *(pkt.Fin or pkt.Rst)* **then**
  $Submit(stream)$;
  $TCB\_Termination(stream)$;
  $return$;
end
$ReorderPacket(pkt, stream)$;
if $(stream.size >= BlockSize)$ **then**
  $Submit(stream)$;
end

The main function of ReorderPacket is to sort the one-stream-affiliated packets according to their TCP sequence number and drop the repeated packets that have the same sequence number. Instead of being processed after a batch of packets belonging to a stream have been received, the packets are maintained upon being received. The reasons for ordering upon reception are as follows: (i) the batch processing method could lead to computing burst, which is detrimental to the smooth process, and (ii) out-of-order packets are actually rare because most of the arrived packets are ordinal and consecutive. As a result, processing packets one by one saves more computational resource.

As the data are submitted to the host, all the packets must be ensured to be ordinal and consecutive. We use ordering to express the sequence of the packet and continuity to denote whether there is any packet that should arrive but has not yet. When the packets arrive, their order can be ensured by sorting the sequence number and modifying the pointers of the list used for attaching packets, but the continuity cannot be ensured, because of the disordered arrival phenomena. To determine whether the data can be submitted, a counter discontinuity number (DCN) is used to identify whether the received packets are consecutive or not. DCN is the counter of gaps between adjoining packets for a stream.

The larger the DCN is, the less the degree of continuity is. Take the following scenario as an example: from one direction of a stream, if packets 1–5 and 7 have been received (the numbers here are the order numbers of the packets been sent out, instead of the sequence numbers of the TCP transport level), the DCN of the stream is 1, because there is a gap between packets 5 and 7. If only packets 1–4 and 7 have been received, the DCN of the stream is still 1, as even though there seems to be two packets missing between packets 4 and 7, it is impossible to determine how many packets can fill in the gap at the time of reception of packet 7, and the only known fact is that there is a sequence number gap between packets 4 and 7 from the TCP perspective.

All the packets are linked up so that packets with smaller sequence number are in the front of the linked list and the packets with bigger sequence number are in the tail of the linked list. Bidirectional links for the packets are needed, because the packets need to be submitted from NPU to CPU according to the sequence number. Yet when a packet arrives, locating the inserting point from the reverse direction of the corresponding stream may gain better performance, because the gaps exist scarcely; and even when a gap exists, it might be filled quickly.

Algorithm 2. ReorderPacket (pkt,stream).

Input: pkt : new arrived packet
Input: $stream$: packet's corresponding stream
Output: $stream$: stream after packet being inserted
$position \leftarrow locate(stream, packet)$;
if $exist(packet,stream)$ **then**
  drop(pkt);
end
$Insert(pkt, stream)$;
if *!continuity(pkt,pkt.prev) and*
  !continuity(pkt,pkt.succ) **then**
  $DCN++$;
end
if *continuity(pkt,pkt.prev) and*
  continuity (pkt,pkt.succ) **then**
  $DCN--$;
end

The determination of two packets belonging to one stream are consecutive or not (continuity function in Algorithm 4.3) is only needed to figure out the addition of sequence number of the former packet and its length. If the result is equal to the sequence number of the latter packet, they are consecutive; if the former packet's sequence number plus its packet's length is less than the current packet's sequence number, they are inconsecutive; and if the former is bigger than the latter, an error takes place, and a warn will be sent to the software running on the host.

From Algorithm 4.3, the assembly process employs "first reassembly policy" [11]. Theoretically, different assembly policies should be adopted in terms of different destination servers, but how to avoid attack is another challenge, which is beyond the scope of this paper.

