Architectural Breakdown of End-to-End Latency in a TCP/IP Network

Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli
Intel Corporation
Steen.Larsen@intel.com

Abstract

Adoption of the 10GbE Ethernet standard has been impeded by two important performance-oriented considerations: 1) processing requirements of common protocol stacks and 2) end-to-end latency. The overheads of typical software based protocol stacks on CPU utilization and throughput have been well evaluated in several recent studies. In this paper, we focus on end-to-end latency and present a detailed characterization across typical server system hardware and software stack components. We demonstrate that application level end-to-end latency with a 10GbE connection can be as low as 10 microseconds for a single isolated request. The paper analyzes the components of the latency and discusses possible significant variations to the components under realistic conditions. We note that methods that are used to optimize throughput can often be responsible for the perception that Ethernet based latencies can be very high. Methods to pursue reducing the minimum latency and controlling the variations are presented.

1. Introduction

The introduction of the 10GbE Ethernet as an IEEE 802.3 standard in 2002 has led to a re-evaluation of data center infrastructures with particular attention to the server systems architecture. The capability of typical server systems in efficiently terminating TCP/IP streams has been evaluated and led to several types of solutions in the industry. Several methods of network acceleration ranging from state-full offloads (TOEs [1,2]) to stateless offloads with platform assists (I/OAT [3,4]) focusing exclusively on CPU utilization and network throughput have been proposed. Very limited advancements have been proposed to comprehensively address the latency between systems in a data center under Ethernet-based networking. We also observe that throughput oriented innovations may have skewed the perception of latency in Ethernet based networking [4].

The focus of this paper is the 'end-to-end' latency between applications running on two different systems. In Section 2, we present general motivation for addressing end-to-end latency and review previous attempts to quantify network latency. We note that certain classes of high-performance computing (HPC) applications seek latencies of just a few microseconds and have motivated innovations such as MPI, Myrinet and Infiniband. These innovations have tended to address Ethernet limitations but only by completely bypassing typical TCP/IP Ethernet systems architecture. Emerging NVRAM based storage subsystems or storage caches are also likely to motivate aggressive latency reductions for a broader class of applications.

Our study is divided into two main areas: we first present a detailed characterization under ideal circumstances, and then evaluate practical aspects that dilate latency. After explaining the motivation in sections 3, section 4 contains a description of our experimental setup and methodology. For our studies, we instrumented the source code of a Linux stack running with nanosecond resolution timers on the latest Xeon processor systems. Sections 5 and 6 quantify latency components associated with the application interface to system software, the network stack and hardware latencies between CPU cores, memory and I/O subsystems. We observe that Ethernet based latency with 10GbE can be as low as 10µs. Latency data can be very sensitive to the workload, software stack and system hardware assumptions. In Section 7, we identify the primary sources of variation that significantly effect overall average latency. In a heterogeneous application environment, a system may be exposed to throughput oriented connections and latency sensitive connections. In current 1GbE and 10GbE adapters with no specific expectations on latency, techniques to reduce the rate of interrupts may significantly bloat overall latency. Three other issues discussed in this paper are head-of-queue effects,

contention for various system resources, and core affinity. Each of the significant factors that influence latency is addressed comprehensively in Section 8. We envision a set of techniques that collectively can achieve deterministic 10µs latencies for Ethernet based networked systems and under a wide range of circumstances.

2. Background (Motivation & Prior Work)

We are primarily focused on data center environments where latency differences of few microseconds can have a significant impact on application performance. Transcontinental distances of the wild Internet involve speed of light dependency and multiple complex packet hops where latencies can easily get above many orders of magnitude beyond a data center. A packet traveling 4800Km (3000 miles) accounting only for wire delay would take 27ms, while a packet in a data center traveling 50 meters would take 278ns to propagate. Within a datacenter the system-to-system latency has the wire speed as a small fraction of the overall latency.

Latency is orthogonal in definition to throughput. Increasing throughput means increasing the number of messages launched, processed or received. Network and system capacity can be increased, but latency itself may or may not be affected by changes in throughput. The importance of latency relative to throughput is very application specific. In a latency sensitive application, a compute resource being used by the application is stalled while waiting for an access to data to return from a network location. If an application thread (a hardware context) stalls after making a request for data and no other thread is available to be scheduled or can be scheduled to hide the latency, the application is latency dependent. If the latency can be hidden by scheduling other threads in place of the stalled thread, the application can become more throughput oriented.

