# Improving the packet send-time accuracy in embedded devices

Ricky K. P. Mok, Weichao Li, and Rocky K. C. Chang

Department of Computing, The Hong Kong Polytechnic University
{cskpmok|csweicli|csrchang}@comp.polyu.edu.hk

**Abstract.** A number of projects deploy Linux-based embedded systems to carry out large-scale active network measurement and network experiments. Due to resource constrains and the increase of network speed, obtaining sound measurement results from these low-end devices is very challenging. In this paper, we present a novel network primitive, `OMware`, to improve the packet send-time accuracy by enabling the measurement application to pre-dispatch the packet content and its schedule into the kernel. By this pre-dispatch approach, `OMware` can also reduce the overheads in timestamp retrievals and sleeping, and the interference from other application processes.
Our evaluation shows that `OMware` can achieve a microsecond-level accuracy (rather than millisecond-level in a user-space tool) in the inter-departure time of packet trains, even under heavy cross traffic. `OMware` also offers optimized call for sending back-to-back packet pairs, which can reduce the minimum inter-packet gap by 2 to 10 times. Furthermore, `OMware` can help reduce the error of replaying archived traffic from 40% to at almost 19%.

## 1 Introduction

Linux-based embedded devices are ubiquitous. For example, many homes use home routers or WiFi APs for sharing the residential broadband access. Some of them run OpenWrt [27], a popular Linux distribution for networked embedded devices, which allows developers to re-use the software tools implemented for the PCs via cross compilation. Due to their low cost, several projects, such as BISMark [1], SamKnows [8], and RIPE Atlas [6], employ them as vantage points to measure the Internet performance or gauge the network service quality of residential broadband. ARM-based single-board computers, such as Raspberry Pi [5], are also used in sensor network and embedded cloud research.

Obtaining sound measurement results from these resource-constrained devices is however very challenging. A fundamental requirement is to send out (probe or archived) packets onto the wire according to their pre-determined schedules. Inaccurate packet send times will distort the scheduled probe patterns (e.g., Poisson, periodic, and Gamma renewal) in active measurement which may result in non-optimal probing [9]. Inaccurate packet send times can also directly affect the measurement results for timing-sensitive measurement, notably packet-pair capacity [20, 12] and available bandwidth [19].

A major source of send-time inaccuracy is the high overhead for these devices to move packets between user and kernel space and in executing the sleep and timestamping function calls. These overheads will widen the gap between the scheduled send time and the actual time of delivering the packet to the wire. Another problem is to contend resources with other running processes (E.g., firewall, NAT, and DNS request forwarding in a residential broadband router). Due to the CPU context switching, the measurement tool will experience highly fluctuated overheads which cannot be calibrated easily. A traffic generator may even fail to send the expected pattern when the CPU consumption is high [10].

In this paper, we propose `OMware`, a new network primitive to improve the send-time accuracy. Its main novelty is on utilizing the sleep period typically required for a packet sending process to copy packets from user space to kernel and construct the `sk_buff` structure [24] for the network card driver. As a result, the first pre-dispatching phase "absorbs" these operations' overheads. In addition, `OMware` offers optimized function calls for sending back-to-back packet pairs, which can improve the accuracy of capacity and available bandwidth measurement [20, 12, 11, 21].

We evaluate `OMware` with two OpenWrt routers (NETGEAR WNDR 3800 and TP-LINK WR1043ND) and perform a set of experiments under different levels of cross traffic to investigate the improvement in network measurement. The results show that `OMware` can achieve a microsecond-level accuracy (rather than millisecond-level in a user-space tool) in the inter-departure time (IDT) of packet trains even under heavy cross traffic. Besides, the packet sending delay can be significantly reduced by 0.2 ms. Furthermore, `OMware` can reduce the IDT in a back-to-back packet pair by 2 to 10 times, therefore enabling the embedded device to measure a much higher capacity.

## 2 Related Works

There are generally two approaches to increase the packet I/O performance—hardware and kernel. The hardware approach adopted by SoNIC [23], NetFPGA [4], and [14] uses programmable network interface cards to improve the precision of packet sending time and receiving timestamp. However, these cards are usually expensive, thus prohibiting them from being used in embedded devices, such as residential broadband routers. Intel recently proposes the DPDK library [18] to boost packet processing performance. However, this library is only supported by their Xeon series CPU which is not available in many embedded systems.

