future internet

MDPI

Article

An Adaptive Throughput-First Packet Scheduling Algorithm for DPDK-Based Packet Processing Systems

Chuanhong Li [1,2], Lei Song [1,2,*] and Xuewen Zeng [1,2]

[1] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; lich@dsp.ac.cn (C.L.); zengxw@dsp.ac.cn (X.Z.)
[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China
* Correspondence: songl@dsp.ac.cn

Abstract: The continuous increase in network traffic has sharply increased the demand for high-performance packet processing systems. For a high-performance packet processing system based on multi-core processors, the packet scheduling algorithm is critical because of the significant role it plays in load distribution, which is related to system throughput, attracting intensive research attention. However, it is not an easy task since the canonical flow-level packet scheduling algorithm is vulnerable to traffic locality, while the packet-level packet scheduling algorithm fails to maintain cache affinity. In this paper, we propose an adaptive throughput-first packet scheduling algorithm for DPDK-based packet processing systems. Combined with the feature of DPDK burst-oriented packet receiving and transmitting, we propose using Subflow as the scheduling unit and the adjustment unit making the proposed algorithm not only maintain the advantages of flow-level packet scheduling algorithms when the adjustment does not happen but also avoid packet loss as much as possible when the target core may be overloaded Experimental results show that the proposed method outperforms Round-Robin, HRW (High Random Weight), and CRC32 on system throughput and packet loss rate.

Keywords: packet scheduling; HRW; DPDK; throughput; packet loss rate

Citation: Li, C.; Song, L.; Zeng, X. An Adaptive Throughput-First Packet Scheduling Algorithm for DPDK-Based Packet Processing Systems. *Future Internet* **2021**, *13*, 78. https://doi.org/10.3390/fi13030078

Academic Editor: Eirini Eleni Tsiropoulou

Received: 29 January 2021
Accepted: 18 March 2021
Published: 19 March 2021

1. Introduction

With the ever-increasing network traffic, the demand for high-performance packet processing systems with different purposes, such as healthcare monitoring [1], traffic accident detection and condition analysis [2], and privacy protection [3], has sharply increased in recent years. High parallelism provided by the multi-core processor makes it a promising option for achieving both high throughput and good scalability since multiple processing cores can be used to deal with the traffic in parallel [4–6]. However, the packet processing systems based on general-purpose multi-core processors usually depend on the TCP/IP stack of the operating system. Frequent switch between user space and kernel space, multiple memory copies, and frequent interruptions degrade the performance of packet processing rapidly [6–11]. Some special network processors equipped with accelerators, such as PowerNP [12] and Cavium [13], can provide extreme performance through targeted optimizations. The expensive price, long development, and poor scalability make them unable to meet various requirements. Fortunately, the emerging high-performance packet I/O engines such as DPDK (Data Plane Development Kit) [14] and Netmap [15], using huge-pages, zero-copy, batch processing, and polling, alleviate or avoid the issues faced by general-purpose multi-core processors, resulting in great performance improvement. The appearance of packet I/O frameworks has made it a reliable

choice to achieve high-performance packet processing on general-purpose multi-core processors. Besides, packet processing based on a programmable data plane is also a promising option since it can provide line-rate packet processing performance [16].

For high-performance packet processing systems based on multi-core processors, the packet scheduling algorithm is vital, since it directly decides the load balance of the system [4,5,17,18], which has a great impact on system throughput performance. A poor scheduling algorithm may overload some cores while staves other cores, thus reducing the performance of the system. According to the scheduling granularity, packet scheduling algorithms can be roughly categorized into two categories: packet-level and flow-level [4,5].

Packet level-based packet scheduling algorithms, such as Round-Robin (RR) [10] and The Least Load First [19], use the packet as the scheduling unit. They distribute each packet independently among cores to achieve perfect load balancing. Although they achieve better load balancing, these methods still have two shortcomings: (1) packets belonging to the same flow are scheduled to different cores, resulting in poor data locality; and (2) since packets of the same flow are processed in parallel on many cores, if flow statistics need to be maintained, synchronization is needed to ensure the consistency between cores, which is costly.

Flow level-based packet scheduling algorithms consider the flow as the scheduling unit. Flow here means a set of bidirectional packets sharing the same N-tuple, for example, the same 5-tuple (source/destination IP, source/destination port, protocol). Generally, hashing is used by flow level schemes to schedule packets from the same flow to a specific core [4,5,20–24]. One or more fields in the packet header are used by the hash function to guide packet scheduling. According to the definition of the flow, packets belonging to the same flow have the same packet header and will be mapped to the same target core. Compared with packet-level methods, packets from the same flow are processed on the same core, thus data locality can be maintained and synchronization is not needed.

Many studies demonstrate that flow-level packet scheduling is more suitable for packet processing applications since the data and instruction locality can be explored to optimize processing performance. However, the dynamic nature of the network traffic and skewed flow size make flow level-based packet scheduling algorithms very easy to overload some cores while the other cores have no work to do [6,25,26], thus resulting in packet loss and lower throughput. Therefore, a hash-based flow-level packet scheduling algorithm cannot guarantee achieving higher throughput and lower packet loss rates.