Many applications may be designed to spawn as many threads as possible to hide latency if the end-user response time is not the primary metric. Several popular throughput-oriented benchmarks such as SPECjAppServer and TPC-C (non-clustered) exhibit such behavior. In these benchmarks, the response time need only be within reasonable limits. High network latencies and high queuing latencies within the system can be tolerated much more in such scenarios. It is critical to note that the scheduling of many threads to hide latency compromises platform efficiency even if response time targets can be met. Scheduling overheads including context switches, thread migration, and contention for shared resources such as caches can be reduced if latency can be reduced and a lower number of threads are active at any given time.

There are several types of applications that have been known to exhibit a greater latency sensitivity compared to bandwidth sensitivity. In particular, we are focused on scenarios where microsecond level difference in latency has an application level significance. Three usage models of interest are recognized as follows:

1) Synchronization latency: This is the latency associated with synchronization messages between multiple threads of an application that has been parallelized to run on different physical compute nodes. Example: Parallel computing
2) Distributed memory access latency: In a distributed memory application, threads running on one compute node access the memory of another compute node in a cluster. i.e. HPC, database clusters, and business performance clusters (BPC)..
3) Storage media access latency: Traditionally, latency to access magnetic tapes or rotational media like disk drives has been several milliseconds. However, with the advent of solid-state devices and the usage of DRAM based caches that front-end magnetic media, latencies can be expected to drop down to microseconds.

Past studies have compared various interconnects and software interfaces with a focus on latency. In an NCSA study, [6] a simple ping-pong test is done comparing TCP/IP messages with Myrinet and Infiniband. For a small message of 64 bytes Infiniband is 5.3µs and Myrinet is 8.3µs compared to 60µs TCP/IP. Infiniband is consistently 10 times lower latency than TCP/IP as messages increase until 64KB where the bandwidth available begins to impact latency.

A significant portion of the latency for TCP/IP comes from the software interface. In this protocol, the two ends that are communicating are not only assumed to be completely asynchronous but are even unaware of each other. When a message arrives at a compute node, a processor must be interrupted and software must discover the application that must process the new message via protocol stacks. Once the discovery takes place, the application must be context switched and the data is copied into the applications buffer before the message can be processed. In addition to this fundamental overhead, several significant sources of

variation may occur. For example, a NIC may use interrupt moderation to amortize the processing overhead of interrupts across a batch of packets. Such techniques could artificially add latency irrespective of the latency sensitivity of a packet within a batch.

In the HPC environment, often the solution is to use an interconnect based on Infiniband or Myrinet. Of the top 500 supercomputers, about 30% use Infiniband or Myrinet interconnects [8]. Latencies between two systems to transmit and receive a message can be as low as 1.29µs with Qlogic InfiniPath HTX host channel adapters. Attempts to reduce latency with TCP offload engines [7] do not yield significant results due in part to the context movement required by TOE. This context is the mapping of IP addresses, TCP ports, and other control flows normally done by the processor executing the application level software. Even with the advent of Ethernet cut-through switches that result in Ethernet switch latencies of 300ns compared to 200ns Infiniband switch latencies, HPC customers are cautious to switch to Ethernet. To summarize, it is generally perceived that Ethernet is about an order of magnitude away from low latency networks such as Infiniband and Myrinet.

3. Problem Statement

It may be obvious why TCP/IP has a much higher latency than Infiniband and Myrinet and other hardware based protocols to efficiently pass point-to-point messages between applications. Key to the differences is that the hardware supports the packetization, segmentation, reassembly and movement of payload data into user space. This is directly accessible by the user space application without a transition from kernel space to user space with memory copies or complex pointer redirections.

On the other hand, TCP/IP has shown remarkable flexibility in its 30 year history. Although certain aspects remain constant (MTU, stack layer structure with TCB and memory descriptors, and a focus of core processing of the stack) several enhancements have streamlined the efficiency of using the incumbent networking protocol. These include interrupt moderation schemes, receive side scaling (RSS) to map to multiple cores efficiently, transmit side offloading (TSO), and TCP/IP offloading schemes. Additionally different protocols have been shown to work and have been widely adopted with the Ethernet hardware layer such as RDMA and layering on top of the TCP/IP stack such as iSCSI.

