System Capability Effects on Algorithms for Network Bandwidth Measurement

Guojun Jin Brian L. Tierney
Distributed Systems Department
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720
g_jin@lbl.gov

ABSTRACT

A large number of tools that attempt to estimate network capacity and available bandwidth use algorithms that are based on measuring packet inter-arrival time. However in recent years network bandwidth has become faster than system input/output (I/O) bandwidth. This means that it is getting harder and harder to estimate capacity and available bandwidth using these techniques. This paper examines the current bandwidth measurement and estimation algorithms, and presents an analysis of how these algorithms might work in a high-speed network environment. This paper also discusses the system resource (hardware and software) issues that affect each of these algorithms, especially running on generic platforms built from off-the-shelf components.

Categories & Subject Descriptors:

B.8.2 [Hardware]: PERFORMANCE AND RELIABILITY — Performance Analysis and Design Aids; C.4 [Computer Systems Organization]: Performance of systems — Design studies, Measurement techniques, Performance attributes, Reliability, availability, and serviceability; D.4.8 [Software]: Operating system — Performance Measurement; C.2.3 [Computer Systems Organization]: Network Operations — Network monitoring; F.2.0 [Theory of Computation]: Analysis of algorithms and problem complexity; G.4 [Mathematics of Computing]: Mathematical software — Algorithm design and analysis; C.2.5 [Computer Systems Organization]: Local and Wide-Area Networks — Buses and High-speed.

General Terms:

Measurement, Algorithms, Performance and Design

Keywords:

Network, Bandwidth, Measure, Estimation, Algorithm, System Capability, Design, Performance

IMC?03, October 27-29, 2003, Miami Beach, Florida, USA.

1. INTRODUCTION

Active measurement is a convenient means to estimate available network bandwidth for ordinary users because it does not require router access. Although passive network monitoring methods such as simple network management protocol (SNMP) [1] can provide detailed statistics on network elements (routers and switches) such as physical bandwidth (capacity [6]) and utilization, they unfortunately require special access privileges which are not usually available to ordinary users.

Algorithms for actively measuring network physical and available bandwidths have been researched for many years. Many tools have been developed, and only a few tools have successfully achieved a close estimation of network bandwidths, especially for networks faster than 100 Mbits/second. The main reason these algorithms fail to accurately estimate high-speed network bandwidth is that they do not take the capabilities of the measurement host system into account.

Ideally, the network bandwidth estimation algorithm should not be dependent on end host performance. If end host capabilities are involved, the measurement will be of the system throughput rather than the network bandwidth. Unfortunately, most current available bandwidth estimation algorithms require that the end hosts have throughput higher than the available network bandwidth. The goal of a network measurement tool should be to measure the available bandwidth of the network path, not the available bandwidth of the measurement host itself.

A number of tools and algorithms do successfully estimate network bandwidth on lower speed networks. *Pathchar* [10] is designed to estimate physical bandwidth of hop-by-hop links. *Clink* [4] and *pchar* [23] are different implementations of pathchar. *Pathload* is for estimating available bandwidth [10]. Nettimer [5] uses a passive algorithm to measure the narrow link capacity of a path, but this algorithm requires that no queuing occur on any network element after this bottleneck link, and thus works only on very idealistic paths. *Netest* [16] measures end-to-end achievable throughput or available bandwidth, whichever is feasible, on networks ranging from asymmetric digital subscriber lines (ADSL) to high-speed networks. *Netest* also analyzes the cross traffic, and thus estimates physical bandwidth of the bottleneck link. This paper will analyze the algorithms used by these tools, as well as new tools *pathchirp* [15] and *igi* [18], and address how system capabilities affect their measurements on different types of networks. Other tools, such as *bprobe/cprobe* [2], *ttcp* [19], *iperf* [21], *netperf* [22], *Sprob* [9], and *Treno* [24], do not

measure available bandwidth but rather *achievable throughput* [16]. Therefore we do not discuss them here.

In addition to analyzing existing algorithms, this paper gives a detailed discussion on end host issues and the techniques required to estimate high-speed networks of the future using PC-based hardware. Also, this paper presents an in-depth analysis on the limitations of how the Berkeley packet filter (BPF) can be used via packet capture library (libpcap) for network measurement, especially on high-speed networks.

2. MEASUREMENT METHOD AND HISTORY

Sending probe packets to networks is a common way to actively measure network bandwidth. Figure 1 characterizes various algorithms used to measure networks, and shows the relationship between tools and algorithms.

For each algorithm, there are two different methods used to probe the network: single packet and packet train (multiple packets). There are several techniques for using these two methods, such as varying the packet size, dispersion, spacing, and so on. [5][6] present arguments on dispersion technology and describes how useful they are. [12] presents ideas on multiple packet techniques. Figure 1 illustrates that packet dispersion is used in a variety of network measurement tools.

Ping is the earliest (1980) and simplest network measurement tool based on a single packet probe. It measures the round trip time (RTT) based on the time to forward a single packet plus the time to get an Internet Control Message Protocol (ICMP) reply packet. Ping results can be used to estimate network congestion by analyzing the RTT variation. Developed in 1988, *traceroute* used a similar mechanism to measure RTT on each hop. In the same year, *ttcp* [19] provided a method using a User Datagram Protocol (UDP) stream to obtain a majority of the path bandwidth in order to estimate the path bandwidth in a highly intrusive manner. In 1991, *netest-1* (netest version 1) used a burst methodology, assuming that a UDP burst can gain most of the bandwidth in 10 RTTs (round trip time) if most of the competing network traffic is Transmission Control Protocol (TCP). The maximum burst size was set between 0.5-1 second. Netest-1 repeats the same test in every 5 seconds with a short UDP/TCP burst instead of a continuous UDP stream.

The methods used by both *ttcp* (UDP mode) and *netest-1* are based on packet trains. *Pathchar* was released in 1997, used a *variable packet size* algorithm to measure link physical bandwidth. Since 2001, many bandwidth estimation tools have been released, most of them designed to measure available bandwidth, and most based on packet dispersion. *Nettimer* [5] is for estimating the narrow link (a router or switch that has the lowest capacity along a path) physical bandwidth. *Pathload* [10] estimates the available bandwidth. *NCS* and *netest* rev. 2 are designed to measure bandwidth as well as the achievable throughput and other important network characteristics. Two new tools, *pathchirp* [15] and *igi* [18], are also for available bandwidth measurement.

All of these tools are *active* measurement tools because they send packets into the network in order to make a measurement. Some tools, such as *ping*, are not intrusive. Packet-train-based tools, such as *ttcp*, can be very intrusive, sending a large number of packets into the network and possibly pushing other traffic aside. Packet trains that are too long can also cause router queues to overflow.

3. SYSTEM RESOURCES

Tools for measuring network bandwidth rely not only on accurate mathematical algorithms but also on well designed implementations that consider all possible effects of host system performance. This section describes such issues.

System resources which affect network bandwidth estimation are the resolution of the system timer, the time to perform a system call, the interrupt delay (coalescing), and the system I/O bandwidth (including memory bandwidth). The timing-related system resources — timer resolution, system call, and

