Mainstream NUMA and the TCP/IP stack

Mark B. Friedman
Microsoft Corporation

Abstract.

Historically, approaches to achieving scalable networking focused on reducing the host processing requirements associated with TCP/IP interrupts. In the many-core era, approaches such as interrupt moderation, jumbo frames, and the TCP Offload Engine prove inadequate because the modest increases in the processing speeds forecast for host computers are not keeping pace with improvements in network bandwidth. The alternative is to distribute the processing of TCP/IP packets across multiple CPUs by session using the technology Microsoft developed for Windows called Receive-Side Scaling (or RSS). This paper motivates the necessity of taking this approach to high-speed networking. Aspects of RSS that are critical to its performance also violate long-cherished software engineering principles of layering embodied in the architecture of the TCP/IP protocol stack. This paper also describes the manner in which RSS needs to be configured to run optimally on server machines with NUMA characteristics. Finally, it discusses the implications of the RSS parallel processing architecture on the performance of the applications that reside one layer above the TCP protocol on machines with NUMA characteristics.

Part 1. Background.

One of the intriguing aspects of the onset of the many-core processor era is the necessity of using parallel programming techniques to reap the performance benefits of this and future generations of processor chips. Instead of significantly faster processors, we are getting more of them packaged on a single chip. To build the cost-effective mid-range blade servers configured in huge server farms to drive today's Internet-based applications, the hardware manufacturers are tying together these complex multiprocessor chips to create NUMA architecture machines– machines with non-uniform memory access speeds. Differences in the speeds in which different memory locations can be accessed is a natural consequence of the complexity of large-scale machines. When the machine is large enough – and here "large" is intentionally a relative term – it is no longer possible for a program to reach every addressable memory location in the same amount of time.

NUMA is nothing new. What is decidedly new is the extent to which previously esoteric NUMA architecture machines are becoming mainstream building blocks for current and future application servers. Historically, NUMA machines were accompanied by complex, hardware-specific programming models if you wanted to build applications that can harness their performance and capacity effectively. For the connected applications of the future, our ability to build programming models that help server application developers deal with complex NUMA architecture performance considerations is the singular challenge of the many-core era. You might say there is an interesting NUMA twist associated with the multi-core, many-core challenge. [1]

In this paper, I will discuss the way both these trends -- multi-core processors and mainstream NUMA architectures – come together to influence the way high speed internetworking works today on servers of various sorts that need to handle a high volume of TCP/IP traffic. These include IIS web servers, SharePoint, Terminal Servers, SQL Servers, Exchange servers, Office Communicator servers, and others. Profound changes were necessary in the TCP/IP networking stack in both Windows Server 2008 and the Microsoft Windows Server 2003 Scalable Networking Pack release to scale effectively on multi-processor machines. These changes are associated with a technology known as Receive-Side Scaling, or RSS. What I want to highlight here is that RSS technology has serious performance implications on the architecture of highly scalable server applications that sit atop the TCP/IP stack in connected system environments.

Let's start by considering what is happening to the TCP/IP software stack in Windows to support high speed networking, which is depicted in Figure 1.

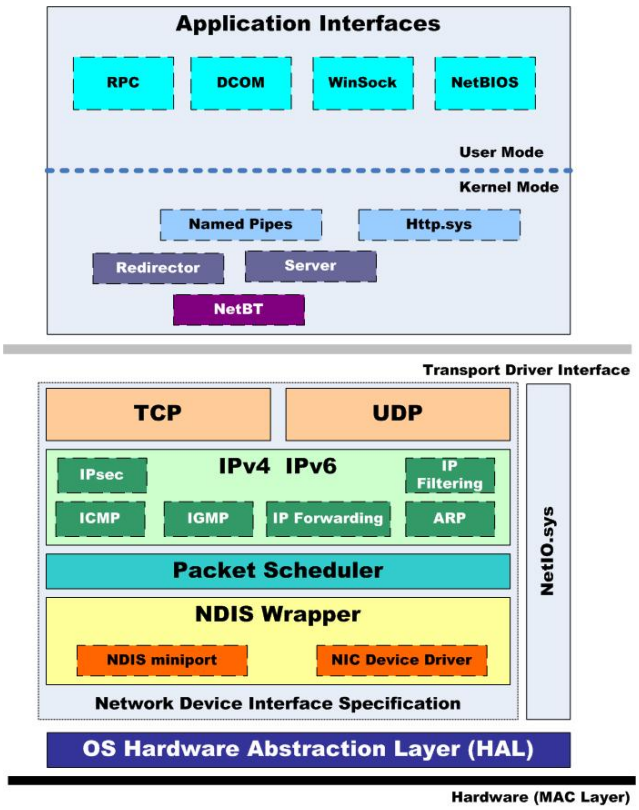

Figure 1. A schematic view of Windows networking, including the standard TCP/IP protocol stack.

The Internet Protocol (IP) and the Transmission Control Program (TCP) are the standardized software layers that sit atop the networking hardware. The Ethernet protocol is the pervasive Media Access (MAC) Layer that segregates the transmission of bits into individual packets. Performance issues with Ethernet arise due to the relatively small size of each packet that is transmitted. The Maximum Transmission Unit (MTU) for standard Ethernet sized packets is 1500 bytes. Any messages that are larger than the MTU require segmentation to fit in standard sized Ethernet packets. (Segmentation on the Send side and reassembly on the Receive side are functions performed by the next higher level protocol in the stack, namely the IP layer.) Not all transmissions are maximum sized packets. For example, the Acknowledgement (ACK) packets required and frequently issued in TCP consist of 50-byte packet headers only, with no additional data payload. On average, the size of packets circulating over the Internet is considerably less than the protocol-defined MTU, due, in part, to the frequency with which TCP requires ACK packets to be sent. The reader who requires additional background into the performance of the TCP/IP stack should refer to [1], especially the sections on Networking.

The performance problems arise because in a basic networking scheme, each packet received by the Network Interface Card (NIC) delivers a hardware interrupt to the host processor, requiring in turn some associated processing time on the host computer to service that interrupt. The TCP/IP protocol is reasonably complex, so the amount of host processing per interrupt is considerable.[1] As transmission bit rates have increased from 10 Mb/sec to 100 Mb to 1 Gb to today's 10 Gb NICs, the potential interrupt rate rises proportionally. The host CPU load associated with processing network interrupts is a long-standing issue in the world of high speed networking. The problem has taken on a new dimension in the many-core era because the network interface continues to get faster, but processor speeds are no longer keeping pace.

[1] Some have argued otherwise. See D. D. Clark, V. Jacobson, J. Romkey and H. Salwen, "An analysis of TCP processing overhead." *IEEE Communications* Magazine, 27(6):23-29, June 1989. This assessment was made prior to the overhaul of the TCP protocol proposed by the very same Van Jacobson that was implemented to address serious scalability issues that Internet technology faced in the early years of its adoption. Taking account of both security and performance considerations, the TCP/IP protocol software stack as implemented today is considerably more complex. *Microsoft Windows Server 2003 TCP/IP Protocols and Services Technical Reference* by Davies and Lee is a useful guide to the full set of TCP/IP services that are provided today, except it does not include the additional functions in the Microsoft Windows Server 2003 Scalable Networking Pack release discussed here. For a recent description of TCP/IP host processor overhead, see Hyun-Wook Jin, Chuck Yoo, "Impact of protocol overheads on network throughput over high-speed interconnects: measurement, analysis, and improvement." The Journal of Supercomputing 41(1): 17-40 (2007)

A back-of-the envelope calculation to figure out how many interrupts/sec a host computer with a 10 Gb Ethernet card potentially needs to handle should illustrate the scope of the problem. For the sake of simplicity, let's assume the 10 Gb Ethernet card has the capacity to deliver data to the application layer at 1 GB/sec.