When a stream ends, times out, or its size exceeds the BlockSize, the packets belonging to the corresponding stream must be uploaded to the CPU. Under the circumstances of a stream ending or timing out, DCN should be zero if all is OK. If it is not zero (meaning some packets have not been received), there is nothing that can be carried out by the reassembly component. Instead, if it

is under the circumstances in which the size of the stream achieves the BlockSize, then the DCN is not zero, and as a result, the reassembly procedure needs to find the gap that causes the DCN to be not zero. We inspect along the linked packets of the stream, if the position of the lost packets are far ahead of that of the last packet (e.g., 8 is an experiential value. It is because the most frequently used default TCP window size is 8 K, which contains less than eight packets at most times); the searching process will stop, and the packets will be submitted, assuming that the lost packets will no longer arrive. On the other hand, if the gap is among the last eight packets, the set of consecutive packets will be submitted, whereas the remaining inconsecutive packets will stay in the list. To sum up, we try to upload only the packets that are consecutive to the host. Packets stored in the SRC are not in order and consecutive, and we maintain the order by the pointers. However, when they are being uploaded, their order and consecution are tried to be ensured. As a result, when the packets arrive to the memory of the host, they are ordered and consecutive if there is no error.

5. IMPROVEMENT

Through the aforementioned analysis, there are basically two issues needed to be tackled. The first is that every arrived packet needs to be used to search the stream table and be hanged to its appropriate position, but for a gigabit Ethernet link, it is only a few microseconds available for a packet to be processed, so it is necessary to accelerate the processing speed. On the other hand, when it is time for the packets to be submitted to the host, the capture ability of the host should be fully utilized. Three techniques are adopted to improve the throughput of the two issues.

5.1. Stream table dispatching technique

There are generally two techniques implemented to organize the TCBs in the stream table: shared stream and separated stream tables. For the shared stream table, all the threads share only one whole stream table, so all the threads need to access the stream table in the global memory. As a result, a lock must be added to the corresponding item of the stream table when one thread is processing the packet. It is no doubt that the accessing competition decreases the performance. On the other hand for the separated stream table, every thread uses its own stream table, but we must use more memory than the shared stream table to hold several tables to make the TCB list within one hash entry not too long.

For both high memory utilization and high performance are required, a new technique is hereby presented. In our approach, a unified hash method for packet dispatching to the threads and obtaining the stream table index is applied, making all the streams grouped under the same stream table index to be always dispatched to the same thread. Therefore, the items of the stream table need not to implement locks, as all the streams hashed to a pointed item will be processed by one specific thread. In addition, if the stream table items assigned to every thread are consecutive and the size assigned to every thread is aligned to the cache blocks, cache hit ratio of the stream table will be high, and overall performance can be improved. The stream table structure is illustrated in Figure 5.

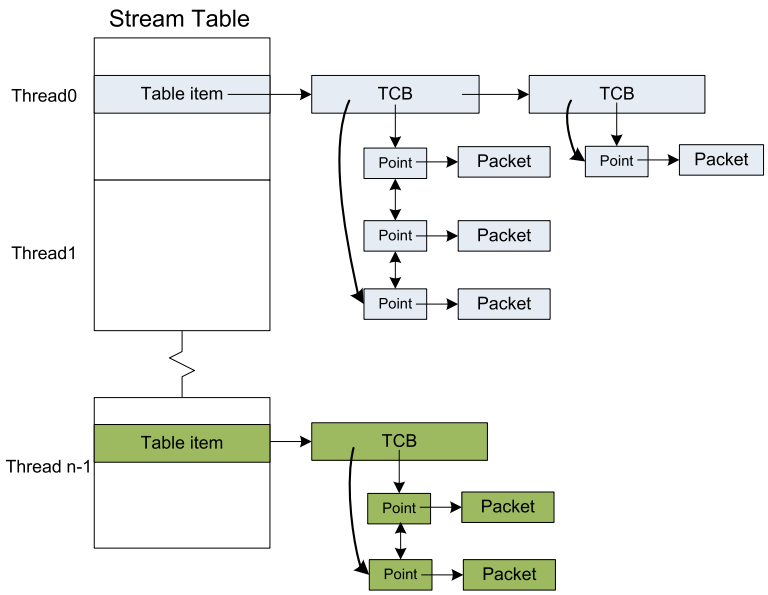

Figure 5. Improved stream table.

DOI: 10.1002/sec

5.2. No-locking timeout

Because a large number of parallel TCP streams are present in the network, the states of a large number of TCBs attached to the stream table must be carefully maintained. To release the memory space of the streams that are not active, three submission mechanisms have been previously introduced: stream timeout, stream termination, and when the size of buffered packet achieves a specified BlockSize.