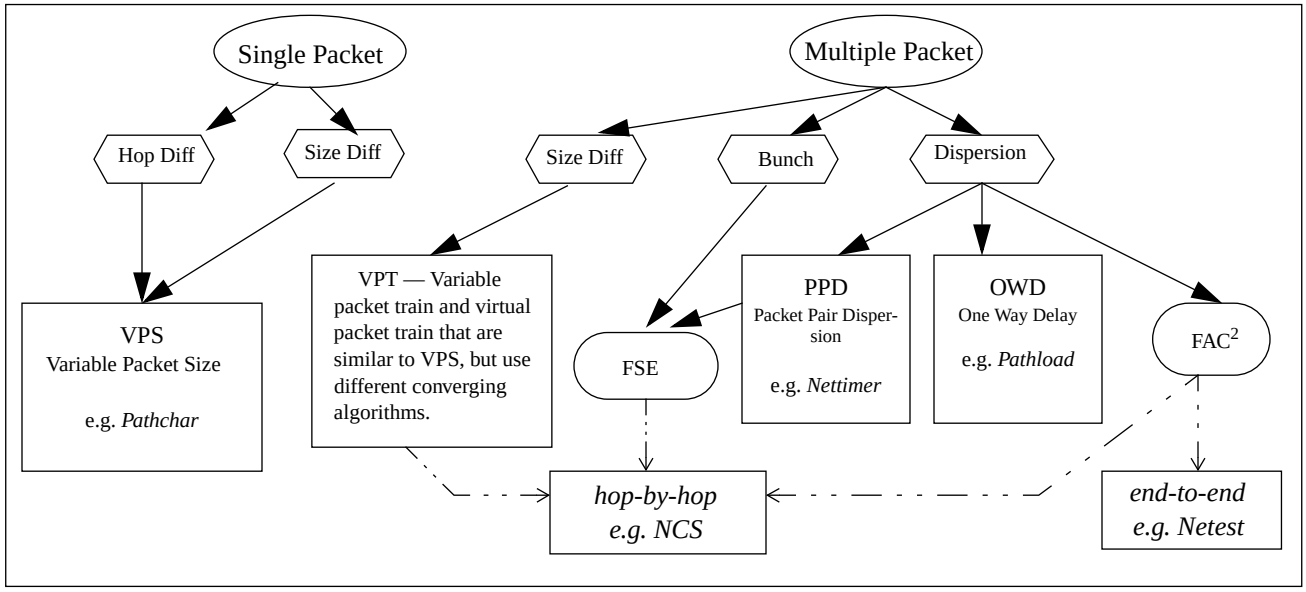

Figure 1. Using packet to probe network; also relations between algorithms and tools

interrupt delay — all affect packet-dispersion-based algorithms much more than they affect packet-train-based algorithms. System I/O bandwidth affects all algorithms equally.

3.1 Interrupt issues

The I/O interrupt interval significantly impacts high-speed NIC (network interface card) performance. For example, if every packet arriving on an 1 Gb/s NIC generates an I/O interrupt, then the system will get interrupted every 12 µs. Most host systems are not able to keep up with this interrupt rate. A common technique to reduce CPU requirements and to increase throughput is called *interrupt moderation.* Many high-speed NICs, including the SysKonnect card, provide interrupt moderation (also known as *interrupt coalescence* or *delayed interrupt*), which bundles several packets into a single interrupt. The idea is that the NIC, on receipt of a packet, does not automatically generate an interrupt requesting the CPU to process the data and releasing buffers for the NIC to get more packets. Instead, the interrupt is delayed for a given amount of time (the *interrupt moderation period*) in hopes of other packets arriving during that time and being serviced by the same interrupt.

Table 1. CPU utilization affected by I/O interrupt

interrupt delay time (coalescing)	% CPU IDLE	% CPU Interrupt	Throughput Mb/s
64 µs interrupt delay for Intel copper GigE (PCI/33 GC-SL) + Intel P4 Xeon 3 GHz CPU	0	92	277
300 µs interrupt delay for above configuration	1	72	515

Table 1 shows how interrupt coalescing affects CPU utilization, thus increasing the network throughput. TCP/IP packets are 1500 bytes for all measurements. The CPU usage was measured by averaging results from UNIX command "*top*" with 1-second refresh rate when running *iperf* with durations of 10 and 30 seconds. The "*vmstat 1*" command was also used to verify the *top* result. In Table 1 we see that the receiving host needed 92% of the CPU to handle I/O interrupts with default interrupt delay settings for the NIC. After the interrupt delay was increased from 64 to 300 µs, the CPU usage dropped to 72% due to the generation of fewer interrupts. This means that the CPU has more time to process packets, so throughput increased 85.9%. Figure 2 shows that tuning the interrupt delay time is not trivial. Tests were done setting the delay from 200 to 550 µs on an Intel P3 Xeon 933 MHz system (64-bit/66MHz PCI). The interrupt delay below 470 µs has no significant impact on CPU usage. Delay values between 470 µs and 475 µs make throughput unstable. This is probably the boundary where the CPU usage is sensitive to the I/O interrupts. 480 µs is the lowest value for getting good throughput on this system for receiving a high-speed TCP stream.

Ideally, the interrupt moderation period is short enough to keep the NIC from running out of buffers and to avoid large delays in packet processing. The maximum interrupt interval (I_time) can be computed as:

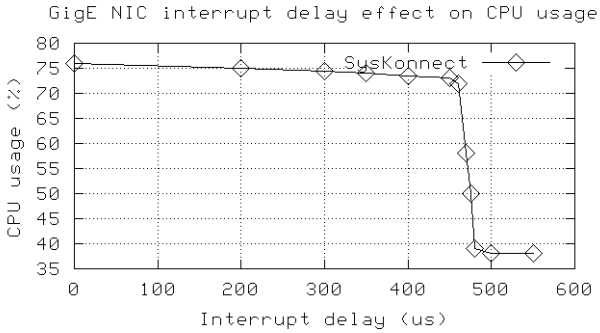

Figure 2. Tuning interrupt delay time

I_time = N * average_packet_size / line_speed

where N is the number of receiver buffer descriptors statically compiled in the network interface device driver. The main drawback of interrupt coalescing is that the kernel is no longer able to assign accurate timestamps to the arriving packets. The problem is that packets are processed a significant amount of time after they arrive at the host. Fortunately, some network cards (for example, SysKonnect) have an onboard timestamp register which can provide information on the exact packet arrival time, and pass this timestamp to the system buffer descriptor. We have modified the FreeBSD SysKonnect driver to allow us to use the NIC timestamp instead of the system clock timestamp [14].

Table 2. Time of Syscall

timestamp is via gettimeofday API and kernel TSC[*] (microtime)		Linux 2.4.1x		FreeBSD 4.8-RELEASE	
		timestamp ns	read/write ns	timestamp ns	read/write ns
Intel Xeon P4	2.4 GHz	900	1400	4409	1206
	2.0 GHz	980	1100	4590 (3567)	130
AMD MP	1730.7 MHz			4195 (4033)	217
AMD XP	1400 MHz	282	506		
Intel P3	746.17 MHz	943	2100	4700	289
	531.83 MHz	970	2050	1800 4.3-R	380 4.3-R

3.2 System call issues

Table 2 lists the time to perform a system call on two operating systems running on various CPUs. The time to perform a system call affects both the outgoing packet pacing and the time to get timestamps for incoming packets. In next section, these numbers are both used to analyze how packet-dispersion-based algorithms are affected by system call time. The syscall time is measured for two areas — getting a timestamp and

[*]: TSC is referred as CTC — the real time counter (RTC), but not the CPU clock counter (CCC).

doing I/O. Each system call is measured in 1 loop (single call), 1000 loops (a value between 1 and the longest loop), and 1800 loops (large enough without hitting the context switch time — 10 ms), and the result is a lower value with the higher frequency. *Gettimeofday* is the UNIX syscall for getting the timestamp. The I/O syscall time is the average of *read* and *write* system calls. The read/write call test is done by reading/writing zero bytes to the stdin/stderr in non-block mode (O_NONBLOCK).

It seems that the cost of getting system time is almost a constant, and the cost is very high compared with the CPU clock rate, especially on the FreeBSD systems. What is the reason for this? Since the CTC (clock time counter — industry standard 8254 counter/timer, also known as *timestamp counter* or TSC) chip is very simple and there is no specific bus lock required to access this chip, the issue is the generic clock access method. To make the operating system code work on different motherboards, the CTC is accessed via the I/O bus, either ISA or PCI, but not directly from the main bus. The highest frequency of these I/O buses for accessing CTC is 33 MHz, and reading the counter register requires 9–16 I/O bus cycles, depending on how the code is implemented. In the best case, reading the CTC register requires about 272.7 ns (9 I/O cycles on 33 MHz PCI bus); while the worst case requires 959 ns (16 I/O cycles on 16.667 MHz ISA bus). [11] confirms this calculation which matches the results in Table 2 for Linux, which uses the same CTC (TSC) read code derived from FreeBSD. However, FreeBSD has two timestamp modes — safe and fast. The fast mode is the same as the Linux CTC read mode — one read per request. The safe mode reads the counter 3–4 times in order to confirm and calibrate the clock accuracy. The results listed in Table 2 for FreeBSD are from a backward compatible implementation that reads two 8-bit time registers (the newer CTC has 16-bit time registers) to form a 16-bit counter. This explains why the cost of reading CTC is so high on FreeBSD systems. This fact indicates that the cost of getting a timestamp is in reading CTC (TSC) rather than in the system call API *gettimeofday*. Therefore, packets timestamped in user space via *gettimeofday* have similar accuracy as timestamped inside the kernel, for example via BPF (Berkeley packet filter). Table 3 shows syscall cost on more operating systems.