To understand the rate interrupts/second need to be processed on a host computer to sustain 1 GB/second throughput with Ethernet, simply divide the throughput in bytes per second by the average packet size. To keep the math easy, assume an *average* packet size is 1k or fewer bytes. (This is not an outlandish assumption. A large portion of the Receive packets processed at a typical web server are TCP ACKs; these are minimum 50 byte headers-only packets. Meanwhile, http Get Requests containing a URL, a cookie value, and other optional parameters can usually fit in a single Ethernet 1500-byte packet -- in practice, the cookie data that most web applications store is often less than 1 KB.) Assuming an average packet size of 1 KB, a 10 Gb Ethernet card that can transfer data at 1 GB/sec rate has the capability of generating 1 million operations/sec on the networked server. Next, assume there is a 1:1 ratio of Send:Receive packets. If 50% of those interrupts are Receive operations, then the machine needs to support 500K interrupts/sec.

Now, if the number of instructions associated with network interrupt processing in the Interrupt Service Routine (ISR) associated with the device, the Deferred Procedure Call (DPC), and the next higher layers in the Network Device Interface Specification (NDIS) stack to support TCP/IP is, let's say 10,000, then the processor load to service TCP/IP networking requests is:

*500,000 interrupts/sec * 10,000 instructions = 5,000,000,000 instructions/second* (1)

which easily exceeds the capacity of a single CPU in the many-core era. If these network interrupts are confined to a single processor, which is the way things worked in days of yore, host processor speed is a bottleneck that will constrain the performance of a high speed NIC.

Of course, instead of wishing and hoping that TCP interrupt processing could be accomplished within 10K instructions in today's complex networking environment, it might help to actually try to measure the CPU path length associated with this processing. To measure the impact of the current TCP/IP stack in Windows Vista, I installed the NTttcp test[2] tool and set up a simple test using the 1 Gb Ethernet NIC installed on a dual-core 2.2 GHz machine running Windows Vista SP1 over a dedicated Gigabit Ethernet network segment.[2] Since the goal of the test was not to maximize network throughput, I specified 512-byte sized packets and was careful to confine the TCP interrupt processing to CPU 0 using the following NTttcp parameters:

```
ntttcpr -m 1,0,192.168.3.51 -a 16 -l 512 -mb -fr -t 120
```

I was also careful to shut down all other networking applications, on my machine for the duration of the test, including both VPN and firewall services.

Here's the output from a 120 second NTttcp run, allowing for both a warm-up and cool down period wrapped around the main test:

Throughput(KB/s)	16,475.553
Throughput(Mbit/s)	131.804
Average Frame Size	764.394
Packets Sent	1,309,892
Packets Received	2,586,923
Packets received/Int)	2
Interrupts/sec	9,494.04
Cycles/Byte	129.3

Table 1. Results from a 120 second test of TCP/IP throughput using the NTttcp benchmarking program.

[2] Note that the point of this exercise was not to try to measure the code path definitely using the very latest hardware and software. In fact, the hardware I tested with was decidedly not representative of a typical server configured for high-speed networking. If you try this on your server machines, you can expect that your mileage will vary. The value of the example is to illustrate the tools you can use to assess the performance your networking configuration and provide a better ballpark figure for the TCP/IP code path than mere guessing.

On the dual-core machine, CPU 0 was maxed out at 100% for the duration of the test – evidently, that was the capacity of the machine to Receive TCP/IP packets and process and return the necessary Acknowledgement packets to the Sender. I will drill into the CPU usage statistics in a moment. For now, let's focus on the interrupt rate, which was about 9500 interrupts/sec or

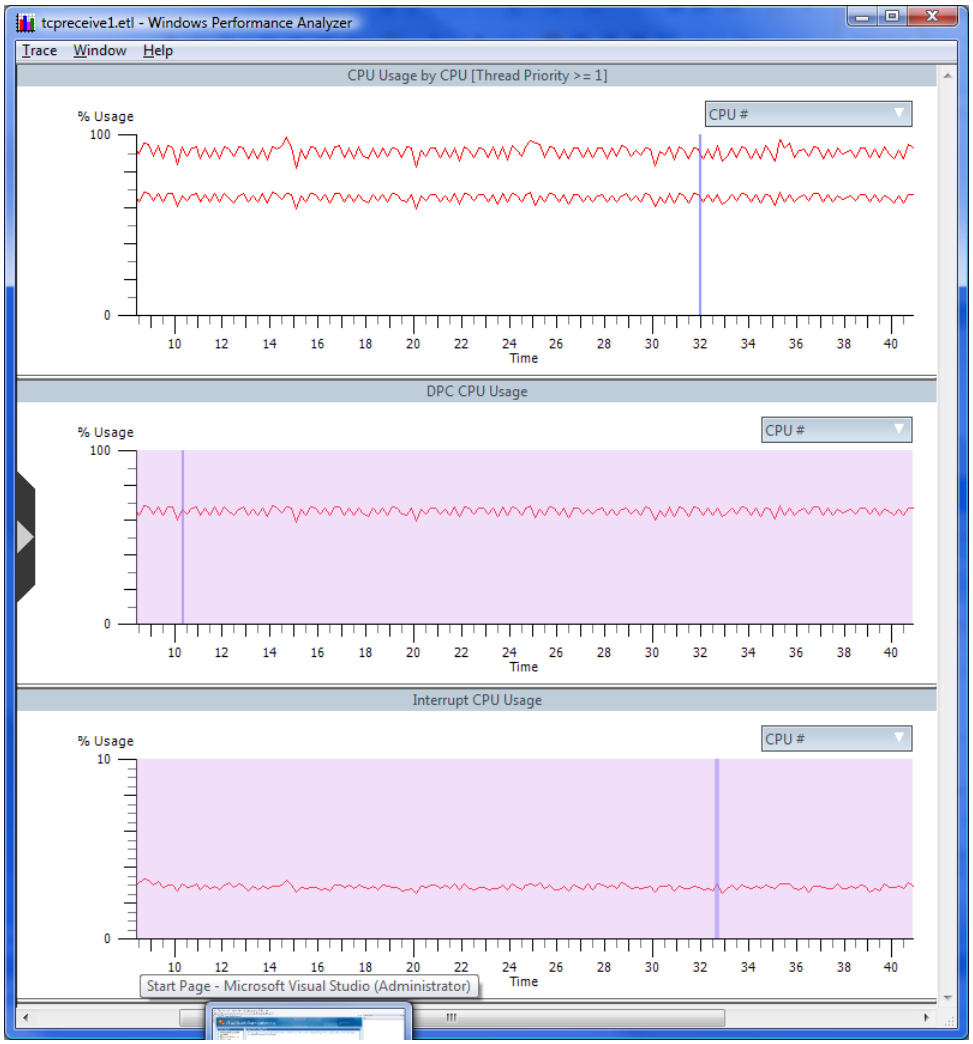

Figure 2. xperview display showing % CPU utilization, % DPC time, and % Interrupt time, calculated from the kernel trace event data recorded during the NTttcp test execution.

slightly more than 100 µsecs of processing time for each Interrupt processed. This being a 2.2 GHz machine, 100 µsecs of processing time translates into 220,000 cycles of execution time per TCP/IP interrupt. Substituting this more realistic estimate of the CPU path length into equation 1 yields

*500,000 interrupts/sec * 200,000 cycles = 100,000,000,000 cycles/second* (2)

a requirement for 100 GHz of host processing power to perform the TCP/IP processing for a 10 Gb Ethernet card running at its full rated capacity.

