Design and Evaluation of Precise Software Pacing Mechanisms for Fast Long-Distance Networks

Ryousei Takano[1,2], Tomohiro Kudoh[1], Yuetsu Kodama[1],
Motohiko Matsuda[1], Hiroshi Tezuka[1] and Yutaka Ishikawa[3,1]
[1] Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology (AIST)
[2] AXE, Inc.
[3] University of Tokyo
Email: takano@axe-inc.co.jp

Abstract— **In this paper, we propose precise software pacing mechanisms at end nodes of communication paths. First, we propose an Inter Packet Gap (IPG) control mechanism, which inserts gap packets between packets. The gap packets are inserted by software at the sender node, and no additional hardware is required. By adjusting the size of the inserted gap packets, the pacing ratio is precisely controlled. Then, we propose an IPG-aware packet scheduling mechanism, in order to merge multiple streams which go through each of different bottleneck links, where the pacing ratios are appropriately maintained.**

These two mechanisms are implemented as a Linux kernel module, and no kernel re-compilation is required for installation. The effectiveness of the pacing mechanism using gap packets is shown by evaluating TCP/IP communication performance in a wide area network emulation environment. As a result, the physical bandwidth of the path is almost fully utilized by using the proposed mechanism.

I. Introduction

There are two principle issues concerning TCP/IP communication over fast long-distance networks: window size control and burstiness control. The former is used to control the amount of transmission in each Round Trip Time (RTT) period. The latter is used to control the amount of transmission more precisely during an RTT period.

The window size control schemes focused on fast long-distance networks have been widely investigated, and many schemes have been proposed. For example, Additive Increase Multiplicative Decrease (AIMD) variant algorithms such as HighSpeed TCP[4], Scalable TCP[6] or BIC-TCP[7] estimate the available bandwidth and determine the window size based on the ratio of packet losses. FAST TCP[8] estimates based on the queuing delay caused by buffering on intermediate routers.

In this paper, we focus on burstiness control. Burstiness control is an operation intended to adjust the transmission rate to the available bandwidth (i.e. target rate) in a fine granularity of time during an RTT period. Effective burstiness control minimizes the possibility of overflow of the buffers of intermediate routers, and realizes a low packet loss rate. Most TCP/IP implementations use ACK-clocking[3] for burstiness control. However, ACK-clocking does not work properly during slow-start at the beginning of a connection, after a packet loss, or when an idle connection resumes. Pacing[9] is a more aggressive burstiness control scheme which directly controls the timing of the transmission of each packet. The packets are transmitted at the $RTT/window_size$ interval based on the target rate on the sender side. The more precise the pacing is, the fewer the packet losses expected at intermediate routers, and the more effectively the physical bandwidth of a network path can be utilized.

We have shown that pacing can increase the utilization efficiency of the physical bandwidth of fast long-distance networks through experiment using trans-pacific networks during the SC2003 Bandwidth Challenge (BWC)[1]. In this experiment, we put special hardware called GtrcNET-1[1] [2] between switches to pace the traffic. GtrcNET-1 controls the Inter Packet Gap (IPG). An IPG is a gap[2] between sequentially transmitted packets. A larger IPG size will decrease the effective bandwidth by $packet_size/(packet_size+IPG)$. A part of the network configuration used in the SC2003 BWC[3] is shown in Figure 1. We employed six nodes each at Phoenix and Tokyo. At Phoenix, the six nodes were subdivided into three groups, and connected to a GbE switch, GtrcNET-1, then to a 10GbE switch (E600) using trunked GbE links. The bottleneck link bandwidth is 2.4Gbps (OC-48 POS), and the RTT is 141ms, on average. In addition, we used HighSpeed TCP and the maximum socket buffer size is set to 8MB. Figure 2 shows results without pacing and with pacing. In the without pacing case, the three GbE links generate traffic exceeding the bottleneck link, and buffer overflow of intermediate routers results in packet losses. In the with pacing case, on the other hand, each of the three GbE links generate stable traffic with 87.5MB/s using GtrcNET-1, and no packet losses occur.

When GtrcNET-1 is used, since the pacing is performed using a hardware clock, extremely precise pacing is realized. However, the use of such special hardware draws additional cost, and such equipment is not always available. On the other hand, some software-based pacing schemes have been proposed. Such schemes realize pacing at the end computing

[1] GtrcNET-1 was formerly called GNET-1.

[2] In most GbE NICs, the IPG is set to 12 bytes, in which case the delay is 96nsec.

[3] In addition to the 2.4Gbps APAN/TransPAC link, we used other two links and achieved a total of 3.78Gbps (97% of the total physical bandwidth) between US and Japan.