In this paper, we propose an adaptive throughput-first packet scheduling algorithm for DPDK-based packet processing systems. Throughput-first means throughput is the primary purpose of our proposed method even though in some cases load imbalance will occur. Combining the feature of DPDK burst-oriented packet reception and transmission, we use Subflow as our scheduling unit and adjustment unit. The definition of Subflow is as follows: A set of packets belonging to the same flow (having the same N-tuple) in the burst is defined as a Subflow, whose size is within the range of [1,bsz_rd], where bsz_rd is the burst size of receiving packets. We use HRW [22] algorithm to guarantee packets belonging to the same Subflow go to the same processing core since packets in the Subflow have the same packet headers and thus have the same packet identifiers used to compute weights of each processing core. By monitoring the occupancy of the core's queue, named the load factor in this paper, the system decides whether the adjustment is needed. It should be noted that the adjustment is only valid for the current burst, not for others. In summary, the main contributions of this work are as follows:

- We first propose to use Subflow as the scheduling unit, which makes the packet scheduling algorithm maintain the advantages of flow-level packet scheduling algorithms in most cases where the adjustment does not happen.
- We first propose to use Subflow as the adjustment unit, which can avoid packet loss as much as possible when the target core may be overloaded.

- An adaptive throughput-first packet scheduling algorithm is proposed and demonstrated that can achieve higher throughput and lower packet loss rate compared with RR, CRC32, and HRW.

The rest of the paper is organized as follows. We introduce some related work in Section 2. Then, we present the Subflow-based packet scheduling algorithm in Section 3. In Section 4, we evaluate our proposed method. We conclude our paper with a discussion of future work in Section 5.

2. Related Works

In recent years, abundant works have been done on packet scheduling algorithms due to the vital role they play in packet processing systems based on multi-core processors [4,5,8,9]. In this section, we briefly summarize these works.

To achieve excellent load balancing, packet-level scheduling algorithms such as Round-Rubin [10] are applied. However, poor cache locality [6,11] and packets reordering [6] make them unsuitable for packet processing applications. On the contrary, flow-level packet scheduling is a more popular choice since it can achieve a high cache hit rate, thus achieving a higher throughput [5,17,27,28]. Among these flow-level schemes, the low overhead makes the hash-based design a canonical scheme that is used to dispatch the packets that share the same N-tuples to a target core [5,18,20,29,30]. There are also many studies on finding the best hash functions [19,20,31]. For example, the authors of [19] evaluated the performance of different hash functions used in Internet traffic splitting and showed that CRC16 outperforms the other hash schemes. A similar result is also found in the literature [6]. Besides, the authors of [21] showed that CRC32 achieves a better load balance performance than CRC16. High Random Weight (HRW) [22] is another well-known hash method. The proven low overhead, better data locality, better load balancing, and minimal disruption make it stand out [17,22].

The hash-based flow level packet scheduling algorithms such as CRC and HRW schedule packets belonging to the same flow to the same processing core. However, the dynamic nature of the network traffic and skewed flow size make these methods vulnerable to lead to load imbalance [6,23]. The authors of [30] demonstrated that hash-based packet scheduling schemes cannot balance the load among the processing cores when the flow size distribution is skewed. Therefore, some adjustments are needed when the load imbalance occurs.

An adaptive HRW is proposed in [23]. Faced with different traffic patterns, it monitors the processor utilization periodically. When the load imbalance is present, which means the processor utilization exceeds the thresholds configured in advance, the AHRW adjusts the weights of the processing cores, which in turn schedules packets to different cores to achieve load balancing. Instead of migrating any flows, the authors of [6,30] first identified the aggressive flows according to the flow statistic and then strictly limited the migration to the top aggressive flows. However, the migrated flows may overload the new allocated core [5]. When the imbalance occurs, Kuang et al. [17] believed that the hashing decision should be adjusted at once, and the new incoming packets should be rescheduled to a new processing core. The real-time weighted queue length of each processing core is the signal to show the load imbalance. Similar methods are also presented in [4,24], except that the authors of [4] used the real system utilization to detect the load imbalance, which is costly.

Our proposed method is similar to these adaptive methods but different in the following aspects: First, the adjustment unit we use is Subflow, not the flow, therefore the adjustment granularity is relatively small. Second, we adopt load factor to depict the core utilization, which is less costly, while some others use the real system utilization. Third, the purpose of our method is to achieve higher throughput and lower packet loss rates, while some others are aimed to achieve better load balancing.

3. The Proposed Packet Scheduling Algorithm

To boost the packet processing performance, emerging packet I/O frameworks apply batch processing technology to amortize the overhead of each packet processing and improve cache locality [25,26,32–34], as does DPDK. Based on the load balancing example provided by DPDK [35], we develop a packet processing system, as shown in Figure 1. Combined with Figure 1, we briefly introduce the process of DPDK burst-oriented packet receiving and sending.

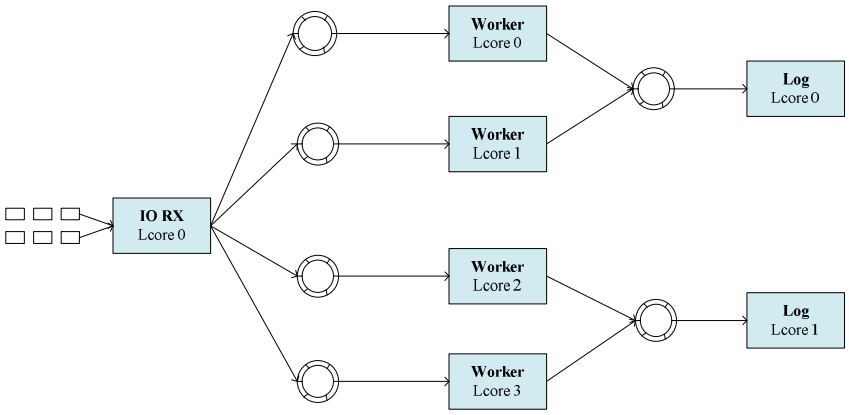

Figure 1. The framework of packet processing system.

