TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 04/15 pp426–439
DOI: 10.26599/TST.2019.9010080
Volume 26, Number 4, August 2021

# Modeling and Analyzing the Performance of High-Speed Packet I/O

Xuesong Li*, Fengyuan Ren, and Bailong Yang

**Abstract:** Recently, 10 Gbps or higher speed links are being widely deployed in data centers. Novel high-speed packet I/O frameworks have emerged to keep pace with such high-speed links. These frameworks mainly use techniques, such as memory preallocation, busy polling, zero copy, and batch processing, to replace costly operations (e.g., interrupts, packet copy, and system call) in native OS kernel stack. For high-speed packet I/O frameworks, costs per packet, saturation throughput, and latency are performance metrics that are of utmost concern, and various factors have an effect on these metrics. To acquire a comprehensive understanding of high-speed packet I/O, we propose an analytical model to formulate its packet forwarding (receiving–processing–sending) flow. Our model takes the four main techniques adopted by the frameworks into consideration, and the concerned performance metrics are derived from it. The validity and correctness of our model are verified by real system experiments. Moreover, we explore how each factor impacts the three metrics through a model analysis and then provide several useful insights and suggestions for performance tuning.

**Key words:** high-speed packet I/O; costs per packet; saturation throughput; latency; modeling

## 1 Introduction

In recent years, the capacities of network links in data centers have witnessed numerous upgrades to fulfill the ever-increasing bandwidth demand of data-intensive applications. Nowadays, high-speed links of multiple 10 Gbps have been widely deployed. However, due to overheads imposed by several costly operations (e.g., interrupts, packet copy, and system calls), keeping pace with such high-speed links is a demanding task for the native OS kernel network stack[1,2].

To achieve line-rate packet processing even for small size packets, novel high-speed packet I/O frameworks, such as netmap[1], Intel DPDK[3], and PF_RING ZC[4], are proposed. These frameworks bypass the kernel network stack and exploit techniques, such as memory preallocation, busy polling, zero copy, and batch processing, to accelerate packet I/O. Because high-speed packet I/O frameworks could attain high throughput and low latency, they are expected an extensive application in data centers. Specifically, they can be used as the fast packet processing engine of software routers[5,6], switches[7,8], and middleboxes[9,10].

In general, costs per packet, saturation throughput, and latency are the most concerned performance metrics for high-speed packet I/O, and these metrics are potentially affected by various factors, including traffic pattern, service capability, and batch processing size. Though rare, the impacts of some factors have been studied in previous work, those studies are mainly carried out by measurements, and only partial factors and performance metrics are covered. Moreover, the interaction among factors remains unknown. What we expect to acquire is a comprehensive understanding of high-speed packet I/O.

In this paper, we concentrate on the modeling

• Xuesong Li is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with Xi'an Research Institute of Hi-Tech, Xi'an 710025, China. E-mail: lixs16@mails.tsinghua.edu.cn.
• Fengguan Ren is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: renfy@tsinghua.edu.cn.
• Bailong Yang is with Xi'an Research Institute of Hi-Tech, Xi'an 710025, China. E-mail: xa_403@163.com.
* To whom correspondence should be addressed.
Manuscript received: 2019-11-15; accepted: 2019-12-31

and performance evaluation of high-speed packet I/O. To address the limitations of measurements, we propose an analytical model to formulate the packet forwarding flow of high-speed packet I/O frameworks. Our analytical model takes the four main techniques (i.e., memory preallocation, busy polling, zero copy, and batch processing) commonly used in the frameworks into consideration, and it is abstracted as an $M^X/G^B/1/(R+B)$ queue with multiple vacations. Given that it is not a common queuing model, we first derive its queue length distribution at steady state. Then, on the basis of the model, we deduce the algebraic formulas of the concerned metrics. The validity and correctness of our model are verified by real system experiments based on Intel DPDK. Moreover, we conduct a series of quantitative analysis to demonstrate how each factor impacts the three performance metrics. The insights and suggestions summarized from model analysis are threefold.

(1) In most cases, batch processing is crucial to improve the concerned performance metrics. However, the max batch processing size does not follow the principle of "the bigger, the better". A max batch processing size greater than 32 does not result in significant gain in either costs per packet or saturation throughput, but increases latency. In addition, dynamically adjusting the max batch processing size is not necessary, because a size of 32 can well balance all three metrics.

(2) The latency reaches a cliff point when the system utilization reaches about 80%. To achieve low latency, we suggest controlling system utilization under this threshold. Simultaneously, we find that the normalized latency-utilization characteristic curve will not be affected by service capability and batch processing setting. Traffic pattern is the only factor that exerts differences on the curve.

(3) Batch processing is not always the best choice. When high-speed packet I/O frameworks serve heavy task applications, such as intrusion detection (in which several hundreds of CPU cycles will be consumed to process a received packet), batch processing can hardly decrease the costs per packet and increase the saturation throughput, but increase the latency by a few microseconds.

The rest of this paper is structured as follows. In Section 2, we introduce high-speed packet I/O and review related work. In Section 3, we establish an analytical model for high-speed packet I/O. We conduct a detailed model derivation, and then the concerned performance metrics are deduced from the model. Subsequently, we present the validation of our model in Section 4. In Section 5, we analyze the impact of various factors on performance metrics in detail. Insights and suggestions to improve the performance of high-speed packet I/O frameworks are also summarized in this section. Finally, we draw the conclusions in Section 6.

## 2 Background and Related Work

In this section, we first introduce high-speed packet I/O and the main techniques that it adopts. Then, the packet forwarding flow of high-speed packet I/O is described. Related studies are also discussed in this section.

### 2.1 High-speed packet I/O

As the native OS kernel network stack is designed as a general-purpose packet processing engine, it prioritizes compatibility rather than performance. As a result, kernel network stack does not perform well when required to cooperate with 10 Gbps or higher speed network links. Nowadays, high-speed packet I/O frameworks, such as netmap, Intel DPDK, and PF_RING ZC, fix this issue by offering a stripped-down alternative to the kernel network stack. These frameworks abandon time-consuming operations (e.g., interrupt, packet copy, and system calls) and mainly use the following techniques to achieve high throughput and low latency:

**(1) Memory preallocation.** This technique allocates all memory resources required to store packets before starting packet reception or transmission. Two ring buffers (Rx ring buffer for reception and Tx ring buffer for transmission), each with a memory of $R$ packets, are allocated when the network driver is loaded, and the memory will be recycled and reused for subsequent packet I/O. With memory preallocation, frequent memory allocation and deallocation could be avoided.

**(2) Busy polling.** Busy polling is defined as the ability to continuously poll the ring buffer without waiting for interrupts from Network Interface Cards (NICs). The New API (NAPI) mechanism of Linux kernel stack, which is introduced in kernel version 2.6, also absorbs the idea of polling. NAPI will switch to polling mode under high traffic load to avoid interrupt handling. Busy polling could mitigate the delay associated with the interrupts, but will burn CPU cycles when there are no packets to process.

**(3) Zero copy.** By mapping the Direct Memory

Access (DMA)-able region (i.e., ring buffer) for NICs to user memory space, packet I/O frameworks could avoid intermediate packet copy among the ring buffer, kernel memory space, and user memory space. With the use of this method, the user application could directly access the memory region that was once restricted, thus potentially entailing risks for the stability of the system.

**(4) Batch processing.** When enabling batch processing, high-speed packet I/O frameworks will retrieve a batch of packets per polling, then process and send them in a group. Batch processing could amortize the overhead of memory management over several packets and improve instruction and data cache locality, prefetching effectiveness, and prediction accuracy.

Furthermore, high-speed packet I/O can better support packet processing on multicore systems by distributing traffic across multiple cores and adopting run-to-completion execution model at each core. In general, we could depict the packet forwarding flow of high-speed packet I/O at each core with Fig. 1. As shown, when packets arrive at the NIC, they will be pushed to the Rx ring buffer via DMA transfer. When the Rx ring buffer is filled with packets, the following forwarding loop will be executed: (a) retrieving a batch of packets via polling the buffer; (b) processing each packet in the batch; and (c) sending the processed packets to Tx ring buffer. When the network link is idle and no packet presents in the Rx ring buffer, the CPU core will still execute the busy polling loop until a packet arrives, then it will again go into the above forwarding loop. We point out where the aforementioned four techniques are used in Fig. 1.