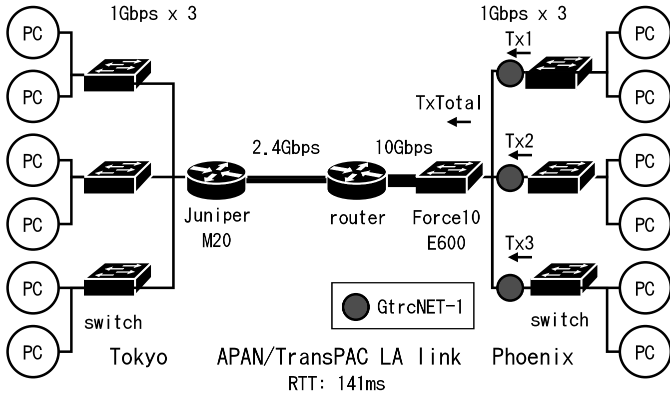
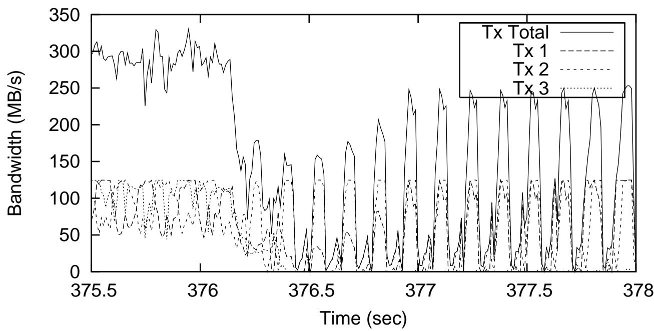
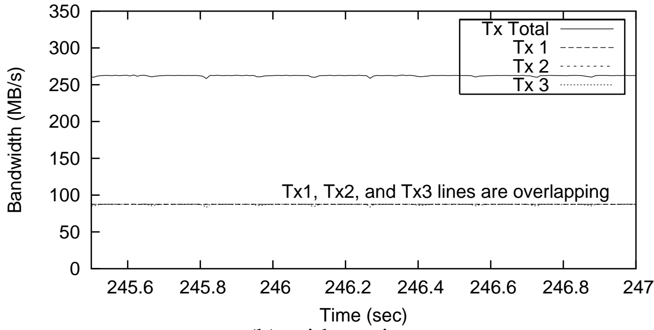


Fig. 1. Trans-pacific file replication experiment using GtrcNET-1 in the SC2003 Bandwidth Challenge



(a) without pacing



(b) with pacing

Fig. 2. The effect of pacing using GtrcNET-1 on the APAN/TransPAC LA Link (300MB/s)

nodes using software timers. However, the pacing ratio is not always precise in these schemes.

In this paper, we propose the design of two mechanisms for precise software pacing at end computing nodes. The mechanisms are: *gap packets* and *IPG-aware packet scheduling*. Gap packets are inserted between successive packets to control the interval of the transmission time of the packets. An IPG-aware packet scheduler aggregates the transmission packets of multiple streams based on the bottleneck links' bandwidths. The proposed mechanisms are implemented as a Linux kernel module. Therefore it can be loaded or unloaded without kernel re-compilation.

The rest of the paper is organised as follows. Section II presents the design of precise software pacing mechanisms. Section III describes the experimental results using a WAN emulation environment and a performance comparison with and without pacing. In Section IV, we discuss related work on TCP burstiness control and software-based pacing schemes. Section V summarizes the paper.

II. Precise Software Pacing

In order to achieve precise software pacing, two mechanisms are proposed in this section: (1) *Gap packets* which precisely adjust the intervals between packets, and (2) *IPG-aware packet scheduling* which coordinates the transmission of packets for each bottleneck link.

A. Gap Packet: Virtual Inter Packet Gap

The key to realizing precise pacing is controlling the starting time of the transmission of each packet. Some schemes which use a software timer to determine the starting time have been proposed [10][11][13]. However, sometimes these are not accurate enough to determine a starting time. We propose a simple yet accurate mechanism to trigger the transmission of a packet. That is, to insert a dummy packet between the *real* packets. We call this dummy packet a *gap* packet. Figure 3 shows the basic idea of the gap packet. A gap packet produces a gap between sequentially transmitted real packets. Packets are sent out from the Interface queue (IFQ) associated with a network interface card (NIC) one after another by the NIC's hardware. Therefore, by registering a gap packet between real packets in IFQ, transmission of the next packet starts after sending a gap packet by NIC. By changing the size of a gap packet, the starting time of the next real packet transmission can be precisely controlled. For example, to control a half rate transmission, a gap packet is inserted between every real packet where the gap packet size is the same as that of the real packets.

The target rate is estimated by the following equation:

$$target_rate = \frac{cwnd \times packet_size}{RTT} \quad (1)$$

where $cwnd$ is the congestion window size.