The packet processing system, as shown in Figure 1, mainly contains three parts:

- Packet Collecting Module: This module contains an I/O core, which is responsible for receiving packets from the NIC and dispatching packets according to some load balancing algorithms. The packet scheduling algorithm proposed in this paper is a function of the I/O core.
- Packet Processing Module: This module is composed of some packet processing cores, called Worker in this paper. These Worker cores are responsible for some tasks such as protocol identifying, flow managing, payload parsing, etc.
- Log Module: This module contains a few cores, which are responsible for managing the information from the Worker cores, merging some logs from the same flow, etc.

DPDK uses a poll-based mechanism to retrieve packets in a burst from the NIC (Network Interface Controller). The burst size configured when the programming initializes can be changed as needed. The number of packets fetched from the NIC should not be larger than the burst size in each polling. The poll-based mechanism can avoid the unpredictable overhead caused by the interruption [36]. Packets are distributed to a processing core, also called a worker, according to some load balancing algorithms such as RSS [37,38], HRW [22], and CRC32. Although packets are retrieved in a burst, they are still processed in a 1-by-1 fashion. Packets dispatched to the same processing core are placed in the out queue allocated to each processing core until the number of packets reaches the transmitting burst size or the timer is expired [25]. At last, packets are delivered to the processing core in a burst. The number of packets sent should not be larger than the burst size. It should be noted that the burst size of receiving packets can be different from that of sending packets.

As mentioned in [30], the locality of network traffic makes hash-based flow-level packet scheduling unable to achieve load balancing between processing cores. Some overloaded cores lead to packet loss, degrading the throughput performance of the packet processing system. However, the packet-level packet scheduling results in poor cache locality, which cannot achieve desirable throughput performance. Thus, a new packet

scheduling algorithm that not only considers the load balancing but also the cache locality is needed to achieve high throughput for packet processing systems.

Combined with DPDK burst-oriented packet receiving and transmitting, we propose an adaptive Subflow-based packet scheduling algorithm. We envision the proposed method has the following properties:

- On the one hand, it can still maintain the advantages of flow-level packet scheduling when the adjustment does not happen.
- On the other hand, when the target core is likely to be overloaded, the proposed scheduling algorithm can change the scheduling decision at once, avoiding the incoming packets scheduled to the overloading core to achieve higher throughput and lower packet loss rates.

From the definition of the Subflow, using hash-based design, packets that belong to the same Subflow in the same burst will be distributed to the same target core. In most cases, where the adjustment is not needed, Subflows from different bursts will also be scheduled to the same target core since these packets have the same 5-tuple. Therefore, the proposed method can maintain the advantages of flow-level packet scheduling when the adjustment does not happen.

When the load factor of the target core exceeds the pre-configured value, meaning that the core is likely to be overloaded, the proposed algorithm will change the scheduling decision at once; hence, packets in the Subflow will not be scheduled to the core. Using Subflow-based adjustment, we reduce the probability of packet loss, thus achieving higher throughput.

The hash-based design is adopted in our proposed method with an adaptive Subflow-based adjustment. The hash algorithm we use is High Random Weight (HRW), first presented in [22], which is used to map a request to a specific server in the server cluster. The lower overhead and better load balancing make it a popular choice for packet scheduling [17,22,23]. In this paper, the definition of HRW function is as follows: Suppose $\vartheta(p_i, \mathrm{C})$ is a pseudo-random weight function, where $\vartheta: \mathrm{P} \times \{1,2,\ldots,\mathrm{bsz\_rd}\} \rightarrow [0, \mathrm{n}]$. Here, C is the set of processing cores, written as $\mathrm{C} = (c_1, c_2, \ldots, c_N)$, where N is the number of them. P is the set of packets fetched from NIC. ϑ is assumed to generate a random value in [0, n] with uniform distribution. For each packet p_i in the burst, the target core is calculated as below:

$$LB_{HRW}(p_i, \mathrm{C}) = c_j, \tag{1}$$

where $c_j = \max_{1 \le j \le N} \vartheta(p_i, c_j)$ and LB_{HRW} is the scheduling function used in this paper.

The packets sharing the same 5-tuples should have the same weight, thus the same target core if no adjustment happens. As mentioned in [4], the pseudo-random weight function plays a vital role in the performance of the HRW algorithm; hence, we follow the one provided in [22]:

$$\vartheta(p_i, \mathrm{C}) = \left(A \cdot \left((A \cdot T + B) \oplus D(p_i)\right) + B\right) mod 2^{31}, \tag{2}$$

where A = 1103515245 and B = 12345. $D(p_i)$ is a 31-bit digest of the packet identifier (ID), and the packet ID is generated based on 5-tuple of the packet. T is the ID of the processing core.

For each packet in the burst, the LB_{HRW} will generate a target value to which the packet should be dispatched. When the target core is obtained, we check the number of packets waiting in its ring, written as n_mbufs. If n_mbufs is larger than zero, i.e., there are some packets queued in the ring waiting to be sent to the target core, we just add the current packet to the ring to maintain the cache locality as much as possible. The adjustment will only be triggered when the following two conditions are both met: (1) n_mbufs is zero; and (2) the utilization of the target core, named *load factor* in this paper, exceeds the pre-configured threshold. Borrowed from the design of the hash function, we use *load*

factor to define the occupancy of the cores' queue as follows: $\tau = \frac{Q_{used}}{Q_{total}}$, where Q_{total} is the total number of rings allocated to the core and Q_{used} is the number of rings occupied by the packets waiting to be processed. We discuss the selection of a suitable threshold in the Experimental Section 4.1.