As the timed-out packets have to be uploaded, a separated timeout thread is used to confirm whether there is any stream is timed out. The timeout thread circularly obtains every item in the stream table and then checks every TCB in the link to determine if there is a timed-out stream. If a timeout occurs, submission of the packets and deletion of the TCB are conducted. The stream table and the TCBs become the critical resources; thus, locks are required because the packet-processing threads need to process the TCBs and the timeout thread also needs to operate the same TCBs.

Lock operation should be avoided according to the experiences on the network device and network security devices, as there is not so much time available to process a packet. For example, only 300 ms are available to process a packet for a gigabit Ethernet link [24]. For the multicored NPUs of Raza Microelectronics Inc. (RMI) and OCTEON, they both have a fast messaging mechanism to implement the synchronization and information transmission among different threads. Therefore, the messaging mechanism can be used to avoid the locks by enabling the timeout thread to send a message to the packet-processing thread and then the packet-processing thread submitting the packets and deleting the TCBs.

5.3. Multichannel virtual queue

The performance of the packet capture is crucial to the performance of the overall traffic analysis system [25,26]; similarly, data block submission is decisive to the performance of stream reassembly. It is obvious that multicore computers are the current dominant trend in computers; thus, how to avoid data copying and enable the data blocks to be distributed to different cores in the host can bring distinct improvements to the overall performance.

Multichannel virtual queue is introduced in [27], which has implemented a novel multicore aware packet capture kernel module that enables monitoring applications to scale with the number of cores. Their technique is to distribute the received packets to several receiving queue and then dispatch them to different CPU cores to process. The objective of multichannel virtual queue is to achieve the fast data capture and load balance, but the generic NICs do not support multi-interruptions, and only the advanced NICs such as Intel 82575 /82576 that support receive-side scaling (RSS) [21] have such a feature. The generic NICs do not have the ability of hash computing, so hash computing must be completed by CPU if multichannel virtual queue is needed to be implemented; subsequently, CPU is required to access the packet head, and as the packet is always new to the CPU, cache will likely to be not hit, which leads to poor performance.

Intel NIC adopts static hash to map the traffic to 128 buckets; therefore, the users may specify which queue should the packets in the 128 buckets be dispatched to. The default method is round-robin (arithmetical complement is utilized, so if three queues are present, the remainder that the bucket number divide 3 is the queue number). The detailed information can be found in [21]. Obtaining the idea from multiple receiving queues and RSS, our approach in SRC is even better than RSS. This is because the hash computing used by Intel NIC is based on the source IP address and the destination IP address and exchanging the source IP address and the destination IP address will cause different value to be produced.

The hash technique that packets belong to the same stream are hashed to the same processing unit is called stream based. However, Deri [23] exploited the feature of the Intel NIC, but he overtook that the packets on different directions for one stream will be dispatched to different core (matching engineer) as the adopted hash algorithm is sensitive to the sequence of the input parameters (which are the source IP address and destination IP address); thus, many of the attacking warnings will not be reported as packets from different direction are sent to different core. We have amended this problem by enabling the NIC's driver to recalculate the hash value if the source address is bigger than destination address, and if the source address is smaller than or equal to the destination address, hardware-distributing mechanism is kept. Although this method is stream based, the performance of the method is only 60% of the method in [23] on the basis of our experiments. Furthermore, Intel NIC only has four fixed queues, but the latest CPU can support eight cores, and the packets in four queues cannot be dispatched to eight cores.

Our approach can tackle the issue as our hash is not sensitive to the sequence of the input parameters and the hash result is scalable to the thread number on the CPU. The host creates several queues organized as ringed buffers and tells the program running on the embedded multicore NPU the number of queues, ring descriptors, length, and head and tail pointers of the ring through the shared memory. NPU then calculates the corresponded queue that the data block of each stream will be dispatched according to the information given by the CPU.

6. MEMORY SIZE REQUIREMENT ANALYSIS

The SRC needs to buffer a large number of packets, maintain TCP connection records for thousands of simultaneous connections, and access these TCBs at high speeds. For such a high-speed and high-density storage, commodity synchronous DRAM chip is the only appropriate choice.