The target rate with gap packets inserted is also defined by the following equation:

$$target_rate' = max_rate \times \frac{packet_size}{packet_size + gap_size} \quad (2)$$

where max_rate is the maximum physical bandwidth of a NIC and gap_size is the number of bytes in the gap packet, including the Ethernet packet header and hardware IPG.

gap_size is calculated as follows, using Equation 1 and 2, so that $target_rate = target_rate'$:

$$gap_size = \frac{max_rate \times RTT}{cwnd} - packet_size \quad (3)$$

A gap packet should not produce any side effects, except for delaying the starting time of the next packet transmission,

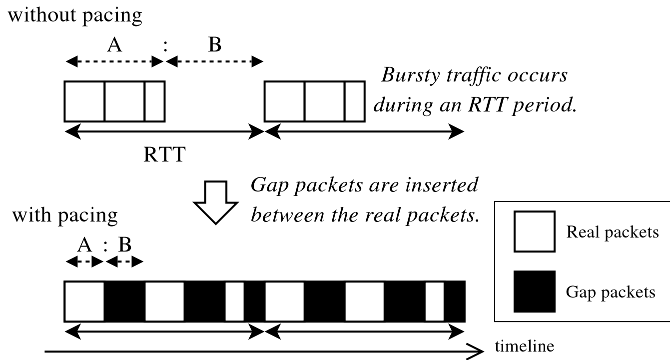


Fig. 3. Gap packet: Virtual Inter Packet Gap

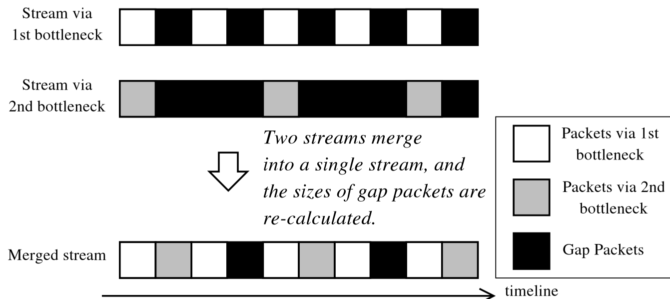


Fig. 4. IPG-aware packet scheduling

and the gap packet should be discarded at the input port of a switch to which the NIC is connected. An unreachable packet can not be used as a gap packet, since a switch floods broadcast packets to all ports when a packet with an unregistered destination is arrived. Therefore, in order to realize a gap packet, the IEEE 802.3x PAUSE packet, with a pause time of zero and the required packet size is employed. IEEE 802.3x flow control from the NIC can not be used in this case. However, since today's PC has enough performance to receive gigabit rate traffic, this function is rarely used in reality.

In a current implementation, the maximum gap packet size is set to 4KB, and so a gap which is larger than 4KB is divided into multiple gap packets. Furthermore, the maximum gap size is set to 64KB.

B. IPG-aware Packet Scheduling

The basic idea behind the IPG-aware packet scheduling mechanism is to schedule packet transmission based on the required IPG that is calculated based on each bottleneck links' bandwidth. If a network has a single bottleneck link, the scheduler only inserts gap packets. However, if the network has multiple bottleneck links, it is necessary to schedule the order of packet transmission and the packet interval. For example, Figure 4 shows two streams whose have the different bottleneck link merge into a single stream, and the sizes of gap packets are re-calculated.

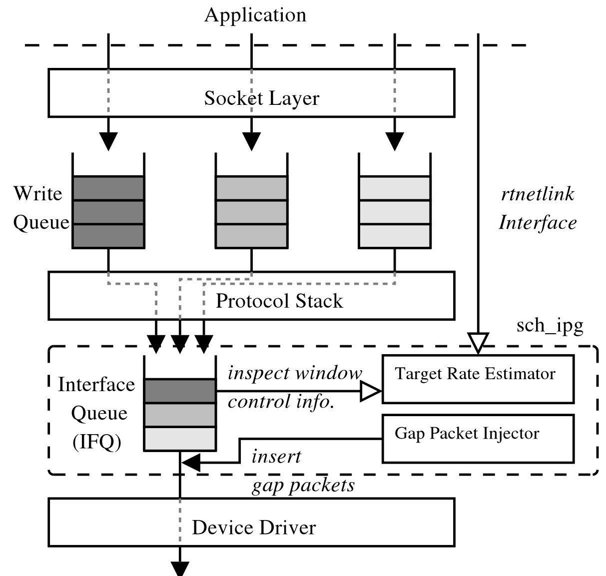


Fig. 5. `sch_ipg` QDisc kernel module

C. Implementation

Figure 5 shows an overview of the implementation in the Linux kernel. Each transmitted packet is queued in the IFQ associated with the output NIC, after processing of the TCP/IP protocol stack. The Linux kernel provides multiple queuing disciplines (QDisc) for the QoS control framework [18]. A QDisc module consists of the IFQ and the queuing algorithm. We implement the IPG-aware packet scheduler on this framework, can call it `sch_ipg`. Because `sch_ipg` is implemented as a loadable kernel module; and it is independent of the NIC, it is portable and easy to deploy.