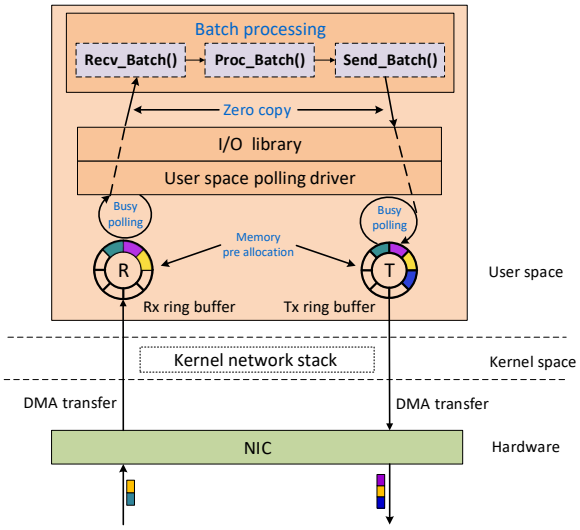



**Fig. 1 Packet forwarding flow of high-speed packet I/O.**

### 2.2 Related work

At present, research on high-speed packet I/O mainly concentrates on the implementation of frameworks[1,3,4,11] and the development of application[8,10,12–16]. As far as we know, few works have focused on the modeling and performance evaluation of high-speed packet I/O.

Gallenmüller et al.[17] compared the transmission efficiency and latency of some main high-speed packet I/O frameworks via measurements. The influences of batch size on saturation throughput and latency were also tested in their work. Similar measurements were conducted on software switches in Refs. [18, 19].

Jarschel et al.[20] provided a performance model of the OpenFlow system. They mainly focused on capturing the delay experienced by packets that have to be processed by the OpenFlow controller in contrast to that be processed by the OpenFlow switch only. Bolla et al.[21] proposed an analytical model to represent the impact of power saving technologies on software routers, but they did not consider the application of high-speed packet I/O in their software routers. Su et al.[22] modeled the process, in which one CPU core polls multiple Rx queues of virtual switch. Unfortunately, batch processing, which is one of the main features of high-speed packet I/O, is not considered in their work. Also, the assumption that packets arrival follows a Poisson process is outdated, which does not characterize the bursty nature of packets arrival in high-speed networks. Research on kernel stack modeling can be found in Refs. [23, 24].

## 3 Queuing Model for High-Speed Packet I/O

In this section, we establish an analytical model to formulate the packet forwarding flow of high-speed packet I/O. Our analytical model takes the four main techniques described in Section 2.1 into consideration. Based on our analytical model, we could obtain expressions for costs per packet, saturation throughput, and latency, all of which are crucial metrics for high-speed packet I/O.

### 3.1 Model description

**Traffic arrival process.** We assume that packets arrive at the system in batches according to a compound Poisson process with the mean batch-arrival rate $\lambda$. This batch-arrival assumption lies in the bursty nature of

packet arrivals in real high-speed networks[25,26], and can effectively approximate the network traffic as presented in Refs. [27,28]. Moreover, considering that statistical characterization of packet interarrival time is well known to have long-range dependency and multifractal statistical features[29,30], the distribution of the arriving-batch size is assumed to follow Zipf's law (which can be regarded as the discrete version of a truncated continuous Pareto distribution)[21]. In detail, the arriving-batch size $X$ is a random variable with Probability Mass Function (PMF) $P(X=j)=x_j, j=1,2,\ldots,$ Probability Generating Function (PGF) $X(z)=\sum_{j=1}^{\infty} x_j z^j$, and expectation $E(X)=\sum_{j=1}^{\infty} x_j \times j$. According to Zipf's law, $P(X)$ is given by

$$P(X=j)=\begin{cases}\dfrac{1}{j^v \sum\limits_{l=1}^{\psi}\dfrac{1}{l^v}}, & j\leqslant\psi;\\ 0, & j>\psi\end{cases} \tag{1}$$

where $\psi$ is the truncated maximum arriving-batch size and $v$ is the burst degree. Both $\psi$ and $v$ will influence the distribution of batch arrival size.

**Polling and vacation**. After the packets enter Rx ring buffer, high-speed packet I/O frameworks retrieve them via polling. Due to the use of **busy polling**, no packets will be returned and processed after an empty polling when the Rx ring buffer is empty. However, an empty polling also consumes CPU cycles, so the framework can only serve the incoming packets until the next polling returns with some packets. From this perspective, an empty polling can be viewed as a vacation during which incoming packets will not be detected and processed. Let $F$ be the CPU frequency and $C_v$ be CPU cycles consumed by a single empty polling, then the single vacation time $V$ is given by

$$V=\frac{C_v}{F} \tag{2}$$

Given that each empty polling always executes the same function call, $C_v$ is a fixed value. Correspondingly, the distribution function of $V$ and its Laplace-Stieltjes transform are given by

$$V(t)=\begin{cases}0, & \text{if } t<V;\\ 1, & \text{if } t\geqslant V,\end{cases}$$

and

$$V^*(z)=\int_0^{\infty}\mathrm{e}^{-zt}\mathrm{d}V(t)=\mathrm{e}^{-Vz} \tag{3}$$

respectively.

**Batch service process**. Recalling that high performance packet I/O frameworks usually adopt **batch processing**, it may retrieve and serve a batch of packets in one polling loop. Generally, an upper bound $B$ is associated with the packet number of batch processing (e.g., $B$ is 32 in Intel DPDK), that is, the framework will retrieve all packets to process when packets in the Rx ring buffer are less than $B$. Otherwise, only $B$ packets will be served, and the remaining packets will be retrieved in subsequent polling.

According to research in Ref. [1], the service time of a batch of packets in high-speed packet I/O is mostly dominated by packet number, and packet size exerts minimal influence. This feature benefits from **memory preallocation** and **zero packet copy**, and it is also verified by our experiments. Let $C_j$ be the random variable for CPU cycles used to serve a batch packets of size $j$ $(0<j\leqslant B)$ and $S_j$ be the corresponding service time. Then

$$C_j=j(C_{\mathrm{IO}}+C_{\mathrm{task}})+C_{\mathrm{call}} \tag{4}$$

$$S_j=\frac{C_j}{F}=\frac{j(C_{\mathrm{IO}}+C_{\mathrm{task}})+C_{\mathrm{call}}}{F} \tag{5}$$

where $C_{\mathrm{IO}}$ is the per-packet I/O CPU cycles, $C_{\mathrm{task}}$ is the per-packet application processing CPU cycles, and $C_{\mathrm{call}}$ is the CPU cycles consumed by receiving and sending function calls. Both $C_{\mathrm{IO}}$ and $C_{\mathrm{call}}$ are fixed values for a specific framework, and $C_{\mathrm{task}}$ is relevant to the application manipulating the packets. Assuming the probability mass function of $C_{\mathrm{task}}$ is given by

$$P(C_{\mathrm{task}}=i)=\tau_i \tag{6}$$

Let $S_j(t)$ be the probability distribution function of $S_j$, then $S_j(t)$ and its Laplace-Stieltjes transform are given by

$$S_j(t)=P\{S_j\leqslant t\}=$$
$$P\left\{\frac{(j(C_{\mathrm{IO}}+C_{\mathrm{task}})+C_{\mathrm{call}})}{F}\leqslant t\right\}=$$
$$P\left\{\frac{C_{\mathrm{task}}\leqslant(t\times F-(jC_{\mathrm{IO}}+C_{\mathrm{call}}))}{j}\right\}=\sum_{i=0}^{r}\tau_i$$

and

$$S_j^*(z)=\int_0^{\infty}\mathrm{e}^{-zt}\mathrm{d}S_j(t) \tag{7}$$

where $r=\lfloor(tF-(jC_{\mathrm{IO}}+C_{\mathrm{call}}))/j\rfloor$.

