IJCCS (Indonesian Journal of Computing and Cybernetics Systems)
Vol.14, No.4, October 2020, pp. 331~340
ISSN (print): 1978-1520, ISSN (online): 2460-7258
DOI: https://doi.org/10.22146/ijccs.54170

Resource Modification On Multicore Server With Kernel Bypass

Dimas Febriyan Priambodo*[1], Ahmad Ashari [2]
[1]Master Program of Computer Science, FMIPA UGM, Yogyakarta, Indonesia
[2]Department of Computer Science and Electronics, FMIPA UGM, Yogyakarta, Indonesia
e-mail: ***[1]priambododimas@gmail.com**, [2]ashari@ugm.ac.id

Abstrak

Teknologi berkembang sangat cepat ditandai dengan munculnya banyak inovasi baik dari hardware maupun software. Multicore server dengan jumlah core yang semakin banyak menuntut software yang effisien. Kernel dan Hardware yang difungsikan menangani berbagai kebutuhan operasional mempunyai beberapa keterbatasan. Keterbatasan ini disebabkan karena tingkat kompleksitas yang tinggi khususnya dalam menangani kebutuhan sebagai peladen seperti single socket discriptor, single IRQ dan kekurangan dalam pooling sehingga memerlukan beberapa modifikasi. Metode kernel bypass merupakan salah satu metode untuk mengatasi kekurangan dari kernel. Modifikasi pada server ini merupakan gabungan untuk menaikkan throughput dan menurunkan latency server. Modifikasi pada level driver dengan hashing rx signal dan modifikasi multiple receiver dengan multiple ip receiver, multiple thread receiver dan multiple port listener digunakan untuk meningkatkan throughput. Modifikasi menggunakan prinsip pooling baik pada level kernel atau level program digunakan untuk menurunkan latency. Gabungan modifikasi ini menjadikan server menjadi lebih dapat diandalkan dengan peningkatan througput rata-rata sebesar 250,44 % dan penurunan latency 65,83 %.

Kata kunci*— hash rx, multiple ip, multiple port, multiple thread, pooling, bypass kernel*

Abstract

Technology develops very fast marked by many innovations both from hardware and software. Multicore servers with a growing number of cores require efficient software. Kernel and Hardware used to handle various operational needs have some limitations. This limitation is due to the high level of complexity especially in handling as a server such as single socket discriptor, single IRQ and lack of pooling so that it requires some modifications. The Kernel Bypass is one of the methods to overcome the deficiencies of the kernel. Modifications on this server are a combination increase throughput and decrease server latency. Modifications at the driver level with hashing rx signal and multiple receives modification with multiple ip receivers, multiple thread receivers and multiple port listener used to increase throughput. Modifications using pooling principles at either the kernel level or the program level are used to decrease the latency. This combination of modifications makes the server more reliable with an average throughput increase of 250.44% and a decrease in latency 65.83%.

Keywords*— hash rx, multiple ip, multiple port, multiple thread, pooling, kernel bypass*

Received February 11th,2020; Revised July 27th, 2020; Accepted October 30th, 2020

1. INTRODUCTION

Shambharkar (2015) states several important issues to improve performance on multicore systems [1]. Improved performance can be done by avoiding migration in multicore systems and designing special techniques, tools or in the form of scheduling in the task / process / thread accompanied by affinity (mapping between software and hardware) to carry out the process right on the specific core. The literature survey shows that in the default operating system there is a problem that there is a transfer or migration of processes on several different threads and requires some development and appropriate steps that result in an increase in hardware process speed .

