Characterizing the Impact of End-System Affinities On the End-to-End Performance of High-Speed Flows

Nathan Hanford[1], Vishal Ahuja[1], Mehmet Balman[2], Matthew K. Farrens[1], Dipak Ghosal[1], Eric Pouyoul[2] and Brian Tierney[2]

[1] Department of Computer Science, University of California, Davis, CA,
{nhanford, vahuja, mkfarrens,dghosal}@ucdavis.edu

[2] ESnet, Lawrence Berkeley Laboratory, Berkeley, CA,
{mbalman,epouyoul,bltierney}@lbl.gov

ABSTRACT

Multi-core end-systems use Receive Side Scaling (RSS) to parallelize protocol processing. RSS uses a hash function on the standard flow descriptors and an indirection table to assign incoming packets to receive queues which are pinned to specific cores. This ensures flow affinity in that the interrupt processing of all packets belonging to a specific flow is processed by the same core. A key limitation of standard RSS is that it does not consider the application process that consumes the incoming data in determining the flow affinity. In this paper, we carry out a detailed experimental analysis of the performance impact of the application affinity in a 40 Gbps testbed network with a dual hexa-core end-system. We show, contrary to conventional wisdom, that when the application process and the flow are affinitized to the same core, the performance (measured in terms of end-to-end TCP throughput) is significantly lower than the line rate. Near line rate performance is observed when the flow and the application process are affinitized to different cores belonging to the same socket. Furthermore, affinitizing the application and the flow to cores on different sockets results in significantly lower throughput than the line rate. These results arise due to the memory bottleneck, which is demonstrated using preliminary correlational data on the cache hit rate in the core that services the application process.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.4 [**Distributed Systems**]: Client/server; C.2.5 [**Local and Wide-Area Networks**]: Internet (e.g., TCP/IP); C.2.m [**Miscellaneous**]: Network performance analysis

NDM'13 November 17, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2534695.2534697

Keywords

40 Gbps network, ESnet, multi-core affinization, end-system performance, high-speed network, flow affinity, application affinity, RPS, RFS

1. INTRODUCTION

The network speed is continuing to grow; 10 Gbps Network Interface Cards (NICs) are common now, 40 Gbps NICs are available and a standard for 100 Gbps has already been approved. However, for a variety of reasons CPU clock frequencies have plateaued, and major technological breakthroughs will be required for them to get much above 3.5 GHz. Because of this widening gap between the link speed and the CPU clock frequency, it is becoming increasingly difficult for a single core to keep up with the high speed link rates (using current protocols and end-system architectures).

Computer architects have compensated for the fact that CPU clock frequencies are no longer increasing by putting multiple processor cores on each die. To exploit the availability of these cores, NICs are now equipped with multi-queue support, which enables parallelization of network processing across the cores. Unfortunately, the use of multi-queue NICs is not a magic bullet. For example, they do not provide any benefit to a single (TCP or UDP) high speed flow, because the kernel avoids distributing the packet processing of a single flow across multiple cores (since packet reordering is a very expensive operation).

Research shows [1] that the benefit of having multiple cores is nullified if the end-to-end flows do not take into account various affinities, particularly between the process that performs protocol processing of a packet and the application process that consumes the packet. For example, instead of using cores arbitrarily, it is better to have the core that is doing protocol/interrupt processing share the highest level cache with the core doing the application processing. This allows for fewer context switches and improved cache behavior.

In this paper, we carry out a detailed analysis of the performance impact of application affinity in multi-core end-systems. The experiments were performed on the 40 Gbps

ESNet testbed [19] using netperf with the *sendfile* option to ensure that the sender is not the bottleneck. Two commodity x86 end-systems assembled from "off the shelf" components were connected via dedicated circuits with a round trip delay of 95 ms. We show, contrary to conventional wisdom, that when the application process and the flow are affinitized to (run on) the same core, the performance (measured in terms of end-to-end TCP throughput) is significantly lower than the line rate; in the worst case the achieved throughput is approximately half (about 22 Gbps). When the flow and the application are affinitized to different cores within the same socket, we can achieve approximately 35 Gbps. Finally, affinitizing the application and the flow to cores on different sockets also results in significantly lower throughput than the line rate approximately 26 Gbps. We looked the L3-cache hit rate for the core that services the application process, and found that the differences in the cache hit rate to a large extent corroborate the throughput data.

The remainder of this paper is organized as follows. In Section 2, we review the related work. In Section 3, we review the various types of affinities that are important in processing a high-speed in a multi-core end-system. In Section 4 we describe the testbed on which the experiments were carried out. In Section 5, we present the results. Finally, in Section 6, we conclude with a discussion of the next steps.