Yet as a general knowledge, a card must satisfy a stated size requirement (e.g., a peripheral component interconnect (PCI) card must not exceed the size of 31.2 cm × 12 cm). When a stream reassembly component is implemented as a card, the size of card must be reduced. In addition, power supply of SRC is restricted by the PCI express (PCIe) bus. To the best of our current knowledge, maximum capability of one DRAM chip is 4 Gb; if 4 GB storage memory is used without parity check, eight chips are needed. And, boot loader, operation system, stack, and memory shared with host are all needed to allocate memory, so the actual memory used to storage stream table, TCBs, and packets will be even less. In this section, the relationship between memory requirement and link throughput, stream timeout, is analyzed by experiments to identify if it is feasible by the current technique of memory density.

The estimation of the memory requirement for the reassembly is complex, for there are so many factors that affect the memory size. Different networks, different timeout mechanisms, and different data block sizes submitted all influence the memory size. In the campus network of a university, because the video and audio applications are used more than that of the other network environments and the stream life time will also be longer, data submission is likely to be more frequent because the block is more easy to be full, so the memory size needed will be larger. Even in one measure point, measures in different time may obtain different result. The NIDS and NFS softwares running on the host always hope for a big BlockSize (submitting block size), as the larger the BlockSize is, the larger the data amount writing to the disk is for one time. But, an extra large BlockSize will cause the impossible sustained memory requirement that cannot be supported under the current techniques.

To obtain the memory requirement macroscopically, we used the traffic captured in the core switch of our campus network; 1 GbE port of the switch is configured to mirror the traffic that is sent to and received from the boundary router. One Dell server is used to connect to the mirroring port to obtain the bidirectional traffic leaving from and arriving to our campus network.

Our Dell server can only achieve the capturing performance of 300 Mbps, but the actual bidirectional traffic is about 1260 Mbps. To further decrease the data capture quantity and protect the privacy, only the head 64 B are storied. We have captured for 396 s and obtained 13.815 GB dump file. With the captured file, a program is used to analyze the stream number per Mbps throughput. The total packets captured are 177,140,885, whereas the TCP packets among them are 11,221,590. Different numbers of streams for different timeouts are depicted in Table I. The memory requirements per Mbps traffic in different timeout values and different BlockSizes are depicted in Table II. Only the memory size of the packet buffer is considered because TCB space is relatively neglectful to the packet space.

We have found that although there are 748.02 streams per Mbps in the traffic when the timeout is set to 64 s, only 8.3% of streams are out of order, and the average packet ratio out of sequence is 23% among the out-of-order streams, which may be a result of short flows losing most of their packets and reporting a high loss rate.

From the aforementioned experiments, if we want to burden 2 Gbps throughput (bidirectional traffic of a gigabit Ethernet) with the timeout of general 64 s and the BlockSize of 16 K, we need $1.1231 \times 2000 = 2.2462$ GB (the typical memory requirement per 1 Mbps traffic is shown in bold in Table II). As the 4 Gb SDRAM is available, the multicore NPU coprocessing solution is feasible by using eight of such chips under the current technology condition.

7. IMPLEMENTATION AND PERFORMANCE EVALUATION

7.1. Implementation

An SRC is developed using XLS416 produced by RMI. The RMI XLS416 is a multicore, multithread MIPS64 processor with a rich set of integrated I/O. XLS416 has four cores, and every core has four threads, so the total thread number is 16. In our implementation, one thread (referred as timeout thread) is used to manage timeout, and all the other threads (referred as packet-processing threads) execute the same routine, whose job is receiving, assembling, and submitting packets. When the timeout thread finds that any stream has timed out, it will send a message to the corresponding packet-processing thread to notify which stream has timed out. On the other hand, every packet-processing thread circularly checks if there is any timeout message after processing one packet.

