The CSI Journal on
Computer Science and Engineering
Vol. 11, No. 2 & 4 (b), 2013
Pages 24-30
Regular Paper

Parallel Implementation of Linux Packet Filtering

Keyvan Karimi[1] Arash Ahmadi[2] Mahmood Ahmadi[2]

[1]Department of Information Technology, Kermanshah Science and Research Branch, Islamic Azad University, Kermanshah, Iran
[2]Engineering Faculty, Razi University, Kermanshah, Iran

Abstract

Packet filtering specifies which type of traffic is allowed to/from organizational network. Each data packet is compared against a rule set. The number of comparisons that must be performed is increased when the size of the rule set is increased. In high bandwidth networks the packet filtering becomes a time consuming task which can reduce the overall throughput. To solve this problem a wide range of researches have been done to improve the overall throughput of packet filtering firewalls. In this paper, comparison of data packet against the rule set for IPTables is performed in user-space by employing parallel processing capability of Graphics Processing Unit. The results show that the CPU-GPU parallel code brings higher throughput over CPU version of IPTables code. The overall throughput for 80 bytes packet size and rule set size of 10,000 is about 400,000 Packets-Per-Second which is 43 times faster than CPU version code.

Keywords: GPU, IPTables, Packet Filtering, Parallel Processing, Throughput, Packet Delay.

1. Introduction

Packet filtering firewall takes its decision to accept or drop the packets only by investigating the header portion of the incoming or outgoing packets, which contains essential networking information including source IP address, destination IP address, protocol, port numbers, etc. Firewall performs filtering task by comparing the packets against rules of the filter set. Many organizations and corporations need to connect the internet to do their current activities. By growing the size of organizations and diversity of the users who connect to the internet, along with new increasing services which are introduced by these organizations, it is an essential requirement to have more precise control on every aspects of the incoming and outgoing traffic. Accordingly, filtering rule sets are becoming more complicated in terms of interrelations and capacity, which results in a higher computational complexities. Modern firewalls use first-match semantics [1, 2, 3]. Today most of the firewalls use two mechanisms for packet matching: 1) linear search that implements the first match and 2) fast state table lookup that uses hash table and search tree to bypass the rule set. Open source firewalls pf [4] and IPTables [5] use both aforementioned approaches. Linear search over alarge rule set can become very inefficient and make the packet filtering a point of failure, because each packet needs to be checked against every rule until the first matching rule is found. In [6, 7, 8] discovery and effects of last-matching rules of a firewall rule set on firewall performance are examined. The packets that matched these rules consume the most CPU power because the firewall must compare all the rules before that rules. The second method, on the other hand, is more suitable for long TCP connections in which fast state table lookup handles most of the packets. However, stateless connections such as UDP, ICMP traffic and short TCP flows, have to use the rule set and linear search similar to the first method.

In this paper, NFQUEUE target of Linux IPTables packet filter is employed in order to send the packets to user-space and process them by parallel processing capability of Graphics Processing Unit (GPU). The independence nature of the packets is well suited for the parallel processing

approaches. In this study, a fully functional packet filter is created by Linux IPTables packet filter, where the throughput of the proposed method is tested on a laboratory test-bed.

Our parallel design of Linux packet filtering can reach the rate of 400,000 PPS, with 10,000 rules. IPTables could only process about 2500 PPS with the same rule set [9] and on our test-bed PC, the sequential code, it could process about 9000 PPS but the authors used a slower machine than ours.

The rest of the paper is organized as follows: Section 2 reviews related works. Section 3 describes Linux Net filter framework and IPTables tool to define chains of rules for packet matching. In section 4, we briefly discuss about GPU and how GPU is used in high performance general-purpose applications. Section 5 describes our user-space packet-filtering approach which uses parallel computing capability of GPU to accelerate the code. In Section 6 results are presented and discussed. Section 7 is conclusion and future works.

2. Related Works

In this section the related researches in packet filtering is presented. Rovniagin et.al. [9] proposed a geometric matching algorithm that its worst-case space complexity is *O (nd)* where *n* is the number of rules and *d* is the number of fields to match. This algorithm is integrated into the Linux IPTables firewall codes and has been tested on the real traffic loads. The results show that this algorithm can filter over 30,000 PPS on a standard PC with10,000 rules.