Next, I re-executed the test while running the xperf ETW utility that is packaged with the Windows Performance Toolkit[3] to capture CPU consumption by the TCP/IP stack:

```
xperf -on  LATENCY -f tcpreceive1.etl –ClockType Cycle
```

According to the xperf documentation, the LATENCY flags request trace data that includes all CPU context switches, interrupts (Interrupt Service Routines or ISRs) and Deferred Procedure calls (DPCs). As explained in [1], Windows uses a two-step process to service device interrupts. Initially, the OS dispatches an ISR to service the specific device interrupt. During the ISR, further

interrupts by the device are disabled. Ideally, the ISR performs the minimum amount of processing time possible and then schedules a DPC to finish the job, which includes re-enabling the NIC for interrupts. DPCs are dispatched at a lower priority than ISRs, but above most other functions in the machine. DPCs execute with the CPU re-enabled for interrupts so it is possible for the execution time of the DPC to be delayed because it is preempted by the need to service a higher priority interrupt from this or another device.

Gathering the xperf data while the NTttcp test was running lowered the network throughput only slightly – by less than 2%, additional measurement noise that can safely be ignored in this context. The disk was otherwise not being used for anything else during this test and there was an idle CPU available to handle the tracing chores. The overall performance impact of gathering the trace data was minimally disruptive in this situation.

I then loaded the trace data from the .etl file and used the xperview GUI application to analyze it. See Figure 2.

Figure 2 shows three views of the activity on CPU 0 where all the networking processing was performed. The top view shows overall processor utilization at close to 100% during the TCP test, with an overlay of a second line graph indicating the portion specifically associated with DPC processing, accounting for somewhere in excess of 60% busy. The DPC data is broken out and displayed separately in the middle graph, and the Interrupt CPU time is shown at the bottom (a little less than 4%).

xperfview allows you to display a Summary Table that breaks out Interrupt and DPC processor utilization at the driver level, sorted by the amount of processor time spent per module. For the DPCs, we see the following.

Module	Function	Count	Max Duration [ms]	Avg Duration [ms]	Duration [ms]
ndis.sys		425125	1.116595	0.075376	32044.44967
	0x8ac79237	423800	0.797987	0.075552	32019.18583
	0x8ad38209	207	1.116595	0.11752	24.326752
	0x8ad3892f	1117	0.012439	0.000837	0.935867
	0x8ad399b3	1	0.001213	0.001213	0.001213
USBPORT.SYS		8312	0.064506	0.011802	98.100399
tcpip.sys		4154	0.551394	0.009585	39.817004
dxgkrnl.sys	0x8f34e09b	3039	0.528346	0.012848	39.047187
iastor.sys		1221	0.033546	0.015061	18.390545

Table 2. xperfview Summary Table display showing processor utilization by DPC.

Confirming my back-of-the-envelope calculation presented earlier, xperf trace data indicates the average duration of an ndis.sys DPC used to process a network interrupt was 75 μsecs. The total amount of time spent in DPC processing was approximately 32 seconds of the full trace, which lasted about 52 seconds, corresponding to slightly more than 61% busy on CPU 0.

Module	Function	Count	Max Duration [ms]	Avg Duration [ms]	Duration [ms]
ntkrnlpa.exe	0x828d6fa2	423803	0.023875	0.003173	1345.147547
dxgkrnl.sys	0x8f3630ea	3040	0.096759	0.039523	120.151742
USBPORT.SYS	0x8f4098c2	5529	0.025199	0.007241	40.038229
pcmcia.sys	0x82f8deea	4968	0.02401	0.006754	33.558272
iastor.sys	0x8aaa7f6c	4968	0.016345	0.005103	25.353482

Table 3. xperfview Summary Table display showing processor utilization by ISR.

The Summary Table display reproduced in Table 2 serves to confirm a direct relationship between the ndis DPC processing and the kernel mode interrupts processed by the ntkrnlpa ISR. The average duration of an ntkrnlpa ISR execution was just 3 μsecs. Together, the ISR+DPC time was just under 80 μsecs. This leads to a slight downward revision of equation 2:

$$500{,}000 \text{ interrupts/sec} * 175{,}000 \text{ cycles} = 88{,}000{,}000{,}000 \text{ cycles/second} \quad (3)$$

which remains a formidable constraint, considering the speed of current processors.

Finally, I drilled into the processor utilization by process, which showed utilization by the NTttcp process, whose main processing thread was also affinitized to CPU 0, at the receiving end of the interrupt responsible for an additional 11% CPU busy. Allowing for OS scheduling and other overhead factors, these three workloads account for the 100% utilization measured for CPU 0.

Process	Cpu Usage (ms)	% Cpu Usage
Idle (0)	32730.11165	50.4
NTttcpr.exe (3280)	7120.518872	10.97
services.exe (700)	582.050622	0.9
InoRT.exe (2076)	368.519663	0.57
dwm.exe (4428)	300.59808	0.46
System (4)	256.802505	0.4
svchost.exe (1100)	166.479679	0.26
taskmgr.exe (7092)	162.960941	0.25
msiexec.exe (6412)	145.122854	0.22
sidebar.exe (5884)	111.86257	0.17
WmiPrvSE.exe (7952)	93.797334	0.14
csrss.exe (668)	79.331035	0.12
svchost.exe (1852)	70.65799	0.11

Table 4. xperfview Summary Table display showing processor utilization by process.

Note that NTttcp's processing likely represents a minimum amount of application-oriented work per packet. It ignores the data payload contents completely, and its other processing of the packet is pretty much confined to maintaining its throughput statistics. We should also note that is a user mode application, which means that processing the receive packet does require a transition from kernel to user mode. It is possible to implement a kernel mode networking application in Windows using the Winsock Kernel interface – the http.sys kernel model driver that IIS uses is an example – that avoids these expensive processor execution state transitions, but they are the exception, not the rule. (And, when it comes to building HTTP Response messages dynamically using ASP.NET, even http.sys hands off the HTTP Request packet to an ASP.NET User mode thread for processing.)

The point of this set of measurements and calculations is not to characterize the network traffic in and out of a typical web server, but to understand the motivation for recent architectural changes in the networking stack – both hardware and software – to allow network interrupts to be processed concurrently on multiple processors. Those architectural changes are the subject of Part II of this paper.

Part II. Strategies to scale high-speed networking.

In Part 1 of this paper, we looked at the capacity issues that are driving architectural changes in the TCP/IP networking stack. While network interfaces are increasing in throughput capacity, processor speeds in the multi-core era are not keeping pace. Meanwhile, the TCP/IP protocol has grown in complexity so that host processing requirements are increasing, too. The only way for networked computers to scale in the multi-core era is to begin distributing networking I/O operations across multiple processors. Since bigger server machines rely on NUMA architectures for scalability, high speed networking is also evolving to exploit machines with NUMA architectures in an optimal fashion.

Programming ccNUMA machines.

Machines with NUMA (non-uniform memory access speeds) architectures are usually large scale multiprocessors that are assembled using building blocks, or *nodes*, that each contain some number of CPUs, some amount of RAM, and various other peripheral connections. Nodes are often configured on separate boards, for example, or specific segments of a board. Multiple nodes are then interconnected with high speed links of some sort that permit all the memory that is configured to be available to executing programs. There are many schools of thought on what the best interconnection technology is. Some manufacturers favor tree structures, some favor directory schemes, some favor network-like routing. A key feature of the architecture is that the latency of a memory fetch depends on the physical location of the RAM being accessed. Accessing RAM attached to the local node is faster than a memory fetch to a remote location that is physically located on another node.