It is well known that the adjustment means another computing of the hash weight. At the same time, there is the case that the target core generated by the LB_{HRW} is still an overloading core. To avoid this issue, we introduce an adaptive hash function when the adjustment is needed. The final weight function ϑ' can be formulated as: $\vartheta'(p_i, C_{N_i}, \mu) = \mu \cdot \vartheta(p_i, C_{N_i})$, where C_{N_i} is a subset of C.

The adjustment factor μ is defined as $\mu = 1 - \tau$, which shows the available position in each queue. The adaptive scheduling function LB_{AHRW} is depicted as:

$$LB_{AHRW}(p_i, C_{N_i}, \mu) = c_j, \tag{3}$$

where $c_j = \max_{1 \le j \le N_i} \vartheta'(p_i, c_j, \mu)$.

As mentioned above, our method uses the Subflow as the scheduling unit and packets in the same Subflow have the same 5-tuple, thus producing the same processing core. In the case where the adjustment does not happen, we can store the decision for the Subflow to avoid computing the hash weight for each packet repeatedly. Meanwhile, when the adjustment is needed, we still can store the adjustment decision for the Subflow since the Subflow is also the adjustment unit, which means the adjustment for a Subflow happens only once in the current burst. Subsequent packets in the burst belonging to the Subflow follow the same decision. It is negligible that the memory overhead to store the decision since our packet scheduling algorithm is based on the Subflow. Besides, it also can be used repeatedly for different bursts.

Finally, the whole algorithm is described in Algorithm 1.

Algorithm 1 Subflow-based packet scheduling

```
1:  target[bsz_rd] ← {−1}
2:    for each packet in the burst fetched from the NIC
3:    index ← p_i%bsz_rd
4:    c_i ← target[index]
5:    if c_i == −1
6:          c_i ← LB_HRW(p_i, C)
7:          target[index] ← c_i
8:    end if
9:    n_mbufs_i ← Get_pktnum(c_i)
10:    if n_mbufs_i == 0
11:        load factor_i ← Get_utilization(c_i)
12:        if load factor_i > Threshold
13:            c_i ← LB_AHRW(p_i, C_{N_i}, μ)
14:          target[index] ← c_i
15:      end if
16:    end if
17: end for
```

For a burst of packets fetched from the NIC, Line 1 clears the target array to guarantee that the previous scheduling strategy is not active for the current burst. The initial value can be any value except the logical number of Worker cores. For simplicity, we set the initial values of the array to −1, indicating that the scheduling decision is not done. For each packet in the burst, we use the packet ID to mod the size of the array to get the index of the value, thus getting the scheduling decision, as Line 3 shows. If the decision is not done for the packet, we use the load balancing algorithm to get the target processing core

and record the decision in the array. Lines 5–8 show the procedure. When the processing core is obtained, we use the function provided by DPDK to get the number of packets waiting in the out queue, as Line 9 shows. If the number of packets is larger than zero, we apply the greed strategy and directly schedule the packet to the core; otherwise, the queue utilization of the target core is needed, as Line 11 shows. Line 12 shows that, if the load factor of the core is larger than a pre-configured value, the adjustment of the scheduling decision is needed. According to the proposed method in this paper, another processing core is chosen, and the decision is recorded in the array. Lines 12–15 show the procedure. When Line 12 is not satisfied, meaning the target core has enough capacity to handle these packets, we distribute the packet to it directly.

One more thing that should be noted is that the adjustment will lead to packets belonging to the same flow being processed by different processing cores; thus, poor cache performance degrades the throughput of the system. Furthermore, packets may need to be reordered for some applications, but not for ours. In our packet processing system, the processing core is responsible for parsing the payload of the packets, and the information needed is sent to the Log module. The Log module is responsible for the integration of the flow information for users. Therefore, in our system, we pay no attention to the problem of packet out-of-order.

4. Results

In this section, we evaluate our proposed method with RR, the original HRW, and CRC32. The test server we use is an Intel server equipped with two 12-core Intel(R) Xeon(R) Silver 4116 processors running at 2.10 GHz. We use Ixia as the packet generator and the traffic is mirrored to our test server through specific ports on the switch by a 10 Gb/s fiber-based link. The DPDK version we use is 17.05.2. Since our proposed method depends on the Log cores to handle the issue of out-of-order, we assume that the processing capacity of the Log cores is not the bottleneck of the system throughput, which can be easily achieved by reserving enough cores.

4.1. The Selection of Threshold

As mentioned above, our proposed method will reschedule packets when some conditions are met, one of which is the load factor of the core exceeds a pre-configured threshold. The load factor represents the utilization of the queue of the core; thus, the pre-configured threshold should have the same definition, which is also a representation of the utilization of the queue. There is no doubt that the selection of the threshold has a great impact on the adjustment frequency. Therefore, we first verify the probability of packet drop under different thresholds. The results are shown in Table 1.

Table 1. The probability of packet loss rate under different thresholds.

Threshold (%)	Probability of Packet Drop (%)
75	73.4
50	55.4
25	42.4

As we can see, when the threshold is 50%, the average probability of packet drop is 55.4%, i.e., there is a higher probability of packet loss when the occupancy of the core's queue exceeds 50%. The larger is the threshold, the higher is that probability that a packet will be dropped. Meanwhile, a smaller threshold leads to adjustment happening frequently, which incurs a large scheduling overhead. Therefore, in this paper, we set the threshold of 25%. This is, only when the utilization of the core' queue exceeds a quarter will our proposed method adjust the scheduling decision.