Table 3. Syscall time for more O.S.

O.S.	Hardware	gettimeofday	read/write
Solaris 2.8	333MHz Sparc	348 ns	8400 ns
Solaris 2.7	400MHz Sparc	278-295 ns	5300 ns
AIX 4.3.3	RS 6000	> 3000 ns	8500 ns
IRIX 6.2	175 MHz IP28	7946 ns	28162 ns
BSD/OS 4	526 MHz P2	10877 ns	11357 ns
Mac OS X 10.2	1 GHz G4	1937 ns	2043 ns

The system call time also affects our ability to increase the system timer resolution. The current UNIX system timer resolution available to a user (via *gettimeofday*) is 1 µs. The system internal timer resolution is often at 1 nanosecond in modern UNIX systems. However, the time to perform a system call limits the user timer resolution to 1.9 µs on most systems with x86-based CPUs running Linux, and limits it to 9 µs for FreeBSD systems because two *gettimeofday* calls are needed to get the relative time. *Therefore, the design of bandwidth measurement algorithms must take this issue into account.* Note that an onboard NIC timer, as mentioned at the end of the § 3.1, may not improve the timer resolution for this situation because the CTC still needs to be read to obtain the current system time. The onboard NIC timer also needs to be accessed to obtain the relative clock to compute the arrival time for each packet.

4. TIMING EFFECT ON EXISTING ALGORITHMS FOR BANDWIDTH MEASUREMENT

This section describes the algorithms currently used to measure network bandwidths, and analyzes the system resources requirements for each algorithm.

4.1 Single packet oriented algorithms:

Pathchar uses *variable packet size* (VPS) algorithm, including *size differential* (SD) and *hop differential* (HD)[8] methods, to estimate link physical bandwidth. The SD algorithm measures the time difference, ΔT, for a constant ΔS — the size difference for packet size increment, by sending UDP packets from the source host to each network element and measuring the time to get an ICMP response (Figure 3). Thus the transfer rate can be denoted as:

$$R_{Tx} = \frac{\Delta S}{\Delta T}$$

$\Delta S = S_2 - S_1$, S_1 and S_2 are sizes for two different packets
$\Delta T = T_2 - T_1$, T_1 and T_2 are the time to send packets S_1 and S_2 to a router respectively.

This algorithm has a limitation that the maximum ΔT, which depends on the maximum packet-size difference, is limited by the MTU (maximum transfer unit) of the network interface. If the network interface is Ethernet, the maximum size difference ΔS is 1472 bytes. When a link bandwidth (BW) is OC-3 (155 Mbits/sec) or higher, the ΔT will be smaller than

$$1472 \times 8 \div 155 \bullet 10^6 = 75.974\ \mu s$$

A typical multi-hop round-trip time (RTT) is greater than 1 ms, and typical system context switch time is 10 ms. For short distances (RTT < 6 ms), the router queuing has a higher effect on the RTT, and the average queuing delay we have seen by *ping* is around 0.05~0.3 ms for RTT less than 6 ms. In long distance cases, at least every other measurement will be interrupted by one context switch if there is any other process running, and the context switch has more impact on the RTT measurement. Under such circumstances, this RTT fluctuation causes a ±5% error rate (where the context switch may introduce an even higher error rate) in time measurement, so the deviation of RTT (Δ_{RTT}), is greater than 50 µs. Under these circumstances, the time difference becomes

$$\begin{aligned}\Delta T &= T_l - T_s \\ &= (S_l \div BW + RTT_l) - (S_s \div BW + RTT_s) \\ &= \Delta T_{(zero\ traffic)} \pm \Delta_{RTT}\end{aligned} \quad (1)$$

where

S_l and S_s are the sizes of the largest and smallest packets

T_l and T_s are the time to transfer each of these two packets

$RTT = T_{sys} + T_{ps} + T_q + T_{ack}$

T_{sys} is the system call time

T_{ps} is the time to send (copy) a packet from user space to the edge of a network interface card (NIC) or the reverse.

Tq is the queuing time for both directions

Tack is the time for acknowledgment to travel back

Therefore, transmission time is not directly proportional to packet size in the real network.

The time difference between the largest packet and the smallest packet that can be transmitted from a source host to an intermediate router is inaccurate when Δ_{RTT} has a magnitude similar to $\Delta T_{(zero\ traffic)}$, and thus dominates ΔT. So, this algorithm is only good for probing networks with capacity up to OC-3 (155 Mb/s) when the MTU is 1500 bytes (see results on p. 17-18 [26]). In a network where jumbo frames (9 KB) are used, this algorithm may measure capacity up to 1 Gb/s. The main merit of this algorithm is that the source host does not need high-transfer-rate hardware to measure bandwidth on high-speed networks.

Since cross traffic can cause T1 and T2 to vary greatly, a single probe will not get an accurate estimate of available bandwidth. In order to obtain a more accurate result, this algorithm sends a number of different size packets to measure the bit rate for each packet, and then uses linear regression to converge on a result.

Figure 3 shows the VPS timeline for transferring two packets. It shows that R_{Tx} on the first hop represents the link capacity, and R_{Tx} on the remaining hops does not because of store and forward delay. To acquire the time difference between router N and router N+1, hop differential (HD) is needed.

In Figure 3, the times have been shifted so that start time of both packets 1 and 2 are aligned at time 0 on the graph. At hop 1 (source host to router A), these packets leave router A at different times due to the store and forward delay. This means that $\Delta T_B = T_{2B} - T_{1B}$ does not represent the time difference of transferring these two packets from A to B. Figure 3 shows that the store and forward delay between these two packets at

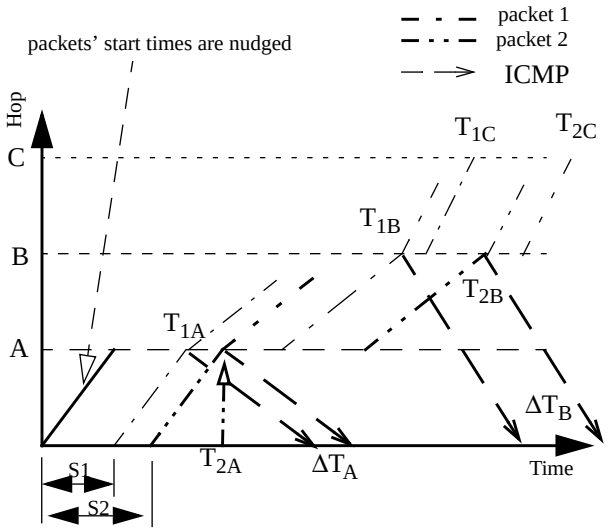

Figure 3. VPS transfer timing of two packets on a

router A is $\Delta T_A = T_{2A} - T_{1A}$. So, the real time difference between transferring these two packets from A to B is:

$$\Delta T_{AB} = \Delta T_B - \Delta T_A = T_{2B} - T_{1B} - (T_{2A} - T_{1A})$$

and the bandwidth of this link is:

$$BW = \Delta S \div \Delta T_{AB}$$

This is the *hop differential* algorithm. This algorithm has two network element (router and switch) related issues. First, different routers may have different ICMP response times. This discrepancy creates difficulties for algorithms based on the hop differential calculation and is the reason why *pathchar* sometimes gives negative results. Second, if any network element that has no ICMP response (e.g.: a layer-2 switch), called a *hidden device*, is immediately before the measured router, the hop differential algorithm will result in a lower bandwidth, which can be computed by a serialization formula:

$$BW = \frac{BW_A \times BW_B}{BW_A + BW_B}$$

BW_A and BW_B are physical bandwidths of router A and B. This is how the HD algorithm can be used to detect hidden devices.