**Buffer capacity**. We assume that the Rx ring buffer has finite capacity $R$, so that at most $(R+B)$ individual packets can be held in the high-speed packet I/O frameworks, where $R$ packets are waiting for service and $B$ packets are being served.

**Queuing model for packet forwarding.** In summary, we can outline that an $M^X/G^B/1/(R+B)$ queue with

multiple vacations could fit into the packet forwarding flow of high-speed packet I/O. Here, $M^X$ stands for the Poisson batch arrival traffic, $G^B$ represents batch packet processing with an upper bound $B$ and the batch service time follows a general distribution, and $R$ denotes the capacity of the Rx ring buffer.

So far, we have constructed a queuing model for high-speed packet I/O. We will demonstrate how to acquire the concerned performance metrics in subsequent subsections. As the first step, we need to obtain the stationary distribution of the queue length (i.e., packet number left in the Rx ring buffer).

The key notations used in our model description and derivation are listed below.

- $\lambda$: Packets batch-arrival rate.
- $X$: Random variable for arriving-batch size.
- $C_v$: CPU cycles consumed by a single empty polling.
- $C_{\text{IO}}$: Per-packet I/O CPU cycles.
- $C_{\text{task}}$: Per-packet application processing CPU cycles.
- $C_{\text{call}}$: CPU cycles consumed by receiving and sending function calls.
- $C_j$: Random variable for CPU cycles used to serve a batch packets of size $j$ $(0 < j \leqslant B)$.
- $V$: Random variable for single vacation time.
- $V(t)$: Distribution function of $V$.
- $V^*(\theta)$: Laplace-Stieltjes transform of $V(t)$.
- $S_j$: Random variable for service time of a batch packets of size $j$ $(0 < j \leqslant B)$.
- $S_j(t)$: Distribution function of $S_j$.
- $S_j^*(\theta)$: Laplace-Stieltjes transform of $S_j(t)$.
- $G_i$: Random variable representing the packet number arriving during the service period of a batch packets of size $i\,(0 < i \leqslant B)$.
- $g_{j|i}$: Probability of $j$ packets arriving during the service period of a batch packets of size $i\,(0 < i \leqslant B)$, $g_{j|i} = P(G_i = j)$.
- $G_i(z)$: The probability generating function (p.g.f) of $G_i$.
- $H$: Random variable representing the packets number arriving during a single vacation period.
- $h_j$: Probability of $j$ packets arriving during a vacation period, $h_j = P(H = j)$.
- $H(z)$: The p.g.f of $H$.
- $L^{(t)}$: Random variable representing the packets number arriving during period $t$.
- $l_j^{(t)}$: Probability of $j$ packets arriving during period $t$.
- $L^{(t)}(z)$: The p.g.f of $L^{(t)}$.
- $P_{i,j}^-$: Probability of $i$ packets left in Rx ring buffer and $j$ packets to be processed at service/vacation starting epoch.
- $n_j^-$: Probability of $j$ packets to be processed at service/ vacation starting epoch.
- $P_{i,j}$: Probability of $i$ packets left in Rx ring buffer and $j$ packets being processed at arbitrary epoch.
- $n_j$: Probability of $j$ packets being processed at arbitrary epoch.

### 3.2 Queue length distribution at service-completion/vacation-termination epoch

As we cannot directly acquire the stationary distribution of an $M^X/G^B/1/(R+B)$ queue with multiple vacations, we will first derive its queue length distribution at service-completion/vacation-termination epoch.

Let $Q_n^+$, $n \in \mathbf{N}$ denote the number of packets in the Rx ring buffer just after the $n$-th service-completion/vacation-termination epoch. Then, the dynamics of $\boldsymbol{Q}^+ = \{Q_n^+ : n \geqslant 1\}$ is given by the recursion:

$$Q_{n+1}^+ = \begin{cases} \min(H, R)\,, & \text{if } Q_n^+ = 0; \\ \min(G_{Q_n^+}, R), & \text{if } 0 < Q_n^+ < B; \\ \min(Q_n^+ - B + G_B, R), & \text{if } Q_n^+ \geqslant B \end{cases} \tag{8}$$

where $H$ and $G_i$ are random variables representing the number of packets that arrive during a vacation period and that arrive during the service period of a batch packets of size $i$ $(0 < i \leqslant B)$, respectively.

Note that the traffic arrival and packet processing are mutually independent, then the process $\boldsymbol{Q}^+$ is an embedded Markov chain. According to Eq. (8), the transition probability matrix of $\boldsymbol{Q}^+$ can be given by

$$\boldsymbol{P} = \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0R} \\ p_{10} & p_{11} & \cdots & p_{1R} \\ \vdots & \vdots & \ddots & \vdots \\ p_{R0} & p_{R1} & \cdots & p_{RR} \end{bmatrix},$$

where

$$p_{ij} = \begin{cases} h_j, & \text{if } i = 0 \text{ and } j < R; \\ g_{j|i}, & \text{if } 0 < i < B \text{ and } j < R; \\ g_{(j-i+B)|B}, & \text{if } i \geqslant B \text{ and } i - B \leqslant j < R; \\ 1 - \sum\limits_{k=0}^{R-1} p_{ik}, & \text{if } j = R; \\ 0, & \text{other}, \end{cases} \tag{9}$$

and

$$h_j = P(H = j),$$
$$g_{j|i} = P(G_i = j)\,.$$

Let $H(z)$ and $G_i(z)$ be the p.g.f. of $H$ and $G_i$, then following the derivation in Ref. [31] and using Eqs. (3) and (7), we obtain

$$H(z)=\sum_{j=0}^{\infty}h_j z^j=V^*(\lambda-\lambda X(z))=\mathrm{e}^{-\lambda V(1-X(z))} \quad (10)$$

$$G_i(z)=\sum_{j=0}^{\infty}g_{j|i}z^j=S_i^*(\lambda-\lambda X(z)) \quad (11)$$

Since $h_j$ and $g_{j|i}$ can be obtained by inverting $H(z)$ and $G_i(z)$, respectively, we can easily calculate the transition probability matrix $\boldsymbol{P}$ by introducing $h_j$ and $g_{j|i}$ into Eq. (9). Let $\boldsymbol{\pi}^+=[\pi_0^+,\pi_1^+,\ldots,\pi_R^+]$ be the stationary probabilities of $\boldsymbol{Q}^+$. Then, $\boldsymbol{\pi}^+$ can be obtained by solving the following equation:

$$\begin{cases}\boldsymbol{\pi}^+=\boldsymbol{\pi}^+P;\\ \sum_{i=0}^{R}\pi_i^+=1.\end{cases}$$

### 3.3 Queue length distribution at arbitrary epoch

To obtain the queue length distribution at arbitrary epoch, we first develop the relationship between queue length distribution at the service–completion/vacation–termination epoch and the arbitrary epoch.

Let $M^-$ and $N^-$ be random variables for the numbers of packets left in the Rx ring buffer and to be processed at the service/vacation starting epoch, respectively. Then, the joint distribution of $(M^-,N^-)$ can be represented by

$$P_{i,j}^-=P(M^-=i,N^-=j)=\begin{cases}\pi_j^+, & \text{if } i=0,j\leqslant B;\\ \pi_{i+j}^+, & \text{if } 0<i\leqslant R-B,j=B;\\ 0, & \text{other}\end{cases} \quad (12)$$

Also, we can represent the distribution of $N^-$ as

$$n_j^-=P(N^-=j)=\sum_{i=0}^{R}P_{i,j}^- \quad (13)$$

Then, we have

$$n_j=\frac{n_j^-S_j}{\sum_{k=0}^{B}n_k^-S_k} \quad (14)$$

where $n_j$ is the probability of $j$ packets being processed by the framework at the arbitrary epoch.