Within one of the new Intel Nehalem many-core microprocessor, all the processor cores and their logical processors can access local memory at a uniform speed. Figure 3 is a schematic diagram depicting a 4-way Nehalem multiprocessor chip that is connected to a bank of RAM. The configuration of processors and RAM shown in Figure 4 is a building block that is used in

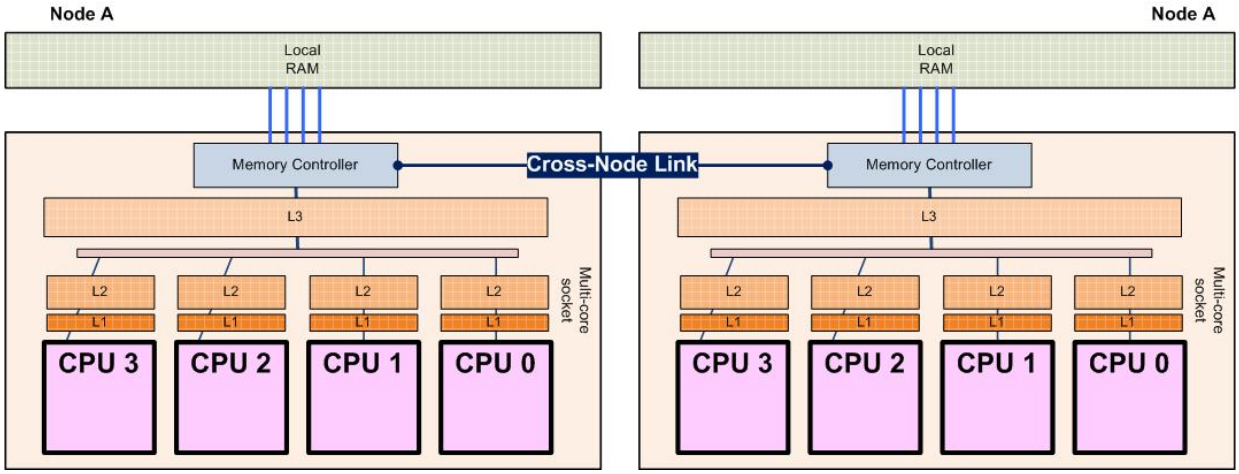

Figure 3. A two-NUMA server showing a cross-node link that is used when a thread on one node needs to access a remote memory location.

creating a larger scale machine by connecting two or more of such nodes together.

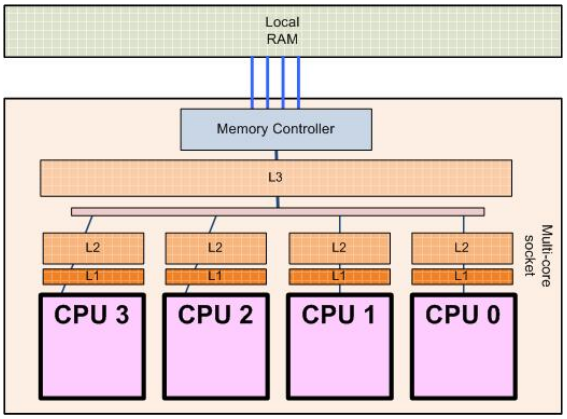

Figure 4. A schematic diagram depicting a NUMA node showing locally-attached RAM and a multi-core socket.

A two-node NUMA server is illustrated in Figure 3, which shows a direct connection between the memory controller on node A and the memory controller on node B. This is the relatively simple case. A thread executing on node A can access any RAM location on either node, but an access to a local memory address is considerably faster. The latency to access to a remote memory location is several times slower. (Definitive timings are not available as of this writing because early versions of the hardware are just starting to become available.)

As the number of nodes increases, it is no longer feasible for every node to be directly connected to every other node, nor can each bank of RAM that is installed be accessed in a single hop. The specific technology used to link nodes may introduce additional variation in the cost of accessing remote memory. From any one node, it could take longer to access memory on some nodes than others. For instance, some nodes may be accessed in a single hop across a direct link, while other accesses may require multiple hops. Some manufacturers favor routing through a shared directory service, for example. Your mileage may vary.

Specifically, in the Intel architecture, manufacturers are supplying a cache coherent flavor of NUMA servers (ccNUMA). Cache coherence is implemented using a snooping protocol to ensure that threads executing on each NUMA node have access to the most current copy of the contents of the distributed memory. Details of the snooping protocol used in Intel ccNUMA machines are discussed here[4].

AMD has taken a somewhat different tack in building its multi-core processors. For communication on chip between processors, AMD uses a technology known as HyperTransport, which is a dedicated, per-processor 2-way high speed link. Multiple

processors cores are then linked on the chip in a ring topology as depicted in Figure 5. The ring topology has the effect of scaling the bus bandwidth that is used as an interconnect linearly with the number of the processors. But the architecture leads to NUMA characteristics. A thread executing on CPU 0 can access a local memory location, a remote memory location that is local to CPU 1 at the cost of one hop across the HT link, or a remote memory location that is local to CPU 2 at the cost of two hops across HT links.

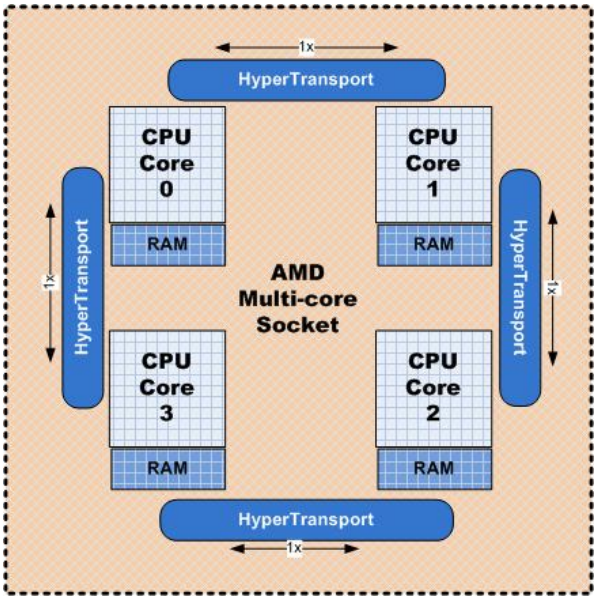

Figure 5. The AMD approach to multi-core processors has NUMA characteristics. A program executing on CPU 0 that accesses RAM that is local to CPU 2 requires two hops across the HyperTransport links that connect the processors in a ring.

Historically, application development for NUMA machines meant understanding the performance costs associated with accessing remote memory on a specific hardware platform. Since manufacturers employ different proprietary interconnection schemes in their multi-tiered NUMA machines, application developer are challenged to find the right balance in exploiting a specific proprietary architecture that may then limit the ability to port the application to a different platform in the future. It may be possible to connect nodes in a NUMA machine in an asymmetric configuration, for example, where the performance cost function associated with accessing different memory locations is decidedly irregular.

To scale well, a multi-threaded program running on a NUMA machine needs to be aware of the machine environment and understand which memory references are local to the node and which are remote. A thread that was running on one NUMA node that migrates to another node pays a heavy price every time it has to fetch results from remote memory locations. The difficulty that programmers face when trying to develop a scalable, multi-threaded application for a NUMA architecture machine is to comprehend its memory usage pattern and how it maps to the NUMA topography. When NUMA considerations were confined to expensive, high-end supercomputers, the inherent complexities developers faced in programming them were considered relatively esoteric concerns. However, in the era of many-core processors, NUMA is poised to become a mainstream architecture.

In theory, it is possible to craft an optimal solution when threads and the memory they access are *balanced* across NUMA processing nodes. In order to achieve an optimal balancing of the machines resources without overloading any of them, programs need to understand the CPU and memory resources that individual tasks executing in parallel require and understand how to best map those resources to the topography of the machine. Then they require a suitable scheduling mechanism to achieve the desired result. Achieving an optimal balance, as a practical matter, is not easy, in the face of variability in the resources required by any of execution threads, a complication that may then require dynamic adjustments to the scheduling policy in effect.

The Windows OS is already NUMA-aware to a degree and, thus, supports a NUMA programming model. For example, once dispatched, threads have node affinity and tend to stay dispatched on an available processor within a node. Windows OS memory management is also NUMA-aware, maintaining per node allocation pools. The OS not only resists migrating threads to another node, it also tries to ensure that most memory allocations are satisfied locally using per node memory management data structures.