4.2 Packet dispersion based algorithms:

Packet dispersion happens between any packet pair — both a single packet pair and packet pairs within a packet train. This section describes three algorithms based on the packet dispersion: *single spacing, constant spacing, and variable spacing*.

Single spacing (packet pair) —

Packet pair dispersion (PPD) is used in *nettimer* to analyze the bottleneck link capacity. *Nettimer* uses a passive measurement method to look at incoming packet pairs from a given source host. This algorithm is demonstrated by the dotted line box in the lower left corner of Figure 4. The PPD algorithm says that if a pair of packets travels back-to-back through a bottleneck

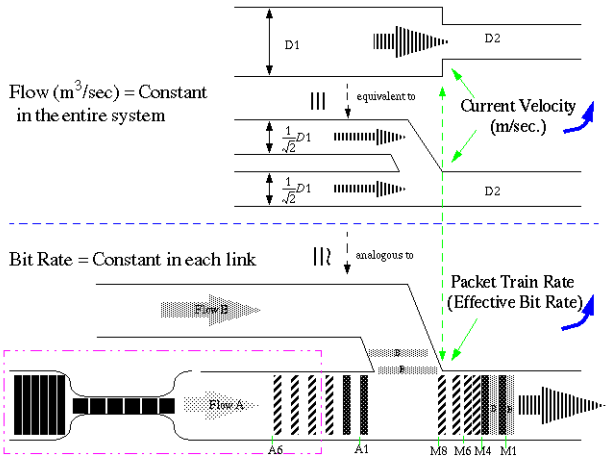

Figure 4. Packet pair and Fluid Spray Effect (FSE)

link, the last bits of the two packets are further separated. After they leave the bottleneck link, this separation will remain until they arrive at the destination. So, the PPD represents the narrow (bottleneck) link's capacity. This is true if and only if no cross traffic occurs at the later routers. The Internet almost always has cross traffic, which causes the fluid spray effect [13] (FSE — Figure 4 and § II.C) when many traffic streams come in from different interfaces and are routed out at another interface with all the packets bunched together, so that the PPD theory does not apply.

This algorithm requires only one resource — accurate system timer resolution, since it is a passive monitoring method. To accurately measure packet pair dispersion, the incoming PPD must be greater than the time for executing at least four system calls, two for getting the arrival time of each packet and two for reading each packet. A mandatory restriction is that the network device cannot have interrupt coalescing enabled, otherwise the packet arrive timestamps will be incorrect. For example, all incoming packets may have the same timestamp.

Constant spacing (self-loading periodic streams)

Pathload uses packet trains with evenly spaced packets which detect *one-way delays* (OWD)[6] to measure available bandwidth. Theoretically, this algorithm may accurately measures available bandwidth. The actual measurement result will vary, especially when measuring a high-speed network, due to the hardware capability and implementation. As discussed in § 5, to use packet trains to measure bandwidth, both sending and receiving hosts must have higher I/O bandwidth than the available network bandwidth. To cause OWD, the probe stream must have a higher transfer rate (R_{snd}) than the available bandwidth (A_{bw}). The difference between R_{snd} and A_{bw} depends on the A_{bw} and NIC speed — the higher the speed, the larger the difference of between the R_{snd} and A_{bw}. For example, if the A_{bw} is 900 Mb/s, the OWD requires 920 Mb/s R_{snd}; but for 9000 Mb/s A_{bw}, the OWD requires 9200 Mb/s rather than 9020 Mb/s R_{snd}. Intuitively, this difference is directly proportional to the resolution of the system timer. The higher the resolution of the system timer, the smaller the difference required to determine the OWD; thus the result is more accurate.

The minimum time needed to distinguish the delay can be either a fixed amount of time or some percentage of the time needed to finish the burst transfer. If the receiver is current PC hardware with 1 µs time resolution and a timestamp timer on the NIC, a few microseconds (for getting the system time) or the time for two system calls (read data and get system time), whichever is greater, can be the lower limit for the time difference. The basic requirement for this algorithm is that the source host needs to have a higher transfer rate than the available bandwidth.

Pathload uses pair-wise comparison test (PCT) and pair-wise difference test (PDT) metrics and statistics to detect the OWD trend. This algorithm builds a region, called the *gray area*, that can be adjusted to estimate the A_{bw}. Metric results above the gray area represent a strong OWD trend, and below the gray area represent no OWD trend. Thus, the gray area is the range of estimated available bandwidth. Pathload can estimate the path's available bandwidth in this manner without requiring a high-resolution timer.

igi [18] — initial gap increasing — uses the similar algorithm with modified method — PTR (packet transmission rate) in order to make measurement more efficiently. However, it omitted some implementation issues such as timer resolution and interrupt coalescing, therefore, the results are not accurate on hosts with GigE NICs (see *Comparison results* at the end of this section-*B*).

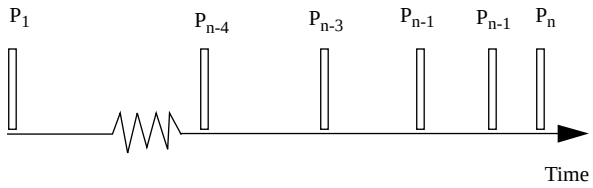

Figure 5. Transmitting timeline of variable spacing packet

Variable spacing (gaps increase crossing the packet train)

The transmitting timeline of the variable spacing packet train is shown in Figure 5. This algorithm is used in *pathchirp* for available bandwidth measurement. It assumes that once the transmit rate of any packet pair, P_x and P_{x+1}, within the packet train is the same as the available bandwidth, the remaining packet pairs after packet P_x, which have a higher transmission rate, will be further separated by cross traffic. That is, the dispersion of the rest of the packet pairs at the receiver will be larger than the spacing at the sender side. If the packet dispersion increases consistently, then the sending rate at packet pair P_x and P_{x+1} is the available bandwidth.

This algorithm is efficient for estimating end-to-end available bandwidth because it can theoretically measure the available bandwidth in one round trip time. Requirements of system resources for this algorithm, however, are very high. Basic requirements for this algorithm include the maximum host I/O bandwidth and accurate timing system. The measurement systems (both sending host and receiving host) need to be idle, so the sender can pace out a packet train with precise timing and the receiver can time the incoming packet accurately. The timing requirement is similar to that of *nettimer*. Due to the time required by the system call described in Table 2, and the I/O interrupt delay (coalescing) caused by high-speed network devices (1 Gb/s or higher), the variable spaced packet train is also difficult to generate on systems equipped with a high-speed network interface, especially under the FreeBSD system, which can only accurately pace out packets up to 470 Mb/s on typical mid-range hardware due to the large system call overhead.

Accurately detecting the gap increase is difficult when the network speed is higher than 300 Mb/s and/or packet-pair rate is higher than 50% of the network speed. A couple of key issues make this measurement difficult. First, the number of packets that can be sent is limited by the timer resolution and the range of packet speed. For example, measuring GigE network where utilization is 50%. The period of a 500 Mb/s packet pair is 24 µs, and the period of a 1 Gb/s packet pair is 12 µs. Because the system timer resolution is 1 µs, the maximum number of packets that can be sent between the 500 Mb/s packet pair and the 1 Gb/s packet pair is 11packets. Due to other system overhead, the number of accurately spaced packet pairs is about a half of that, or 5 packets. Under this circumstance, cross traffic can either compact or spread out this small packet fleet, causing undeterminable results. Second, when the packet train tail is short (e.g.: 5 packets),

even though the cross traffic is less than the path capacity minus this probe traffic, it will most likely increase packet-pair gaps in these probe packets when both traffics encounter each other, causing under estimation of the bandwidth. See case 1 and case 2 illustrated in Figure 6. Furthermore, by default, interrupt coalescing on Gigabit or high-speed NICs bunches 5 to 12 packets together for one interrupt service. Without a user accessible onboard timer, the system will think all the packets arrived at the same time.