`sch_ipg` provides two components: (1) a target rate estimator; and (2) a gap packet injector. The target rate estimator calculates the pacing target rate from Equation 3, where $cwnd$ and RTT are retrieved from the data structure of the window control (i.e. struct `tcp_opt`). The gap packet injector inserts gap packets based on the target rate, when a dequeue request is received from the device driver. Here, gap packets are generated using the `skb_clone` function, in order to reduce memory copy operations for the packet payload.

Furthermore, in order to adapt a network topology, `sch_ipg` is implemented as a classfull QDisc[18] which may have sub-QDiscs for bottleneck links. A sub-QDisc has a target rate for the associated bottleneck link. `sch_ipg` makes decisions about how to coordinate among sub-QDiscs based on these rates, and it inserts gap packets, if necessary.

III. Evaluation

We present performance results using the proposed precise software pacing mechanism[4] in a WAN emulation environment.

[4] We evaluate only a gap packet mechanism, because an IPG-aware scheduling mechanism has not been implemented completely.

TABLE I
HOST HARDWARE SPECIFICATIONS

Processor	Pentium4 2.8GHz
Mother Board	Intel D865GLC
Main Memory	1GB (DDR400)
NIC	Intel 82547EI (CSA)

TABLE II
HOST SOFTWARE CONFIGURATION

OS	RedHat Linux 9
Kernel	Linux 2.4.27 + Web100 2.4.0
TCP Protocols	Scalable TCP (included in Web100 2.4.0) FAST TCP version 20040421
Max Socket Buffer	32MB

A. Experimental Setting

The sender and receiver hosts are connected via GtrcNET-1. Here, GtrcNET-1 is used not for pacing, but for WAN emulation. GtrcNET-1 is a fully programmable network testbed, which consists of an FPGA, 4 SRAM blocks and 4 Gigabit Ethernet ports. Based on the configuration of the FPGA, it provides functions such as WAN emulation, traffic monitoring, and traffic shaping at gigabit wire speed. In this experiment, GtrcNET-1 emulates a single bottleneck link whose bandwidth and RTT latency are 62.5MB/s and 200ms, respectively. Further, GtrcNET-1 emulates TailDrop buffer management.

Table I shows the specifications of the host hardware, and Table II shows the configuration of the host operating system. We use desktop class PCs, comprised of an Intel Pentium 4 2.8GHz processor, 1 GB of memory (DDR400), an Intel D865GLC mother board, and an on-board Intel 82547 GbE NIC. All PCs are running Redhat Linux 9.0 and the Linux kernel 2.4.27, with the Web100[16] 2.4.0 patch and Tom Kelly's SACK-tag patch[19] applied. The SACK-tag patch fixes a bug in retransmission processing which retrieve entire the write queue for every ACK. The Web100 exposes the statistics inside the TCP stack itself through an enhanced standard Management Information Base (MIB) for TCP. The TCP congestion control algorithms are Scalable TCP or FAST TCP. This configuration also enables *WAD_IFQ* in order to ignore *send stalls* at the IFQ, where it has the same effect as increasing IFQ size (txqueuelen). The maximum socket buffer size is set to 32MB. From the point of bandwidth distance product, 32MB is large enough for this experimental setting.

B. Preliminary Validation of Effects of Gap Packets

The proposed mechanism achieves precise pacing by adjusting the sizes of gap packets. In order to verify this, we measured the bandwidth while varying the size of the gap packet. Figure 6 shows how well this mechanism can approximate the actual bandwidth to the theoretical bandwidth. This result shows that it can pace packet transmission at the target rate precisely.

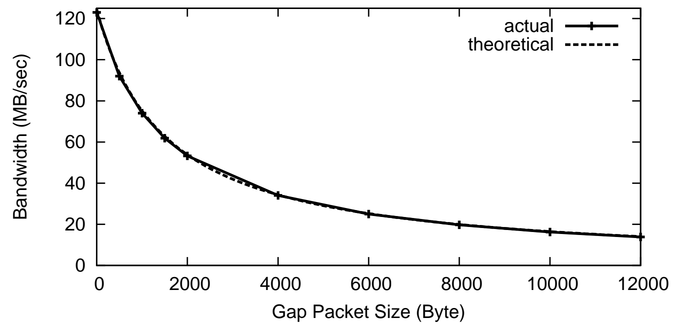


Fig. 6. Bandwidth while varying the size of the gap packet

C. Performance in a WAN emulation Environment

We evaluated whether the proposed mechanism can effectively reduce packet losses on fast long-distance networks. In order to verify this, we measured the bandwidth behavior from both macroscopic and microscopic points of view. The following results were measured by the *iperf*[20] over a period of 5 minutes. Iperf results show TCP throughput. However, `sch_ipg` estimates the target rate at a data link layer. Therefore, the overhead of the TCP/IP protocol header must be considered.