Windows also provides a number of NUMA-oriented APIs that applications can use to keep their threads from migrating off-node and also enable them to direct memory allocations to a specific physical processing node. For more information on the NUMA support in Windows, see the MSDN Help topic "NUMA Support."[5]

To help application developers deal better with the complexities of NUMA architectures in the future, the Windows NUMA support needs to evolve. One potential approach would be for the OS to attempt to calculate a performance cost function at start-up that it would then expose to driver and application programs when they start up and run. Conceivably, the OS might also need to adjust this performance cost function to response to configuration changes that occur dynamically, such as any power management event that affects memory latency. These changes would then have to be communicated to NUMA-aware drivers and applications somehow so they could adapt to changing conditions.

Multiprocessing and the networking stack

By necessity, both the hardware and the software devoted to processing network traffic need to evolve in the many-core era to become multiprocessor-oriented. On servers that have NUMA architectures, that multiprocessing support needs to acquire a NUMA flavoring. The technology that allows network interrupts to be processed concurrently across multiple processors includes support for

- multiple Descriptor Queues in the networking hardware,
- Extended Message Signaled Interrupts (MSI-X) to allow hardware interrupts to be serviced concurrently on more than one processor, and
- the software support in Windows known as Receive-Side Scaling (or RSS).

With ccNUMA architecture machines becoming more mainstream, it is clear that multi-processor support should also include being NUMA-aware.

For an idea of how fast the NICs are getting, a typical 1 Gb Ethernet card supports 4 transmit and 4 receive interrupt queues and can spread interrupts across as many as 4 host processors for load balancing under RSS. A 10 Gb Ethernet card necessarily supports even high levels of parallelism. For example, the dual-ported Intel 82598 10 Gigabit Ethernet Controller provides 32 transmit queues and 64 receive queues per port, which can be mapped to a maximum of 16 processor cores. Note that this increase in parallel processing capacity is only a 4x improvement over recent 1 Gb Ethernet cards, which is probably inadequate to exploit the increased bandwidth fully.

Let's consider briefly some of the ideas to improve TCP/IP performance that have been implemented in the recent past. The strategies discussed here either increase the efficiency of host computer processing of TCP/IP packets or attempt to off load some of these software functions onto networking hardware. These strategies have proven effective, but they do not offer enough capacity relief to keep pace with the steady advance of networking speeds. One way out of the current mismatch between high speed networks and the host processing requirements they generate is a parallel processing approach where TCP/IP interrupts are distributed across multiple CPUs.

Interestingly, some of the changes outlined here to streamline TCP/IP host processing fly in the face of a major precedent. The layered architecture of the networking stack is widely regarded as one of the storied accomplishments of software engineering. Several of the efforts to improve the performance of host processing of TCP/IP interrupts involve shattering the strict isolation of components that the layered networking model advocates.

In theory, at least, the layered approach simplifies complex software that is designed to function smoothly in diverse environments. Layering also supports development of components that can proceed independently and in parallel. In principle, each layer defines and adheres to a standard set of services, or *interfaces*, that it provides to the component in the layer immediately above it. An upper layer communicates with a level below it using only this predefined set of abstract interfaces. (The set of services provided and consumed by two adjacent layers, in effect, defines a *contract*.) Furthermore, in the design of the networking protocol, components are isolated. It is an article of faith among software engineers that layered architectures, when properly defined and implemented, greatly contribute to the robustness and reliability of the software built using those design principles.

TCP/IP Layers			
Data unit	**Layer**	**Function**	**Example**
Data	5. Application	Application-specific	HTTP, SMTP, RPC, SOAP, etc.
Segment	4. Transport	End-to-end connections (sessions) and reliable delivery	TCP/ UDP

Packet/Datagram	3. Network	Logical addressing and routing; segmentation & re-assembly	IP
Frame	2. Data link	Physical addressing (MAC)	Ethernet, ATM
Bit	1. Physical	Media, signal and binary transmission	Optical fiber, coax, twisted pair

Table 4. The TCP/IP layered networking model.

Table 4 is a standard representation of the layered networking model used in TCP/IP, which has gained almost universal acceptance in computer-computer communications. Take the ubiquitous IP layer, for instance. IP implements a Best Effort service model to deliver packets from one station to another using routing. By design, it is connectionless, session-less and unreliable. Delivery of packets to the correct destination is not guaranteed, but IP does take a "best effort" approach to accomplish this. For the applications that require these services, the higher-level TCP Host protocol guarantees that packets are delivered reliably and in order to the designated application layer above it. It does this using a session-oriented protocol that preserves the state of the messaging-passing session between packets. TCP has also evolved complicated, performance-oriented flow and congestion control mechanisms that are beyond the scope of the current discussion.

The layering approach to networking introduces one additional and crucial design constraint. When one station is transferring a message to another, only components at the same level in the protocol stack can exchange data and communicate with each other. For example, only the TCP component in the receiver is supposed to be able to understand and process information placed into the TCP packet header by the sender. However, both the TCP Offload Engine and Receive-Side Scaling utilize knowledge of what is going on the upper layer TCP protocol down in the Data Link (or MAC) layer in the receiver, a serious violation of the principle that the layers in the protocol stack remain totally isolated from each other. Apparently, this is a case where the serious performance issues trump the pure design principles, and the evolution of TCP/IP has always been sensitive to practical issues of scaling. It is not that you absolutely cannot violate the contract that governs the ways layers communicate, but it is something that should be done very thoughtfully so that your once clean interfaces don't start to look like swiss cheese.

A crucial factor that works to encourage breaking with precedent is that the protocols from the TCP layer down to the hardware all adhere very strictly to the standards in order to promote interoperability. This strict compliance has the effect of hardening the services and interfaces between these layers in cement. This rigidity actually reduces the risk of side effects when a lower level component presumptuously usurps a service that architecturally is defined as the responsibility of some higher level.

Another factor is also at work. Within a layer, components that conform to the same contract layer can, in theory, be freely substituted for each other. This principle of the layered approach is supposed to promote development of a profusion of components that implement different sets of services, but still adhere to the strict requirements of the standard. In fact, the need for interoperability severely limits the proliferation of components that can be freely substituted for each other. Ethernet is almost always the hardware used at the bottom of the stack due to its superior cost/performance. Ethernet is always followed by IP, which is then usually followed by TCP. UDP can be freely substituted for TCP at the host processing layer, but only when the TCP services that ensure reliable delivery of packets and flow control can be dispensed with. In practice, there is very little variety among the components you will see operating in every networking protocol stack. TCP/IP over Ethernet, being ubiquitous, achieves the highest possible degree of interconnectivity and interoperability.

With the TCP/IP stack so pervasive and so stable and dominant, it then becomes possible to think the unthinkable. It becomes difficult to resist the temptation to violate the principle of isolation if you can demonstrate a big enough performance win. Having the Ethernet layer peek into the TCP packet headers and optimize their processing is acceptable when the violation of this sacrosanct principle of layering yields sufficient performance or scalability improvements.

Recent performance improvements to TCP/IP host processing.

Before we drill into the current set of architectural changes to the networking stack, let's explore briefly some of the more successful strategies for reducing the host computer processing requirements associated with TCP/IP interrupt processing that have been explored in the past. Stateless processing associated with the IP layer were some of the earliest functions identified that could be performed on the NIC and reduce the amount of host processing needed. These offloaded functions include Checksum and segmentation for large Sends, both of which were supported in the Windows 2000 timeframe. Because the IP protocol is stateless and connectionless, there are virtually no side effects to performing these functions on the NIC, even if it does violate the principle of strict isolation between layers in the protocol stack.

Another set of performance improvements that have been implemented recently do potentially generate serious side effects that must be handled rather delicately. These include:

- interrupt moderation
- jumbo frames
- TCP Offload engine (TOE)

We will consider into these three approaches next, discussing some of the potential side effects, performance trade-offs, and other issues they raise. It is probably also worthwhile to mention netDMA, which is the Windows support for Intel's I/O Acceleration Technology (I/OAT), in this context. I/OAT makes targeted improvements in the processor memory architecture to improve the efficiency of NIC-to-memory transfers.