XLS416 has three frequency models: 800 M, 1.0 G, and 1.2 G; for the best of the performance, we used the XLS with 1.2 GHz. XLS416 integrates 8 GbE interfaces or two 10 GbE interfaces. To further save the printed circuit board (PCB) size and consider that the 10 GbE may be the mainstream backbone link of the enterprise network, two 10 GbE interfaces are adopted to SRC. Our SRC is equipped with 4 GB of 533 MHz DRAM and one PCIe 1.1×4 bus connecting to the host. The interface chip is VSC8486-11 that connects the fiber module and the XLS through XAUI. Dual-inline memory module (DIMM) chips are used instead of DIMM strips, for the former occupies less printed circuit board space and the stability is better than the latter. The total card's power consumption is under 26 W.

Table I. The stream number per Mbps throughput according to different timeout.

Timeout (s)	1	2	4	8	16	32	64	128	256
Stream number per Mbps throughput	82.38	133.76	184.52	246.52	360.74	581.34	748.02	839.33	890.50

Table II. The memory required per 1 Mbps traffic.

	1 K	2 K	4 K	8 K	16 K	32 K	64 K	128 K	256 K
1 s	0.0128	0.0232	0.0338	0.0509	0.0698	0.0919	0.1292	0.1296	0.1296
2 s	0.0180	0.0320	0.0465	0.0699	0.0968	0.1293	0.1876	0.1881	0.1881
4 s	0.0279	0.0493	0.0714	0.1062	0.1478	0.2026	0.3021	0.3026	0.3026
8 s	0.0465	0.0807	0.1174	0.1727	0.2411	0.3219	0.5113	0.5120	0.5120
16 s	0.0753	0.1336	0.1967	0.2889	0.3985	0.5336	0.8822	0.8835	0.8835
32 s	0.1218	0.2174	0.3209	0.4690	0.6507	0.8762	1.5031	1.5056	1.5056
64 s	0.2133	0.3777	0.5560	0.8125	1.1231	1.5347	2.6942	2.6981	2.6981
128 s	0.3941	0.6981	1.0266	1.5036	2.0889	2.8569	4.9828	4.9908	4.9908
256 s	0.7507	1.3315	1.9598	2.8739	4.0122	5.4226	9.5093	9.5251	9.5261

Unit: megabytes.

At the software level, there are a stream reassembly program running on the SRC and a driver running on the host. Programs running on the SRC are merged to one image with the RMI operating system and are burned into the Flash. We provide an SRC_API extending from Libnids [14]. In addition to the features of the Libnids, our SRC_API can be used to obtain the statistics and notify software running on the SRC the number of the analysis threads running on the host, timeout limit of the stream, and the BlockSize. To achieve this goal, 2M space is used to share information between the CPU and NPU, and every thread running on the CPU is assigned 64 MB memory space for packet capture.

We also develop an application level main program based on SRC_API to test the system performance, and this program just obtains the stream data blocks and then drops them, instead of conducting any analysis of the stream.

7.2. Evaluation

The test topology is depicted in Figure 6. Dell PowerEdge R710 server with a Xeon 2.13 GHz E5606 CPU and total 16 GB ECC DDR3 (4×4 GB) are used to host the SRC. R710 server has a PCIe $\times 8$ bus that can be used to hold the joint of the SRC. Red Hat Enterprise Server 64 b with a 2.6.18-92.el5 kernel is used as the operation system. An IXIA XM2 with an Xcellon-Ultra NP 10 GbE load module is used to construct the evaluation environment. Both the packet and application level tests can be used by IXIA XM2. The packet test was conducted by