4.2. Packet Distribution

The packet distribution is a direct flection of the load balancing performance of those hash-based packet scheduling algorithms. In our tests, the length of packets generated by Ixia are almost the same; therefore, we show the number of packets on each processing core (Worker) under different network loads to describe the load balancing performance of different scheduling algorithms, as shown in Figure 2.

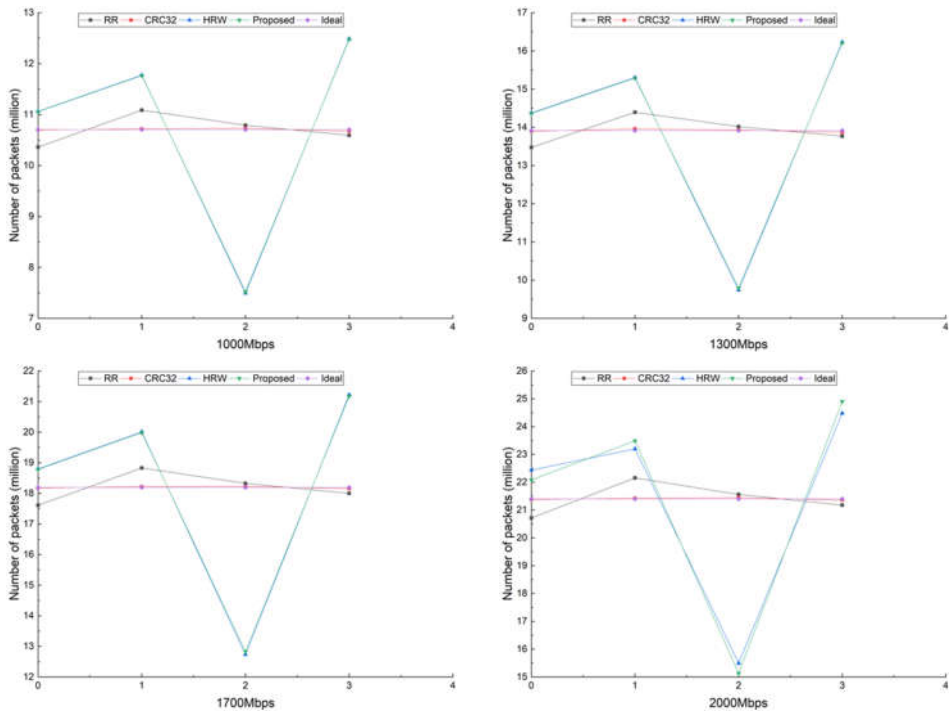

Figure 2. Number of packets on each processing core under different throughputs.

The Ideal curve represents the theoretical optimal allocation; in other words, the number of packets on each processing core is $\left\lceil \frac{pkt\_num}{core\_num} \right\rceil$, where $core\_num$ is the number of processing cores. In our test, $core\_num$ is 4. Intuitively, RR would achieve a similar distribution with Ideal; however, to be consistent with our Subflow-based packet scheduling, RR is performed on each burst, not for the whole traffic. Since the number of packets fetched from the NIC in the burst varies with each retrieval, the load distribution of RR is as Figure 2 shows.

As shown in Figure 2, the number of packets on each core is a little unbalanced for HRW and the proposed method, compared with RR and CRC32. As mentioned above, the primary purpose of our proposed method is to achieve higher throughput and lower packet loss rates. Only when the load imbalance leads to packet loss will the scheduling decision be adjusted. As we can see, CRC32 achieves an excellent load distribution, approaching the Ideal distribution, which shows that it is not a bad idea to use it as the scheduling algorithm for a packet processing system.

4.3. Throughput and Packet Loss Rate

The throughput of each method is evaluated in this subsection. Figure 3a shows the throughput when the packet loss is zero. while Figure 3b shows the result when the packet loss is not larger than one of ten thousand. In our tests, we maintain three counters on the I/O core: the first one is used to record the number of packets received; the second one is used to record the number of packets distributed to all Workers; and the last one is used

to record the number of packets dropping due to lack of position. Therefore, here, the packet loss is the real packet dropping for the entire traffic.

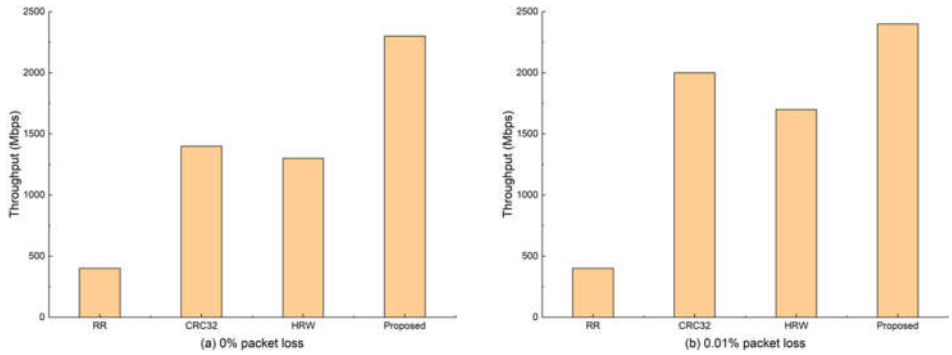

Figure 3. Throughput for different packet loss rates.

If we restrict the packet loss to zero, as shown in Figure 3a, the throughput performance of our proposed method achieves an average improvement of 4.75, 0.64, and 0.77 times, respectively, compared with RR, CRC32, and HRW. If a slight packet loss is allowed, where the packet loss rate is not larger than one of ten thousand, the throughput performance of our method still has an average improvement of 4.75, 0.2, and 0.41, respectively, compared with RR, CRC32, and HRW.