Employing GPU for packet filtering is introduced in [10] and the results is obtained in packet delay per number of firewalls and per number of rules. Unfortunately, the delay that API causes to exchange packets from kernel-space to user-space is not considered. It only calculates the raw elapsed time that packets spend to process on GPU but our approach is tested in a laboratory test-bed and by considering the time that packets spend to be received from network interface and to be processed on GPU then issue the verdicts. Another work is [11] that authors have used and employed parallel capability of GPU to process packets in user-space. It modifies user-level interface for packet I/O engine to remove high overhead caused by the I/O engine. In IP forwarding, only the destination IP address is checked but in packet filtering firewalls at least four to five fields have to be checked.

Another work on GPU accelerated software router that is achieved significant boost on speed is [12]. Christiansen and Fleury [13] introduced Interval Decision Diagrams for packet filtering using first order logic. They construct a logic formula by the set of rules based on the integer intervals that introduces the logarithmic search time but exponential build algorithm. In [14] parallel firewall design is introduced. It has reviewed different design aspects on firewall throughput and packet delay.

In this work, the stateless portion of IPTables Linux packet filtering is mapped and implemented on a GPU. It should be noted that this part is running in sequential manner on the Linux kernel which we have implemented it in parallel using GPU in user space.

3. Linux Net Filter/IPTables

Net filter [15] refers to a framework within the Linux kernel that can be used to hook functions into the networking stack. On the other hand, IPTables uses the Net filter to hook functions designed to perform operations on packets (such as filtering) into the networking stack [5]. You can think of Net filter as a framework that IPTables builds firewall functionality on it. The kernel module named ip_tables which is a component of Net filter provides table-based system for defining firewall rules that can filter or transform packets. The table can be administered using the IPTables user-space tool. System administrator can define tables containing chains of rules for the treatment of packets. IPTables searches chains of rules for matching with each packet sequentially. There are five predefined chains that a table may not contain all of them: PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING. IPTables has the following built-in tables: Filter, NAT, Mangle and Raw. The Filter table is used for filtering purposes and consists of three chains: INPUT, OUTPUT and FORWARD. Figure 1 shows a sample IPTables rule set that is defined in the Filter table. This sample rule set consists of six rules that the top rule in each chain has the maximum priority and it will be processed first. Every rule in filter table has four main sections:

- Table, that the rule should be added to
- Chain, that the rule should be inserted to
- Match, the criteria
- Target, which specifies the action on matched packets

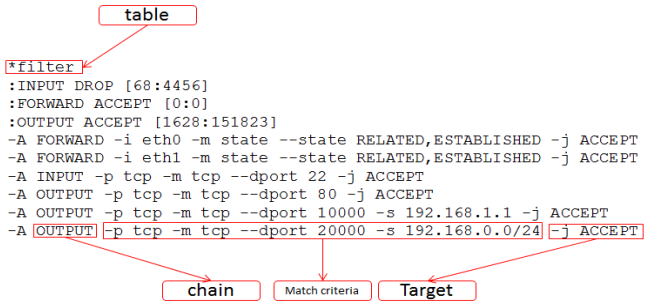

Figure 1. Excerpts from IPTables firewall configuration file, showing three chains and six rules

In the Filter table of IPTables, each chain has a default action that when the packets did not match to any rule, it will be executed on packets. The default action is DROP or ACCEPT.

4. Overview on GPGPU

General-Purpose computing on Graphics Processing Unit (GPGPU) refers to the use of GPU processing capability for general purpose processing. A vast number of processing cores available in current generation of GPU architectures is well suited for parallel computing purposes. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model created by NVIDIA Corporation and implemented by the GPUs that it produces. CUDA gives the developers the ability to use latest NVIDIA GPUs for computation like CPUs. Unlike CPUs, the GPUs

have a parallel throughput architecture that create and execute many concurrent threads. CUDA programs consist of some phases that are executed on both CPU and GPU sides. The phases that have little or no parallelism are implemented on host code (on CPU) and the phases that have rich amount of parallelism are implemented on device code (on GPU). A simple code execution of CUDA program is illustrated in figure 2 CUDA-accelerated libraries, compiler directives, and extensions to industry-standard programming languages, including C, C++, are prepared for the CUDA platform accessible. Therefore, C/C++ programmers can use 'CUDA C/C++' that is compiled with 'nvcc' command to accelerate their applications on GPU [16].