As a result, with the growing importance of low latency in applications, what generates the latency becomes important. The overall goal would be to improve TCP/IP if possible and if not reasonable, propose proper alternatives based on the importance of latency to an application. The first question is:

What is the breakdown of minimal latency components to pass a message from one system's application to another system's application?

To answer this we break down the sequence of operations between an application preparing a message and sending it out the wire. Over a short wire, we observe the receiving of the message and follow the processes needed to pass the message to the receiving application.

Once the minimal latency is understood between two systems in a simple connection, we can look at variations upon the message latency. In other words, what is the variation and scope beyond the minimum latency in the previous question?

By looking at how typical TCP/IP transactions encounter events that prolong latency, we can understand what can be done to improve TCP/IP latency.

4. Experimental Setup

Figure 2 below shows the logical experimental configuration which is a basic back-to-back fiber optic connection between two Intel Xeon 2.13GHz 5138 based platforms. To avoid core affinity mapping concerns, only a single core was enabled. The NIC installed is an Intel 82571 1GbE based card using a PCIe x4 link [10] to the Intel® 5000P chipset. [5] This is a 1Gbps fiber optic NIC.

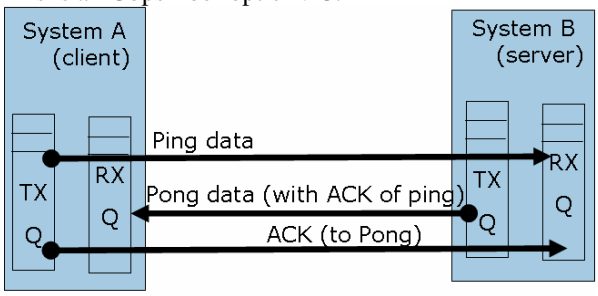

Figure 2

On the software configuration, the two systems are executing Linux 2.6.18 RC3 kernels. NAPI [11] has been turned off so there is no interrupt moderation in the simple message tests being run. NetPIPE 3.6.2 was used to send a simple TCP/IP ping-pong message between the two systems. A complete test therefore constitutes 3 packets on the link layer:

1. Ping to server
2. Pong response to client (with ACK of ping)

3. ACK to server for pong.

It should be noted that the stacks on each system sees three transactions. The client processes a ping transaction and receives ACK and pong transactions. The server receives a ping and ACK and sends a pong (with ACK). Figure 3 illustrates this with how the client system reacts to the ping-pong test.

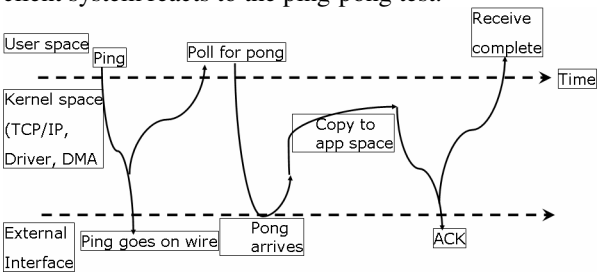

Figure 3

The NetPIPE application is configured to run a single byte of information from the client to the server and pong back. This results in a 64 byte packet seen on the wire which is the minimum TCP/IP packet size.

With the application in a running configuration, a breakdown of application, stack components and driver latencies can be derived by adding kernel probes into the source code of the application and kernel. These probes enable and disable performance monitoring event registers such as CPU_CLK_UNHALTED [9] counting the processor core clock cycles. For a simple application where a single packet is being sent and a single packet is being received, there is enough control to deterministically enable/disable the counter.

Below is an example code of counting cycles during the TCP sending copy operation. Between the Start and End of instrumentation is the normal Linux kernel transaction.

```
  Start_Instrumentation(skb,          \
ACROSS_TCP_SENDMSG_COPY);
  err = skb_copy_to_page(sk, from, \
skb, page, off, copy);
  End_Instrumentation(1,              \
ACROSS_TCP_SENDMSG_COPY);
```

There is instrumentation latency added with this approach. When the sampling is done, before sampling real data, an empty Start/End sequence is made to calculate the number of core cycles used to Start and Stop the instrumentation. This time is subtracted from any data such as the example above.