The kernel approach runs on commercial PCs and optimizes the operating system's kernel to increase the performance. Examples include PF_RING [15] for improving packet capturing performance, and nCap [16], netmap [29], and kTRxer[30] for improving both sending and receiving speed. `Epoll` in Linux and `kqueue` in FreeBSD are mainly for improving the event notification mechanism, which can enhance the performance of packets reception. On the other hand, pktgen [26] aims at a high-speed packet sending. However, they do not consider the accuracy of packet send time. Using real-time Linux (RTLinux [7]) is a possible solution to increase the packet send-time accuracy. For example, Kiszka et al. propose RTnet [22] for hard real-time networking. However, running RTLinux on residential broadband router may significantly affect the performance of running other network services.

In wireless sensor network community, Österlind and Dunkels [28] proposed to pre-copy packet to improve the packet forwarding throughput in 802.15.4 networks, but the application cannot send packets at any dedicated time.

## 3 Background

Linux-based embedded devices, such as home routers and private NASes (Network Access Storage), can be found in many homes today and of low cost. Some of them support OpenWrt, which is one of the popular and active Linux distributions specifically for embedded devices. Furthermore, the packages of several network measurement tools, including D-ITG, `httping`, and `hping`, are readily available on public repositories. Developers can also run their own tools via cross compilation. However, the computational power of these devices are far lower than commodity PCs. Table 1 shows the detailed configurations of three testing devices, including NETGEAR WNDR 3800, which has the same configurations as a BISMark-enabled router, and a reference PC.

**Table 1.** The configurations of the testing devices.

| Device Model | CPU/Chipset (Clock Freq.) | RAM |
|---|---|---|
| Raspberry Pi | BCM2835 (700 MHz) | 512 MB |
| TP-LINK WR1043ND | AR9132 (400 MHz) | 32 MB |
| NETGEAR WNDR3800 | AR7161 (680 MHz) | 128 MB |
| Reference PC | Intel Core2Duo (1.86 GHz) | 2 GB |

Note: All embedded devices are running OpenWrt 12.09.1.
All devices expect Raspberry Pi support 1 Gbps Ethernet.

Due to resource limitations, the performance and accuracy of these devices are not satisfactory, especially in today's high speed network. We have identified three basic operations—timestamp retrieval, sleep, and packet transmission—could cause performance degradation. These operations are commonly used in network tools. In the following, due to page limitation, we can only show the performance issues in packet transmission, which cause the most significant impact.

### 3.1 Packet sending performance

We define the packet sending performance by the time period between the calling of `sendto()` and the packet is put on wire, because some tools may regard the calling of `sendto()` as the packet sending time. Previously, Rizzo showed that the time period 950 ns in his high-end FreeBSD host (Intel i7-870 with 10 Gbit NIC) [29]. However, we found that tens of microseconds are required in the embedded devices.

Instead of forcing the functions to return early [29], our benchmark program repeatedly flushes out 100,000 identical TCP packets using the raw socket (i.e., `sendto()`). Besides, the packet's TCP/IP header and checksums are pre-computed to mitigate any overhead from these operations. We repeat the experiment with five packet sizes, which are {40, 200, 500, 1000, and 1500} bytes. All the packets are captured by an endace DAG card directly connected to the device. We then analyze the IDTs between packets to estimate the overall sending performance.

Figure 1 shows the average packet IDTs against the packet sizes. We can see that the performance of the three embedded devices is about one order of magnitude slower than a commodity PC. For example, the average packet IDT for 40-byte packets is 2.64 $\mu$s, while the NETGEAR router is 41.7 $\mu$s. The Raspberry Pi performs the worst among the embedded device, because the Raspberry Pi's ethernet interface connects to CPU via the USB interface and results in poor performance. Unlike the reference PC, the performance is fairly stable across the packet sizes in all three embedded devices. The average packet IDTs for TP-LINK and NETGEAR only respectively increase by 5% and 8% as the packet size increases from 40 bytes to 1500 bytes.