Each of these approaches has worked to a degree, but none has produced enough of a breakthrough in performance to address the underlying condition, the growing mismatch between host processing requirements and network bandwidth.

As noted earlier, the CPU load associated with the processing Ethernet packets with TCP/IP at a server is a long-standing and persistent performance problem that has escaped a satisfactory solution in the past. For many years, the thrust of conventional solutions to the problem was straightforward – namely, any means possible for reducing the number of interrupts that the host computer needs to process. Two of the more effective approaches to reducing the number of interrupts are to use some form of *interrupt moderation* or so-called *jumbo frames*, basically larger packets than the Ethernet standard supports. Both approaches are effective to a degree, but also have serious built-in limitations and drawbacks.

Interrupt moderation on the NIC is widely used today to reduce the host interrupt processing rate. It is successful, but only to the extent of addressing the processing load associated with each interrupt, which, as indicated in the protocol overhead measurements discussed earlier, is relatively minor. A NIC that supports interrupt moderation can delay the host interrupt for up to a specified amount of time with the hope that the NIC will receive additional networks packets to process during the delay. Then, instead of each packet causing an interrupt, the host processor can process multiple packets in a single interrupt. In the measurements reported in Part 1, interrupt moderation was used to cut the host processor interrupt rate in half. When you consider as we have earlier, the potential rate of networks interrupts that a 10Gb Ethernet card can drive, some form of interrupt moderation on the NIC becomes essential for the smooth operation of the host processor.

Interrupt moderation helps, but not enough to relieve the bottleneck at the host CPU. The host processing associated with the TCP/IP protocol appears to scale as a function of *both* the number of interrupts and the amount of data being transferred between the NIC and the host computer. As the average size of data payloads increases, the processing bottleneck shifts to memory latency. See, for example, the bottleneck analysis presented in an Intel white paper entitled "Accelerating High-Speed Networking with Intel® I/O Acceleration Technology." [6]

Interrupt moderation should be used cautiously in situations where the fastest possible network latency is required, such as two communicating infrastructure servers connected to the same high-speed networking backbone. It also has to be implemented carefully to ensure it does not interfere with the TCP congestion control functions that try to measure round trip time (RTT).

Jumbo frames. Sending data across the wire in so-called *jumbo frames* also significantly reduces the number of host interrupts. And there is little question that the size of the Ethernet MTU is sub-optimal for many networking transmission workloads. Consider the relatively large data payloads that routinely need to be transferred between a back-end database machine and the clusters of front-end and middle tier machines in a typical clustered, multi-tier web service application today. Using jumbo frames of, say, 9K payloads on the high speed network backbone linking these servers leads to a 6:1 reduction in the number of host processor interrupts required to transfer sizable blocks of data. When servers are connected to a Storage Area Network (SAN) using iSCSI, even larger frames are desirable.

In fact, jumbo frames appears to be such a simple, effective solution within the confines of the data center that it naturally leads to consideration of what other aspects of the TCP/IP protocol that are sub-optimal in that environment could also be modified. For example, when there is frequent high speed communication between very reliable components, the TCP/IP requirement to acknowledge positively the receipt of every packet is overkill, and it very tempting to break with the standard and relax that requirement. The superior cost/performance of high speed Ethernet-based networking makes it very tempting to consider as an alternative interconnect technology to use with both SANs and High Performance Computing (HPC) clusters. In both these cases there are alternatives linkage technologies that outperform TCP/IP that are also considerably more expensive. For a further discussion of this issue in the context of SAN performance, see [7]. And for the HPC flavor of this same discussion, see Jeffrey Mogul's "TCP offload is a dumb idea whose time has come." [8]

Unfortunately, using non-standard jumbo frames introduces a significant compatibility problem that severely limits the effectiveness of the solution. The great majority of network clients will reject frames larger than the standard Ethernet MTU of 1500 bytes. In effect, you can send jumbo frames between specific host computers that are equipped to handle them on a dedicated backbone segment readily enough, but you cannot reliably send them to just any machine connected using the IP internetworking layer. So implementing jumbo frames requires more complicated routing schemes. TCP/IP RFC 2923 section

2.1, which is supported in Windows XP SP3, Vista, and Windows Server 2008, allows two TCP peers to negotiate the largest size MTU that can be transmitted between them. But the connectionless and stateless IP routing mechanism means that no single packet transmitted between station A and B need follow the same route twice. Given that the precise route to the destination station is dynamically constructed for each packet, any intermediate router that did not support jumbo frames would reject any non-standard packets it received and prevent successful transmission to the receiver.

TCP Offload Engine. A TCP Offload Engine (or TOE) is another solution that has been implemented to reduce the host processing required for TCP/IP interrupts. As the name suggests, in this approach, certain TCP/IP protocol functions are performed directly on the NIC, either reducing the amount of processing that must be performed in the host machine, or eliminating host interrupts associated with certain TCP/IP housekeeping operations entirely. Areas where significant performance gains are experienced with TOE include the elimination of expensive memory copy operations, offloading segmentation and reassembly (a function of the IP layer), and offloading some of the TCP housekeeping functions that ensure reliable connections (mainly, ACK processing and TCP retransmission timers). Moving these functions onto the NIC results in a reduction of the total number of interrupts that need to be processed by the host machine. Potential performance benefits associated with TOE, which are considerable, are quantified in [9].

TCP Offload Engine, however, is a grievous violation of the layered architecture of the networking protocol stack. The TCP Chimney Offload feature that provides TOE support in Windows, for example, required an extensive re-architecture of the TCP/IP stack. TOE introduces many breaking changes. See the KnowledgeBase article 951037 [10] entitled "Information about the TCP Chimney Offload feature in Windows Server 2008" detailing the many limitations, reflecting what networking functions can & can't safely be offloaded in which computing environments. For instance, any networked machine that enforces an IPsec-based security policy where it is necessary to inspect each individual packet cannot use TOE. Neither is TOE currently compatible with either server virtualization technology or common forms of clustering based on virtual IP addresses.

The modest benefits in many environments and the complexities introduced due to explicit violations of the layered model of the network protocol argue against a general TOE solution. Another strong criticism of the TOE approach is that it merely moves the bottleneck from the host processor to the NIC. As RSS penetrates the market for high-speed networking, I believe that interest in the TOE approach will wane. If you do have a processing bottleneck on the host machine as a result of high-speed networking, with an RSS solution, at least, the bottleneck is visible, and there are inexpensive mechanisms to help deal with it. A processing bottleneck on the NIC is opaque and resists any capacity solution other than to swap in a more expensive card, assuming one exists, and hope that the new one is significantly faster and more powerful than the old one.

Intel I/OAT. Intel's I/OAT introduces memory architecture improvements that give the NIC access to a dedicated DMA (direct memory access) engine for copying data between host memory and the NIC. These architectural changes are known as the Intel QuickData Technology DMA subsystem. With both interrupt moderation and the TCP Offload of IP segmentation and re-assembly, the processor tends to receive fewer interrupts to process, but each interrupt results in larger amounts of data that needs to be processed by the host. The networking protocol stack services the initial interrupt from the NIC and examines the Receive data block while it is running in kernel mode. Most data blocks associated with networking I/O subsequently need to be copied into the networking application's private address space. A performance analysis showed that, especially with larger blocks of data, this second memory-to-memory copy operation was responsible for a very large portion of the host processor load. An Intel white paper describes this analysis in some detail. [6]

Ultimately, the result of this performance analysis was the set of I/OAT architectural improvements that permit this second memory-to-memory operation to be performed by a DMA provider engine (located on the Northbridge chip set currently) that requires no additional host processor bandwidth. The memory copy operation occupies the memory controller, but does not consume Front Side Bus bandwidth, which also frees up the host processor to perform other CPU tasks. Interestingly, Windows support for this technology, described in the Driver Development Kit (DDK) documentation [11], is actually very general, but to date the only netDMA client available is the tcpip.sys kernel mode driver that processes networking interrupts. It ought to be possible for disk I/O controllers to also exploit I/OAT architectural improvements sometime in the future. However, data blocks associated with disk I/O, which are cached by default in the system address space in Windows, are not necessarily subject to multiple copy operations, depending on the cache interface used.