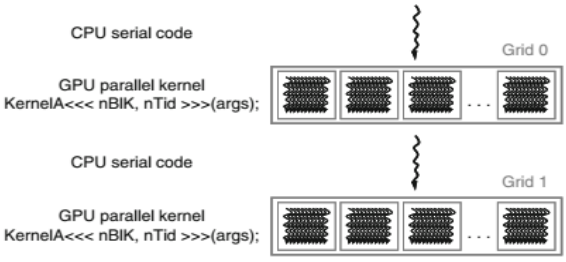

Figure 2. Execution of a CUDA program [17]

In contrast to general-purpose processors that have to satisfy requirements from legacy operating systems, applications and I/O devices that make memory bandwidth more difficult to be increased, simpler memory models and fewer legacy constraints of GPUs have made it easier to achieve higher memory bandwidths. The more recent NVIDIA GT500 chip supports about 327.7 GB/s.

5. Implementation

5.1. IPTables/Lib Net Filter_Queue Packet Filtering

In addition to IPTables, Net filter project also provides a set of libraries that can be used to perform different task in user-space [5]. One of these libraries is Libnetfilter_queue that is used for packet filtering in user-space. The NFQUEUE target in IPTables rules is used to queue the packets to user-space. Receiving queued packets from kernel Nfnetlink_queue subsystem and then issuing verdicts to the kernel Nfnetlink_queue subsystem is performed by Libnetfilter_queue. For example, the following rule will ask for a decision to the listening user-space program for TCP packets going to the Linux box:

IPTables -A INPUT –p tcp -j NFQUEUE --queue-num 0

In user-space, the program uses the Libnetfilter_queue library to connect to queue 0 and receive the messages from kernel. It should be noted that:

- A fixed length queue is implemented as a linked list of packets
- An integer value is integrated to each packet as an index
- The packets are released when the user-space program issues the verdict to the corresponding index value

Multiple packets are read by the size of the program buffer. Then the packet buffer is sent into the GPU to be processed in parallel in order to specify the action that should be applied on each packet. The action that should be applied on each packet is saved in a verdicts array and is returned to host. Then, Libnetfilter_queue issues this verdicts array. Issuing the verdicts by Libnetfilter_queue for packets can be performed in any order, for example receiving packets in 1, 2, 3 order and issuing them in 2, 3, 1 order. Packet flow in Net filter is depicted in figure 3 and we can find the position of the packet-filtering program which is the abstract view of the test-bed. Moreover, It shows that how IPTables and Libnetfilter_queue cooperate to perform packet filtering in user-space.

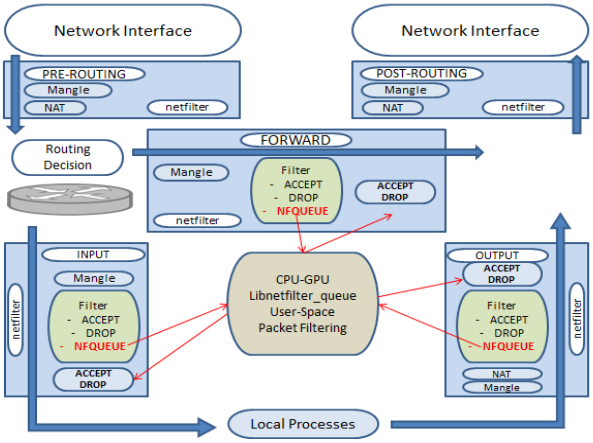

Figure 3. Packet flow in Net filter and the position of our CPU-GPU user-space program

Multiple threads are created in user-space on CPU to distribute traffic among threads by NFQUEUE target and then each thread sends its packet buffer to GPU. The following command is utilized to send different connections to different threads on CPU. Multi-threading is utilized on CPU level to gain parallel processing capability of CPU along with GPU.