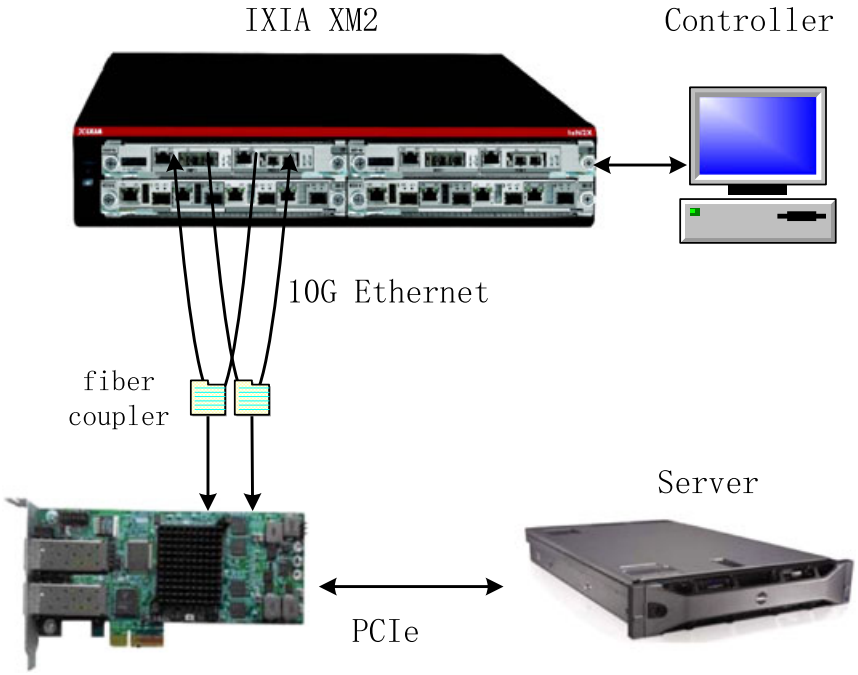

Figure 6. Stream reassembly performance test environment.

IxExplorer, and the application level test was conducted by IxChariot [28].

Some counters are used in the SRC, and the host can obtain them through the shared memory; afterwards, the counters are compared with statistics in the IxExplorer to detect if any packet drop occurs. The maximum traffic that does not cause any packet dropping is considered as the throughput. By using this approach, we cannot obtain the exact throughput in every situation, a step of 0.01 Gbps has been used as an increment to obtain the critical point.

Several factors that may affect the performance are the computing ability of the NPU, I/O performance of the 10 GbE interfaces in NPU, memory size, bus bandwidth, and capture ability of the host application. The memory space has been analyzed in Section 6. The maximum processing ability can be obtained by the packet test.

Although the theoretical bandwidth of a PCIe 1.1×4 may achieve 10 Gbps, its actual performance is limited by the 8 b/10 b coding mode, as well as the performance of host and SRC. To obtain the utmost limit of the data uploading, throughput from SRC to host is tested to determine the maximal packet transfer throughput. A modification making the software running on the SRC do not reassemble the packet is conducted to test the performance with different packet length and different number of capture threads (queues). The result is described in Table III.

We can draw a conclusion from Table III that more capture threads yield better performance. The results also testify the necessity of using a multichannel virtual queue technique. On the other hand, once nine threads are used, instead of increasing the performance, the capture performances decrease a little than when the eight threads are used, because the E5606 CPU has four cores and eight threads in total; thus, more than eight threads may cause thread switching and lead to performance debasement. The column of the one thread shows that when the packet length is less than 1024, the performance is almost linear to the packet length. Longer packet produces higher performance, which is because the packets are DMAed to the memory of the CPU one by one, and the longer the packets are, the larger the data quantity is; therefore, the average transfer time for 1 B decreases. However, when the packet length is short, the key performance limit is the startup time of DMA. In general, the highest performance that the SRC can obtain is a little less than 6 Gbps when the largest packet length and 8 capture threads are used, which is approximately the same as that in [25]. As a general knowledge, the average packet length is between 300 to 400 B in a regular internet link. If 8 threads are used to capture the data, the capture performance is above 4 Gbps.

To conduct the stream reassembly test, the SRC needs to obtain the bidirectional traffic to reconstruct the packets. The IXIA tester can produce bidirectional traffic indeed, but the two 10 GbE interfaces connect with each other resulting in that no traffic is obtained by the SRC. A method must be employed to obtain the bidirectional traffic and inject it into the SRC. We use two fiber couplers to splitter the light; for every coupler, one input port of the coupler is used to connect an interface of the tester to obtain the traffic, and for the two other output ports, one is used to connect the interface, and the other one is used to inject traffic into the SRC.