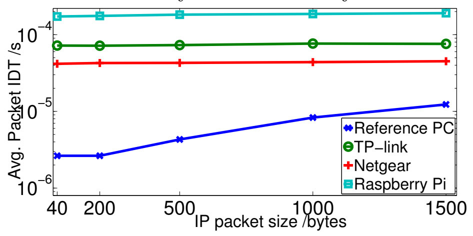


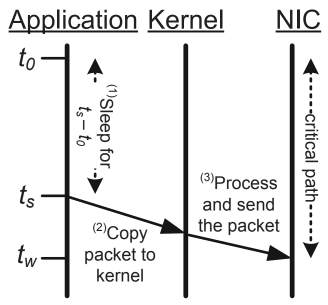
**Fig. 1.** The average packet IDT against packet size on all devices.

## 4 Pre-dispatch Programming Model

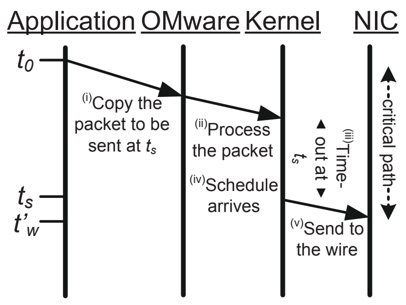
We survey several network tools listed in Table 2. We find that these tools are often implemented with similar kind of function calls in packet I/O, sleep and timestamp retrieval. We further investigate their source code and programming flows. These tools often adopt a *sequential* programming model to schedule the sending of packets. Figure 2(a) and 2(b) illustrate a timeline comparison between the sequential model and our proposed pre-dispatch model, respectively. The application in the figures refers to a network tool running on the user space. For both model, at time $t_0$, we assume the application has prepared the packet content to be sent at a future time, $t_s$. The packet appears on the wire at $\{t_w, t'_w\}$ in {sequential, pre-dispatch} model. Therefore, the sending time errors are $(t_s - t_0)$ or $(t'_s - t_0)$ for sequential or pre-dispatch model, respectively.

**Table 2.** Examples of function calls used in network tools.

| Tools | Packet I/O | Sleep | Timestamp Retrieval |
|---|---|---|---|
| D-ITG [13] | POSIX Socket | `select()` and polling | `gettimeofday()` |
| httping [2] | POSIX Socket | `usleep()` | `gettimeofday()` |
| Iperf [3] | POSIX Socket | `nanosleep()` | `gettimeofday()` |



(a) Sequential Model.



(b) Pre-dispatch Model.

**Fig. 2.** Timeline comparison between the sequential and pre-dispatch approaches.

We first consider the sequential model in Figure 2(a). The applications using this model are usually implemented using POSIX socket for packet I/O and a family of `sleep()` functions for spacing out packets. We summarize this model into three major steps.

(1) The application prepares the packet content, computes the sleep period (i.e., $t_s - t_0$, for $t_s > t_0$) and goes into sleep mode.
(2) After the sleep period is over, the packet content is copied to the kernel using socket.
(3) The packet headers are filled by the TCP/IP stack and finally sent to the network card.

On the other hand, our pre-dispatch model, as shown in Figure 2(b), divides the packet sending process into two major phases. The tool first prepares and copies the packet to the `OMware` before the scheduled sending time, $t_s$. Then, the `OMware` sends the packet when $t_s$ arrives. We can describe the details with five steps:

(i) Once the packet is ready and the sending time is determined, the application can immediately invoke the packet sending call in the `OMware` API, which takes the pointer of packet and the sending time as the input.
(ii) The `OMware` processes the packet, which includes adding ethernet header and constructing `sk_buff` structure.
(iii) If the packet sending time does not arrive (i.e., current time $< t_s$), `OMware` will add the packet sending operation as a kernel task triggered by a high resolution timer. Otherwise, the packet should be sent immediately.
(iv) When the scheduled send time $t_s$ arrives, an interrupt will be generated to trigger the callback routine of sending the processed packet.
(v) As the packet has been processed, it can be put onto the wire quickly. The `OMware` API then acknowledges the application on whether the process is successful.