1) Average bandwidth and number of TailDrop packets: First, we measured the average bandwidth and the number of packet losses in a bottleneck link router while varying the TailDrop buffer size, i.e. the size of FIFO before the bottleneck. Assuming that the TailDrop buffer size is large enough, the router is robust with respect to bursty traffic. Tables III and IV show the results using Scalable TCP and FAST TCP. Tables III (a) and IV (a) show the results without pacing. In this case, packets are frequently lost, and thus, the bandwidth is low. Tables III (b) and IV (b), in contrast, show the results with pacing, in which case zero or only a few packets are lost, and thus, the bandwidth is uniform. Using Scalable TCP, packets are still lost during the steady state. It is more likely the target rate estimation misses the mark. Further, the results shown in Table IV (b) is slightly lower than the available bandwidth. Thus, the improvement of the target rate estimation is still an open issue.

2) Microscopic behavior during the slow-start phase: We further measured the behavior of the bandwidth with a finer resolution (500 μsec), where we use the fine resolution traffic monitoring feature of GtrcNET-1. The TailDrop buffer size is set to 128KB. Figure 7 shows the result during the slow-start phase , with and without pacing, using Scalable TCP. The bandwidth is plotted by bar chart, and the $cwnd$ is plotted by line chart. Without pacing, packet losses occur at about two seconds after the start. Here, bursty traffic occurs over the bottleneck link bandwidth, whose peak bandwidth reaches about 123MB/s, however the average bandwidth during an RTT period is only about 3MB/s. By performing fast recovery after a packet loss, $cwnd$ is set to half the size of the $cwnd$ before the packet loss. And then, the sender state enters the congestion avoidance phase. With pacing, on the other hand,

TABLE III

SCALABLE TCP: AVERAGE BANDWIDTH AND NUMBER OF TAILDROP PACKETS

	(a) without pacing		(b) with pacing	
Buffer (KB)	BW (MB/s)	TailDrops	BW (MB/s)	TailDrops
64	17.3	3071	56.8	41
128	19.9	2403	55.8	791
256	23.0	1526	57.1	0
512	29.9	2946	57.0	374

TABLE IV

FAST TCP: AVERAGE BANDWIDTH AND NUMBER OF TAILDROP PACKETS

	(a) without pacing		(b) with pacing	
Buffer (KB)	BW (MB/s)	TailDrops	BW (MB/s)	TailDrops
64	12.9	1699	54.7	0
128	20.4	1328	54.7	0
256	21.0	1231	54.7	0
512	57.1	0	54.7	0

the traffic is paced under the bottleneck link bandwidth, and no congestion exists. Thus, $cwnd$ is ten times larger than that without pacing, and the bandwidth is fifteen times higher. In addition, when the bandwidth is under about 20MB/s, little bursty traffic remains because the maximum gap size is set to 64KB.

Figure 8 shows the result during the slow-start phase, with and without pacing, using FAST TCP. The aggressiveness of increasing $cwnd$ during the slow-start phase is less than that of Scalable TCP, because FAST TCP performs burstiness control at the TCP protocol layer. Without pacing, bursty traffic occurs over the bottleneck link bandwidth, and this results in packet losses. With pacing, on the other hand, the traffic is paced under the bottleneck link bandwidth, and no congestion exists.

3) Microscopic behavior during the congestion avoidance phase: Figure 9 shows the result during the congestion avoidance phase, with and without pacing, using FAST TCP. Without pacing, bursty traffic causes packet losses, and then the sender periodically shrinks the $cwnd$ before the macroscopic average bandwidth reaches the bottleneck link bandwidth. Thus, the inefficient bandwidth utilization result in the poor performance of the TCP/IP communication. With pacing, on the other hand, the physical bandwidth of the bottleneck link is fully utilized, and then the TCP/IP communication achieves high and stable throughput.