To test the stream assembly performance of SRC, we change the tester to the application test mode and use the HTTP as the traffic load. Two XM2 ports are used to emulate the traffic between one server and multiclients. To make full use of the transferring bandwidth, eight capture threads in the host are used. We test the performance under different core number circumstances to obtain the relationship between the NPU core number and stream reassembly performance. Because every core has four threads, when one core is tested, one thread is used as timeout thread, and the other three threads are used to reconstruct the packets; when two cores are tested, one thread is used as timeout thread, and the other seven threads are used to reconstruct the packets; and so forth. Owing to the traditional NIDS used Libnids [14] to conduct its stream reassembly, we compared our SRC's performance with the Libnids, an Intel 82599 is used as an Ethernet card, and the test environment is the same as Figure 6; the results are depicted in the last column of Table IV.

Table IV. Throughput of stream reassembly card and Libnids.

Packet length(byte)	1 core	2 cores	3 cores	4 cores	Libnids
64	0.22	0.33	0.61	0.64	0.45
128	0.25	0.47	0.98	1.18	0.48
256	0.59	1.02	2.50	3.11	0.82
512	0.75	1.51	2.66	3.48	0.93
1024	1.30	2.73	3.12	3.70	0.99
1500	1.45	2.98	3.11	3.85	1.21

Unit: gigabits per second.

Table III. Throughput of packet capture.

Packet length (byte)	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads	9 threads
64	0.13	0.22	0.27	0.44	0.45	0.46	0.52	0.76	0.75
128	0.26	0.34	0.58	0.87	0.88	0.9	1.12	1.59	1.57
256	0.53	0.67	1.02	1.89	1.96	2.31	2.79	4.10	4.05
512	1.09	1.88	1.97	2.86	3.07	3.71	4.03	4.77	4.78
1024	1.97	3.42	3.56	4.47	5.24	5.29	5.31	5.45	5.42
1500	2.80	4.73	4.96	5.69	5.68	5.68	5.73	5.91	5.86

Unit: gigabits per second.

To check if disordered packets affect the performance remarkably, we adjust the packet sequence of the HTTP reply and make the one or two packet(s) with larger zsequence sent out previously, and the ratio of the disordered stream is about 10%. Even through the SRC needs to reorder the packets, no remarkable influence to the performance has been identified, and we speculate that this is because the SRC only needs to adjust the pointer of the list instead of moving blocks of memory.

It is also observed that both more cores and longer packets can lead to higher performance. If all the cores are used, the performance is close to that of the packet capture, which means that when all the threads in the NPU are turned on to reassemble the packets, the performance is nearly close to the PCI transferring ability; and if the PCI multiplying factor is 8, the performance will be higher than the current implementation. Because the average packet length is between 300 and 400 B in real network environment, the throughput will be higher than 3 Gbps, whereas the throughput of Libnids is between 0.82 and 0.93 Gbps with the relatively high-performance Intel NIC 82599. In other words, we can use one SRC instead of three high-performance servers to accomplish the same stream reassembly. From the spending viewpoint, the cost of one SRC is less than $1000, which is much lower than two high-performance servers.

8. CONCLUSION

Transmission control protocol packet reassembly is crucial to NIDS and NFS. However, its performance becomes the bottleneck as the matching performance increases. In this paper, through the analysis on why the performance is very low in the traditional stream reassembly, we have identified that the emerged multicore NPU can hide the delay through parallel DRAM access. With the aforementioned discovery, a coprocessing stream reassembly framework based on multicore NPU has then been introduced as a card; in this way, both the packet capture and stream reassembly can be solved by a card. In addition, to enhance the performance, we brought forward stream table dispatching, no-locking timeout, and multichannel virtual queue to improve the performance of the proposed SRC scheme. Furthermore, RMI XLS416 was used to implement a coprocessing SRC, and we applied an IXIA XM2 tester to evaluate its performance in the situation with different packet sizes and different numbers of processing cores. The results have demonstrated that our scheme is around three times better than Libnids when SRC is used in the current predominant server.

ACKNOWLEDGEMENT

This work has been supported by the National High-Tech Research and Development Plan of China under Grant No. 2011AA01A103.