The major difference between the two models is when the program starts to wait (i.e., (1) and (iii)) for the scheduled time. The pre-dispatch model utilizes part of the sleep time to handle time consuming operations, such as (i) and (ii).

Therefore, the system can take a shorter critical path in sending packets and improve the throughput.

## 5 Evaluation

In this section we evaluate the packet send-time accuracy, pre-dispatching period, packet-pair accuracy, and packet send timestamp accuracy on a testbed. To support the pre-dispatch model, we have implemented `OMware`, which is a loadable kernel module for Linux. `OMware` provides a set of APIs for network tools. We cross-compile `OMware`, so that our experiments can run on two home routers, NETGEAR WND3800 and TP-LINK WR1043ND, both of which are installed with OpenWrt 12.09.1.

### 5.1 Testbed and Test Suite

We setup a testbed, as shown in Figure 3, to emulates a network environment with cross traffic. The WAN port of the OpenWrt router, $D0$, is directly connected to an endace DAG Card 4.5G2 [17] with 1 Gbps Ethernet for capturing the traffic sending from $D0$. The server installed with the DAG card, $S0$, runs the `dagfwddemo` program, so it can forward the traffic from $D0$ to a Linux host, $S1$, and the cross traffic. $X0$ and $X1$ are two Linux hosts for generating cross traffic using D-ITG, where $X0$ is behind the NAT provided by $D0$. The cross traffic is unidirectional UDP flows generated by D-ITG [13]. Each flow is configured with Pareto distributed packet inter-arrival times and uniform distribution of packet size over $\{40, 1500\}$ bytes. The bitrate for each flow is about 2200 kbps, and the packet sending rate is about 352 packets/s.

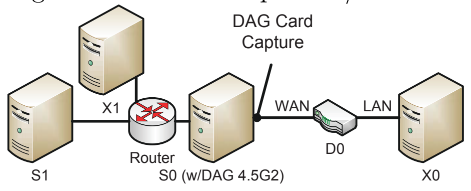


**Fig. 3.** Testbed for the performance tests.

We have implemented a simple network measurement tool running with `OMware` and different programming models for comparing the performance and timing accuracy between our approach and raw socket. Table 3 lists the details of a test suite. The packet train tests send a train of evenly spaced TCP data packets to the WAN port. According to their memory capacities, the train has 100 packets for the NETGEAR router and 50 packets for the TP-LINK router. The packet pair tests send 50 and 25 pairs of back-to-back packets to the WAN port. Both tests use different packet sizes, inter-departure times between packets or packet pairs, and degree/ direction of cross traffic. The parameters used are listed in Table 4. The packet send time is recorded by both the measurement tool using `OMware` and the DAG card.

**Table 3.** The test suite for evaluating `OMware`.

| Methods | Packet pattern | Library | Model | Description |
|---|---|---|---|---|
| *OIR* | Packet train | `OMware` | `OMware` (initial pre-dispatching) | The tool prepares all the probe packets and their sending timestamps in advance and sends them to `OMware` for pre-dispatching the sending of the packets. |
| *OFR* | Packet train | `OMware` | `OMware` (on-the-fly pre-dispatching) | The tool uses `clock_nanosleep()` with absolute timestamp to sleep until $\phi$ $\mu$s before the scheduled send time. Then, it prepares the probe packets and sends them to `OMware` for pre-dispatching the sending of the packets. |
| *OSM* | Packet train | `OMware` | Sequential | This method is a special case of *OFR* method where $\phi$ is zero. |
| *RSM* | Packet train | POSIX | Sequential | This method uses raw socket for sending packets. Similar to *OSM*, `clock_nanosleep()` with absolute timestamp is used for spacing the probe packets. |
| *TOM* | Packet pairs | `OMware` | `OMware` | This method employs the packet pair sending function in `OMware` to send a sequence of packet pairs with initial pre-dispatching. |
| *TRW* | Packet pairs | POSIX | Sequential | This method uses raw socket to send a sequence of packet pairs. |