Let $M$ and $N$ be the random variables denoting the numbers of packets left in the Rx ring buffer and being processed at the arbitrary epoch, respectively, and $P_{i,j}=P(M=i,N=j)$. Then,

$$P_{i,j}=P(N=j)\times P(M=i|N=j)=n_jP(M=i|N=j).$$

For $i<R$,

$$P_{i,j}=n_j\Big(\sum_{k=0}^{i}P(M^-=k|N^-=j)g_{i-k|j}^{\mathrm{e}}\Big)=n_j\sum_{k=0}^{i}\frac{P_{k,j}^-}{n_j^-}g_{i-k|j}^{\mathrm{e}} \quad (15)$$

For $i=R$,

$$P_{R,j}=n_j\Big(\sum_{k=0}^{R}P(M^-=k|N^-=j)\sum_{x=R-k}^{\infty}g_{x|j}^{\mathrm{e}}\Big)=n_j\sum_{k=0}^{R}\frac{P_{k,j}^-}{n_j^-}\sum_{x=R-k}^{\infty}g_{x|j}^{\mathrm{e}} \quad (16)$$

where $g_{i|j}^{\mathrm{e}}$ is defined as the stationary probability that $i$ packets arrive during an elapsed service time of $j$ packets. According to Ref. [32], $g_{i|j}^{\mathrm{e}}$ is given as follows (for brevity, we let $S_0=V$, then $g_{i|0}^{e}$ represents the stationary probability that $i$ packets arrive during an elapsed vacation time):

$$g_{i|j}^{\mathrm{e}}=\int_0^{\infty}l_i^{(t)}\frac{P(S_j\geqslant t)}{E(S_j)}\,\mathrm{d}t \quad (17)$$

where $l_i^{(t)}$ is the probability of $i$ packets arriving during period $t$. In a similar way with that we get $h_j$ and $g_{j|i}$, we can obtain $l_i^{(t)}$ by inverting

$$L^{(t)}(z)=\sum_{i=0}^{\infty}l_i^{(t)}z^i=\mathrm{e}^{-\lambda t(1-X(z))}.$$

### 3.4 Concerned performance metrics

For high-speed packet I/O, costs per packet, saturation throughput, and latency are the performance metrics that are of utmost concern. We derive these metrics with the model description in Section 3.1 and the stationary probabilities obtained in Section 3.3.

**(1) Costs Per Packets (CPPs).** It is defined as average CPU cycles used to forward (including receiving, processing, and transmitting) a packet. Since batch processing with different batch sizes brings about different overheads, CPPs will change with traffic pattern, traffic load, and max batch processing size.

Realizing that $n_j^-(0\leqslant j\leqslant B)$ is exactly the distribution of batch processing size, then the CPP is given by

$$\mathrm{CPP}=\frac{\sum_{j=1}^{B} n_j^- E(C_j)}{\sum_{j=1}^{B} j\times n_j^-}=\frac{\sum_{j=1}^{B}\left(j(C_{\mathrm{IO}}+E(C_{\mathrm{task}}))+C_{\mathrm{call}}\right)n_j^-}{\sum_{j=1}^{B} j\times n_j^-}=C_{\mathrm{IO}}+E(C_{\mathrm{task}})+\frac{C_{\mathrm{call}}}{b} \tag{18}$$

where $b=\left(\sum_{j=1}^{B} j\times n_j^-\right)/\left(\sum_{j=1}^{B} n_j^-\right)$ is the average batch processing size.

**(2) Saturation Throughput (ST).** It is a metric defined as the maximum packet processing rate achieved by the frameworks when the CPU core is saturated.

"Saturation" means that the Rx ring buffer is always filled with a mass of packets. Then, $B$ packets will be returned from every polling. Accordingly, saturation throughput is denoted by

$$\mathrm{ST}=\frac{B}{E(C_B)/F}=\frac{F}{C_{\mathrm{IO}}+E(C_{\mathrm{task}})+C_{\mathrm{call}}/B} \tag{19}$$

**(3) Latency.** It is the queuing and processing delay experienced by a packet.

Let $\bar{L}$ be the average number of packets in the queuing system (including the ones left in the Rx ring buffer and being processed by the framework). Then, from the definition of average queue length we know that

$$\bar{L}=\sum_{i=0}^{R}\sum_{j=0}^{B}(i+j)P_{i,j} \tag{20}$$

Substituting Eqs. (15) and (16) into Eq. (20), after simplification we obtain

$$\bar{L}=\sum_{j=0}^{B} n_j\left(j+\int_0^{\infty}\frac{P(S_j\geqslant t)}{E(S_j)}\lambda E(X)t\,\mathrm{d}t\right)+n_B\sum_{i=0}^{R}\frac{iP_{i,B}^-}{n_B^-} \tag{21}$$

Let $\bar{W}$ be the average waiting time of a packet, namely, latency. Then, by using Little's law, we have

$$\bar{W}=\frac{\bar{L}}{\lambda^*} \tag{22}$$

where $\lambda^*$ is the effective packet arrival rate and equal to the packet processing rate, i.e.,

$$\lambda^*=\frac{\sum_{j=0}^{B} jn_j^-}{\sum_{j=0}^{B} E(S_j)n_j^-} \tag{23}$$

## 4 Model Validation

In this section, experiments are conducted to validate our analytical model. We compare the three performance metrics obtained from our model with the results from a real high-speed packet I/O framework—Intel DPDK. Intel DPDK is chosen as it is an open-source software and supports numerous commodity NICs.

### 4.1 Experiment setup

Our DPDK experiment testbed consists of two servers: one traffic generator and one forwarder, and both servers are equipped with a dual-port Intel X520-SR2 network interface card (Intel 82599ES 10 Gb Ethernet Controller). The generator and the forwarder are connected via a 10 Gb Ethernet link. The generator runs on Intel Core i7-6700 CPU @ 3.40 GHz and has 8 GB DDR4–2133 MHz memory. The forwarder runs on a dual socket Intel Xeon E5-2620 v3 CPU and has 64 GB DDR4–1866 MHz memory. The available CPU core frequencies of the forwarder are 1.2 GHz to 2.4 GHz with 0.1 GHz step. Hyper-threading and turbo boost of the forwarder are disabled to make the measurements consistent and repeatable.

At the beginning of each experiment, we initialize a DPDK instance, which is pinned to a dedicated core, on the forwarder. Then, with the help of MoonGen (a software packet generator)[33], the generator starts to send packets to the forwarder. When packets arrive at the forwarder, the following receiving–processing–sending loop will be executed:

- A batch of packets is retrieved from the Rx ring buffer ($B$ packets will be retrieved at most);
- The source and destination MAC address of each packet are updated, and a certain number of CPU cycles (i.e., $C_{\mathrm{task}}$) of processing are executed;
- All processed packets are sent in batch to the traffic generator via the same link.

MoonGen is also used to measure the packet latency by sending extra timestamped packets periodically (about $10^4$ timestamped packets per second). The latency measurement utility of MoonGen employs the hard timestamping features of commodity Intel NICs. To send Poisson batch arrival traffic, we

modify the codebase of MoonGen to add the corresponding functionality. The modified source files of MoonGen and the sample application code of DPDK instance can be found in our GitHub repository (https://github.com/bigstone09/MAPHSPIO).

The values of some important parameters used in our model are listed in Table 1. Among these parameters, the values of $C_{\text{IO}}$, $C_{\text{call}}$, and $C_v$ are obtained through measures based on Intel DPDK; the Rx ring buffer size $R$ remains the default value of DPDK, i.e., 512; the values of $\psi$ and $v$ are acquired from a CAIDA 10 Gbps link trace[34] using least-squares fitting. The Cumulative Distribution Function (CDF) of burst size distribution of the traffic trace is shown in Fig. 2. We can find Zipf's law with $\psi = 20$ and $v = 0.701$ can fit the trace statistics well. Also, we fix the packet size to 64 bytes. Thus, a 10 Gbps link could carry up to 14.88 Mpps (100%) of packet rate.