Research by Bo (2016) [2], Diener et al. (2016) [3], Paul, Bhattacharjee and Rajesh (2014) [4], states that performance improvement is NUMA (Non-Uniform Memory Access). NUMA can improve performance by reducing thread transfer through partitioning memory. Contrary to some previous studies Majo & Gross, (2013) [5] shows the error of the researcher's view of the NUMA concept which actually adds to problems in internal memory and reduces process performance. Tang et al., (2013) [6] conducted NUMA research on gmail backend server and frontend google search, giving the same result, which was a decrease in performance. The decrease in performance is indicated by a data decrease of 15% with the AMD Barcelona platform on Gmail Backend and by 20% with Intel Westmere for Web-search frontend.

The system built is a modification to avoid migration in multicore systems as well as testing to provide data on the difference in processing speed between shared cores and dedicated cores that often exist in a multicore processor. Modifications made based on kernel deficiencies in the server computer into a better system are indicated by increased throughput and decreased latency. Modifications made include the driver level which is intended to modify the signal settings by modifying the hashing algorithm into a multi queue to overcome the shortcomings of a single IRQ in the kernel. Modifications are also made to the kernel base of the port listener so that there is no lock on the default kernel such as analysis (Rivera et al., 2014) [7] with binding sockets so that each port can be used simultaneously by all CPUs without locking and waiting processes that increase load from the processor. Modifications were also made to reduce latency with pooling techniques on data packets so that it is expected to be able to avoid context switching while addressing kernel shortages in flow control.

2. METHODS

The study began by examining the differences between shared cores and dedicated cores by creating a workload looping program so that a time difference graph in milli seconds (ms) is obtained. The results of determining the speed difference determine the server recommendations later among multicore developments that use shared core or known as Hyper Threading (HT) on Intel. The next step determines the amount of default server system throughput using a simple sender and receiver program. The default standard number is used as a parameter for the success of the modification in steps called kernel bypass later. Modifications that consist of hashing signal receive which is intended to divide the signal into several queues which by default are one queue on the ethernet card. Modifications are made by using multiple IP (Internet Protocol) receivers which are intended to divide the load and the interrupt into several parts so that the throughput is increased and to carry out affinity on the thread so it is called a multi-thread receiver. This modification also overcomes the limitation of accepting interrupts, which is almost impossible to accept data packets larger than each CPU processing because it is an interrupt for an ethernet or peripheral. The next modification is performed by using a bind port that is intended to create a multi socket listener so that it adjusts the analysis (Rivera et al., 2014) [7].

To increase server reliability, an additional measurement parameter is given, namely RTT (Round Trip Time) or the time required for the server to process the packet and return it. RTT is obtained by modifying existing sender and receiver programs that are used to indicate the time in milliseconds (ms). Modifications in this stage only use the principle of pooling. Pooling used is applying 2 types of pooling in the kernel by calling the so_busy_pooling function and using pooling inside code principle in the body program.

2.1. Modifications to Reduce Throughput

Resource Utilization Multicore Server with Kernel Bypass take over the main functions of the kernel on the network using a new method by trying the hashing technique on the signal that aims to divide the queue to be directed at the specific CPU as shown in Figure 1 but the compatibility of the ethernet card is main problem with this hashing method.

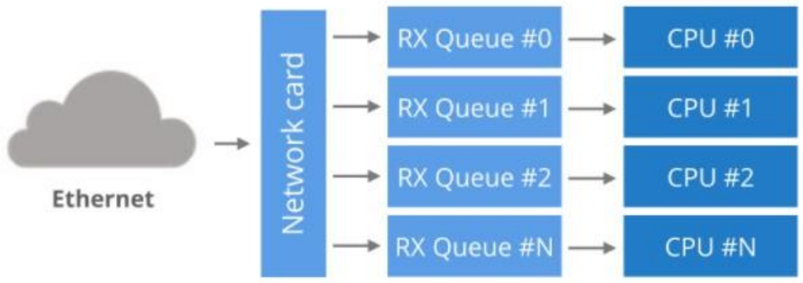
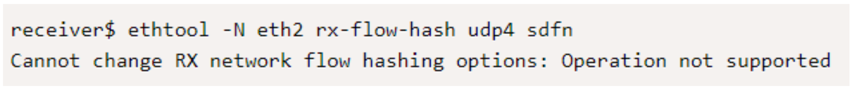