**Table 4.** The parameters used in evaluating packet sending performance.

| Parameters | Values |
|---|---|
| No. of cross traffic flows, $\rho$ | 0, 1, 5, 10, 20, 30 |
| Direction of cross traffic, $\boldsymbol{\rho}$ | WAN$\rightarrow$LAN, WAN$\leftarrow$LAN |
| IP packet size, $\lambda$ (bytes) | 40, 200, 500, 1000, 1500 |
| Expected inter-departure time, $\alpha$ ($\mu$s) | 0, 10, 100, 1000, 10000, 100000 |
| Pre-dispatching period for the *OFR* method, $\phi$ ($\mu$s) | 0, 100, 500, 1000 |

## 5.2 Packet Send-Time Accuracy

We use the timestamps from the packet capture to compute the actual packet IDT sent from the router by $IDT = ts_{n+1} - ts_n$, where $ts_n$ represents the timestamp of the $n^{th}$ packet in the packet train. Figure 4 shows a log-log plot of average packet IDT against the expected IDT, $\alpha$, in the idle NETGEAR router. The error bars plot the 95% confidence interval of data. We can see that the *OIR* method outperforms the other three, especially in very small packet IDT (10 and 100 $\mu$s). But the variation for 10 $\mu$s case is quite large, as this IDT is close to the limit of the system. The *OFR* method becomes more accurate when the expected IDT increases to 100 $\mu$s as the pre-dispatching can take place after sending the first few packets. The *OSM* and *RSM* methods improve their accuracy when the IDT is larger than 1 ms.

Figures 5(a) to 5(d) are four box-and-whisker plots respectively showing the summary of data of the *OIR*, *OFR*, *OSM*, and *RSM* methods where the expected packet IDT is set to 1 ms. In each box-and-whisker plot, the top/bottom of the

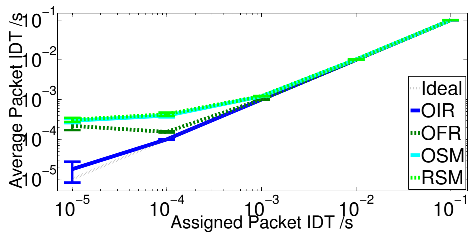


**Fig. 4.** The log-log plot of measured average packet IDT against assigned one for OIR, OFR ($\phi$=1000 $\mu$s), OSM, and RSM ($\rho = 0$, $\lambda = 40$ bytes, $\alpha \geq 10\mu s$, NETGEAR).

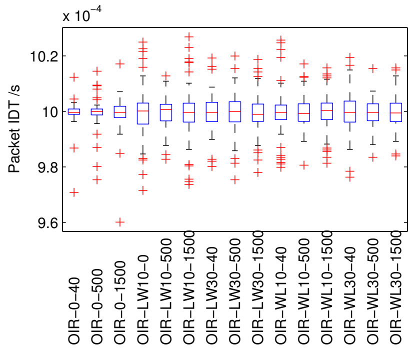
box are given by the 75th/25th percentile, and the mark inside is the median. The upper/lower whiskers are the maximum/minimum, respectively, after excluding the outliers. The outliers above the upper whiskers are those exceeding 1.5 of the upper quartile, and those below the minimum are less than 1.5 of the lower quartile. Each figure shows 15 test cases with different degrees/directions of cross traffic and packet sizes. For example, *OIR-0-500* on the x-axis in Figure 5(a) represents the results obtained from *OIR* method under '0' cross traffic (idle) and sending 500-byte IP packets; WL*XX* or LW*XX* represents the experiment runs with *XX* flows of cross traffic in WAN→LAN or WAN←LAN direction, respectively.

We can see that the *OIR* method is the most stable against the cross-traffic. Most of the IDTs fall within ± 20 $\mu$s of the true value. The *OFR* method also shows an accurate median value. But the cross traffic slightly affects this method's accuracy. The inter-quartile range increases with the number of cross traffic flows. Without adopting the pre-dispatching technique, the *OSM* method shows even larger inter-quartile range (about 1 ms) for all cases, which is caused by the inaccuracy of sleep function. Finally, the *RSM* method shows the worst result. All the IDTs suffer from at least 0.2 ms inflation. Besides, this method is also susceptible to cross traffic interference. When the WAN←LAN cross traffic is heavy (e.g., LW30), the inter-quartile range shows a six-fold increase. To summarize, traditional method (*RSM*) experiences larger delay and variance in sending packets than `OMware`-based methods.