Due to these factors and discussions from previous sections, the variable spacing packet train algorithm may not be able to measure bandwidth above OC-12 (622 Mb/s). A larger MTU (also called Jumbo Frame) can improve the measurement condition for current network bandwidth. However, the Large MTU has not yet been standardized. Assume that a 32KB MTU (the largest IP frame) might be accepted by network standard in a few years, it could make variable spacing algorithm 20 times better. However, if this MTU size then lasts more than a dozen years, the network bandwidth may increase more than 20,000 times. So, we cannot simply wait for MTU increasing to help algorithms functioning properly.

Comparison results:

Current tests show that *igi* consistently underestimates available bandwidth on 1 Gb/s paths. For example, *igi* gave results of 219 Mb/s on a one-hop 1 Gb/s path that was only 7% utilized, and 336 Mb/s on an 8-hop 1 Gb/s path what was 27% utilized (utilization was measured by *netest*). *Pathchirp* (1.3.3 release) does not produce any result if the measurement period is less than 30 seconds or the parameter is greater than 250 m using the "-u" option on the one-hop path. When used without any option, pathchirp measurement results were between 70.58–94.3 Mb/s on the same path. One-hop tests were done between two Linux 2.4.20 testbed hosts: (1) a dual Intel Xeon 2.2 GHz CPU with Syskonnect 9843 SX GigE fiber NIC and the maximum network system throughput (MNST) of 1 Gb/s on a 64-bit/66 MHz PCI; and (2) an AMD 1.4 GHz CPU with Netgear GA620T GigE copper NIC and MNST of 710 Mb/s on a 32-bit/33 MHz PCI. The maximum throughput on this one-hop path is about 690 Mb/s. The 8-hop tests were done between the dual Xeon host and a dual AMD MP host (1.4 GHz CPUs) that also runs 2.4.20 Linux with Syskonnect 9843 SX GigE fiber NIC (MNST of 725 Mb/s on 32-bit/33 MHz PCI). After upgraded Linux to 2.4.21 on the one hop testbed, the peak throughput is increased to 732 Mb/s, and *igi* gives a better result of 339 Mb/s A_{bw}, where *pathload* reports 676 Mb/s A_{bw}, and *netest* report the maximum throughput is 705 Mb/s. *Igi* seems to be more system dependent.

4.3 Packet train based algorithms

Packet-train-based tools do not measure the packet pair dispersion inside the packet trains. Packet train algorithms attempt to determine the amount of cross traffic rather than the amount of packet dispersion caused by the cross traffic [Figure 6]. Therefore, packet-train algorithms are less sensitive to the resolution of the system timer and less affected by I/O interrupt delays (coalescing). Also, packet train algorithms do not rely on any single packet pair; therefore increasing the packet length can help to overcome timer resolution problem when measuring very high-speed networks.

A well-known issue with using packet trains is how to measure the capacity of links beyond the narrow link. [13][8] proposed a solution to this problem based on the *fluid spray effect* (FSE), summarized here.

FSE theorem: Assume that two packet trains, both traveling at speeds lower than the network capacity, encounter each other at a router. If the aggregate rate equals or exceeds the router's capacity, all packets are bunched together to form a new stream. When this stream leaves the router, its train rate is at the outgoing router interface (line) speed. This is shown in the lower right in Figure 4, which also shows that if an incoming train is long enough, a pair of packets or a "*subtrain*" within this train will travel at the line speed when it leaves the router.

FSE happens almost everywhere on the Internet. The packet bunching effect is different at each router because each router has different bandwidth and cross traffic. This packet bunching extent can be fed back to the source host via an ICMP message, as the ICMP message will carry packet dispersion information on each router back to the source host, and the source host can use this information to compute each router's physical bandwidth. This method allows packet-train-based methods to measure hop-by-hop link capacity beyond a narrow link.

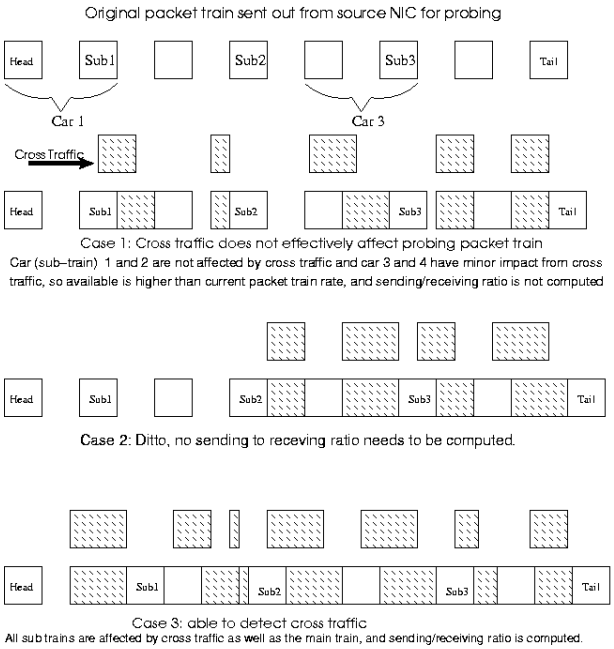

Figure 6. Cross traffic effect to the packet train rate

Using packet trains for end-to-end network bandwidth measurement requires fewer system resources. *Netest* uses feedback adaptive control and feedback asymptotic convergence (FAC^2) algorithm [13] to measure end-to-end available bandwidth and analyze cross traffic, then computes the bottleneck capacity of the path. Figure 6 illustrates the FAC^2 principle of how cross traffic affects the packet train rate from source to destination. The *netest* client on the sending host sends out a constant spaced packet train at the rate recommended (feedback adaptive control) by the *netest* server running on the receiving host. The *netest* server measures the incoming rate for each *car* [8] (measurement unit in number of

MTUs, see Figure 6) and the rate for the entire packet train. When the adjusted sending rate to receiving rate ratio is close to 1, the receiving rate is the available bandwidth. The theoretical convergence time is 5 RTTs [13].

Because algorithms based on packet trains measure the arrival time of each car instead of each individual packet, the time resolution issue is simplified. Packet-train-based algorithms can adjust the car size to fit the system time resolution, while packet-dispersion-based algorithms have to rely on how accurately the system can measure the biggest single packet (one MTU). Therefore, packet-train-based algorithms work better in high-speed network environments, especially for end-to-end bandwidth measurement. [13] mathematically proves that a packet-train-based algorithm, FAC^2, can measure available bandwidth accurately. Using an emulation network testbed [20] we have verified that the accuracy of FAC^2 is close to 99% when path utilization is below 70%. In high-utilization cases, FAC requires more probes to converge to an accurate result, as shown in Figure 7.

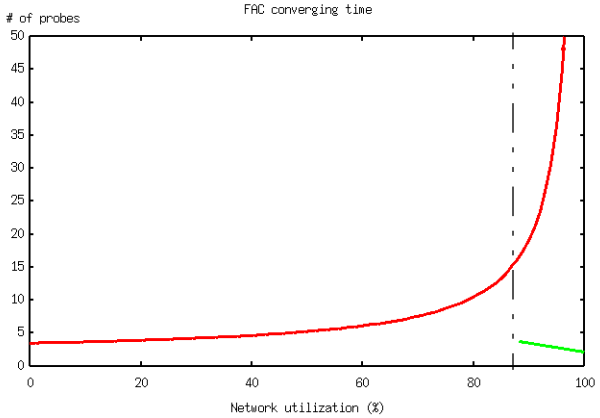

Figure 7. Feedback Asymptotic Convergence

5. FUTURE BANDWIDTH MEASUREMENT

In this section we discuss the system hardware issues required to estimate new high-speed networks such as OC-48 (2.4 Gb/s) and 10 Gb/s Ethernet.

Single packet, packet dispersion, and packet train are different techniques to probe a network for measuring bandwidth. In order to measure high-speed networks, the single packet method requires a high resolution timer due to packet size constraints. A similar issue applies to the packet dispersion algorithms. For example, a 1514-byte packet transmitted through a 10 Gb/s NIC takes about 1.21 µs, and this packet traveling through a 1 Tb/s NIC takes only 12.1 ns. Current UNIX timer resolution is 1 µs, which makes it impossible to measure any incoming packet over 3 Gb/s due to the additional overhead of system calls. When the receiving interrupt is coalesced, the packet dispersion is impossible to measure.