When the TailDrop buffer size is large enough, such as 512KB, as shown in Table IV, packet losses do not occur, and then the sender state changes to the steady-state. Figure 10 shows the result during the congestion avoidance phase, where the TailDrop buffer size is set to 512KB. Without pacing, $cwnd$ is uniform, however the microscopic bandwidth is sometimes unsteady. There might be three reasons: the interrupt coalescing of the NIC[14], the delayed ACK, and the ACK compression phenomenon[15]. These cause an increase in the amount of packet transmission, and result in small bursty packet transmissions. Using our proposed software pacing mechanism, the microscopic bandwidth is quite steady. These

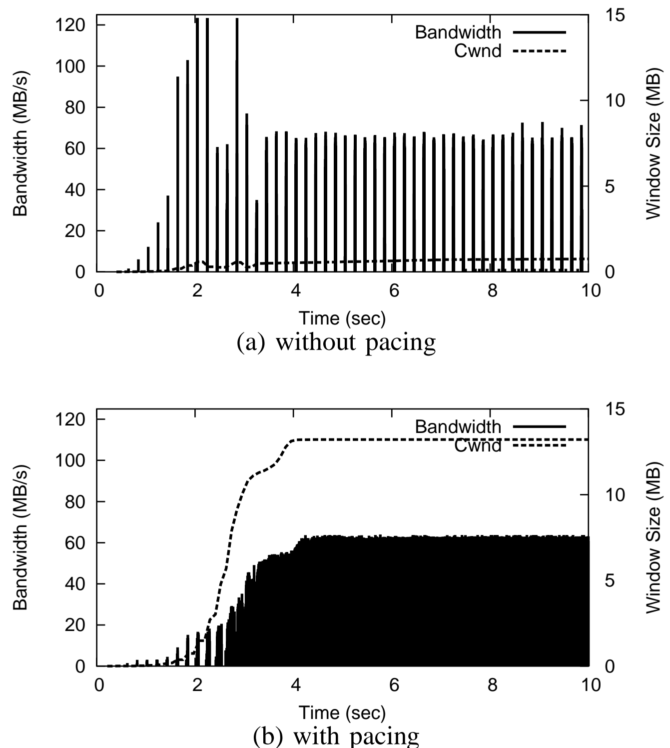


Fig. 7. Scalable TCP: Bandwidth and congestion window size during the slow-start phase (buffer size: 128KB)

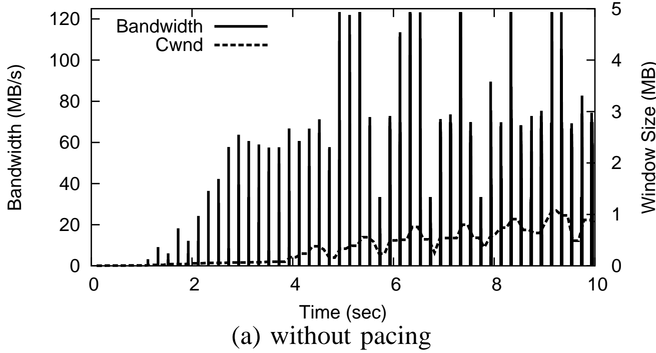
results also suggest that this is an effective mechanism to improve TCP/IP communication performance on fast long-distance networks.

D. Memory Bandwidth Overhead

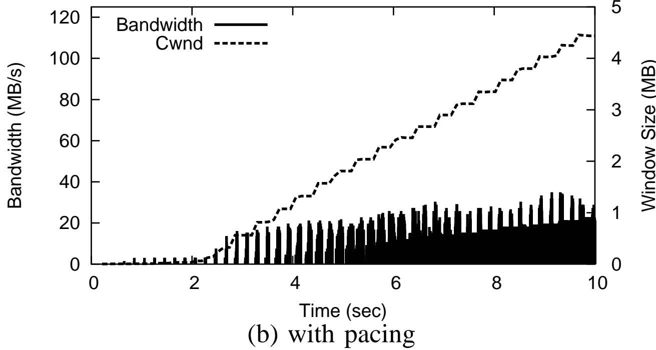
To transmit a packet, the NIC performs Direct Memory Access (DMA) to the main memory to copy the packet body. Such DMA transfers can degrade the performance of applications, since the DMA access may conflict with memory accesses from the CPU.

We evaluated the memory bandwidth overhead when the proposed software pacing mechanism is used, since the NIC performs DMA not only to transmit real packets, but also to transmit gap packets. We measured the memory bandwidth by using a benchmark program called *mbench*, which simply does block copies between two 1MB buffers using the memcpy library function. In addition, GtrcNET-1 emulates a bottleneck link whose bandwidth and RTT latency are 125MB/s and 200ms, respectively. When mbench is executed, iperf generates bulk transfers as follows:

1) *mbench only*
   mbench is executed without iperf.
2) *socket buffer*
   The socket buffer size is limited to a maximum of 15MB so as to limit the Iperf transmission rate to 62.5MB/s. No software pacing is used here.
3) *pacing*
   Software pacing is used to limit the Iperf transmission



(a) without pacing



(b) with pacing

Fig. 8. FAST TCP: Bandwidth and congestion window size during the slow-start phase (buffer size: 128KB)

TABLE V

MEMORY BANDWIDTH OVERHEAD WITH PACING

	Memory Bandwidth (MB/s)	Iperf Throughput (MB/s)
mbench only	1935.5	-
socket buffer	1236.3	58.38
pacing	1043.0	58.38
full rate	923.0	116.13

rate to 62.5MB/s.