## 4.2 Validation result

To make the validation more convincing, we conduct experiments under two different settings: Setting-A, where $B = 8$, $C_{\text{task}} = 100$ CPU cycles, and $F = 2.0$ GHz; and Setting-B, where $B = 32$, $C_{\text{task}} = 200$ CPU cycles, and $F = 2.4$ GHz. We vary the traffic load from 1% to 99% and then compare the CPP and latency, under different loads, the results are depicted in Figs. 3a and 3c, respectively. We vary the CPU frequency from 1.2 GHz to 2.4 GHz, then compare the saturation throughput (in this case, we ignore the frequency constraint in Setting-A and Setting-B), the results are depicted in Fig. 3b.

**Table 1 Values of some used parameters.**

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| $C_{\text{IO}}$ | 45 cycles | $R$ | 512 packets |
| $C_{\text{call}}$ | 43 cycles | $\psi$ | 20 |
| $C_v$ | 24 cycles | $v$ | 0.701 |

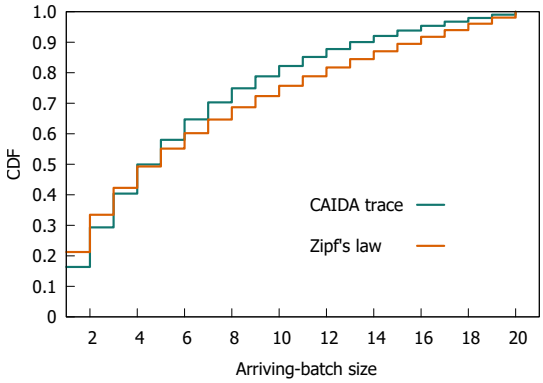



**Fig. 2 CDF of batch size distribution.**

### 4.2.1 CPP

As shown in Fig. 3a, the CPP deduced from our model coincide with that from experiments in both settings.

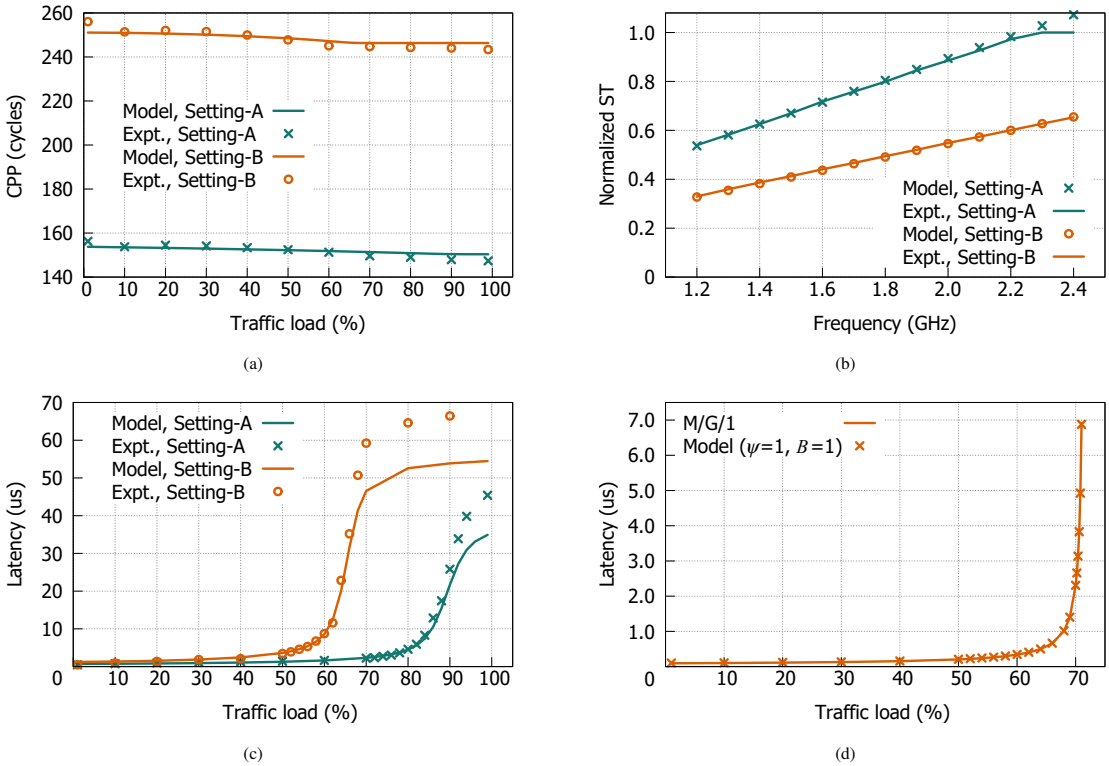



**Fig. 3 Validation of analytical model ("Model" for results of analytical model, "Expt." for results of DPDK experiment).**

We can also find that the CPP just slightly decrease as the traffic load grows. This is mainly due to the batch arrival characteristics of traffic, which allows the shared overhead to be amortized by several packets even under low traffic loads.

#### 4.2.2 ST

Figure 3b shows that ST measured in DPDK exactly matches that given by our model. For simplicity and clarity, we normalize the throughput with the link capacity (14.88 Mpps). It is noteworthy that even if the model could give a normalized throughput higher than 1 (due to higher service capability), measures from the DPDK are always bounded by link capacity (i.e., $\leqslant 1$). This is why the ST given by our model and the experiments at Setting-A is deviated when frequency is greater than 2.2 GHz.

#### 4.2.3 Latency

As depicted in Fig. 3c, latencies derived from the model are generally consistent with those from DPDK measurements under low loads. Although a difference exists under high loads, the latency evolution from both sources shows a similar trend. Actually, we cannot accurately measure the latency experienced by packets in DPDK. Our approach is to first measure the round-trip time between the generator and forwarder under different traffic loads, and then subtract a base round-trip time (round-trip time of a single packet when $C_{\text{task}} = 0$) to get the latency that packets experience. Considering the possible measurement errors, the latency approximation given by our model is still satisfactory. We also found that our model tends to underestimate the latency. The additional latency in real system experiments mainly comes from memory access overhead, which is not considered in our model.

When $\psi$ and $B$ are both equal to 1 (Poisson traffic and non-batch processing), our analytical model approximately degenerates to an M/G/1 queue. Therefore, we also compare the latency deduced from our degenerated model with that from the M/G/1 queue in Fig. 3d. The results also confirm that our model derivation is correct and credible.

In summary, our model can well formulate the packet forwarding flow of high-speed packet I/O. When compared with the results from real system experiments, our model can provide very close CPP and ST. Although errors exist with regard to latency, latencies derived from our model show similar evolution trends with that from system experiments, which is also valuable in performance analysis.

## 5 Performance Analysis

In this section, we will demonstrate how different factors impact the performance of high-speed packet I/O. Several insights and suggestions gained from our model analysis are also provided.

### 5.1 Impacts of various factors

In general, configurable factors that may have an impact on performance include traffic pattern (truncated maximum arriving-batch size $\psi$ and bursty degree $v$), service capability (CPU frequency $F$ and per-packet processing load $C_{\text{task}}$), and maximum batch processing size $B$. The default parameter settings for our analysis are $\psi = 20$, $v = 0.7$, $F = 1.6$GHz, $C_{\text{task}} = 100$ CPU cycles, and $B = 32$. When one factor is discussed, others maintain the default values.

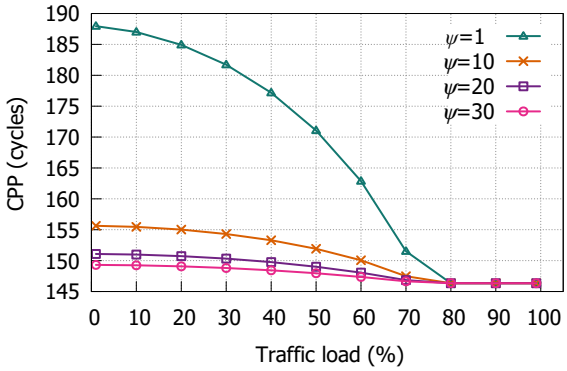
#### 5.1.1 Impacts of traffic pattern

As traffic pattern determines the batch size of packet arrival, it will also influence the distribution of batch processing size. Then according to Eqs. (18)–(22), traffic pattern may impact both CPP and latency, but has no impact on ST. We analyze the evolution of CPP and latency when $\psi = 1$ (Poisson traffic and non-batch arrival), 10, 20, 30 and $v = 0.2, 0.7, 1.2$. The results are shown in Figs. 4 and 5, respectively.