Figure 1.Hashing signal process

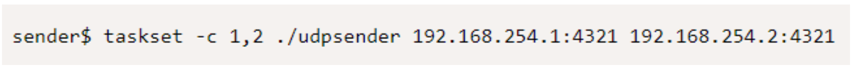
This modification is carried out with the rx-flow-hash command followed by the division tupple, i.e. ip address with the udp4 command and tupple port with the sdfn command as show in figure 2. In this paper command sdfn not support. As describe above compability of ethernet card is main problem to do more.



```
receiver$ ethtool -N eth2 rx-flow-hash udp4 sdfn
Cannot change RX network flow hashing options: Operation not supported
```

Figure 2. rx flow hash modification

The next modification uses multi IP (Internet Protocol) aimed at dividing the load and dividing the CPU load so as to overcome CPU limitations in processing or processing IRQ (Interupt Re Quest). This modification is common to perform and is not a new modification. In this study the modification was undertaken by adding a new ethernet card with different IPs and the sender was directed into 2 different Ips in figure 3



```
sender$ taskset -c 1,2 ./udpsender 192.168.254.1:4321 192.168.254.2:4321
```

Figure 3. multiple ip modification

The concept of multi threading sender was adopted to make the next modification as multi threading receiver. Same concept diferrent object to process more througput with couple with multiple IP modification. This modification is performed with taskset -c 1,2 ./udpreceiver1 0.0.0.0:4321 2 which shows activating CPU 1 and 2 to run the udp receiver program. The use of multi-thread receiver is the application of research (Rivera et al., 2014) [7]. The application of this modification allows a decrease in throughput so that it requires a modification to the socket discriptor to avoid multiple cores locking describe in next modification.

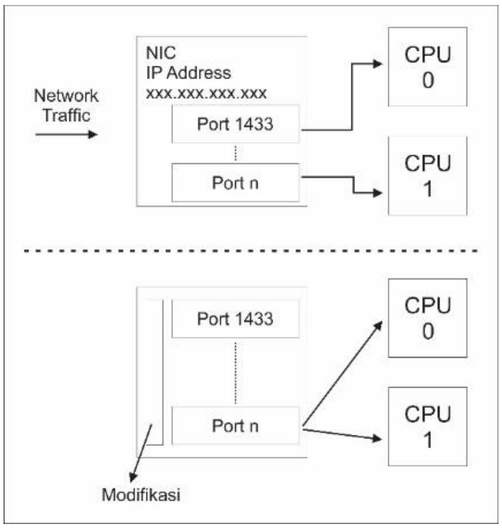
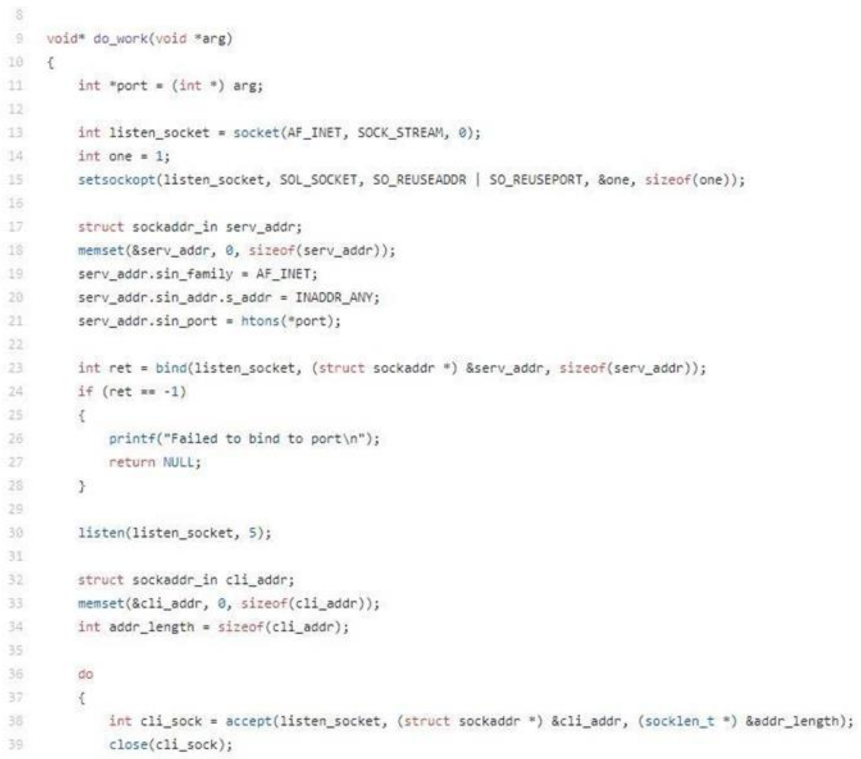