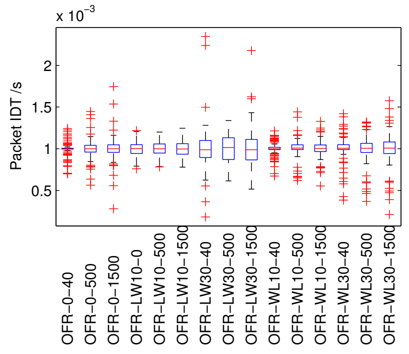
## 5.3 Pre-dispatching Period

Another important issue is the length of pre-dispatching period in the *OFR* method. Some stateful measurement tools, such as OneProbe [25], requires the information from the previous probe packets to generate a new one. Preparing all probes packets at the beginning of the measurement becomes infeasible. As shown in the previous section, `OMware` cannot pre-dispatch probes if the packet send time is too close to the current time. Therefore, we test four different pre-dispatching periods and examine their effects on the packet send-time accuracy.

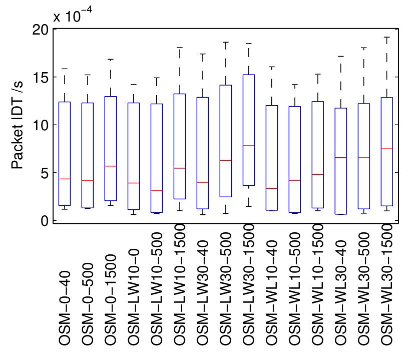
Figures 6(a) and 6(b) show the CDFs of the packet IDTs with different pre-dispatching periods using an expected packet IDT of 10 $\mu$s and 1000 $\mu$s,



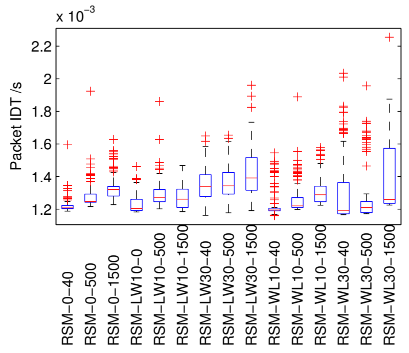
(a) *OIR*.



(b) *OFR* ($\phi$=1000 $\mu$s).



(c) *OSM*.



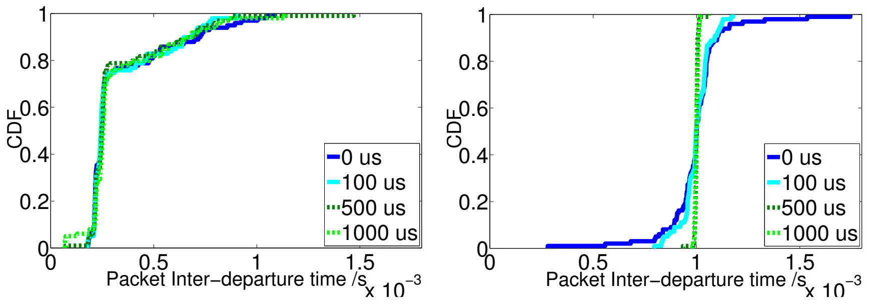
(d) *RSM*.

**Fig. 5.** The box-and-whisker plots of packet IDTs using different methods ($\alpha = 1$ ms, NETGEAR router).

respectively. When the expected packet IDT is very small (e.g., $\alpha = 10$ $\mu$s), the pre-dispatching period cannot improve the accuracy. It is because the requested IDT is insufficient for `OMware` to finish the pre-dispatching phase before the scheduled send time. However, when the expected packet IDT increases to 1 ms, the fluctuation of the packet IDTs can be significantly decreased when the pre-dispatching period increases to 500 $\mu$s (as shown in Figure 6(b)). We also found similar pattern in other cases. Therefore, we conclude that the pre-dispatching period of 500 $\mu$s is sufficient for completing the first part of packet dispatchment in this router.