REFERENCES

1. Nicholas W, Vern P, Gonzalez JM. The shunt: an FPGA-based accelerator for network intrusion prevention. In *Proceedings of the 2007 ACM/SIGMA 15th international symposium on Field programmable gate arrays*. ACM: Monterey, California, USA, 2007; 199–206.
2. Lu R, Lin X, Liang X, *et al*. Secure provenance: the essential of bread and butter of data forensics in cloud computing. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM: Beijing, China, 2010; 282–292.
3. Martin L, Gregory SJ. The case for hardware transactional memory in software packet processing. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM: La Jolla, California, USA, 2010; 1–11.
4. Vasiliadis G, Antonatos S, Polychronakis M, *et al*. Gnort: high performance network intrusion detection using graphics processors. In *Proceedings of the Recent Advances in Intrusion Detection*. Springer: Cambridge, Massachusetts, USA, 2008; 116–134.
5. Giorgos V, Michalis P, Spiros A, *et al*. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. Springer: Saint-Malo, France, 2009; 265–283.
6. Vasiliadis G, Michalis P, Sotiris I. RMIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM: Chicago, Illinois, USA, 2011; 297–308.
7. Lin CH, Liu CH, Chang SC. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *Conference of IEEE Globecom*. IEEE: Houston, Texas, USA, 2011; 1–5.
8. Meiners CR, Patel J, Norige E, Torng E, *et al*. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security*. USENIX Association: Washington, DC, USA, 2012; 8–23.
9. Egorov S, Savchuk G. SNORTRAN: an optimizing compiler for snort rules. http://www.fidelissec.com/snortran/SNORTAN-wp.pdf
10. Ptacek T H, Newsham T N. Insertion, evasion, and denial of service: Eluding network intrusion detection. DTIC Document, 1998.
11. Novak J. Target-based fragmentation reassembly. Sourcefire, 2005.
12. Fraleigh C, Moon S, Lyles B, *et al*. IPacket-level traffic measurements from the sprint IP backbone. *IEEE Network* 2003; **17**(6):6–16.

13. Lan K, Heidemann J. A measurement study of correlation of internet flow characteristics. *Comput Networks* 2006; **17**(1):46–62.
14. Libnids. http://libnids.sourceforge.net
15. Tcpflow. http://afflib.org/software/tcpflow
16. Pan J, Mark JW, Shen S. TCP performance and behaviors with local retransmissions. *The Journal of Supercomputing* 2002; **23**(3):225–244.
17. Kim H, Heidemann J. Counting network flows in real time. *Comput Networks* 2006; **17**(1):46–62.
18. Wagener G, Dulaunoy A, Engel T. Towards an estimation of the accuracy of TCP reassembly in network forensics. In *Proceedings of Future Generation Communication and Networking*. IEEE: Sanya, Hainan, China, 2008; 273–278.
19. Zhang M, Ju J. Space-economical reassembly for intrusion detection system. In *Proceedings of International Conference on Information and Communications Security*. Springer: Huhehaote, Neimenggu, China, 2003; 393–404.
20. Sarang D, Vern P. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium*. USENIX Association: Baltimore, Maryland, USA, 2005; 65–80.
21. Intel 82575EB GbE Controller Software Developer Manual. http://www.intel.com
22. Cascallana GA, Lizarrondo EM. Collecting packet traces at high speed. In *Proceedings of Workshop on Monitoring, Attack Detection and Mitigation*. Citeseer: Tübingen, Germany, 2006; 1–6.
23. Deri L. Improving passive packet capture: beyond device polling. In *Proceedings of International System Administration and Network Engineering*. SANE: Amsterdam, Netherlands, 2004; 85–93.
24. Lunteren JV, Engbersen T. Fast and scalable packet classification. *IEEE J. Sel. Areas Commun.* 2003; **21**(4):560–571.
25. Deri L, Engbersen T. nCap: wire-speed packet capture and transmission. In *Proceedings of the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON)*. IEEE: Nice, France, 2005; 47–55.
26. Smith M L, Loguinov D. Enabling high-performance internet-wide measurements on windows. In *Proceedings of Passive and Active Measurement*. Springer: Zurich, Switzerland, 2010; 121–130.
27. Francesco F, Luca D. High speed network traffic analysis with commodity multi-core systems. In Proceedings of the 10th Annual Conference on Internet measurement, Melbourne, Australia, 2012; 218–224.
28. IXIA. http://www.ixiacom.com/