IPTables –A INPUT –p tcp –j NFQUEUE –queue-balance 0: x

In this command the different TCP connections is handled by different CPU threads to process in user-space. The "0: x" at the end of the command specifies the number of threads that has to be created on CPU to handle "x" different connections.

The synchronization between CPU threads is performed by a Boolean variable to avoid sending data packets on GPU at the same time. Figure 4 shows the multi threads program model.

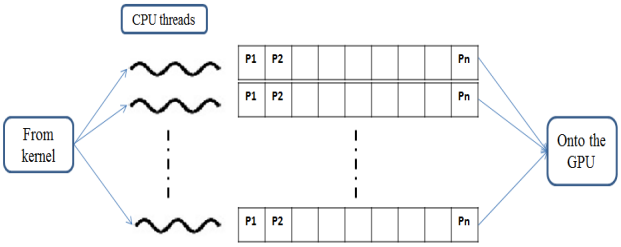

Figure 4. Exploit multiple cores and threads of CPU

5.2. Data set

The first portion of data set is filter set. Filter set is extracted from real filter set. Class bench [18] is employed to create synthetic filter set which has characteristics of real filter set. It also has Trace generator produce a sequence of packet headers that can be used to trigger our real filter set. The filter sets that are produced are 100, 1000, 5000, and 10,000 rule sets. Every filter has 5-tuple which is formatted as Figure 5:

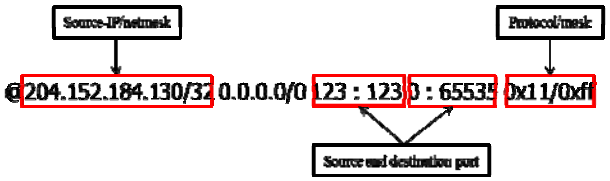

Figure 5. A sample filter which is created from {source/destination}/ {net mask}, {source, destination} ports and protocol/mask

Second portion of data set is traffic. The data packet is generated by the tools that are introduced in test-bed section.

5.3. GPU Programming Model

Considering independent nature of each packet in network communications, it can be concluded that parallel processing of multiple packets can benefit from high parallel processing capability of GPU by using CUDA parallel computing platform. Therefore, we receive the packets and queue them in our packet array data structure. Acceleration of packet processing is performed by calling 'buffer size' multiply 'rule set size' number of threads in the GPU. Using this configuration, multiple packets about buffer size are processed once.

Every thread has two features: thread index and block index, thread index and block index can be three-dimensional in which the dimensions are x, y and z. In ourimplementation, we have employed the x dimension of block index to specify each packet and combination of the y dimension of block index and x dimension of thread index to specify each rule. In this implementation, each thread checks a packet with a rule for matching. We have a buffer of packets and another array (rule_matched array) corresponding to this buffer array along with rule array. The rule array is sent in the beginning of our program into the GPU memory. The rule_matched array has the same size of packet buffer and each element of this array corresponds to the each packet in the packet buffer that specifies the first rule packet matched with. Each thread when finds a matching rule with a packet, writes its thread number to the corresponding element in the rule_matched array. Because the packets may matched more than one rule in firewall rule set and threads run concurrently, we have to apply a mechanism to solve this problem. For this reason, we have used CUDA 'atomic Min' Atomic Operations to solve the problem. While a thread wants to write to that memory location, it locks the memory location and writes its rule number. Therefore, the first rule that has the maximum priority in firewall rule matching concept is written to the rule_matched array. Figure 5 shows parallel model that is used to rule matching on GPU.