The packet train technique has no size restriction for its car [Figure 6], therefore, the time resolution is not crucial. However, it still requires that the source host must have a higher sending rate than the available bandwidth, and have the ability to control the burst size and sending rate. The high sending rate may sound trivial, since modern CPUs and NICs are fast. In fact, it is more complicated.

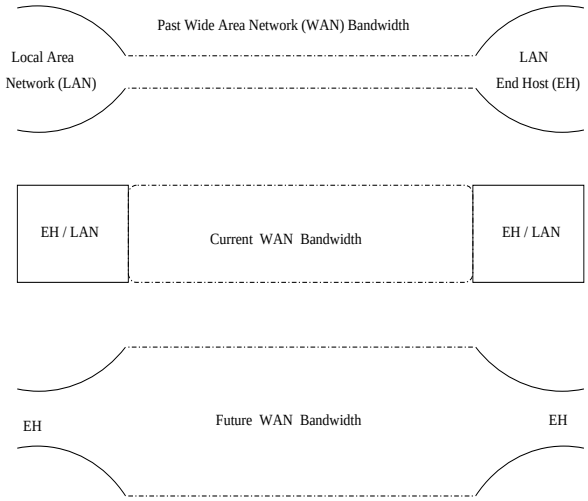

Figure 8. Network and system bandwidth change scale

Currently, the end host I/O bandwidth is similar to the network bandwidth. The end host is the main factor limiting network application throughput in the future. A host's memory, I/O bus, NIC, and operating system affect the throughput. Thus, a method to determine if the end hosts are capable of measuring the available bandwidth is a required part of bandwidth estimation algorithm design.

In the past 10 years, network speed has increased by a factor of 1000; CPU clock speed has increased by more than a factor of 30; memory clock speed has increased by almost a factor of 20. Memory bandwidth, however, has increased by only a factor of 10, and PCI I/O bus bandwidth has increased by only a factor of 8. If these growth rates continue for the next decade, the end host will certainly be the throughput bottleneck for network applications. The growth of network to system bandwidth scale is shown in Figure 8.

The main bottleneck in current systems is at the memory and I/O subsystem. Figure 9 shows the data path for sending data from user memory to the NIC. For a system equipped with a 64-bit/66MHz PCI bus, if the memory bus is 266 MHz, the total time needed to transfer data from a user buffer to the NIC is 6 memory cycles: the 2 fixed cycles plus 4 memory cycles per bus cycle (266/66). However, if the memory bus is 533 MHz, then 10 cycles are required (2 + 533/66). The

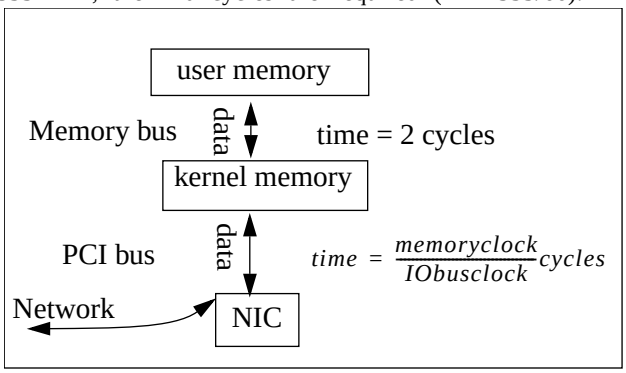

Figure 9. Hardware data path for packets

generic formula for calculating the I/O throughput from memory and I/O bus frequency is:

$$IOthroughput = \frac{MemoryBandwidth}{(PCI + Memory \times 2)cycles}$$

$$= \frac{MemoryBandwidth}{\frac{MemoryClock}{IOBusClock} + 2} \qquad (2)$$

Let us apply this formula to a real case. An ASUS K7V motherboard is equipped with VIA 868 PCI controller that has a 133 MHz memory bus, and it produces 144 MB/s memory copy bandwidth (288 MB/s memory bandwidth). The newer generation VIA PCI controller, VT400, has a 400 MHz memory bus and produces 326 MB/s memory copy bandwidth for the ASUS A7V8X motherboard. Both motherboards have a 32-bit/33 MHz PCI bus. According to equation (2), a VIA 868 system can have a maximum 384 Mb/s (48 MB/s) network throughput, while a VT400 motherboard can have only 369 Mb/s network throughput. In fact, due to DMA overhead (see the next paragraph), the VT400 motherboard only has 300 Mb/s network throughput, while the same NICs can produce 2 or 3 times higher throughput on other motherboards. Therefore, we can see that simply increasing the memory clock speed does not necessarily result in an equivalent increase in the data transfer rate from user space to the NIC. [17] presents some specific tuning ideas on how to speed up the 10-Gigabit NIC performance.

Direct memory access (DMA) operation overhead is related to the PCI burst size because each DMA transfer needs to acquire the bus (bus arbitration), set the address, transfer data and release the bus. The total clock cycles needed for a burst transfer is

total clock cycles = 8 + (n-1) + 1 (Idle time on bus)

where n *is the number of data transfers per burst, and*
8 *is the overhead of burst operation*

Table 4 shows the latency and transfer rate of using different burst sizes for data transferring across the PCI bus. It clearly shows that as the burst length increases, the transfer rate increases. PCI-X extends the burst size to a few kilobytes, which can improve the I/O performance where large burst sizes can be applied.

Table 4. Latency for Different Burst Length Transfers (32-bit/33MHz PCI)

Burst Size	Total Bytes Transferred	Total Clocks	Transfer Rate (MB/s)	Latency (ns)
8	32	16	60	480
16	64	24	80	720
32	128	40	96	1200
64	256	72	107	2160

An interesting issue in improving network transmission and receiving is the use of the so-called zero-copy implementation. A common misconception is that zero-copy implementations may double the network transfer. However, zero-copy implementations only help I/O performance when the I/O bus clock rate is close to the memory bus clock rate. As the memory to I/O bus clock ratio increases in the future, zero-copy will not be very helpful for host I/O performance. The zero-copy implementation *will* help the system to reduce the CPU usage, because the user to kernel space memory copy is done by the CPU. So when the I/O performance is CPU bound, zero-copy will improve throughput. A key issue with a zero-copy implementation is memory page mapping. To map an I/O memory buffer to a user data buffer, the buffer size must be equal to the memory page size, typically 4 KB, controlled by memory controller (hardware). This requires that I/O data be in 4-KB data blocks, which does not map well to 1500-byte packets of current Ethernet-based networks. When Jumbo Frames of at least 4 KB are used, then a zero-copy implementation will be possible.

The percentage of performance that will be increased can be derived from equation (2) above and substituted into equation (3) below:

$$percentage = \frac{newThroughput - oldThroughput}{newThroughput}$$

$$= \frac{\frac{MemoryBandwidth}{\frac{MemoryClock}{IOBusClock}} - \frac{MemoryBandwidth}{\frac{MemoryClock}{IOBusClock} + 2}}{\frac{MemoryBandwidth}{\frac{MemoryClock}{IOBusClock}}}$$

$$= \frac{2 \times IOBusClock}{MemoryClock + 2 \times IOBusClock} \qquad (3)$$

A zero-copy implementation is not really *zero* memory copy, it only eliminates the memory copy from user space to the kernel or inversely. It will never eliminate the I/O bus transfer (DMA), which is the major bottleneck to the I/O performance. That is, only two memory cycles are eliminated in zero-copy implementation. If I/O bus speed is 66 MHz and the memory bus is 133 MHz, the performance will be increased by

$$percentage = \frac{66 \times 2}{133 + 66 \times 2} = 49.8\%$$

but if the memory bus increased to 400 MHz, then the maximum percentage of throughput improved by zero-copy implementation is

$$percentage = \frac{66 \times 2}{400 + 66 \times 2} = 24.8\%$$

Another method to on increase system performance is to use SMP (symmetric multiple processors). High-end I/O buses are most likely only supported on SMP motherboards. For example, most x86-based single CPU motherboards only have 32-bit/33MHz PCI bus, while all x86-based SMP motherboards support PCI-X and 64-bit/66MHz PCI buses.