Figure 4. Listener port modification Process

Resource Modification on Multicore Server with Kernel Bypass using a program to overcome the struggle for lock as shown in Figure 4 is using multi listener so that each thread uses a different listenner with the result that even though using the same port there is no bottleneck. This multi listenner modification is undertaken before the accept () function. Accepting connections generally does not fork or bind socket, so all cores use waiting mode. In this study also applied affinity to IRQ NICs as a companion so as to improve the performance of the process as carried out by (Tsai et al., 2012) [8].



```
void* do_work(void *arg)
{
    int *port = (int *) arg;

    int listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    int one = 1;
    setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &one, sizeof(one));

    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(*port);

    int ret = bind(listen_socket, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
    if (ret == -1)
    {
        printf("Failed to bind to port\n");
        return NULL;
    }

    listen(listen_socket, 5);

    struct sockaddr_in cli_addr;
    memset(&cli_addr, 0, sizeof(cli_addr));
    int addr_length = sizeof(cli_addr);

    do
    {
        int cli_sock = accept(listen_socket, (struct sockaddr *) &cli_addr, (socklen_t *) &addr_length);
        close(cli_sock);
```

Figure 5. source code of binding port

2. 2 Modifications to Reduce Latency

This paper use modified client server program to show latency. In client program send packets and measure the time until response back. Server waits for packets and echoes them back to the source. In the system design to reduce latency, modifications will be made to split the queue from interrupt rx to be directed to the CPU list which aims to reduce excessive interrupt handling on 1 (one) core. Interrupts will be placed sequentially on the CPU, RX queue # 0 to CPU # 0, RX queue # 1 to CPU # 1 and so on as shown in Figure 6. This is also another control besides the irq_balance solution.

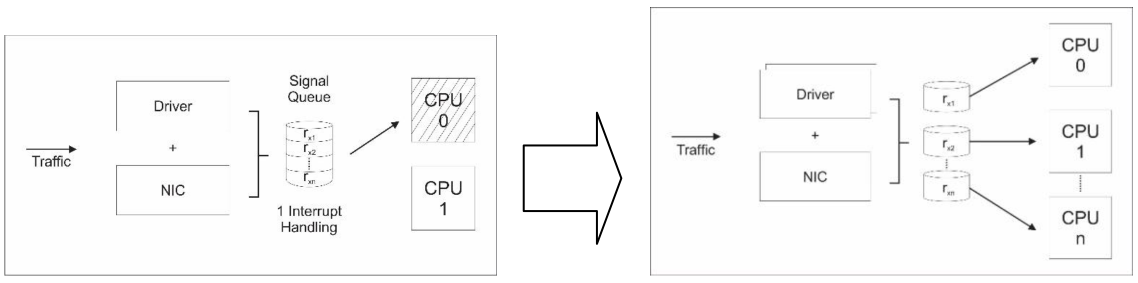
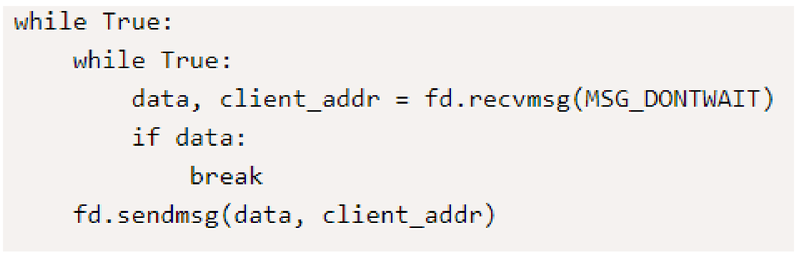