There is a certain amount of variation in the cycle count. The exact core clock cycle when the counter is enabled and disabled depends on the state of the processor when the event counter is called. Even in a controlled environment as a single ping-pong test, up to 30% variation can be seen in processor clock cycles. As a result, averaging each sample over 10,000 ping-pong tests was captured and of these at least 7 were visually inspected for isolating extreme cases such as system warm up or other potential configuration perturbance. The average of these averages was then used for each latency component discussed in section 6. Although we are concerned with the minimum latency of each component, we cannot simply take the minimum, since a minimum in one component (i.e. TCP) will result in a non-minimum of another component (i.e. IP)

Although the variation in the core clock counter prevents an absolute minimum latency value to be measured, some checks can be done. One was to re-run the same test on another pair of client-servers. Accounting the difference between a 2.13GHz core and 3.0GHz core, certain latency components were checked with very similar clock counts in this different environment. Additionally the L2 miss performance register was tested, and all cases the last level cache misses of the Xeon processor matched in every ping-pong test executed.

In the measurements made, only a single client system was used to break down the latency components. The time needed to send a packet is based on the client sending the ping. The time needed to receive a packet is based on the client receiving the pong packet.

There are hardware based latencies that cannot be instrumented with software noted in Table 1 and 2. These are based on estimated latencies based on previous hardware measurements.

5. Overview of Latency

The NetPIPE application responds with a measurement of time to complete the ping-pong round trip time. This time is divided by two to represent the time required to send a packet and receive the packet.

For the Intel 82571-based 1Gbps back-to-back configuration described above, NetPIPE reports 14µs. This is the round trip ping-pong transaction of 28µs. The next section will do a detailed breakdown of components leading to this 14µs.

With 10Gbps NICs becoming available, a back-to-back configuration with PCIe x8 cards and 3.0GHz Xeon 5160 cores was studied. Again turning NAPI off, and using a single core configuration, the one-way latency reported by NetPIPE for a TCP single byte message was repeatedly 10.2µs. A certain amount of speedup from the faster core is expected, but since

small 64 byte packets are being used, there is as expected little benefit from a 10Gbps link.

6. Detailed Latency Breakdown

First let us take the latency breakdown from the transmission of a packet through the physical interface

Table 1

Description of transmit packet activities	Source	Time (ns)
Application sends a message to the socket interface	Measure	950
TCP prepares a datagram to IP layer	Measure	260
IP layer calls network device driver	Measure	550
Netdev calls precise hardware implementation	Measure	430
Basedriver execution and hand control to NIC	Measure	400
Core IO write propagation delay to wake up NIC	Estimate	180
NIC to process core write and fetch descriptor of packet to transmit	Estimate	580
NIC, based on descriptor, fetches packet header/payload and sends packet to PHY	Estimate	400
Total Transmit packet time		**3,750**

and ending with the receive latencies. To begin with the latency component breakdown, the NetPIPE application needs to prepare a transmit request for the ping operation. This takes 950ns to send a message to the socket interface (on a connection that has already been established) with the `sock_write()` function. This then calls the `tcp_sendmsg()` to begin the TCP transmission.

Once the TCP layer begins, the application buffer is copied into kernel space and pushed into the transmit queue. At this point, after 220ns, the `ip_queue_xmit()` can be called to initiate the IP layer.

The IP layer takes 450ns to do various tasks such as routing, segmentation, and IP header processing before calling the network device driver with `dev_queue_xmit()`. The network driver consumes 430ns to construct the output packet queue entry and calling `e1000_xmit_frame()`. The e1000 1Gbps NIC driver sets up a DMA transfer with the 82571 NIC by indicating a packet is pending transmission. This is done by writing to a NIC control register, taking 400ns to complete. The value written to the NIC is a pointer into a ring of descriptors in main memory which the NIC can fetch the packet descriptor of the packet to transmit.

A time of about 180ns is needed to propagate the tail pointer register write from the core to the NIC. The NIC, once it knows a packet is needed to transmit, fetches a descriptor from the memory (via DMA) that defines where the packet header and payload reside in memory. Since the 82571 is running at a slower speed of about 135MHz, it takes about 180ns to interpret the descriptor. A second DMA read by the NIC is generated to fetch the actual packet data. Each 64-byte cache line access to memory takes an estimated 400ns to propagate from the PCIe signals pins to memory and back. As a result, the hardware latency is 1,160ns for the NIC to launch the packet onto the fiber interface. The summation of the 3,750ns packet transmit process is shown in Table 1.