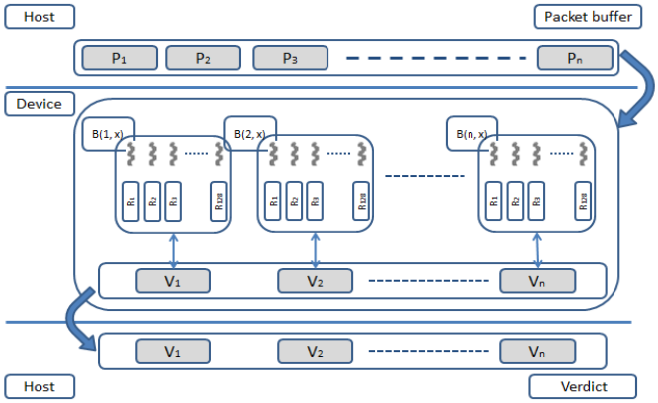

Figure 6. Processing packets on GPU and return the verdict result to Host

The following equations show the relation between packets and rules with thread and block indexes:

$$\text{Rule\_number} = \text{ThreadIndex.x} + \text{BlockIndex.y} * \text{BlockDim.x} \quad (1)$$

$$\text{Packet\_number} = \text{blockIndex.x} \quad (2)$$

Every thread that runs the code to find a matching rule with the packets uses these two values to point the buffer array and rule array. A pseudo code on GPU is as follows. Initial value of verdicts array is set to zero.

```
Func void comparison_gpu(verdicts, packet_buffer)
{
   If packet_num.proto is equal rule_num.proto
      Continue;
   Else
      Break;
   If packet_num.dport is equal rule_num.dport
      Continue;
   Else
      Breake;
   Then if packet.num.sport is equal rule_num.sport
      Continue;
   Else
      Break;
   Then if packet_num.source_ip is in range
rule_num.source_ip
      Continue;
   Else
      Break;
   Then if packet_num.dest_ip is in range rule_num.dest_ip
      atomicMin(verdicts[packet_num] , rule_num);
}
```

When a matched is occurred the thread number is written into verdicts array. It is important to mention that since multiple rules could be matched, to prevent of writing a wrong value to verdicts array the memory location must be locked. Locking the memory location is performed by special CUDA functions, called atomic functions. The function that is employed here is atomic Min. Each matched thread to a rule calls this function and the function compares the thread number (rule number) with current value, if the value of

thread is smaller than current value of verdicts then thread number is written to verdicts. Therefore, the rule with highest priority is obtained.

The two factors that should be considered for each software program in high performance environment are time and space complexities of the algorithm. As it can be observed that the algorithm complexity is O (N/M) that N is the number of packets and M is the number of GPU cores and it can be concluded that the time complexity is O (n) but as it is mentioned in [11] the launching latency for a single thread is 3.8 µs and 4.1 µs for 4096 threads (only 10% increase). Therefore, the time complexity is not increased by a percent of N.

The rule set is sent to the GPU at the start of program and this work is performed only once during the program. In addition, every time buffer is filled packet buffer is sent to GPU. Therefore, the space complexity is about (buffer-size * size-of-packet) + (rule-set-size * size-of-rule).

5.4. Test-Bed

User-space packet-filtering program is run on a commodity PC that has a 2.5 GHz Intel Core i5 CPU with a 1 Gbps Ethernet interface. NvidiaGeforce GTX 580 graphic processor is employed to processing packets on packet filter program. The packet generator is a PC with a single 1 Gbps Ethernet interface which is named Ubuntu1 and the firewall is Ubuntu2 PC which is shown in figure 6 Both PCs ran Ubuntu Linux 11.10. The two PCs are connected together by a crossover Ethernet cable according to figure 6 We have configured our test-bed to generate random source and destination IP addresses, source and destination ports by flooding the packets.

All the packets that traffic generator generates are 80 bytes TCP packets with no TCP flags set. On the firewall PC that the program was running, we have configured the program to finish when it filters a specific amount of packets and then it prints the time elapsed. When we received the packets, and sent them into the GPU for processing then we simply set the verdicts to drop all the packets. We used Ostinato open source package in Ubuntu, which is a cross-platform network packet crafter/traffic generator to generate traffic. By this tool (Ostinato) we can create and customize different traffics and generate all the traffics at the same time to any interface.