Figure 6. Interupt handlilng modification

Modifications also use the pooling technique of network traffic to avoid context switching. Pooling settings are done to avoid increasing processing time when a process is undertaken by a different CPU. Pooling is performed in two stages, that is through pooling the kernel that has been integrated in the linux kernel 3.11 or SO_BUSY_POLL.

The next Pooling stage is the one that is implemented in the receiver's body program so that pooling becomes more effective. Pseudo Code show in figure 7. All modification is effective as indicated by a reduction in the average latency to the default of 65.83%.



```
while True:
    while True:
        data, client_addr = fd.recvmsg(MSG_DONTWAIT)
        if data:
            break
    fd.sendmsg(data, client_addr)
```

Figure 7. modified server program

3. RESULTS AND DISCUSSION

3.1. Throughput

Table 1 Comparison of throughput modification

No	Default	NUMA		Multiple IP		Multiple thread		Binding port	
		Throughput (pps)	Gap (%)	Throughput (pps)	Gap (%)	Througput (pps)	Gap (%)	Throughput (pps)	Gap (%)
1	451,3k	430k	4,72	609k	34,94	495k	9,68	1114k	246,84
2		428k	5,16	657k	45,58	480k	6,36	1147k	254,16
3		432k	4,28	649k	43,81	461k	2,15	1126k	249,50
4		425k	5,83	653k	44,69	486k	7,69	1138k	252,16
5		427k	5,38	640k	41,81	490k	8,58	1142k	253,05
6		431k	4,50	651k	44,25	477k	5,69	1129k	250,16
7		428k	5,16	655k	45,14	485k	7,47	1131k	250,60
8		431k	4,50	623k	38,05	491k	8,80	1136k	251,72
9		433k	4,05	641k	42,03	481k	6,58	1121k	248,39

Table 1 shows the overall results of the modifications made in increasing throughput. Modifications made can beat the NUMA method that has been used to increase throughput. The use of multiple receiver threads experiences obstacles as exemplified by (Rivera et al., 2014)[7], adopted to modification with binding port that can provide significant results in the process. The results of the overall modification as set out in table 1, if calculated in an average will result in an increase of 250.44%.

3.2. Latency

Table 2 Comparison of latency modification

no	Default (ms)	NUMA	Gap (%)	Busypool (ms)	Gap (%)	Pooling inside code (ms)	Gap (%)
1	57,224	56,646	1,01	49,886	12,82	37,426	65,40
2	60,170	55,216	8,23	50,224	16,53	40,399	67,14
3	61,892	54,891	11,31	50,960	17,66	39,086	63,15
4	58,672	53,228	9,28	49,706	15,28	36,128	61,58
5	60,065	49,781	17,12	50,676	15,63	38,412	63,95
6	57,640	52,248	9,35	50,469	12,44	39,018	67,69
7	61,042	52,817	13,47	49,102	19,56	40.124	65,73
8	58,671	50,890	13,26	51,610	12,03	38,798	66,13
9	61,672	51,743	16,10	49,718	19,38	41,006	66,49

Table 2 shows the overall results of the modifications made in reducing latency. Modifications made after pinning and comparing using the so_busy_pool kernel. Pooling which are performed on the body server program avoid blocking on.