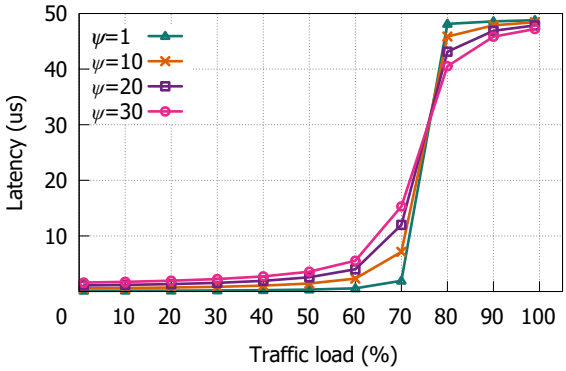
When the traffic load is not particular high, the CPP will decline with the increase of $\psi$ and decrease of $v$, because a higher $\psi$ or lower $v$ implies more bursty incoming traffic and consequently results in a larger batch processing size. Then, the shared overheads could be amortized by more packets, and fewer CPU cycles will be consumed in an average sense. When traffic overloads, traffic pattern hardly affects the CPP because the batch processing size has reached its limit when overloading.

With respect to latency, it suffers an increase when we enlarge $\psi$ or shrink $v$ under low and medium traffic loads. The latency increases because more bursty traffic leads to a longer queuing time, and a larger batch processing size increases the processing time. The impact of traffic pattern on latency under overloads is opposite to that under low loads, because more bursty traffic will increase the packet loss rate, and thereby decrease the queuing time.

From Fig. 4a, we also find that whether the traffic is batch arrival or not has a significant impact on the CPP. Because the CPP under low loads when $\psi = 1$ is much higher than that when $\psi > 1$. From Fig. 5, we can also

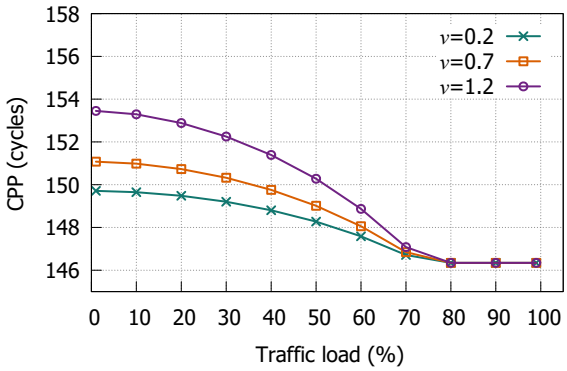



(a) Impact of $\psi$ on costs per packet

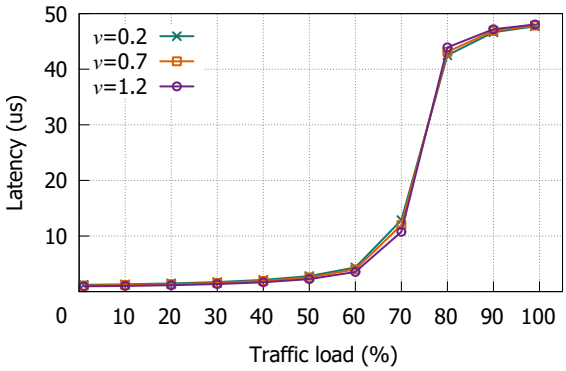



(b) Impact of $\psi$ on latency

**Fig. 4 Impact of max arriving-batch size $\psi$.**




(a) Impact of $v$ on costs per packet




(b) Impact of $v$ on latency

**Fig. 5 Impact of bursty degree $v$.**

find that the CPP and latency are less sensitive to bursty degree $v$.

### 5.1.2 Impacts of service capability

CPU frequency $F$ and per-packet processing load $C_{\text{task}}$ together determine the service capability. According to Eqs. (18)–(23), the impacts of $C_{\text{task}}$ on all three performance metrics are obvious, i.e., the larger the $C_{\text{task}}$, the higher (lower/higher, respectively) the CPP (saturation throughput/latency, respetively). Since these impacts are easy to understand, we do not show them in figures.

The impacts of $F$ on ST and latency are also apparent, because its role is exactly the opposite of that of $C_{\text{task}}$. However, it is not the case when it comes to CPP. As revealed in Fig. 6, the increase in frequency also leads to an increase in the CPP, and this phenomenon is particularly evident under a moderate traffic load. The underlying reason is that a large frequency (high service capability) will reduce the probability of a long queue length, which results in the decrease of average batch processing size (i.e., $b$ in Eq. (18)), and then the increase of CPP.

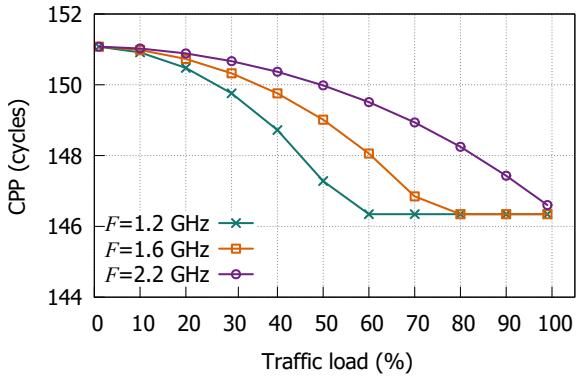
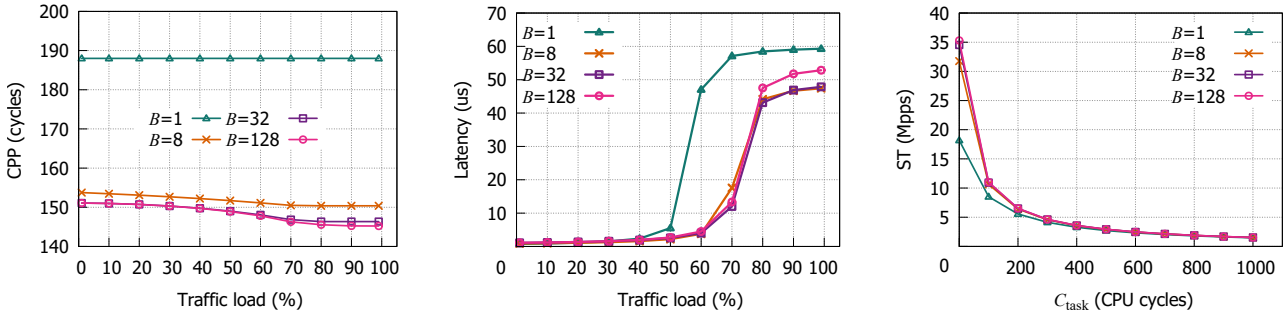



**Fig. 6 Impact of frequency $F$ on CPP.**

### 5.1.3 Impacts of max batch processing size

We depict the evolution of costs per packet and latency under different traffic loads with $B = 1$ (non-batch processing), 8, 32, and 128 in Figs. 7a and 7b, respectively. The saturation throughput with different $B$ and $C_{\text{task}}$ is demonstrated in Fig. 7c.

As can be seen, batch processing is crucial for high-speed packet I/O, because all three performance metrics




(a) Impact of $B$ on costs per packet (b) Impact of $B$ on latency (c) Impact of $B$ on saturation throughput

**Fig. 7 Impact of max batch processing size.**

are the worst when $B = 1$. Meanwhile, we should also note that the marginal benefit of batch processing declines with the increase of max batch processing size $B$. When $B > 32$, continuing to increase $B$ will hardly bring about a reduction in CPP or an augmentation in ST, but increase the latency. In addition, Fig. 7c also shows that the impact of $B$ on ST is fading with the increase of $C_{\text{task}}$.