With the Subflow-based adjustment, some packets are rescheduled to the core, which is capable of handling them, leading to no packet loss, and thus improves the throughput performance of the packet processing system. As for other algorithms tested, overloaded cores result in packet loss, thus reducing the throughput performance. One may find that, under two cases, the throughput of RR is the same, which is lower than the other flow level-based packet scheduling algorithm, demonstrating that flow level-based packet scheduling algorithm is more suitable for packet processing applications.

To further evaluate the effectiveness of the proposed method, we use some special cases where the traffic is dispatched to one or two cores, i.e., there are one or two cores overloading while the others have no traffic. The test results are shown in Figures 4 and 5.

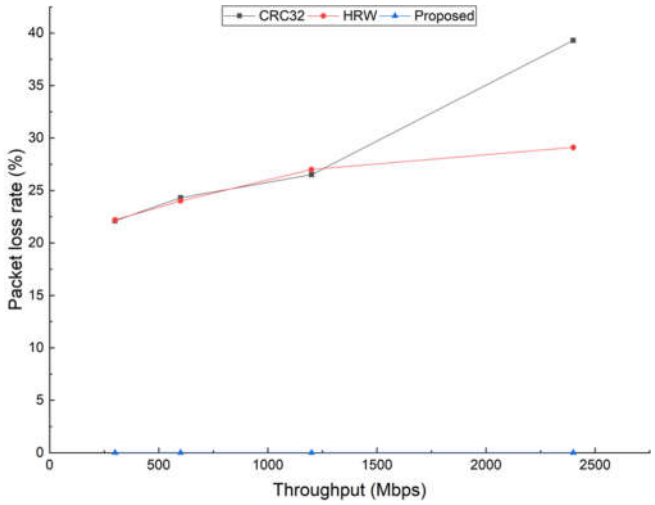

Figure 4. Packet loss rates under different throughputs when one core is overloaded while others have no traffic.

Figure 4 shows the packet loss rates under different throughputs when one core is overloaded and the others have no traffic. Even when the throughput is 300 Mbps (Million

bits per second), the packet loss rate is larger than 20% for HRW and CRC32. For our proposed method, the packet loss remains zero from 300 to 2400 Mbps. Under this special case, the improvement in throughput of our proposed method is around one order of magnitude when the packet loss rate is restricted to zero.

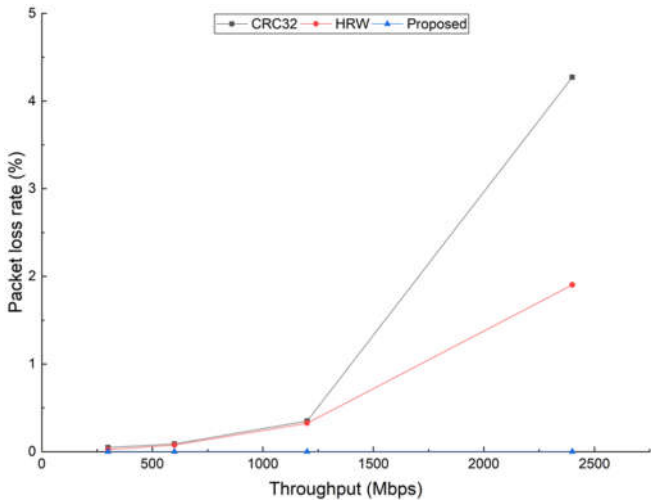

Figure 5. Packet loss rates under different throughputs when two cores are overloaded while others have no traffic.

Figure 5 shows the case where two cores overloaded, i.e., all traffic is distributed to two cores and the remaining have no traffic to process. The result is similar to Figure 4. When the throughput is 300 Mbps, there is a slight packet loss for HRW and CRC32, while, for our proposed method, the packet loss rate the packet loss remains zero from 300 to 2400 Mbps. It is interesting to note that, according to the results in Figures 4 and 5, under the condition that the throughput is the same, the packet loss rate is far smaller for the two cores overloading case than that of one core overloading. The reason is that, compared with the single traffic generated by Ixia for one core overloading, the packets generated for the two cores overloading case are dispatched to each core interleaved, giving the cores some gaps to breathe, thus having a lower packet loss rate.

The results in Figures 4 and 5 further demonstrate the effectiveness of our proposed method. With the Subflow-based adjustment, we not only can reduce the packet loss rate to zero when the throughput is low but also can improve the throughput to a high level.

4.4. Scheduling Overhead

Although the adjustment can reduce packet loss rate and improve the throughput, as shown in Algorithm 1, the adjustment also needs one more HRW, causing extra overhead of the packet scheduling algorithm. Therefore, we test the overhead of each scheduling algorithm. The result is shown in Figure 6.

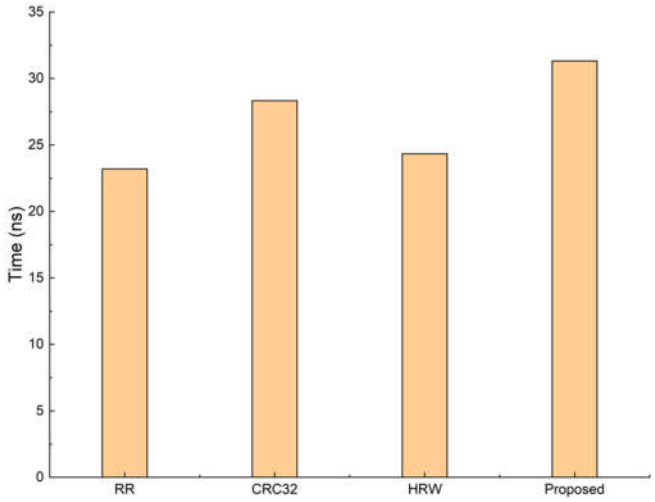

Figure 6. Average time spending on distributing a packet of different methods.