The physical adapter (PHY) basically translates to optical waves and tunes the signal to pre-emphasize or de-emphasize the signal using DSP algorithms. This is needed to counteract the number of “zeros” or “ones” in a serial pattern that may skew the receiving end incorrectly. The latency specifications on 1GHz PHYs are usually <10ns. The short fiber connection between the two machines of 3meters has a latency of light of an estimated 10ns. Since the fiber speed is 1Gbps, an additional 672ns are needed to propagate the 64-byte minimum packet size and 20 bytes of inter-frame gap and preamble. The total wire time including the PHYs is estimated at 702ns.

For the receiving process, in this case the receiving of a pong packet from the server, is similar to the reverse of transmit. The NIC, upon filtering a packet as its own packet, will start writing into main memory based on the contents of a pre-fetched packet descriptor. This filtering takes 200ns at the 135MHz NIC speed at which point the NIC will start the DMA write into main memory. Immediately after launching the DMA, the NIC will interrupt the processor that pending receive data is ready. Based on the ordering rules of the PCIe specification [10] the interrupt will not be serviced before the write is finished into main memory to maintain data coherence. This combined DMA and interrupt latency is calculated as 900ns based on the Intel® 5000P chipset specification, resulting in a total 1,100ns before core software is engaged.

The interrupt handler takes about 270ns to switch out the current context and start to determine the source of the interrupt. Most current systems follow a complex process of reading potential interrupt cause

registers (ICR) to determine exactly what generated the interrupt. In the 82571 configuration this takes 1000ns for the read to propagate to the PCIe device and respond to the core. With emerging NICs (that use MSI-X messages) to vector the core directly to the interrupt source instructions to service the interrupt this can soon be counted as zero time. The core, once it enters the `e1000_intr()` routine takes 300ns to process the descriptor pointers in the ring and start calling the `netif_rx()` SoftIRQ. The SoftIRQ is analogous to the Windows deferred procedure calls (DPC) implementation of drivers and takes 1,287ns. Part of this time is needed for the kernel to schedule the call.

Table 2

Description of receive packet activities	Source	Time (ns)
MAC filter determines target packet is for this machine	Estimate	200
NIC starts DMA packet header and payload into memory	Estimate	400
NIC interrupts core with MSI-X packet to APIC	Estimate	500
Hardware MSI-X interrupt service routine to parse what caused interrupt	Estimate	270
Interrupt cause register read requirement	Measure	1,000
ISR packet processing of descriptor to update receive queue	Measure	300
SoftIRQ (deferred procedure call in Windows)	Measure	1,287
TCP and IP receive side processing	Measure	570
Wakeup application to process socket information	Measure	1,274
Kernel to application space data copy.	Measure	208
ACK the pong received by the remote sender	Measure	1,117
Application receive message overhead to register completion	Measure	621
Total receive packet time		**7,747**

After updating the input packet queue, the `ip_rcv()` is called starting the TCP/IP receive process. Both TCP and IP layers take 570ns. The application then needs to be scheduled taking 1,274ns. This is the normal configuration where the application is blocked, basically sleeping and not actively polling the socket on data availability. In the NetPIPE application, once the application is woken up, this triggers the 208ns copy operation from kernel space to memory space. Since the packet is a single cache line, this time falls within the 80ns fully buffered DIMM (FBD) latency to read and then write the data. At this time the acknowledge (ACK) to the pong packet is generated to provide a complete network connection taking 1,117ns. Finally the application needs to register that the ping-pong test is complete taking 621ns. The complete receive sequence is seen below in table 2 for a total of 7,747ns.

Combining transmit, wire and receive breakdown latencies, a total of instrumented, analytical and specification based time is 12.2µs compared to the 14µs reported by the NetPIPE application. This 14% difference is due to the different reporting mechanisms and cumulative error in estimation and measurement.

7. Sources of Variation on Minimum Latency

Variations on minimum latency discussed above are factors that add latency to messages between two systems, often by a 10X multiplication of time in a 10Gbps network. This is also termed message jitter or skew. This section will explore four major causes for variation which adds significantly to average TCP/IP latencies.

Interrupt moderation is a method to reduce the number of processor context switches due to interrupts. Consider a 10Gbps NIC that in the process of bidirectional 64-byte packets would need to interrupt each 26ns. Instead of interrupting on each packet, packets can be grouped together for more efficient processing. While this process reduces the number of interrupts the core needs to deal with, the cost is that latency critical packets are delayed. By default Windows specifies a 250µs window of interrupt moderation to accumulate packet tasks and Linux has a 125µs window. These figures seriously dwarf the 14µs minimum latency mentioned above.