We also claim that there is no need to dynamically adjust the max batch processing size and $B = 32$ is recommended, because it could well balance the CPP, ST, and latency. $B = 32$ is the default setting of Intel DPDK as well as an empirical value, which is now proven to be reasonable by our model.

### 5.2 Insights and suggestions

**(1) Cliff point of latency**. If we look back at Figs. 4b, 5b, and 7b, we may find that the latency increases gently when traffic load is low, but increases sharply when traffic load is higher than a specific value (this value varies with $\psi$, $v$, $F$, $C_{\text{task}}$, and $B$), i.e., there exists a cliff point. Intuitively, we believe that the cliff point is more relevant to the system utilization. According to queuing theory, the utilization, denoted by $\rho$, is the probability that CPU is busy handling incoming packets. Thus, we can write

$$\rho = \sum_{j=1}^{B} n_j = 1 - n_0 \tag{24}$$

After obtaining the expression of utilization, we replot the evolution of latency with utilization under various configurations in Figs. 8–11. It is noteworthy that directly comparing the latency evolution at different $C_{\text{task}}$, $F$, or $B$ is not that fair, for the maximum latency under different conditions is varying. So we normalize the latency with the maximum value on each condition in Figs. 10 and 11.

Two insights could be summarized from the

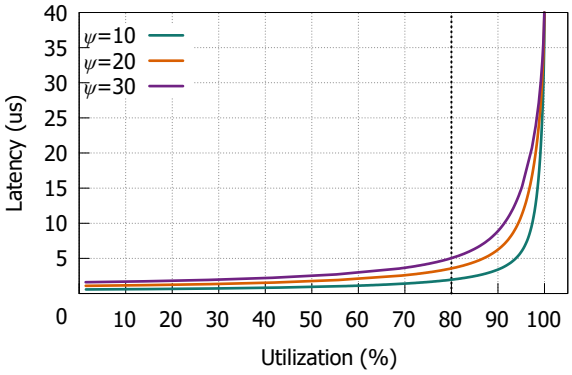



**Fig. 8 Evolution of latency with utilization when $\psi$=10, 20, and 30.**

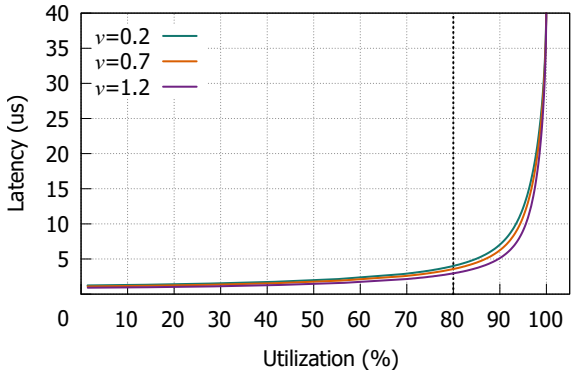



**Fig. 9 Evolution of latency with utilization when $v$=0.2, 0.7, and 1.2.**

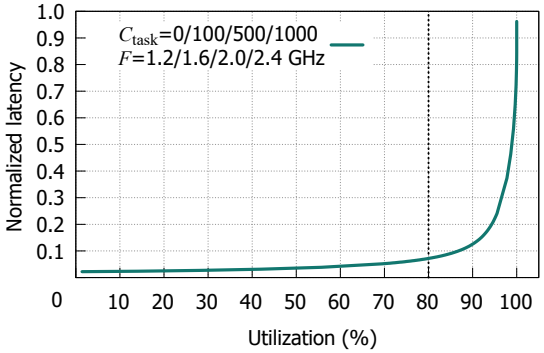



**Fig. 10 Evolution of normalized latency with utilization when service capabilities varying.**

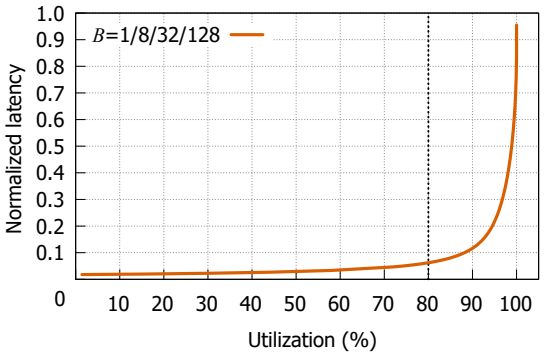



**Fig. 11 Evolution of normalized latency with utilization when *B*=1, 8, 32, and 128.**

evolution:

- In general, the latency is always increasing sharply when the utilization exceeds about 80%. This cliff value is of great significance, because it provides guidance for balancing the utilization and latency. In light of this, we could consolidate traffic processed by multiple underloaded cores into one core, thus saving equipment investment and energy consumption without incurring a large latency increase.
- Given different per-packet processing loads, CPU frequency, or max batch processing size, the normalized latency-utilization characteristic curve remains the same. The only factor that affects the curve is traffic pattern.

**(2) Is batch processing always good choice?** As illustrated in Fig. 7, batch processing could achieve better CPP, latency, and ST. Although the comparative advantage is always true for transmission efficiency and ST, it is not always the case when it comes to latency.

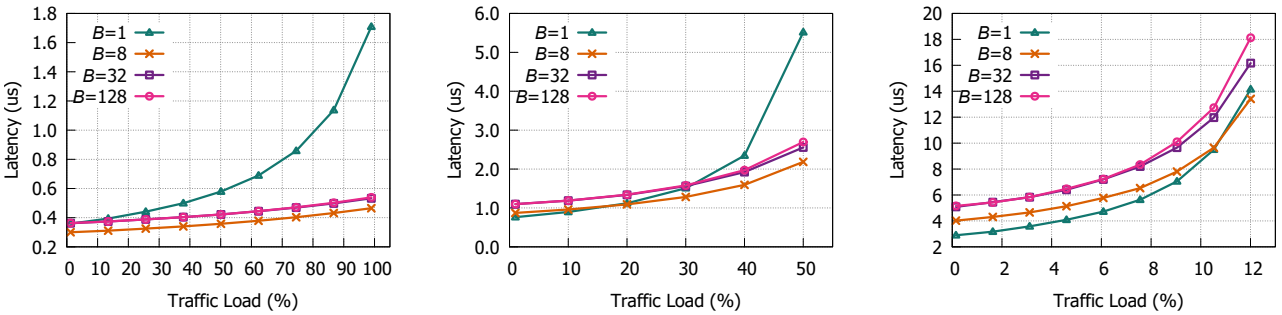
We draw the latency versus traffic load when $B$ =1, 8, 32, 128 and $C_{\text{task}} = 0, 100, 1000$ in Fig. 12 (only parts with utilization below 80% are shown.). It can be seen that with the increase of $C_{\text{task}}$, the comparative advantage of batch processing on latency is gradually lost. When the processing load is up to 1000 cycles, non-batch processing even possesses a few microseconds of advantage over batch processing. The reasons are twofold. When $C_{\text{task}}$ is small, the relative CPP of batch processing are much higher than those of non-batch processing. Thus, the queuing delay of non-batch processing is larger. However, their processing time is not very different, and then batch processing has an advantage. Conversely when $C_{\text{task}}$ is large, batch and non-batch processing have similar CPP and then similar queuing delays. But batch processing will lead to an obviously longer processing time, thus highlighting the advantage of non-batch processing.

In summary, when high-speed packet I/O frameworks serve light task applications (e.g., software switch/router), batch processing is preferred. When heavy task applications, such as intrusion detection, are service recipients, non-batch processing deserves a recommendation.