Parallelizing TCP/IP

In the many-core era, the host processor overhead associated with processing TCP/IP interrupts is not a capacity problem, since CPU cycles on the host computer are plentiful and becoming more plentiful all the time. The problem is that the individual processors themselves are not fast enough, nor are they growing much faster. To craft a solution that works in the many-core era, there is a clear need to enhance the hardware and software in the TCP/IP protocol stack to run in parallel across multiple processors and take advantage of the available capacity. There are two hardware and software technologies that are associated with that capability today:

- Extended Message-Signaled Interrupts (MSI-X): a hardware technology that allows the NIC to support multiple interrupt vectors, enabling multiple processor cores to handle interrupts from the NIC simultaneously.

- Receive-Side Scaling (RSS): the protocol used in the NDIS driver software to manage multiple interrupt vectors and communicate to the hardware to ensure that session-oriented TCP packets are delivered in sequence to a processor-specific interrupt queue.

MSI-X and RSS work together to allow the processing of TCP/IP Receive packets to scale in parallel across multiple processor cores

Message Signaled Interrupts (MSI-X). MSI-X is an architectural change that allows a device to send interrupts to be processed on multiple CPUs. Historically, on the Intel architecture, devices were limited to sending interrupts to a single target.

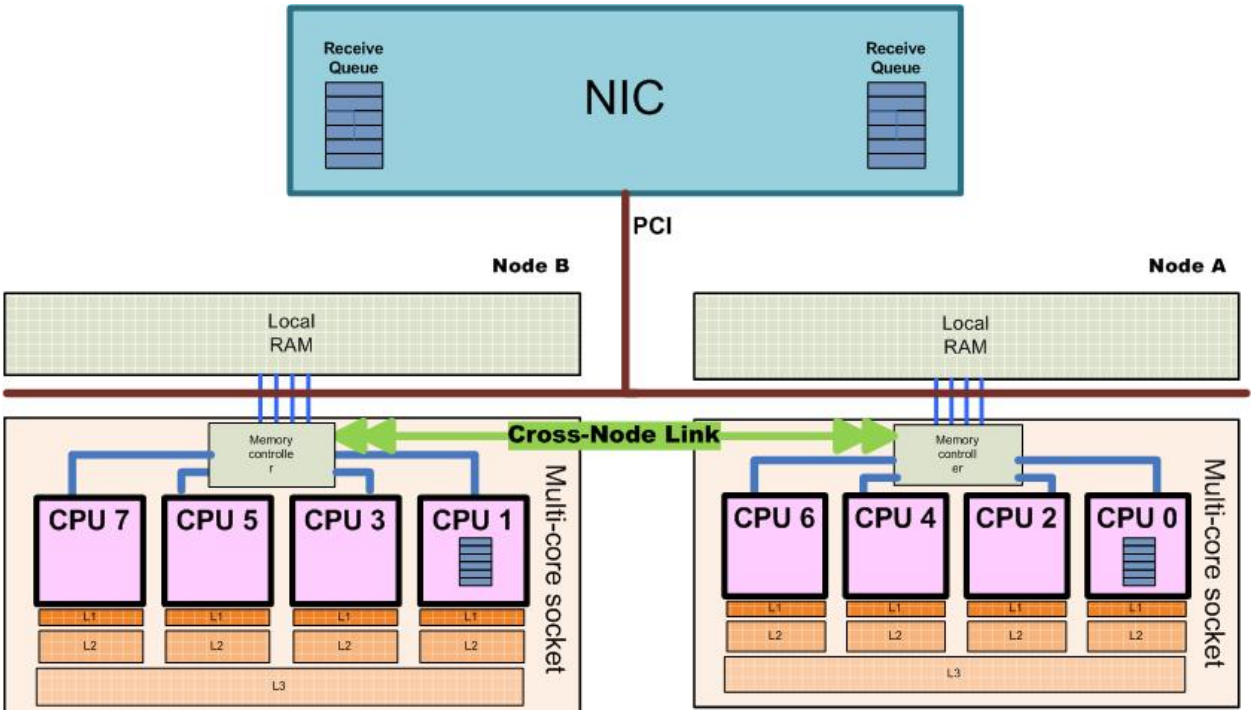

Figure 6. Two NUMA nodes in a Windows Server machine configured to use MSI-X and RSS to process TCP/IP Receive packets across multiple processors.

Concentrating all hardware interrupts on a single processor boosts the instruction execution rate of the Interrupt Service Routine (ISR) by increasing the chances of a processor cache warm start. In the many-core era, limiting the device to one processor that it can interrupt is a severe capacity constraint. MSI-X capabilities allow the NICs to scale on many-core processors.

One key feature of Windows' support for MSI-X devices is the ability to specify a policy that automatically assigns MSI-X interrupts to CPUs based on the OS's understanding of the underlying NUMA topology of the machine. An NDIS-driver that supports MSI-X devices can specify an *IrqPolicySpreadMessagesAcrossAllProcessors* policy that automatically distributes interrupts across an optimal set of eligible processors. On some NUMA machines, the performance of the device connection is affected by the underlying topology of the multi-node connections. For instance, certain device-to-processor node connections may be low latency local ones, while others are higher latency remote connections. For performance reasons, you want NIC interrupts to be processed on nodes that are connected locally and access local memory on that node exclusively. For optimal scalability, you then want to balance device interrupts across all the NUMA nodes that are interconnected. The *IrqPolicySpreadMessagesAcrossAllProcessors* policy understands these performance considerations, and distributes the device interrupts to the right set of processors automatically.

Figure 6 illustrates one way the *IrqPolicySpreadMessagesAcrossAllProcessors* policy could be used to distribute interrupts from the NIC across nodes in a simple NUMA machine. A server with two quad-core sockets is shown, with each socket connected to a block of local RAM. Memory accesses from a processor core to local RAM are considerably faster than an access to remote memory attached via a bridge to the other multi-core socket. An optimal configuration is to process TCP/IP interrupts on CPU 0 on the first node and on CPU 1 on the second node, as depicted, balancing the networking I/O load across nodes.

While Receive-Side Scaling (RSS) does not require MSI-X, the two technologies normally go hand-in-hand. We restrict the RSS discussion here to the manner in which MSI-X devices are supported, which is both the simplest and most common case.

Receive-Side Scaling (RSS). RSS complements the Windows support for MSI-X. It allows the workload associated with processing network interrupts to be spread across multiple CPUs. With RSS, the DPC routine that we have seen is responsible for

performing the bulk of the host processing is also scheduled to run on the same CPU where the interrupt service routine (ISR) just ran. Concentrating all the work associated with network interrupt processing on the same CPU improves instruction execution rates because data associated with the packet is likely to remain in the processor caches. It also dramatically reduces delays spent in unproductive spin lock code associated with serialization. Optimistic, non-blocking per processor locking strategies are effective under these circumstances. By default under RSS, even the Send processing associated with an ACK message is also processed on the same CPU where the Receive was processed to take advantage of the same performance considerations.

There is one complication, however, that arises when network interrupts are distributed across multiple CPUs that RSS is forced to address. If packets are distributed randomly across multiple CPUs, this can conflict with the important function of the TCP protocol that guarantees delivery of data in sequence to the application. Suppose packets for a group of TCP connections are processed across two CPUs and one CPU in the bunch is lightly loaded while the other is heavily loaded. Older packets received on the lightly loaded CPU could easily be processed first. Receiving packets out of order in TCP triggers Fast Retransmits, for example, that could degrade both the network and delay the application, not to mention serialization delays before TCP can safely notify the application layer that Request data is available for processing.