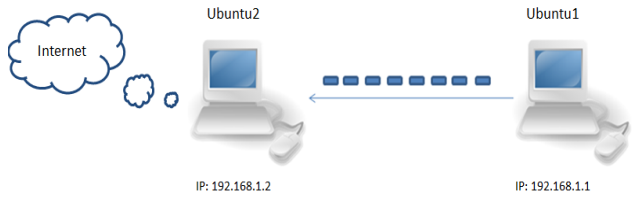

Figure 7. Tested Scenario

6. Results

We ran our test-bed for different rule set sizes of 100, 1000, 5000, and 10,000 rules which are the real rule set. In addition, we have considered the impacts of the different buffer sizes on the packet delay and throughput on GPU. Accordingly, the results are divided into three sections. In the first section, the impacts of different rule set sizes and buffer sizes on packet delay for GPU is considered. Therefore, in this section the packet delay in µs (microsecond) is only provided for GPU. In the second section, we have compared packet delay between CPU and GPU-CPU. A packet is received from network interface and sent to user-space then processed on GPU and finally returned to network interface. The time duration of this process is defined as packet delay. In order to perform a fair comparison based on packet delay between CPU and CPU-GPU, the buffer size is considered as one packet because CPU processes the packets one by one (sequentially).

So, we have considered one packet buffer size in this section. The final section is about throughput. By changing buffer size and rule set size the throughput of the IPTables Linux packet filter on GPU is calculated. The buffer size is considered to be 1000 packets which gives an acceptable packet delay. 1000 packets are buffered every time and are sent to GPU at once. For each state, we ran the program more than 30 times and the average value of them is calculated.

Figure 7 shows the packet delay for different rule set size and buffer size on GPU. The horizontal axis shows the number of rules in the filter set files and the vertical axis shows the packet delay in microsecond. Although increasing buffer size may result in boosting throughput, it increases the packet delay. We can see by increasing buffer size and ruleset size the packet delay is increased too. We want to show how much buffer size can be increased while we have had an acceptable packet delay.

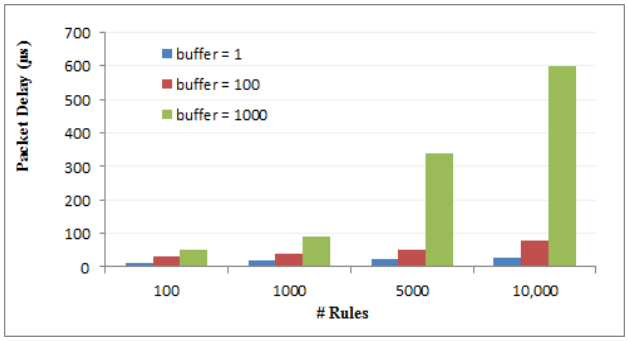

Figure 8. Packet delay on GPU for different rule set size and buffer size

Figure 8 shows the packet delay between CPU and CPU-GPU code. The results demonstrate that by increasing rule set size the packet delay is increased for CPU version code while in CPU-GPU code the amount of increasing is fewer. When the amount of parallelization of GPU computing is small, buffer size of one packet and 100 rules, CPU has much better performance than GPU.

While the amount of parallelization is increased, one packet and 10,000 rules, latency for CPU version code is increased and GPU gives better performance. The vertical axis shows the packet delay in microsecond and the horizontal axis shows the number of the rules. Smaller packet delay is obtained in rule set size with more than 10,000 rules.

In the last section we have examined throughput which the results are shown in figure 9 The X axis shows the number of rules and the Y axis shows the throughput in kilo

packet per seconds (KPPS).The results in figure 9 show that we have at least 2.3 times speedup and we obtain about 43 times speedup for 10,000 rules set size in the GPU over the CPU version code. The throughput for 10,000 rules in sequential code is about 9000 PPS while the throughput for our GPU accelerated code is about 400,000 PPS for the same rule set size. The results that is obtained here for CPU version code is comparable with Net filter Performance Testing [19].