2. RELATED WORK

Internet characterization using analytical modeling, simulation, and empirical measurements has received a lot of attention.Many researchers have modeled high speed traffic (such as video) on wireless networks [21, 8] or using dedicated custom interconnects in multicores [22]. But very few studies have investigated the impact of high speed traffic on end-hosts, especially when they are connected using dedicated networks. The receive livelock phenomena, wherein a machine spends all its cycles in hard and soft interrupt contexts, was first reported by Mogul et al [17]. They found that when a machine enters the receive livelock state, most of the packets are dropped, and no forward progress is made. As a consequence, NAPI [24] and NIC interrupt throttling are enabled by default in most of the kernels and NICs, respectively. A NAPI enabled kernel is helpful because it does not operate in a purely interrupt-driven mode - at low data rates, interrupts are enabled, while at high data rates the kernel switches to polling. Marian et al [16, 15] characterized packet loss and bandwidth degradation for end-systems connected over a lambda network, and in [2] it is shown that a lightly loaded 10 Gbps WAN can transform a slow and steady flow into a bursty flow. This means the receiving end-system has to cope with a bursty flow at 10 Gbps, which leads to packet losses in the end-system, and degraded performance.

The design focus for commodity end-systems is on CPU and memory intensive applications, since that is where the majority of the processor time is spent. In general, I/O in commodity end-systems does not get a lot of attention, and as a result these systems are inefficient for network processing. In [13], it was found that the receive side network processing for 10 Gbps Ethernet could easily overwhelm two cores of an Intel Xeon Quad Core processor.

Apart from this fundamental mismatch, the key barrier to running high speed networking applications on commodity multi-core systems is the memory stalls that are incurred during packet processing [15]. A lot of research has been performed related to trying to improve network I/O performance by mitigating the memory stalls [7, 11, 12, 14]. However, in [10] it was clearly demonstrated that such techniques do not offer any benefit for single flows, because the performance is CPU limited. In the same paper, it was also shown that cache affinity should be taken into account in order to benefit from these techniques. In particular, it was shown that if affinity is taken into account, a standard LINUX network stack runs 32% faster for various I/O sizes.

Receive-side Scaling (RSS) is designed to allow uniform performance increase. NICs apply a filter and send packets to different queues. That enables packets from each flow to be sent to separate receive queue [23]. As a results, packet processing is distributed among different CPUs. Receive Packet Steering (RPS) is essentially a simplified RSS, done in software (i.e., what the NIC does with the on-board RSS tables and interrupts and DMA queues is done in RPS using extra linked-lists in the host memory). Since it is in software, it is necessarily called later in the datapath - thus, while RSS selects the queue and hence the CPU that will run the hardware interrupt handler, RPS selects the CPU which will perform protocol processing above the interrupt handler.

Optimizations like RSS [20], RFS [5], and RPS [6] are not able to achieve high throughput even when using multiple parallel flows. What is needed is an approach which provides a more informed usage of cores within a multi-core system when doing network processing [1]. Cores should not be chosen arbitrarily, but rather cores should be chosen that share the lowest possible level of the cache structure[1]. For example, when a given core (e.g. core A) is selected to do the protocol/interrupt processing, the core that shares the L2 cache with core A should execute the corresponding user-level application. Doing so will lead to fewer context switches, improved cache performance, and ultimately higher overall throughput.

Irqbalance scatters interrupts across cores based on the load statistics. In one sense, Irqbalance is a variant of round-robin scheduling. Foong et al [4] and Narayanswamy et al [18] have analyzed the effect of processor affinity on networking performance of multicore systems. They demonstrated the limitation of Irqbalance and the benefit of RSS. In [9], a cache aware scheduling mechanism was proposed, but their focus was on processor utilization rather than performance. Their scheme works only for TCP and requires modification at the kernel level. In [25], a different design for the network stack has been proposed which offloads the network stack processing to a dedicated core. In this way, it eliminates the unnecessary sharing of network state among multiple cores.

With the advent of techniques like Direct Cache Access(DCA), processor affinity has become very important. Using the LINUX system call interface, a user-space process can affinitize itself to a certain CPU. However, the socket interface

[1]In this document we consider the L1 cache to be at a lower level (closer to the core) than the L2 cache, L2 lower than L3, etc.

does not provide any information regarding which CPU to choose. The default operation of the kernel scheduler is that it dynamically chooses the CPU on which the user-space process will run, and can potentially migrate the process to a different CPU altogether.