4. CONCLUSIONS

Systems developed using kernel bypass are able to increase server capabilities as shown by increasing throughput and decreasing latency compared to the default kernel. The results of the experiment, the increase in average throughput was 250.44% and the decrease in average latency was 65.83%.

REFERENCES

[1] Shambharkar, S. A. (2015). A Study on Setting Processor or CPU Affinity in Multi-Core Architecture for Parallel Computing. International Journal of Science and Research, 4(5), 2013–2016.

[2] Bo, Z. (2016). Analysis of the Resource Affinity in NUMA Architecture for High Performance Network. 2016 5th International Conference on Measurement, Instrumentation and Automation, 547–550.

[3] Diener, M., Cruz, E. H. M., Alves, M. A. Z., Navaux, P. O. A., Busse, A., & Heiss, H. U. (2016). Kernel-Based Thread and Data Mapping for Improved Memory Affinity. IEEE Transactions on Parallel and Distributed Systems, 27(9), 2653–2666. https://doi.org/10.1109/TPDS.2015.2504985

[4] Paul, M. V. V., Bhattacharjee, R., & Rajesh, R. (2014). Traffic capture beyond 10 Gbps: Linear scaling with multiple network interface cards on commodity servers. Proceedings - 2014 International Conference on Data Science and Engineering, ICDSE 2014, 194–199. https://doi.org/10.1109/ICDSE.2014.6974636

[5] Majo, Z., & Gross, T. R. (2013). (Mis) Understanding the NUMA Memory System Performance of Multithreaded Workloads. IEEE International Symposium on Workload Characterization (IISWC), 1–8.

[6] Tang, L., Mars, J., Zhang, X., Hagmann, R., Hundt, R., & Tune, E. (2013). Optimizing Google's warehouse scale computers: The NUMA experience. Proceedings - International Symposium on High-Performance Computer Architecture, 188–197. https://doi.org/10.1109/HPCA.2013.6522318

[7] Rivera, D., Ach, E., & Bustos-jim, J. (2014). Analysis of Linux UDP Sockets Concurrent Performance. 2014 33rd International Conference of the Chilean Computer Science Society, 65–69.

[8] Tsai, W. Y., Huang, N. F., & Hung, H. W. (2012). A port-configuration assisted NIC IRQ affinitization scheme for multi-core packet forwarding applications. GLOBECOM - IEEE Global Telecommunications Conference, 2547–2552. https://doi.org/10.1109/GLOCOM.2012.6503500

[9] Abel, F., Hagleitner, C., & Verplanken, F. (2012). Rx stack accelerator for 10 GbE integrated NIC. Proceedings - 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, HOTI 2012, 17–24. https://doi.org/10.1109/HOTI.2012.18

[10] Angelo, G. D., Marchetti-spaccamela, A., & Cnr, I. (2016). Multiprocessor Real-Time Scheduling with Hierarchical Processor Affinities. 2016 28th Euromicro Conference on Real-Time Systems, 237–247. https://doi.org/10.1109/ECRTS.2016.24

[11] Fusco, F., & Deri, L. (2010). High Speed Network Traffic Analysis with Commodity Multi-core Systems. proceedings of the 10th ACM SIGCOMM conference on Internet measurement, 218–224.

[12] Galagan, V., Yurchenko, O., Preobrazhensky, E., Zhuravkov, P., & Dombrougov, M. (2013). Multi-gigabit intel-based software routers. Proceedings - RoEduNet IEEE International Conference. https://doi.org/10.1109/RoEduNet.2013.6714193

[13] Gu, Q., Wen, L., Dai, F., Gong, H., Yang, Y., Xu, X., & Feng, Z. (2014). StackPool: A high-performance scalable network architecture on multi-core servers. Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013, 17–28. https://doi.org/10.1109/HPCC.and.EUC.2013.13

[14] Hanford, N., Ahuja, V., Farrens, M., Ghosal, D., Balman, M., Pouyoul, E., & Tierney, B. (2014). Analysis of the Effect of Core Affinity on High-Throughput Flows. 4th International workshop on Network-Aware Data Management, 9–15.