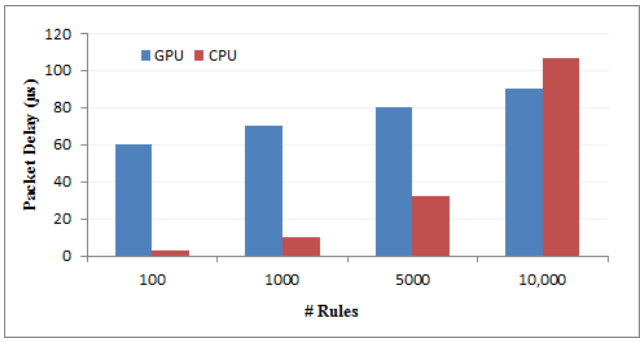

Figure 9. Packet delay between CPU and CPU-GPU for different rule set size

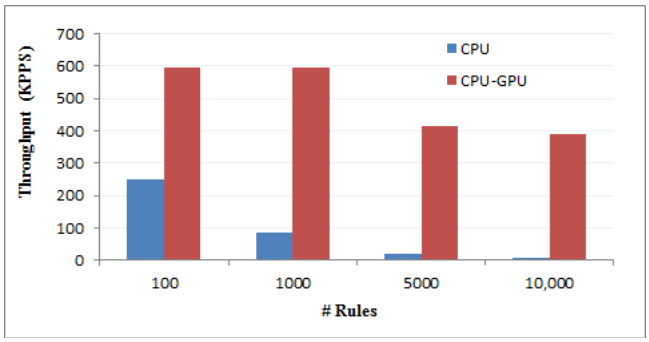

Figure 10. Throughput for different rule set size on CPU and CPU-GPU

The throughput of Libnetfilter_queue API for 80 Bytes packet sizes can reach up to 600,000 PPS without considering other processing overheads. Therefore, the throughput of CPU-GPU packet-processing code is limited up to 600,000 PPS. The program gains 400,000 PPS throughput on CPU-GPU for 10,000 rules that is anyway 43 times faster than CPU code of IPTables in kernel.

7. Conclusion and Future Works

In our work while the IPTables preserves the state full fast packet matching we have improved the naive slow linear matching for large rule set and it is well suited for heavy loaded networks. Because the Libnetfilter_queue library uses Net link sockets for inter-process communications (IPC) between the kernel and user-space therefore Net link sockets are not suitable for high performance packet processing. Therefore, our user-space packet filtering throughput is limited up to 600,000 PPS.

Consequently, the total bandwidth for 80 Bytes packet size is 256 Mbps while the corresponding IPTables linear search algorithm can only reach up to 5.8 Mbps bandwidth. The integration of high performance packet I/O with accelerated GPU packet processing can be the future work for removing packet I/O inefficiency. Employing a better packet-matching algorithm on GPU can make more parallelism acceleration to eliminate CPU processing power inefficiency for rich features and free open-source packet filter firewalls like IPTables.

References

[1] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley Longman Publishing, 2003.

[2] D. D. Welch-Abernathy, *Essential Check Point Firewall-1 NG: An Installation, Configuration, and Troubleshooting Guide*, Pearson Higher Education, 2004.

[3] H. Bidgoli, *Handbook of Information Security, Information Warfare, Social, Legal, and International Issues and Security Foundations*, John Wiley and Sons, 2006.

[4] D. Hartmeier, "Open BSD Packet Filter," http://www. openbsd. org/faq/pf, June 2006.

[5] N. Team, *The Net Filter/IPTables Project*, Cambridge, MA, 2004.

[6] K. Salah, K. Sattar, Z. Baig, M. Sqalli, and P. Calyam, "Resiliency of Open-source Firewalls Against Remote Discovery of Last-matching Rules," *Proc. IEEE Intl Conf. Security of Information and Networks*, pp. 186-192, 2009.

[7] K. Salah, K. Sattar, Z. A. Baig, and M. H. Sqalli, "Discovering Last-matching Rules in Popular Open-source and Commercial Firewalls," *Intl Journal of Internet Protocol Technology*, vol. 5, no. 2, pp. 23-31, 2010.

[8] K. Salah, K. Sattar, M. Sqalli, and E. Al-Shaer, "A Probing Technique for Discovering Last-matching Rules of a Network Firewall," *Proc. IEEE Intl Conf. Innovations in Information Technology*, pp. 578-582, 2008.