The head-of-queue effect is also a problem. As discussed with the ring buffer, to transmit and receive a packet, the OS sees a serial sequence of packets to service. These packets are serviced in order received or order of pending transmits. As a result, a latency-critical message can be blocked by other protocol requests. In a similar manner this head-of-queue effect is seen in the large buffers of today's NICs. The buffers are needed to support the bursty traffic patterns of today's network, and it is not uncommon to see a 320KB transmit buffer on a 10Gbps NIC. On such a NIC, if the buffer is full and a latency sensitive packet

L is attempting to be sent, it would take the PHY 20µs to drain the buffer at a 10Gbps rate before packet L is presented on the wire.

System bandwidth contention is another issue. Although there can be contention on the processor interface and memory interface, the most visible conflicts are seen on the slower PCIe interfaces. In the case of the 10Gbps NIC above, 28 outstanding PCIe transactions are supported. Each NIC based PCIe transaction on a PCIe x8 configuration can take up to 138ns for large packets. As a result the PCIe interface can block for up to 4µs. This is one of the more extreme bandwidth contentions, but a cumulation of contentions on the various physical interfaces can be considered to occur frequently.

A fourth area to consider is application core affinity. If an incoming packet is being processed on a core, and is not the core running the application, context switch latency is seen for the packet processing core to hand over to the application core. In Linux this is observed to add 2µs and Windows 8µs typically to the complete system-to-system latency. [16]

There are remotely possible conditions that will affect latency as well. Overall processor load will also play a role, but to some extent will fall into the figures listed above. There could be a page fault to disk or a complex OS context switch, but this is considered extremely rare in a modern datacenter. An additional rare occurrence would be a link layer contention or packet retransmission.

In summary, the four estimations mentioned above can add 282µs to minimum 12µs message latency. It can be stated that less than 5% of potentially expected latency that can be contributed to the deterministic requirements of the application, stack, driver, hardware and wire in a datacenter.

8. Methods to Reduce Variation

The primary task in reducing latency in a TCP/IP environment is to first tag latency critical messages. Once they are tagged, there must be a method to detect and classify them both in transmit and receive path. Once latency critical messages are detected, there is need to prioritize over other messages. These methods cannot be done in isolation, but the latency reduction methods need to be propagated to the entire network and systems on the network.

To help address interrupt moderation, New Application Programming Interface (NAPI) is in current use of Linux 2.5/2.6 kernels [11, 12]. This attempts to intelligently monitor the receiving packet flow, and in a low packet per second scenario allows interrupts on a frequent basis. As packets per second increase and the system cannot respond to interrupts efficiently, polling by the kernel of the receive descriptor ring is started. This assumes high packet rates will continue, and if packet rates drop to a more intermittent rate, interrupt based signaling of receive packets can resume and active polling by the kernel driver is stopped.

The Intel e1000 driver supports a form of adaptive interrupt moderation [13] that attempts to classify incoming traffic. The interrupt throttling is then based on the class determined, such that large amounts of packets will generate fewer interrupts and if the class changes to small amounts of packets (or small packets), less moderation will be placed on interrupt generation. This can be extended to include heuristics that trigger based on throughput and packet sizes such that appropriate interrupts are generated at the optimal time. Work by Hansen and Jul [17] ties the operating system scheduler to the asynchronous data arrival to reduce the overall system-to-system latency.

An additional approach in Data Center Ethernet (DCE) [14] is to tag particular flows to differentiate as low latency flows. By having different virtual channels over the same Ethernet interface, different channels can be applied with different interrupt schemes. One option is simply to have certain TCP ports as being low latency and interrupt upon receive traffic regardless of any interrupt moderation control. This can be extended to complex TCP connection information to low latency flows.

As the network interface logic moves onto the die of the core processing TCP/IP, interesting opportunities arise in how to notify the core of pending receive traffic. This could be in the form of having complex monitor/mwait instructions or schemes to map receiving data into a temporary cache in the coherent domain.