## 5.4 Packet-pair Accuracy

`OMware` provides a dedicated API for sending back-to-back packet pairs. A smaller gap between the two packets can enable us to measure a higher capacity using packet pair based methods (e.g., [20, 11]). Figures 7(a) and 7(b) plot the CDFs

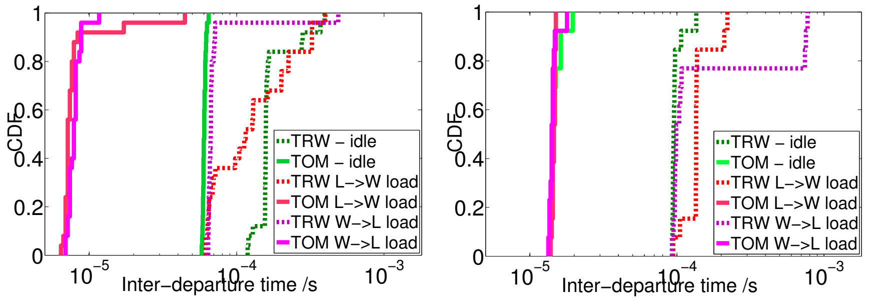


(a) $\alpha = 10\mu$s.

(b) $\alpha = 1000\mu$s.

**Fig. 6.** The CDFs of packet inter-departure time of the OFR method with different pre-dispatching period ($\rho = 0$, $\lambda = 1500$ bytes, NETGEAR router).

of the back-to-back packet pairs' IDTs under different degrees of cross traffic in the NETGEAR and TP-LINK router, respectively. We set a 1 ms gap between each pair to mitigate the influence from the previous pair. We can see that the NETGEAR router can achieve a minimum IDT of 6.44 $\mu$s, while the TP-LINK one only can reach 13.6 $\mu$s. They can achieve 2 to 10 times improvement against the raw socket version under the same condition.



(a) NETGEAR.

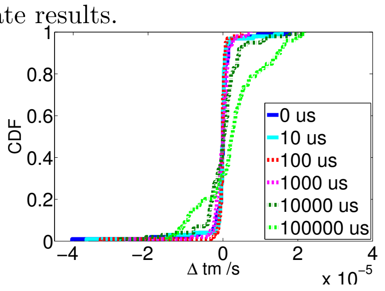
(b) TP-LINK.

**Fig. 7.** The CDFs of back-to-back packet pairs' inter-departure time. ($\alpha$ =1 ms and $\lambda$ =40 bytes).

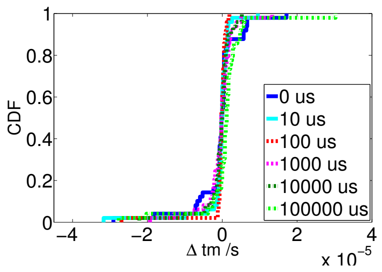
### 5.5 Packet Send Timestamp Accuracy

In most cases, the measurement tools cannot rely on external timestamping device, such as DAG card, to provide precise packet send timestamp. The tools have to rely on the send timestamp reported by `OMware`. To appraise the accuracy of the timestamps, we subtract the packet IDTs computed by two time sources, $\Delta tm = IDT_{OMware} - IDT_{DAG}$, where $IDT_{OMware}$ and $IDT_{DAG}$ are the packet IDTs of the same pair of packets, but computed using the timestamps reported by the `OMware` and those captured by the DAG card, respectively. Figures 8(a) and 8(b) show the CDFs of $\Delta tm$ for the NETGEAR and TP-LINK routers.

We can see that the packet IDT difference computed by the two time sources are very close. `OMware`'s timestamp accuracy can generally reach micro-second level. Therefore, the measurement tools can use `OMware`'s timestamp to compute accurate results.



(a) NETGEAR.



(b) TP-LINK.

**Fig. 8.** The CDF of the difference between packet IDTs computed by the DAG card capture and send timestamps reported by `OMware` ($\rho = 0$ and $\lambda = 40$ bytes).

## 6 Conclusions

This paper proposed a novel network primitive to improve the packet send-time accuracy. The model employs a two-phase approach to allow pre-dispatch of packets to reduce the impact from the low packet sending performance. Our implementation, `OMware`, allows the tools to buffer probe packets and their send times in the kernel before their actual send time. Hence, the packet send-time accuracy and sending rate can be significantly improved.