PCI-X will help to increase the I/O performance, but when using SMP systems, one must be aware that plugging in two or more CPUs will reduce the system memory bandwidth. This is due to the bus arbitration. For example, just plugging a second CPU on a SMP motherboard without using it can reduce memory bandwidth by 10-15%. Activating the second CPU with an SMP OS kernel will reduce memory bandwidth even more, up to 20%.

Summary

Achieving a fast enough packet sending rate to measure high-speed networks is not trivial on current (or even near-future) hardware. Therefore, as part of the design of network bandwidth estimation algorithms, host hardware, memory bandwidth, CPU power, I/O bus bandwidth, and NIC speed all need to be considered. This allows an algorithm to determine if a given host is capable of measuring bandwidth. To measure available bandwidth, both hosts must be able to handle data transfer rates higher than the available bandwidth. Otherwise, only the maximum throughput of the slower end host can be measured.

Implementation of the algorithm is another important factor affecting the data transfer rate. The implementation also depends on the operating system. For example, assume that a system has 1000 MB/s memory bandwidth, and one system call costs 1 µs. Sending a 20 KB UDP datagram from user memory to NIC memory takes 100 µs + 1 µs. If this datagram is sent as 20 1 KB datagrams, then the total time will be 100 µs + 20 µs. The second method reduces the transfer rate by approximately 20%. So, in algorithm design and implementation, both hardware and software issues must be considered.

In summary, our study shows that algorithms are more robust if they have lower system resource requirements. Algorithms restricted by timing related system resources will have difficulty measuring network bandwidth on high-speed networks. One possible solution to timing-related problems is to directly access a system real-time clock register in order to obtain the accurate time. A generic solution is to use a time-insensitive method such as packet trains to build new algorithms.

6. USING BERKELEY PACKET FILTER

The Berkeley packet filter (BPF) provides an interface to network data link layers in a protocol-independent fashion. All packets traveling on the network the host is attached to are accessible via this mechanism. BPF also can timestamp each packet as it arrives. However, obtaining an accurate packet arrival time is a very difficult task when the NIC speed is 1 Gb/s or higher; and using current PC-based hardware to capture all packets in a high-speed flow is non-trivial.

Time related issues — interrupt moderation and the cost of obtaining system time — were discussed in § 3. These both introduce timing errors during timestamping of every incoming packet. Errors occurring via get system time function, *gettimeofday()*, may be correctable when the error rate is low, but errors caused by interrupt moderation are not correctable. In measuring a 1 Gb/s network, the maximum time for receiving two contiguous 1500-byte IP packets is 12 µs. According to the discussion in § 3, the system time function (getting time from the clock time counter or CTC) will introduce an 8% error (based on 959 ns cost of *microtime* kernel function). When measuring a 10 Gb/s network, the maximum packet spacing is 1.2 µs, so the 959 ns cost will result in an 80% error. Since the cost of the *microtime* function is relatively constant, this error can be calibrated by subtracting the *microtime* function cost from the packet arrival time.

The interrupt moderation causes non-correctable time errors because all incoming packets are collected and DMAed (direct memory access) into system memory without timestamping until an I/O interrupt occurs. Once the interrupt moderation time has expired, the network device driver is triggered to process all these packets in the system memory. This timestamp is useless because the actual packet arrival time is unknown. In § 3, we mentioned that an onboard NIC timer is one possible solution for this problem, but onboard NIC timer technology is not suitable for a general purpose network measurement tool. because very few NICs have an onboard

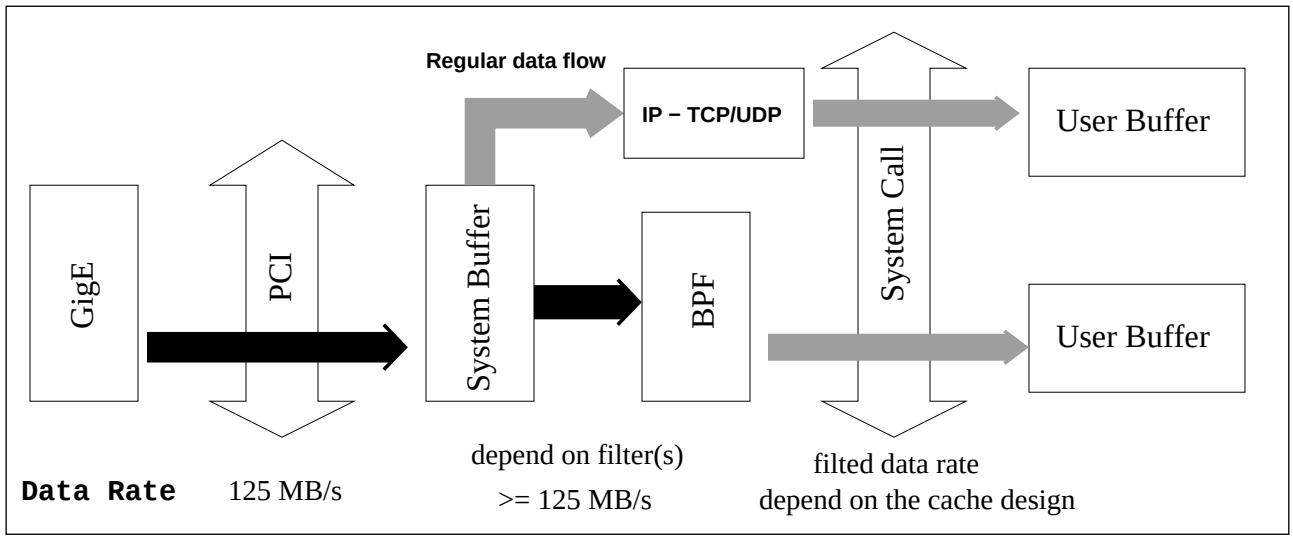

Figure 10. Hardware data path for incoming packets

timer. Also, using an onboard NIC timer requires modifying the device driver for all NICs (with onboard timers) and BPF catchpacket() + bpf_mtap() functions in all required operating systems (For example, see [25] for the source code for modifications of using the SysKonnect NIC under FreeBSD).

Therefore using BPF is not a useful solution for general purpose network measurement tools. It requires that developers have a very deep knowledge of the device driver, operating system kernel, application code strategy, and algorithm development. It also requires a specific NIC and a specific operating system to run. This prevents the tool from being used on different systems.

Without completely understanding both operating system and hardware system design, one might think that using BPF to capture packets and timestamp them may be easier than doing the same task at user level. In fact, using BPF to capture packets often requires higher system resources, such as CPU and memory bandwidth. Figure 9 is the data path for a normal network traffic flow through the end host system. When the BPF device is opened, the NIC operates in promiscuous mode which allows all packets on that network segment to be captured and copied into system memory. This behavior changes the data path on the system. Figure 10 shows the new data path, and this data path is highly sensitive to cache size and data processing speed, including the CPU and memory bandwidth.

Therefore, building a BPF-based capturing system requires even more careful system design. A key issue when NICs are operating in promiscuous mode on a host is that the amount data transferred from NIC into system memory via DMA can be at the maximum NIC capacity, not only the amount of data sent to this host. Of course, this depends on whether or not the host is connected to a shared or switched link. The design must consider all the traffic from the network unless the host is directly connect via a dedicated link. Let R_{cp} be the capture data rate, which is less than or equal to the NIC capacity (R_{nic}), the total memory bandwidth required for non BPF based capturing system is:

$$\begin{aligned} Bandwidth &= (PCI + MemoryCycle \times 2) \times R_{cp} \\ &= PCI \times R_{cp} + MemoryCycle \times R_{cp} \times 2 \end{aligned}$$

and for BPF based capturing system is:

$$\begin{aligned} Bandwidth = {} & PCI \times R_{nic} + MemoryCycle \times R_{cp} \times 2 \\ & + CacheableMemoryCycles \times R_{nic} \\ & + L3L4MemoryCycles \times R_{nic} \end{aligned}$$

PCI is in equivalent memory clock cycles (*MemoryCycle*)
L3L4MemoryCycles is memory bandwidth for IP-TCP/UDP.