The head-of-queue example in NIC data buffer can also be addressed with DCE which will formulate the order of transmit flows and potentially reorder based on latency priority the order of the transmitted packets, bypassing the 20µs mentioned above. Head-of-queue latency impacts are also being addressed with Receive-Side Scaling (RSS) [15] which generates a hash table based on the n-tuple of the flow. This can be the mapping of source IP address and port and destination IP address and port. Based on this hash, different flows can be mapped to different available processors. In this manner a single core, or ring of descriptors, does not become a bottleneck for latency critical messages.

As the NIC moves onto the same die as the core, interesting methods to control the right data being available to the right core such as Direct Cache Access (DCA) [3] can be explored.

The third latency variation discussed is bandwidth contention. The obvious method to affect this is provide more system bandwidth such as bringing the NIC closer to the core associated with processing TCP/IP traffic. Another method is to use DCA based on knowledge of when the core will need the data, avoiding memory bandwidth contention.

The proper alignment of cores to application and packet processing will also reduce the latency. In supporting high throughput it may be appropriate to sequester a core for efficient packet processing. To ensure low latency, attaching the application core to be the same core that processes the packets can reduce up to 8µs in variation.

In short there are many aspects that are under development to make TCP/IP system-to-system latency have much more determinism.

9. Conclusions

End-to-end latency between applications is emerging as an increasingly important metric in data centers. Low latency may not only be a requirement for niche HPC applications but also for much more common applications that are storage intensive and when solid-state storage technologies are adopted.

In current available commercial and relatively inexpensive server systems that communicate via Gb Ethernet we have measured a 12us latency to transmit a message between two machines. In our experimental analysis, we have accounted for all of the significant contributors to this latency. We have observed that much of this time is spent in the application/stack/driver but there is also a significant component in hardware. Further substantial reduction in latency would require simplification of existing driver to OS interface and also the application to system software interface. Hardware latency can be reduced by integration of the network interface eliminating intermediate components such as chipsets.

One of the most significant issues with Ethernet latencies has been the variability. However, it is practical to implement a set of methods to classify latency sensitive packets and to prioritize them throughout the system is possible. Technologies such as adaptive interrupt moderation, DCE, RSS, and NIC integration will significantly bridge any remaining gap between TCP/IP based Ethernet communication latency and other specialized solutions such as Infiniband and Myrinet.

10. References

[1] A. Foong, T. Huff, H. Hum, J. Patwardhan and G.Regnier. “TCP performance re-visited”, Proc. of the IEEE Intl. Symposium on Performance of Systems & Software, Austin, Mar 2003.

[2] J. Mogul, “TCP Offload Is a Dumb Idea Whose Time Has Come,” Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX), Usenix Assoc., 2003;www.usenix.org/events/hotos03/tech/full_papers/mogul/mogul.pdf

[3] R. Huggahalli et.al. “Direct Cache Access for High Bandwidth Network I/O” International Symposium on Computer Architecture (ISCA), 2005 http://www.cs.wisc.edu/~isca2005/papers/02A-02.PDF

[4] G Regnier et. al. “TCP Onloading for Data Center Servers” IEEE Computer Nov 2004.

[5]http://download.intel.com/design/chipsets/datashts/31307103.pdf

[6] http://vmi.ncsa.uiuc.edu/performance/pmb_lt.php

[7] Feng, W.; Balaji, P.; Baron, C.; Bhuyan, L.N.; Panda, D.K.; Performance characterization of a 10-Gigabit Ethernet TOE; High Performance Interconnects; Proceedings. 13th Symposium Aug. 2005

[8] http://www.top500.org/stats/28/connfam/

[9] IA-32 Intel® Architecture Software Developer’s Manual Volume 3: System Programming Guide Appendix A, June 2005.

[10] PCI Express Base Specification Revision 2.0, 2006; http://www.pcisig.com

[11] http://linux-net.osdl.org/index.php/NAPI

[12]http://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf

[13]http://download.intel.com/design/network/applnots/ap450.pdf

[14]http://www.ieee802.org/3/ar/public/0503/wadekar_1_0503.pdf

[15]http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx

[16] A. Foong, J. Fung, and D. Newell, “An In-Depth Analysis of the Impact of Processor Affinity on Network Performance,” Proc. IEEE Int’l Conf. Networks, IEEE Press, 2004.

[17] J. Hansen and E. Jul, “Latency Reduction using a Polling Scheduler,” Proc. of the Second Workshop on Cluster-Based Computing, ACM-SIGARCH 2000.