4) *full rate*

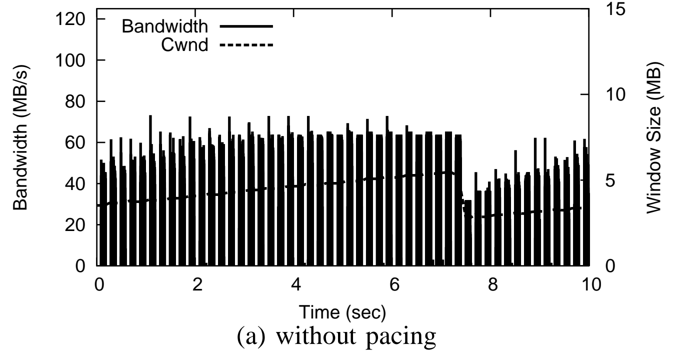
Iperf transmits with no limits (i.e. 125MB/s).

Table V shows the result. Iperf shows almost the same transmission rate for *socket buffer* and *pacing*. However, the NIC performs DMA to generate gap packets in *pacing*. Therefore, the memory bandwidth of *pacing* is smaller than that of *socket buffer*. On the other hand, the NIC performs the same amount of DMA in *pacing* and *full rate*. However, the memory bandwidth is larger for *pacing*. This is because the CPU performs memory copies to send a real packet.

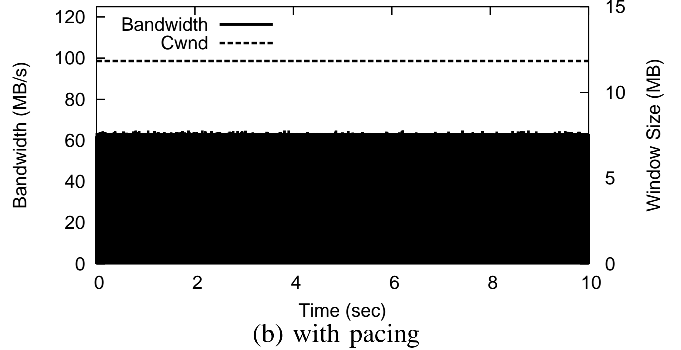
This result shows that the proposed software pacing mechanism decreases available memory bandwidth for application programs, but the decrease is smaller than for the case where full rate transmission is performed.

IV. RELATED WORK

On TCP/IP communication over fast long-distance networks, bursty traffic during the slow-start phase can overrun the buffer of the bottleneck link router with a large number of packets, hence large packet loss occurs. Limited slow-start[5] is a modification of the slow-start behavior for use with TCP connections with large congestion windows. This requires some sort of a priori knowledge of the bottleneck link bandwidth. The proposed mechanism does not require a priori knowledge, because the target rate is estimated from TCP window control information.



(a) without pacing



(b) with pacing

Fig. 9. FAST TCP: Bandwidth during the congestion avoidance phase (buffer size: 128KB)

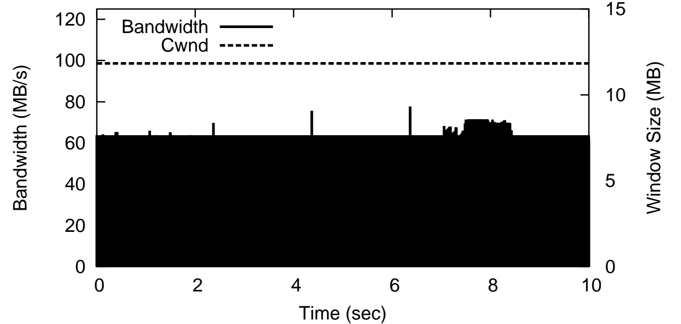


Fig. 10. FAST TCP: Bandwidth during the congestion avoidance phase (without pacing, buffer size: 512KB)

While some NICs have a function to set the IPG size, the size can not be changed dynamically according to the previously sent packet size. In addition, such implementation is dependent on NICs and device drivers. The software-based approach, on the other hand, controls the timing of

transmission of packets using a software timer, and requires no additional hardware. Antony et al[17], however, reported pacing may be hard to implement at the device driver level since it requires that the operating system maintains a timer per TCP flow with μsec resolution, which could incur lots of overhead.

Clusterd Packet Spacing[12][13] controls the transmission interval between packets during the slow-start phase using a software timer. In [13], the pseudo software interrupt function 'tasklet' in Linux is used to implement a high-resolution timer. Using the tasklet, the timing at which to insert a packet into the IFQ can be scheduled with 1 μsec accuracy. This approach divides transmission packets into small portions, but still has small burstiness. In other words, such an approach cannot control the transmission rate at a high resolution, and may cause extra overhead.