From the above equations, we see that if the data capture rate is at the NIC capacity, the BPF-based capturing system needs more memory bandwidth to do the filtering than the non-BPF-based system. This bandwidth required for filtering depends on the BPF filter size, BPF buffer size, cache size, and how fast an application can drain the filtered data (the CPU speed). When the data capture rate is less than the NIC capacity, a BPF-based capturing system requires extra memory bandwidth to handle the unwanted data DMAed into system memory, and this amount of bandwidth cannot be reduced. The tunable bandwidth is the amount (CacheableMemoryCycles × Packet Rate) required by the filtering process. In general, the BPF buffer size should be about one half of the cache size (built-in CPU cache controller), assuming that CPU speed and incoming packet rate (the number of packets per second, not the data rate) are moderate. If the CPU is capable of faster filtering and the capturing application can drain the filtered data immediately, then the BPF buffer size should be larger than one half of the hardware cache size. If the packet rate is higher, the BPF buffer should be relatively smaller. The idea is to keep the packet header access cost in one CPU clock cycle instead of in one memory clock cycle, which is N times greater than the CPU clock cycle (N is the ratio of the CPU clock rate to memory bus rate). For example, a capture system has a 933 MHz Intel P3 Xeon CPU with 256 KB cache to capture 1 Gb/s network traffic. A 141 KB BPF buffer gives the best system performance, capturing 242,718 packets per second (average 515 bytes per packet) for tcpdump to write results to a fast local disk (100 MB/s). When the BPF buffer was increased to 512 MB, the capture rate dropped to 226,222 packets per second. With a 32 KB default BPF buffer set by libpcap, the capture rate is 45,244 packets per second using the same average packet size.

Above discussions are for using BPF on a receiving host. When using BPF on a sending host, it requires more memory bandwidth. Besides accommodating all incoming traffic, the sending host also needs to send traffic and to do filtering on outgoing traffic. This is the reason why running *tcpdump* on a sending host cannot completely capture all outgoing traffic. Therefore, when designing a network measurement system using BPF, the algorithm has to consider if the tool is for measuring sending traffic or receiving traffic. These factors illustrate that BPF is a useful mechanism for capturing packets for data analysis, but may not be suitable for measuring bandwidth on a high-speed network.

7. CONCLUSION

An important issue for implementing available bandwidth algorithms is the speed of the measurement host compared to the physical bandwidth of the network. Current high-speed network bandwidth exceeds most available system I/O bandwidth, and this will likely continue for the foreseeable future. One should not expect that simply faster CPU or memory will make the measurement job easier.

Two issues arise when determining how to measure available bandwidth on a high-speed network:

1) Can a slow end host measure a network bandwidth that is higher than the host NIC bandwidth and/or the I/O bus of the end hosts? Existing algorithms are only able to measure the network capacity, but not available bandwidth. Current algorithms for measuring available bandwidth require that the end hosts have higher throughput than the available network bandwidth. Therefore, new available bandwidth estimation algorithms are needed. One possible solution is to measure physical bandwidth, then estimate cross traffic, thus computing the available bandwidth.

2) It is important to take into account the system timer resolution when designing available bandwidth measurement

algorithms. When the network capacity is high, the time to transmit or receive a packet becomes very short. Therefore, it is not possible to measure available bandwidth using packet pair dispersion algorithms on very high speed networks. Current experience shows that using packet trains is an excellent alternative for building algorithms to measure the network bandwidth in the future.

Based on the above analysis and discussion, we conclude with the following advice for designers and implementers of high-speed network available bandwidth estimation algorithms and tools:

1. Most existing available bandwidth algorithms and tools are only accurate up to speeds of 100-150 Mbits/second. A very few algorithms work on speeds up to 1-2 Gbits/second. Future algorithms should target network speeds of 10 Gbits/second or higher.

2. When designing and implementing available bandwidth estimation algorithms and tools, one must be very aware of the system hardware issues described in this paper.

3. The packet train is currently the best methodology for building algorithms to measure high-speed network bandwidth.

4. Tools that attempt to measure available bandwidth should attempt to determine whether or not the measurement host is the bottleneck, and report this fact when it is. New algorithms are needed that do not require a high-throughput end host to measure network available bandwidth.

8. ACKNOWLEDGMENTS

This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. This is report no. LBNL-48556. See disclaimer at http://www-library.lbl.gov/disclaimer.

References

[1] Uyless Black, Network management standards: SNMP, CMIP, TMN, MIBs, and object libraries. New York: McGraw- Hill, c1995.

[2] R.L. Carter and M.E.Crovella, “Measuring Bottleneck Link Speed in Packet-Switched Networks,” Performance Evaluation, vol. 27,28, pp. 297-318,1996.

[3] Kevin Lai and Mary Baker. Measuring Bandwidth. In Proceedings of IEEE INFOCOM, March 1999.

[4] Allen B. Downey, Using pathchar to estimate Internet link characteristics, proceedings of SIGCOMM 1999, Cambridge, MA, September 1999, 241-250.

[5] Kevin Lai and Mary Baker, “Nettimer: A Tool for Measuring Bottleneck Link Bandwidth”, Proceedings of the USENIX Symposium on Internet Technologies and Systems, March 2001.

[6] C. Dovrolis, P. Ramanathan, D. Moore, What do packet dispersion techniques measure? In Proceedings of IEEE INFOCOM, April, 2001.

[7] Thomas J. Hacker, Brian D. Athey, The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network, Aug. 2001.

[8] G. Jin, G. Yang, B. Crowley, D. Agarwal, Network Characterization Service (NCS), HPDC-10 Symposium, August 2001

[9] Stefan Saroiu, SProbe: A Fast Technique for Measuring Bottleneck Bandwidth in Uncooperative Environments. Available: http://sprobe.cs.washington.edu

[10] Manish Jain and C. Dovrolis, Pathload: A Measurement Tool for End-to-end Available Bandwidth, PAM, March, 2002.

[11] Attila Pasztor´, Darryl Veitch, PC Based Precision Timing Without GPS, Sigmetrics, June 2002

[12] Attila Pásztor, Darryl Veitch, Active Probing using Packet Quartets, IMW, Nov. 2002

[13] Jin, Guojun, “Algorithms and Requirements for Measuring Network Bandwidth”, technical report LBNL-48330, 2003

[14] Deb Agarwal, José María González, Guojun Jin, Brian Tierney, “An Infrastructure for Passive Network Monitoring of Application Data Streams”, PAM, April 2003

[15] Vinay Ribeiro, Rudolf Riedi, Richard Baraniuk, Jiri Navaratil, Les Cottrell, “pathChip: Efficient Available Bandwidth Estimation for Network Paths”, PAM, April 2003

[16] G Jin, B Tierney, “Netest: A Tool to Measure the Maximum Burst Size, Available Bandwidth and Achievable Throughput”, ITRE, August 2003

[17] Justin (Gus) Hurwitz, Wu-chun Feng, “Initial End-to-End Performance Evaluation of 10-Gigabit Ethernet”, Hot Interconnect 11, August 2003

[18] Ningning Hu, Peter Steenkiste, “Evaluation and Characterization of Available Bandwidth Probing Techniques”, IEEE JSAC Special Issue in Internet and WWW Measurement, Mapping, and Modeling, 3rd Quarter, 2003.

[19] ftp://ftp.arl.mil/pub/ttcp

[20] http://dsd.lbl.gov/NCS/back/emn.html#EMN_LAB

[21] http://dast.nlanr.net/Projects/Iperf

[22] Netperf: A Network Performance Benchmark. Available: http://www.netperf.org/netperf/training/Netperf.html

[23] pchar: A Tool for Measuring Internet Path Characteristics. Available: http://www.employees.org/~bmah/Software/pchar

[24] http://www.psc.edu/networking/treno_info.html

[25] http://dsd.lbl.gov/SCNM/FreeBSD_mods.html

[26] http://www.caida.org/projects/bwest/presentations/mtgjun02/L2effects.pdf