[15] Hanford, N., Ahuja, V., Farrens, M., Ghosal, D., Balman, M., Pouyoul, E., & Tierney, B. (2015). Improving network performance on multicore systems : Impact of core affinities on high throughput flows. Future Generation Computer Systems. https://doi.org/10.1016/j.future.2015.09.012

[16] Hanford, N., Farrens, M. K., Pouyoul, E., & Tierney, B. (2014). Characterizing the Impact of End-System Affinities On the End-to-End Performance of High-Speed Flows. ACM/IEEE symposium on Architechtures for Networking and comunications System, 259–260.

[17] He, P., Wang, J., Deng, H., & Zhang, W. (2010). Balanced locality-aware packet schedule algroithm on multi-core network processor. Proceedings of the 2010 2nd International Conference on Future Computer and Communication, ICFCC 2010, 3, 248–252. https://doi.org/10.1109/ICFCC.2010.5497642

[18] Huang, C., Yu, X., & Luo, H. (2010). Research on high-speed network data stream capture based on multi-queue NIC and multi-core processor. ICIME 2010 - 2010 2nd IEEE International Conference on Information Management and Engineering, 2, 248–251. https://doi.org/10.1109/ICIME.2010.5477440

[19] Jie, L., Shuhui, C., & Jinshu, S. (2016). Implementation of TCP large receive offload on multi-core NPU platform. 2016 International Conference on Information and Communication Technology Convergence, ICTC 2016, 258–263. https://doi.org/10.1109/ICTC.2016.7763481

[20] Jin, H. W., Yun, Y. J., & Jang, H. C. (2008). TCP/IP performance near I/O bus bandwidth on multi-core systems: 10-Gigabit ethernet vs. multi-port gigabit ethernet. Proceedings of the International Conference on Parallel Processing Workshops, 87–94. https://doi.org/10.1109/ICPP-W.2008.33

[21] Li, Y., & Qiao, X. (2011). A Parallel Packet Processing Method on Multi-core Systems. 2011 10th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, 78–81. https://doi.org/10.1109/DCABES.2011.11

[22] Nelms, T., & Ahamad, M. (2010). Packet Scheduling for Deep Packet Inspection on Multi-Core Architectures. Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on, 1–11. https://doi.org/10.1145/1872007.1872033

[23] Orosz, P. (2012). Improving Packet Processing Efficiency on Multi-core Architectures with Single Input Queue. Carpathian Journal of Electric and Computer Engineering, 5, 44–48.

[24] Sibai, F. N. (2010). Simulation and performance analysis of multi-core thread scheduling and migration algorithms. CISIS 2010 - The 4th International Conference on Complex, Intelligent and Software Intensive Systems, 895–900. https://doi.org/10.1109/CISIS.2010.17

[25] Tsujita, Y., Hori, A., & Ishikawa, Y. (2014). Affinity-Aware Optimization of Multithreaded Two-Phase I / O for High Throughput Collective I / O. international Conference on High Performance Computing & Simulation, 210–217.

[26] Velkoski, G., Ristov, S., & Gusev, M. (2013). Affinity-aware HPC Applications in Multichip and Multicore Multiprocessor. Information Technology Interfaces (ITI), Proceedings of the ITI 2013, 95–100.

[27] Zou, H., Sun, X., Ma, S., & Duan, X. (2012). A Source-aware Interrupt Scheduling for Modern Parallel I / O Systems. 2012 IEEE 26th International Parallel and Distributed Processing Symposium, 156–166. https://doi.org/10.1109/IPDPS.2012.24