Given this complication, RSS distributes connections, not individual packets. RSS has a mechanism that sends all the packets associated with any one TCP connection to the same processor. This preserves the order of delivery of received data packets, which avoids needless requests for TCP retransmits. Crucially, the processor associated with the specific connection must be communicated to the NIC, which must arrange Received packets into the correct message queues accordingly, prior to signaling the host processor by raising an interrupt. This coordination, of course, is another violation of the isolation principle of the layered networking stack. It is worth noting that nasty side effects can arise as a result of this willful violation of the layered networking architecture; see, for example, KB927168 [12] documenting a conflict between RSS and Internet Connection Sharing on Vista that was later fixed in WS2008 and Vista SP1.

To achieve good performance, however, it is absolutely necessary for the NIC to schedule all the packets for the same TCP session to same host processor. It can only do that by peeking into the TCP header and finding the port indicator, which it then uses to calculate the right CPU to deliver the packet to. This calculation is based on a hash table that is passed to the NIC by the NDIS driver software. RSS even includes a capability to adjust the load across the CPUs that are enabled for processing NIC interrupts dynamically. The protocol stack in Windows can re-balance the interrupt load by modifying the hashing table passed to the NIC that is used in determining the proper CPU. This mechanism can be used in case some CPUs remain overloaded for an extended period of time because, for example, some TCP connections are more chatty and persistent than others.

Speaking of maintaining a balanced system, long-running tasks such as large file copies associated with a single ftp, SMB or media server session present inherent difficulties under RSS. The general problem is that the throughput of any one session is ultimately limited by host processor speed. With many-core processors, it is important to figure out how to use parallel data divide-and-conquer techniques to break long serial operations into smaller sub-tasks that can be executed concurrently. Providing the capability to spread long, data-intensive operations across multiple TCP sessions is one possible approach.

For further technical details on RSS, see [13]. One interesting aspect of the RSS specification is that the DPC, not the ISR, is responsible for re-enabling the processor for more interrupts from the NIC. This prevents the NIC from sending any more Receive packets to the processor until the previous set has been completely processed. This effectively acts as both a serialization mechanism and a form of interrupt moderation that adaptively adjusts the delay between interrupts based on the specific processing load at the CPU.

Part 3. Summary and Conclusions.

For many years, the effort to improve network performance on Windows and other platforms focused on reducing the host processing requirements associated with the need to service frequent interrupts from the NIC. In the many-core era where the clock speeds of processors are constrained by power considerations, this strategy is inadequate to the growing host processing requirements that accompany high-speed networking. It is necessary to augment technologies like interrupt moderation and TCP Offload Engine that improve the efficiency of network I/O with an approach that allows TCP/IP Receive packets to be processed in parallel across multiple CPUs. Together, MSI-X and RSS are technologies that enable host processing of TCP/IP packets to scale in the many-core world, albeit not without some compromises with the prevailing model of networking using isolated, layered components.

Using MSI-X and RSS, for example, the Intel 82598 10 Gigabit Ethernet Controller mentioned at the start of Part 2 can be mapped to a maximum of 16 processor cores that could then be devoted to networking I/O interrupt handling. Capacity-wise, this is still not sufficient processing capacity to handle the theoretical maximum load equation 3 predicts for a 10 Gb Ethernet card, but it does represent a substantial scalability improvement.

With this understanding of what MSI-X and RSS accomplishes, let's return for a moment to our NUMA server machine shown in Figure 6. With MSI-X and Receive-Side Scaling, CPU 0 on node A and CPU 1 on node B are both enabled for processing network

interrupts. Since RSS schedules the NDIS DPC to run on the same processor as the ISR, even at moderate networking loads, CPU 0 and 1 for all practical purposes become dedicated to the processing of high priority networking interrupts.

Numerous economies of scale accrue using this approach. The same RSS process that sends all Receive packets from a single TCP connection to a specific CPU for processing improves the efficiency of that processing. The instruction execution rate of the TCP/IP protocol stack is enhanced significantly through this scheduling mechanism that enforces localization. Ultimately, TCP/IP application data buffers need to be allocated from local node memory and processed by threads confined to that node. Recently used data and instructions that networking ISRs and DPCs issue tend to reside in the dedicated cache (or caches) associated with the processor devoted to network I/O. Or, at the very least, they migrate to the last level cache that is shared by all the processors on the same NUMA node.

Ultimately, of course, the TCP layer hands data from the network I/O to an application layer that is ready to receive and process it. The implications of RSS for the application threads that process TCP receive packets and build responses for TCP/IP to send back to network clients ought to be obvious, but I will spell them out anyway. For optimal performance, these application processing threads also need to be directed to run on the same NUMA node where the TCP Receive packet was processed. This localization of the application's threads should, of course, be subject to other load balancing considerations to prevent the ideal node from becoming severely over-committed while other CPUs on other nodes are idling or under-utilized. The performance penalty for an application thread that must run on a different node than the one that processed the original TCP/IP Receive packet is considerable because it must access the data payload of the request remotely. Networked applications need to understand these performance and capacity considerations and schedule their threads accordingly to balance the work across NUMA nodes optimally.

Consider the ASP.NET application threads that process incoming HTTP Requests and generate HTTP Response messages. If the HTTP Request packet is processed by CPU 0 on node A in a NUMA machine, the Request packet payload is allocated in node A local memory. The ASP.NET application thread running in User mode that processes that incoming HTTP Request will run much more efficiently if it is scheduled to run on one of the other processors on node A, where it can access the payload and build the Response message using local node memory.

There is currently no mechanism in Windows today for kernel mode drivers like ndis.sys and http.sys to communicate to the application layers above them and specify the NUMA node on which that packet was originally processed. Communicating that information to the application layer is another grievous violation of the principle of isolation in the network protocol stack, but it is a necessary step to improve the performance of networking applications in the many-core era where even moderately sized server machines have NUMA characteristics.

References.

[1] Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." Dr. Dobb's Journal, March 1, 2005. http://www.ddj.com/architect/184405990

[2] NTttcp performance testing tool: http://www.microsoft.com/whdc/device/network/TCP_tool.mspx

[3] Windows Performance Toolkit (WPT, aka xperf): http://msdn.microsoft.com/en-us/library/cc305218.aspx

[4] David Kanter, "The Common System Interface: Intel's Future Interconnect," http://www.realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT082807020032&mode=print

[5] Windows NUMA support: http://msdn.microsoft.com/en-us/library/aa363804.aspx

[6] Intel white paper: Accelerating High-Speed Networking with Intel® I/O Acceleration Technology

[7] Mark B. Friedman, "An Introduction to SAN Capacity Planning," *Proceedings*, Computer Measurement Group, Dec. 2001.

[8] Jeffrey Mogul's "TCP offload is a dumb idea whose time has come," *Proceedings* of the 9th conference on Hot Topics in Operating Systems - Volume 9, 2003. http://portal.acm.org/citation.cfm?id=1251059&dl=ACM&coll=portal&CFID=71988909&CFTOKEN=98964748

[9] Dell Computer Corporation, "Boosting Data Transfer with TCP Offload Engine Technology."

[10] Microsoft Corporation, KB 951037, http://support.microsoft.com/kb/951037

[11] Microsoft Corporation, Windows Driver Development Kit (DDK) documentation, http://msdn.microsoft.com/en-us/library/cc264906.aspx

[12] Microsoft Corporation, KB 927168, http://support.microsoft.com/kb/927168

[13] Microsoft Corporation, NDIS 6.0 Receive-Side Scaling documentation, http://msdn.microsoft.com/en-us/library/ms795609.aspx