V. Conclusion

This paper has shown the design and evaluation of precise software pacing mechanisms: (1) gap packets which adjust precisely the interval of packets; and (2) IPG-aware packet scheduling which coordinates the transmission of packets for each bottleneck link. Experiments in a WAN emulation environment with one bottleneck link show that gap packets can precisely control IPG to the target rate, and TCP/IP communication performance using Scalable TCP, as well as FAST TCP, is improved. As a result of this experiment, we have shown that precise software pacing can be realized without any special hardware, and without Linux kernel recompilation.

Towards practical use of this work, an adaptation to various network topologies is more important. In this paper, we presented experimental results in a simple network setting which has a single bottleneck link. We plan to evaluate and discuss details of an IPG-aware packet scheduling algorithm on a multiple bottleneck link network.

Acknowledgment

A part of this research was supported by a grant from the Ministry of Education, Sports, Culture, Science and Technology (MEXT) of Japan through the NAREGI (National Research Grid Initiative) Project.

References

[1] O. Tatebe, H. Ogawa, Y. Kodama, T. Kudoh, S. Sekiguchi, S. Matsuoka, K. Aida, T. Boku, M. Sato, Y. Morita, Y. Kitatsuji, J. Williams, and J. Hicks, "The Second Trans-Pacific Grid Datafarm Testbed and Experiments for SC2003," IEEE/IPSJ SAINT 2004 Workshops, pp.26-30, January 2004, http://datafarm.apgrid.org/.

[2] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe, and S. Sekiguchi, "GNET-1: Gigabit Ethernet Network Testbed," IEEE Cluster 2004, September 2004, http://gtrc.aist.go.jp/gnet/.

[3] V. Jacobson, "Congestion Avoidance and Control," ACM SIGCOMM '88, pp.314-329, August 1988.

[4] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649, December 2003, http://www.icir.org/floyd/hstcp.html.

[5] S. Floyd, "Limited Slow-Start for TCP with Large Congestion Windows," RFC 3742, March 2004.

[6] T. Kelly, "Scalable TCP: Improving Performance in Highspeed Wide Area Networks," Computer Communication Review, Vol.32, No.2, April 2003, http://www-lce.eng.cam.ac.uk/ ctk21/scalable/.

[7] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control for Fast Long-Distance Networks," IEEE INFOCOM 2004, March 2004, http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/.

[8] C. Jin, D. X. Wei, and S. H. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance," IEEE INFOCOM, March 2004, http://netlab.caltech.edu/FAST/.

[9] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," IEEE INFOCOM, pp.1157-1165, March 2000.

[10] V. Visweswaraiah and J. Heidemann, "Improving Restart of Idle TCP Connections," USC TR 97-661, November 1997.

[11] D.Lacamera, "TCP Spacing Linux Implementation," http://www.danielinux.net/projects/tcp_spacing.php.

[12] M. Nakamura, J. Sembon, Y. Sugawara, T Itoh, M. Inaba, and K. Hiraki, "End-node transmission rate control kind to intermediate routers - towards 10Gbps era," PFLDnet2004, February 2004.

[13] H. Kamezawa, M. Nakamura, J. Tamatsukuri, N. Aoshima, M. Inaba, K. Hiraki, J. Shitami, A. Jinzaki, R. Kurusu, M. Sakamoto, and Y. Ikuta, "Inter-layer coordination for parallel TCP streams on Long Fat pipe Networks," SC2004, November 2004.

[14] A. Shaikh and K. Christensen, "Traffic Characteristics of Bulk Data Transfer using TCP/IP over Gigabit Ethernet," IEEE 2001 International Performance, Computing, and Communications Conference, pp.103-111, April 2001.

[15] J. C. Mogul, "Observing TCP Dynamics in Real Networks," ACM SIGCOMM'92, pp.305-317, April 1992.

[16] M. Mathis, J. Heffner, and R. Reddy, "Web100: Extended TCP Instrumentation for Research, Education and Diagnosis," ACM Computer Communications Review, Vol.33, No.3, pp.69-79, July 2003, http://www.web100.org/.

[17] A. Antony, J. Blom, C. de Laat, J. Lee, and W. Sjouw, "Microscopic Examination of TCP flows over transatlantic Links," iGrid2002 special issue, Future Generation Computer Systems, vol.19 issue 6, 2003.

[18] K. Wehrle et al., "The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel," Pearson Prentice Hall, 2004.

[19] T. Kelly's SACK-tag patch, http://www-lce.eng.cam.ac.uk/ ctk21/code/.

[20] Iperf, http://dast.nlanr.net/Projects/Iperf/.