3. PACKET PROCESSING IN MULTI-CORE END-SYSTEMS AND RELATED AFFINITIES

A NIC performs the following tasks when a new packet is received [27]: 1) the symbol stream is read off the physical medium; 2) clock recovery is performed, and the data is unscrambled to decode the 64b/66b line encoding; 3) the Ethernet frames are assembled and placed in the on-board memory (used as a FIFO). An Ethernet frame is dropped if its frame check sequence (CRC32) is incorrect - otherwise, frames are transferred to the host memory using Direct Memory Access (DMA), and the corresponding packet descriptors are stored in the receive ring buffer. This transfer can fail if the system bus does not have the required bandwidth, or if the receiving end-system is not able to cope with the high incoming data rate.

If the DMA fails, the packet can remain in the on-board memory if there is room; if not, it is dropped. It is important to note that modern NICs are fast enough to receive packets at line rate, so when packets are lost it is because of some shortcoming of the receiving end-system. Packets which have been transferred to the memory are further processed by various protocol handlers (e.g. IP, TCP) in the kernel network stack. Using the 5-tuple flow identifier (destination port number, destination IP address, source port number, source IP address, and protocol), the user-space application is identified, and after en-queueing the packets into the corresponding socket buffer the user-space application is awakened. A user-space application is allowed only a limited amount of socket buffer space - when this limit is reached, incoming packets are dropped.

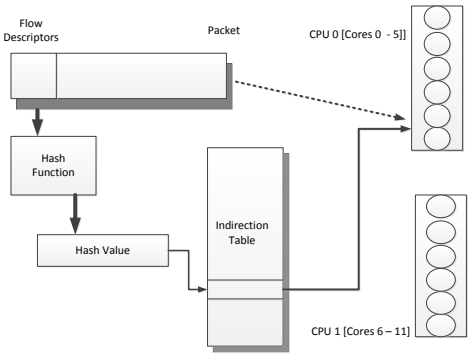

Figure 1: Review of RSS.

For multi-core end-systems, NICs are currently equipped with multi-queue support such as Receive Side Scaling (RSS) [20], which allows parallelization of network processing across multiple cores. Figure 1 shows the basic operation of RSS, which supports multiple receive queues and uses a hashing function and an indirection table to assign packets of the same data flow to a single queue. Additionally, Message Signal Interrupt (MSI/MSI-X) is used to assign a dedicated interrupt to a queue, and flow-pinning support is used tie an interrupt to a specific core.

The OS maintains, in the system memory, a ring buffer for each receive queue. The following steps are taken when a new packet arrives [28],

1. A hash function is applied to a flow identifier (typically, the 5 tuple consisting of the source and destination IP addresses, the source and destination port numbers, and the protocol). The hash result is used as an index to an indirection table to determine the receive queue and hence the core that will perform the protocol processing for the packet;
2. The NIC assigns the incoming packet to the corresponding receive queue;
3. The NIC DMAs received packets from a receive queue to the the corresponding ring buffer;
4. The NIC interrupts cores with nonempty queues. An interrupted core performs the protocol processing of the packets in the ring buffer.

3.1 Affinities

Based on the above description of the packet receive process, we can define following types of affinities.

1. **Flow Affinity:** The goal of flow affinity is to ensure that packets belonging to the same TCP/UDP flow are processed by the same core. Since TCP is a stateful protocol, distributing the packet processing among multiple cores has high overhead due to the need to access shared data in memory, and other synchronization overhead. Furthermore, since TCP guarantees inorder delivery, data reordering by the application process or by the kernel can also significantly tax the CPU resources.
2. **Interrupt Affinity:** The goal of interrupt affinity is to ensure that interrupts (network related or otherwise) of the same type should be processed by the same core. OS features such as IRQ balance, that redistribute interrupts (for example in round-robin fashion to distribute interrupt load across the cores), have poor performance and are often disabled.
3. **Application Affinity:** The goal of application affinity is to ensure that the application process that consumes the network data is bound to a particular core, usually based on load-balancing. Considering the modern CPU topologies, there are a number of choices where the user-space process may run with respect to the network stack: 1) Same core: The user-space process and the network stack process run on the same core. 2) Hyperthread: The user-space process and the network stack process run on peer hyperthreads of the same core. 3) Peer Core: The user-space process and network stack process run on different cores that share the last level cache(LLC) (basically, two cores on the

same chip). 4) Different Chip: The user-space process and network stack process run on cores that belong to different chips.

4. **Cache Affinity:** The goal of cache affinity is to ensure that instead of using cores arbitrarily, cores should share the lowest possible level cache: one core for protocol/interrupt processing, for example, and another for application processing. This allows for fewer context switches and improved cache behavior.