In our experiments, the average time spending on distributing each packet is 23.3, 28.4, 24.4, and 31.4 ns for RR, CRC32, HRW, and the proposed method, respectively. The overhead of our proposed method is 1.29 times that of HRW, which is acceptable considering the improvement in throughput. Compared with RR and CRC32, the increase in overhead is 0.35 and 0.11 times, respectively.

5. Conclusions and Discussion

To handle the ever-increasing network traffic, high-performance packet processing systems based on multi-core processors are a promising option. However, how to schedule packets to each processing core is crucial to make full use of the processing capacity of the system. In this paper, we propose an adaptive throughput-first packet scheduling algorithm for DPDK-based packet processing systems. Combined the burst-oriented packet receiving and transmitting, we propose to use the Subflow as the scheduling unit as well as the adjustment unit. By monitoring the occupancy of the core's queue, the system makes a scheduling decision as well as an adjustment decision if needed. With Subflow-based scheduling and adjustment, the proposed method gives the overloaded cores some gaps to breathe, thus reducing packet loss rate and improving the throughput of the system. The experimental results show that, compared with HRW, the throughput of our method achieves an average improvement of 0.77 times. Compared with RR and CRC32, the improvement is 4.75 and 0.64 times, respectively. Under some special cases, when only one core or two cores is overloaded, our method can achieve zero packet loss rate even under the throughput of 2400 Mbps, while CRC32 and HRW drop packets when the throughput is 300 Mbps. The extra overhead incurred by the adjustment is 0.29 times that of HRW, which is acceptable considering the improvement in throughput and packet loss rate.

It is well known that the adjustment of the scheduling decision may incur out-of-order packets. We assume that the out-of-order problem is solved by the Log module by merging these logs from the same flow in this paper. However, the passive effects on the Log module should be further evaluated in future work. At the same time, when the adjustment is needed, some other factors such as the protocol the target core must process should be considered to maintain cache affinity as much as possible. Moreover, with different configurations of hardware and software platforms, the effectiveness of the proposed method should be further evaluated in future works since these factors affect the performance of packet processing systems.

Author Contributions: Conceptualization, C.L.; methodology, C.L..; software, C.L.; validation, C.L., L.S.; writing—original draft preparation, C.L.; writing—review and editing, C.L., L.S., and X.Z.; visualization, C.L.; supervision, L.S.; project administration, X.Z.; funding acquisition, X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science and Technology Major Project, grant number XDC02070100.

Acknowledgments: We also gratefully acknowledge the editor and all reviewers for their valuable suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ali, F.; El-Sappagh, S.; Islam, S.M.R.; Ali, A.; Attique, M.; Imran, M.; Kwak, K.S. An intelligent healthcare monitoring framework using wearable sensors and social networking data. *Futur. Gener. Comput. Syst.* **2020**, *114*, 23–43.
2. Ali, F.; Ali, A.; Imran, M.; Naqvi, R.A.; Siddiqi, M.H.; Kwak, K.S. Traffic accident detection and condition analysis based on social networking data. *Accid. Anal. Prev.* **2021**, *151*, 105973.
3. Ko, H.; Lee, H.; Kim, T.; Pack, S. LPGA: Location Privacy-Guaranteed Offloading Algorithm in Cache-Enabled Edge Clouds. *IEEE Trans. Cloud Comput.* **2020**, *7161*, 1–1.
4. Zhang, R.; Wang, J.; Sheng, Y.; Chen, X.; Ye, X. Protocol-aware packet scheduling algorithm for multi-protocol processing in multi-core MPL architecture. *IEICE Trans. Inf. Syst.* **2017**, *E100D*, 2837–2846.
5. He, F.; Qi, Y.; Xue, Y.; Li, J. Load scheduling for flow-based packet processing on multi-core network processors. In Proceedings of the 20th IASTED International Conference Parallel and Distributed Computing and System (PDCS 2008), Orlando, FL, USA, 16–18 November 2008; pp. 41–46.
6. Iqbal, M.F.; Holt, J.; Ryoo, J.H.; De Veciana, G.; John, L.K. Dynamic Core Allocation and Packet Scheduling in Multicore Network Processors. *IEEE Trans. Comput.* **2016**, *65*, 3646–3660.
7. Zha, Q.; Zhang, W.; Zeng, X. TCP/IP protocol stack acceleration technology based on multi-core processor. *J. Netw. New Media* **2013**, *1*, 58–64.
8. Han, S.; Jang, K.; Park, K.; Moon, S. PacketShader: A GPU-accelerated software router. *ACM SIGCOMM Comput. Commun. Rev.* **2010**, *40*, 195–206.
9. He, P.; Wang, J.; Deng, H.; Zhang, W. Balanced locality-aware packet schedule algrorithm on multi-core network processor. In Proceedings of the 2010 2nd International Conference on Future Computer and Communication, Wuhan, China, 21–24 May 2010; Volume 3, pp. 248–252.
10. Guo, J.; Yao, J.; Bhuyan, L. An efficient packet scheduling algorithm in network processors. In Proceedings of the IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies, Miami, FL, USA, 13–17 March 2005.
11. Riché, T.L.; Mudigonda, J.; Vin, H.M. *Experimental Evaluation of Load Balancers in Packet Processing Systems*; Computer Science Department, University of Texas at Austin: Austin, TX, USA, 2004.
12. Allen, J.R.; Bass, B.M.; Basso, C.; Boivie, R.H.; Calvignac, J.L.; Davis, G.T.; Frelechoux, L.; Heddes, M.; Herkersdorf, A.; Kind, A.; et al. IBM PowerNP network processor: Hardware, software, and applications. IBM J. Res. Dev. 2003, 47, 177–193.
13. Cavium The OCTEON processor Available online: https://cn.marvell.com/ (accessed on Mar 10, 2021).
14. Intel Data Plane Development Kit Available online: https://www.dpdk.org/ (accessed on Mar 9, 2021).
15. Rizzo, L. NetMap: A novel framework for fast packet I/O. Proc. 2012 USENIX Annu. Tech. Conf. USENIX ATC 2012 2012, 101–112.
16. Lee, J.; Ko, H.; Lee, H.; Pack, S. Flow-Aware Service Function Embedding Algorithm in Programmable Data Plane. *IEEE Access* **2021**, *9*, 6113–6121.
17. Kuang, J.; Bhuyan, L.; Xie, H.; Guo, D. E-AHRW: An energy-efficient adaptive hash scheduler for stream processing on multi-core servers. In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, Brooklyn, NY, USA, 3–4 October 2011; pp. 45–56.
18. Lukas, K. Load Sharing for multiprocessor network nodes. Available online: https://silo.tips/download/load-sharing-for-multiprocessor-network-nodes-2#(accessed on Dec 14, 2020).
19. Cao, Z.; Wang, Z.; Zegura, E. Performance of Hashing-Based Schemes for Internet Load Balancing. In Proceedings of the IEEE INFOCOM 2000, Tel Aviv, Israel, 26–30 March 2000; Volume 1, pp. 332–341.
20. Shi, W.; Kencl, L. Sequence-preserving adaptive load balancers. In Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, San Jose, CA, USA, 3–5 December 2006; pp. 143–152.
21. Zhao, J.; Guo, Z.; Zeng, X.; Song, M. High-Performance Implementation of Dynamically Configurable Load Balancing Engine on FPGA. *IEEE Commun. Mag.* **2020**, *58*, 62–67.
22. Thaler, D.G.; Ravishankar, C. V. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.* **1998**, *6*, 1–14.