[9] D. Rovniagin, and A. Wool, "The Geometric Efficient Matching Algorithm for Firewalls," *Proc. IEEE Intl Conf. Electrical and Electronics Engineers in Israel*, pp. 153-156, 2004.

[10] M. S. Gaur, V. Laxmi, K. Cahndra, and M. Zwolinski, "Acceleration of Packet Filtering Using GPGPU," *Proc. IEEE Intl Conf. Security of Information and Networks*, pp. 227-230, 2011.

[11] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated Software Router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195-206, 2011.

[12] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP Routing Processing with Graphic Processors," *Proc. IEEE Intl Conf. Design, Automation and Test in Europe*, pp. 93-98, 2010.

[13] M. Christiansen, and E. Fleury, "An Interval Decision Diagram Based Firewall," *Proc. IEEE Intl Conf. Networking*, pp. 64-70, 2004.

[14] E. W. Fulp, "Parallel Firewall Designs for High-speed Networks," *Proc. IEEE Intl Conf. Computer Communications*, pp. 1-4, 2006.

[15] http://www.Netfilter.org/projects/Libnetfilter_queue/doxygen, June 2013.

[16] NVIDIA Corporation, http://www.nvidia.com/object/cuda_home_new.html, May 2013.

[17] D. B. Kirk, and W. H. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach*: Newnes, 2012.

[18] D. E. Taylor, and J. S. Turner. "Classbench: A Packet Classification Benchmark," *IEEE/ACM Trans. Networking*, vol. 15, no. 3, pp. 499-511, 2007.

[19] J. Kadlecsik, and G. Pásztor, "Netfilter Performance Testing," *Proc. IEEE Intl Conf. Computer Communications*, pp. 14-20, 2004

Keyvan Karimi received B.Sc. degree in computer software engineering from Tabriz University, Tabriz, Iran, in 2009. He received M.Sc. degree in information technology engineering from Islamic Azad University of Kermanshah, Science and Research Branch, Kermanshah, Iran, in 2013. He was involved in high performance packet filtering firewall project on GPU, Karin Saze, Kermanshah, Iran from 2011 to 2014. His research interests include packet filtering firewalls, parallel processing, high performance packet classification algorithms, and network security.
E-mail: kaiwan.karimi@gmail.com

Arash Ahmadi received the B.Sc. and M.Sc. degrees in electronics engineering from Sharif University of Technology and Tarbiat Modares University, Tehran, Iran, in 1993 and 1997, respectively, and the Ph.D. degree in electronics from the University of Southampton, U.K., in 2008. He was with Razi University, Kermanshah, Iran, as a Faculty Member. From 2008 to 2010, he was a Fellow Researcher with the University of Southampton. He is currently an Assistant Professor in the Electrical Engineering Department, Razi University. His current research interest includes digital hardware design and optimization, high-level synthesis, bio-inspired computing and memristors.
E-mail: aahmadi@razi.ac.ir

Mahmood Ahmadi received the B.S. degree in Computer Engineering from Isfahan University, Isfahan, Iran in 1995. He received the M.Sc. degree in Computer Engineering from Tehran Poly Technique University, Tehran, Iran in 1998.From 1999 to 2005, he was a faculty member at Razi university in Kermanshah in Iran. In October 2005, he joined the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS), Delft University of Technology, Delft, The Netherlands, as a fulltime Ph.D. student. He got his PhD in May 2010. Currently, he is working as assistant professor at Computer Engineering Department of Razi University of Kermanshah in Iran. He is member of IEEE. His research interests include Computer architecture, network processing, Bloom filters, performance modeling and reconfigurable computing.
E-mail: m.ahmadi@razi.ac.ir

Paper Handling Data:

Submitted: 26.04.2014
Received in revised form: 08.11.2014
Accepted: 15.11.2014
Corresponding author: Keyvan Karimi,
Department of Information Technology, Kermanshah Science and Research Branch, Islamic Azad University, Kermanshah, Iran.