## 6 Conclusion

This paper concentrates on the modeling and performance evaluation of high-speed packet I/O. To acquire a comprehensive understanding of high-speed packet I/O frameworks, we develop an analytical model to characterize their packet forwarding flow. Our model takes the four main techniques adopted by high-speed packet I/O frameworks into consideration, and our concerned performance metrics, i.e., CPP, ST, and latency, are deduced from this model. The validity and correctness of our model are verified by real system experiments. We also quantify the impacts of various factors on the three metrics by model analysis. Insights and suggestions could be briefly summarized as follows:

(1) An excessively large max batch processing size (e.g., $B > 32$) is detrimental. Also, dynamically adjusting the max batch processing size is not necessary, because $B = 32$ could well balance the CPP, ST, and




(a) $C_{\text{task}} = 0$ (b) $C_{\text{task}} = 100$ (c) $C_{\text{task}} = 1000$

**Fig. 12 Interaction of $B$ and $C_{\text{task}}$ on latency.**

latency.

(2) The latency reaches a cliff point when the system utilization reaches about 80%. And only the traffic pattern will influence the cliff value.

(3) Non-batch processing is recommended when the high-speed packet I/O framework serves heavy task applications, that consume more than several hundreds of CPU cycles to process one packet.

## Acknowledgment

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work was supported in part by the National Key Research and Development Program of China (No. 2018YFB1700103), and the National Natural Science Foundation of China (No. 61872208).

## References

[1] L. Rizzo, Netmap: A novel framework for fast packet I/O, in *Proceedings of USENIX Annual Technical Conference*, Boston, MA, USA, 2012, pp. 101–112.

[2] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, mTCP: A highly scalable user-level TCP stack for multicore systems, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014, pp. 489–502.

[3] Intel, Intel Data Plane Development Kit (DPDK), http://dpdk.org, 2012.

[4] ntop, PF_RING ZC (Zero Copy), http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/, 2014.

[5] S. Han, K. Jang, K. Park, and S. Moon, Packetshader: A GPU-accelerated software router, in *Proc. of ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.

[6] S. Gallenmüller, P. Emmerich, R. Schönberger, D. Raumer, and G. Carle, Building fast but flexible software routers, in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Beijing, China, 2017, pp. 101–102.

[7] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al., The design and implementation of open vSwitch, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Oakland, CA, USA, 2015, pp. 117–130.

[8] Nicira, Open vSwitch with DPDK, http://docs.openvswitch.org/en/latest/intro/install/dpdk/2018, 2018.

[9] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, Clickos and the art of network function virtualization, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014, pp. 459–473.

[10] J. Hwang, K. Ramakrishnan, and T. Wood, Netvm: High performance and flexible networking using virtualization on commodity platforms, *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[11] P. Steve and R. David, Introduction to OpenOnload-building application transparency and protocol conformance into application acceleration middleware, http://www.moderntech.com.hk/sites/default/files/whitepaper/SF-105918-CD-1_Introduction_to_OpenOnload_White_Paper.pdf, 2011.

[12] Snabb Switch: Fast open source packet processing, https://github.com/SnabbCo/snabbswitch, 2020.

[13] L. Rizzo and G. Lettieri, Vale, a switched ethernet for virtual machines, in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, Nice, France, 2012, pp. 61–72.

[14] Fd.io, https://fd.io/, 2018.

[15] Cisco, Trex: Realistic traffic generator, https://trex-tgn.cisco.com/, 2015.

[16] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, MICA: A holistic approach to fast in-memory key-value storage, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014, pp. 429–444.

[17] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, Comparison of frameworks for high-performance packet IO, in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Oakland, CA, USA, 2015, pp. 29–38.

[18] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, Assessing soft-and hardware bottlenecks in PC-based packet forwarding systems, in *Proceedings of the 14th International Conference on Networks*, Barcelona, Spain, 2015, pp. 78–83.

[19] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, Throughput and latency of virtual switching with open vswitch: A quantitative analysis, *Journal of Network and Systems Management*, vol. 26, no. 2, pp. 314–338, 2018.

[20] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. TranGia, Modeling and performance evaluation of an openflow architecture, in *Proceedings of the 23rd International Teletraffic Congress*, San Francisco, CA, USA, 2011, pp. 1–7.

[21] R. Bolla, R. Bruschi, A. Carrega, and F. Davoli, Green networking with packet processing engines: Modeling and optimization, *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 110–123, 2014.

[22] Z. Su, B. Baynat, and T. Begin, A new model for DPDK-based virtual switches, in *Proceedings of 2017 IEEE Conference on Network Softwarization*, Bologna, Italy, 2017, pp. 1–5.

[23] L. Zabala, A. Ferro, and A. Pineda, Modelling packet capturing in a traffic monitoring system based on linux, in *Proceedings of 2012 International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, Genoa, Italy, 2012, pp. 1–6.

[24] K. Salah, K. El-Badawi, and F. Haidari, Performance analysis and comparison of interrupt-handling schemes in gigabit networks, *Computer Communications*, vol. 30, no. 17, pp. 3425–3441, 2007.

[25] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese, Efficiently measuring bandwidth at all time scales, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, 2011, pp. 71–84.

[26] T. Benson, A. Anand, A. Akella, and M. Zhang, Understanding data center traffic characteristics, *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[27] P. Salvador, A. Pacheco, and R. Valadas, Modeling IP traffic: Joint characterization of packet arrivals and packet sizes using BMAPs, *Computer Networks*, vol. 44, no. 3, pp. 335–352, 2004.

[28] A. Klemm, C. Lindemann, and M. Lohmann, Modeling IP traffic using the batch Markovian arrival process, *Performance Evaluation*, vol. 54, no. 2, pp. 149–173, 2003.

[29] A. Erramilli, O. Narayan, and W. Willinger, Experimental queueing analysis with long-range dependent packet traffic, *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, pp. 209–223, 1996.

[30] W. Willinger, V. Paxson, and M. Taqqu, Self-similarity and heavy tails: Structural modeling of network traffic, *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, vol. 23, pp. 27–53, 1998.

[31] M. Chaudhry and J. G. Templeton, *A First Course in Bulk Queues*. New York, NY, USA: Wiley, 1983.

[32] R. Nelson, *Probability, Stochastic Processes, and QueueingTheory: The Mathematics of Computer Performance Modeling*. Berlin, Germany: Springer Science & Business Media, 2013.

[33] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, MoonGen: A scriptable high-speed packet generator, in *Proceedings of the 2015 Internet Measurement Conference*, Tokyo, Japan, 2015, pp. 275–287.

[34] CAIDA, The CAIDA Anonymized Internet Traces 2016 Dataset-20160406, http://www.caida.org/data/passive/passive_2016_dataset.xml, 2016.

**Xuesong Li** received the BEng degree from Tsinghua University in 2013, and the MS degree from Xi'an Research Institute of Hi-Tech in 2015. He currently is a PhD candidate jointly trained by Xi'an Research Institute of Hi-Tech and Tsinghua University, under the advising of Prof. Bailong Yang and Prof. Fengyuan Ren. His research interests include high-speed packet I/O and energy-aware data center network.

**Fengyuan Ren** received the BS and MS degrees from Northwestern Polytechnic University, Xi'an, China in 1993 and 1996, respectively. In Dec. 1999, he obtained the PhD degree from Northwestern Polytechnic University. He is now a professor at the Department of Computer Science and Technology, Tsinghua University, Beijing, China. From 2000 to 2001, he worked at the Electronic Engineering Department of Tsinghua University as a post doctoral researcher. In Jan. 2002, he moved to the Computer Science and Technology Department of Tsinghua University. His research interests include network traffic management and control, control in/over computer networks, and wireless networks and wireless sensor networks. He authored / co-authored more than 80 international journal and conference papers. He is a member of the IEEE, and has served as a technical program committee member and local arrangement chair for various IEEE and ACM international conferences.

**Bailong Yang** received the BS and MS degrees from Xi'an Research Institute of Hi-Tech, Xi'an, China in 1990 and 1993, respectively. He received the PhD degree from Xi'an Research Institute of Hi-Tech, Xi'an, China in 2001. He is currently a professor of Xi'an Research Institute of Hi-Tech. His research interests include complex network and computer simulation.