23. Kencl, L.; Le Boudec, J.Y. Adaptive load sharing for network processors. *IEEE/ACM Trans. Netw.* **2008**, *16*, 293–306.
24. Guo. D.; Liao. G.; Nhuyan. L.N; Liu. B. An adapative hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multi-core server. In proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems Association for Computing Machinery, New York, NY, USA, 19 October 2009; pp. 50–59.
25. Miao, M.; Cheng, W.; Ren, F.; Xie, J. Smart Batching: A Load-Sensitive Self-Tuning Packet I/O Using Dynamic Batch Sizing. In Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, Australia, 12–14 December 2017; pp. 726–733.
26. Trevisan, M.; Finamore, A.; Mellia, M.; Munafò, M.; Rossi, D. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Commun. Mag.* **2017**, *55*, 163–169.
27. Li, Y.; Shan, L.; Qiao, X. A parallel packet processing runtime system on multi-core network processors. In Proceedings of the 012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science, Guilin, China, 19–22 October 2012; pp. 67–71.
28. Li, H.; Hu, C. MP-ROOM: Optimal matching on multiple PDUs for fine-grained traffic identification. *IEEE J. Sel. Areas Commun.* **2014**, *32*, 1881–1893.
29. Dittmann, G.; Herkersdorf, A. Network Processor Load Balancing for High-Speed Links. In Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems, San Diego, CA, USA, 14–18 July 2002; pp. 727–735.
30. Shi, W.; MacGregor, M.H.; Gburzynski, P. Load balancing for parallel forwarding. *IEEE/ACM Trans. Netw.* **2005**, *13*, 790–801.
31. Hesselbach, X.; Fabregat, R.; Baran, B.; Donoso, Y.; Solano, F.; Huerta, M. Hashing based traffic partitioning in a multicast-multipath MPLS network model. In Proceedings of the 3rd International Latin American Networking Conference, Ontario, Canada, 2–6 May 2005.
32. Cerovic, D.; Del Piccolo, V.; Amamou, A.; Haddadou, K.; Pujolle, G. Fast packet processing: A survey. *IEEE Commun. Surv. Tutorials* **2018**, *20*, 3645–3676.
33. Emmerich, P.; Gallenmuller, S.; Antichi, G.; Moore, A.W.; Carle, G. Mind the Gap—A Comparison of Software Packet Generators. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Beijing, China, 18–19 May 2017; pp. 191–203.
34. Gallenmüller, S.; Emmerich, P.; Wohlfart, F.; Raumer, D.; Carle, G. Comparison of frameworks for high-performance packet IO. In Proceedings of the 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Oakland, CA, USA, 7–8 May 2015; pp. 29–38.
35. Available online: http://doc.dpdk.org/guides/sample_app_ug/load_balancer.html (accessed on Dec 14, 2020).
36. Trifonov, H. Traffic-Aware Adaptive Polling Mechanism for High Performance Packet Processing. Available online: https://ulir.ul.ie/handle/10344/6246 (accessed on Dec 14, 2020).
37. Shemesh. O. System and Method for Symmetric Receive-Side Scaling (RSS). U.S. Patent No. 8635352, Washington, DC, USA, 21 January 2014; Patent and Trademark Office.
38. Kai, L.I. Traffic dynamic load balancing method based on DPDK. *Intell. Comput. Appl.* **2017**, *7*, 85–89.