Our testbed evaluation results showed that using `OMware` to pre-dispatch packets can provide accurate packet send times. Comparing to raw socket, OMware can reduce the minimum packet inter-departure time by ten times and reduce the variation by 6 times under heavy load cross traffic. In the future, we will compare the performance of `OMware` in more embedded devices and investigate the performance impact to other applications.

**Acknowledgements:** We thank three anonymous reviewers for their valuable comments. We also thank Lei Xue, McKeith Kwok, and Jack Chan for their help in programming the prototype of `OMware` and carrying out experiments. This work is partially supported by an ITSP Tier-2 project grant (ref. no. GHP/027/11) from the Innovation Technology Fund in Hong Kong.

## References

1. Bismark. `http://www.projectbismark.net`.
2. httping. `http://www.vanheusden.com/httping/`.
3. Iperf - The TCP/UDP Bandwidth Measurement Tool. `http://iperf.fr/`.
4. NetFPGA. `http://netfpga.org/`.
5. Raspberry Pi. `http://www.raspberrypi.org/`.

6. RIPE Atlas. `https://atlas.ripe.net/`.
7. RTLinux. `https://rt.wiki.kernel.org/`.
8. Samknows. `http://www.samknows.com`.
9. F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot. On optimal probing for delay and loss measurement. In *Proc. ACM IMC*, 2007.
10. A. Botta, A. Dainotti, and A. Pescapé. Do you trust your software-based traffic generator? *IEEE Commuication Magazine*, 48(9):158–165, 2010.
11. E. Chan, A. Chen, X. Luo, R. Mok, W. Li, and R. Chang. TRIO: Measuring asymmetric capacity with three minimum round-trip times. In *Proc. ACM CoNEXT*, 2011.
12. E. Chan, X. Luo, and R. Chang. A minimum-delay-difference method for mitigating cross-traffic impact on capacity measurement. In *Proc. ACM CoNEXT*, 2009.
13. A. Dainotti, A. Botta, and A. Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.
14. L. Degioanni and G. Varenni. Introducing scalability in network measurement: toward 10 Gbps with commodity hardware. In *Proc. ACM IMC*, 2004.
15. L. Deri. Improving passive packet capture: Beyond device polling. In *Proc. SANE*, 2004.
16. L. Deri. nCap: Wire-speed packet capture and transmission. In *Proc. IEEE E2EMON*, 2005.
17. Endace. DAG packet capture cards. `http://www.endace.com`.
18. Intel. Packet processing on intel architecture. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html.
19. M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Trans. on Networking*, 11:537–549, 2003.
20. R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi. CapProbe: A simple and accurate capacity estimation technique. In *Proc. ACM SIGCOMM*, 2004.
21. J. C. Kim and Y. Lee. An end-to-end measurement and monitoring technique for the bottleneck link capacity and its available bandwidth. *Computer Networks*, 58:158–179, 2014.
22. J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink. RTnet - A flexible hard real-time networking framework. In *Proc. IEEE ETFA*, 2005.
23. K. S. Lee, H. Wang, and H. Weatherspoon. SoNIC: Precise realtime software access and control of wired networks. In *Proc. USENIX NSDI*, 2013.
24. Linux Foundation. `sk_buff`. `http://www.linuxfoundation.org/collaborate/workgroups/networking/skbuff`.
25. X. Luo, E. Chan, and R. Chang. Design and implementation of TCP data probes for reliable and metric-rich network path monitoring. In *Proc. USENIX ATC*, 2009.
26. R. Olsson. pktgen the Linux packet generator. In *Proc. Linux Symposium*, 2005.
27. OpenWrt. `https://openwrt.org/`.
28. F. Österlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *Proc. ACM HotEmNets*, 2008.
29. L. Rizzo. netmap: A novel framework for fast packet I/O. In *Proc. USENIX ATC*, 2012.
30. L. Xue, X. Luo, and Y. Shao. kTRxer: A portable toolkit for reliable internet probing. In *Proc. IEEE IWQoS*, 2014.