Considering only those interrupts that are related to packet processing, flow affinity and interrupt affinity imply the same thing. In this work we focus primarily on the relationship between flow affinity and application affinity.

As pointed out in many papers (including [1] and [28]), while RSS provides the ability to process receive packets belonging to different flows in parallel, it cannot guarantee that a flow will be directed to the same core (or some core with cache affinity) on which its application thread resides. This is because the NIC does not know the relationship between the flow and the application that consumes that data. Furthermore, in many NICs (such as the Mellanox NIC) the hash function is computed in the hardware and cannot be changed. The hash function operates on a four tuple which consists of the source ip address, destination ip address, source port, and destination port. As mentioned in [1], by looking at the output of the hash function for different tuples, it may be possible to select appropriate port numbers that map the flow to the desired core.

4. EXPERIMENTAL TESTBED

Our performance analysis uses resources from ESnet's 100G Testbed [2], which includes a dedicated 100 Gbps link connecting the National Energy Research Scientific Computing Center[3] (NERSC) in Oakland, CA to StarLight[4] in Chicago, IL.

The ESnet Testbed, shown in Figure 2, is a public testbed open to any researcher and includes high-speed hosts at both NERSC and StarLight. The results presented in this paper were collected using two 40 Gbps capable hosts (*diskpt-6,7*) at NERSC. Each host has two 6-core Intel processors, 64GB of system memory and 2 Mellanox 40G NICs. Each 40 Gbps host is based on the Intel Sandy Bridge[3] architecture with PCI Gen-3, supporting double the previous generation bus capacity.

In order to evaluate performance over a high latency path, we configured a 40 Gbps loop from Oakland to Chicago and back. Traffic originating from a host interface at NERSC would reach StarLight and then return to NERSC for a total RTT of 95ms. All of our tests were run along this loopback path unless where otherwise noted. There was no other traffic on the path at the time of our testing.

The overall architecture is shown in Figure 3.

[2]ESnet 100G Testbed [19]
[3]National Energy Research Center http://www.nersc.gov
[4]StarLight: http://www.startap.net/starlight/

All of the hosts ran a 2.6.32-220 Linux kernel. We performed standard host and NIC driver tuning to ensure the best performance for each of our benchmarks[5]. The Maximum Transmission Unit (MTU) on each installed NIC was set to 9000 bytes and Rx/Tx link layer flow control was enabled by default. Each of our experiments involved memory-to-memory transfers to remove disk I/O as a potential bottleneck. In this testbed nearly all host interfaces are connected directly to high-end 100 Gbps Alcatel-Lucent Model SR 7750 border routers, which have a large amount of buffer space.

5. RESULTS

5.1 List of Key Parameters

A summary of the key system parameters that impact the end-to-end performance is listed in Table 1.

5.2 Experimental Approach

The goal of our experimentation was to evaluate the performance of a single TCP flow for a 40 Gbps data transfer between two systems. TCP/IP remains the most popular standard for file transfer and connection management on the internet today, so it was our protocol stack of choice. The software used to simulate the network load was netperf, and it was used in the TCP_SENDFILE configuration, which triggers the zero-copy offload on the sending system in CentOS. We did this to create a situation where the receiver was the bottleneck. In an average test, for example, the active cores on the receiver would be at or near 100% utilization, while the sender CPU would be under 30% utilization.

Netperf is a software tool for benchmarking network performance. It works by keeping the time of the test constant, but measuring the amount of data sent or received, or the number of connections opened and closed. We used the TCP_SENDFILE option, in which the sending process simply generates data and sends it using the sendfile() call in the sockets API.

The System Under Test (SUT) was a standard CentOS image provided by ESnet. The only modifications made to the OS image we were provided were the installation of the latest versions of tcpdump, libpcap, and Intel's Performance Counter Monitor for Sandy Bridge Architectures. Hardware-wise, these systems are composed of dual Intel Xeon E5-2667 6-core processors and 64GB of DDR3-1600 ECC Memory. Herein, we will be referring to the 6-core packages as "sockets" and the individual cores simply as "cores." The NIC is physically connected to Socket 1, and the processors in this system come equipped with what is referred to as the Intel QuickPath Interconnect, or QPI. In the Sandy Bridge architecture, PCI slots are physically connected to one socket. If an application on one socket requires data from a device physically connected to another socket, that data must traverse the QPI between the two sockets. The QPI in our system has a maximum transfer rate of 16 gigabytes per second. While we were unable to test the throughput of the QPI due to the lack of accessibility to the hardware counters in our system's BIOS, we have assumed for the purposes of this experiment that the

[5]Linux TCP tuning: http://fasterdata.es.net/host-tuning/linux/

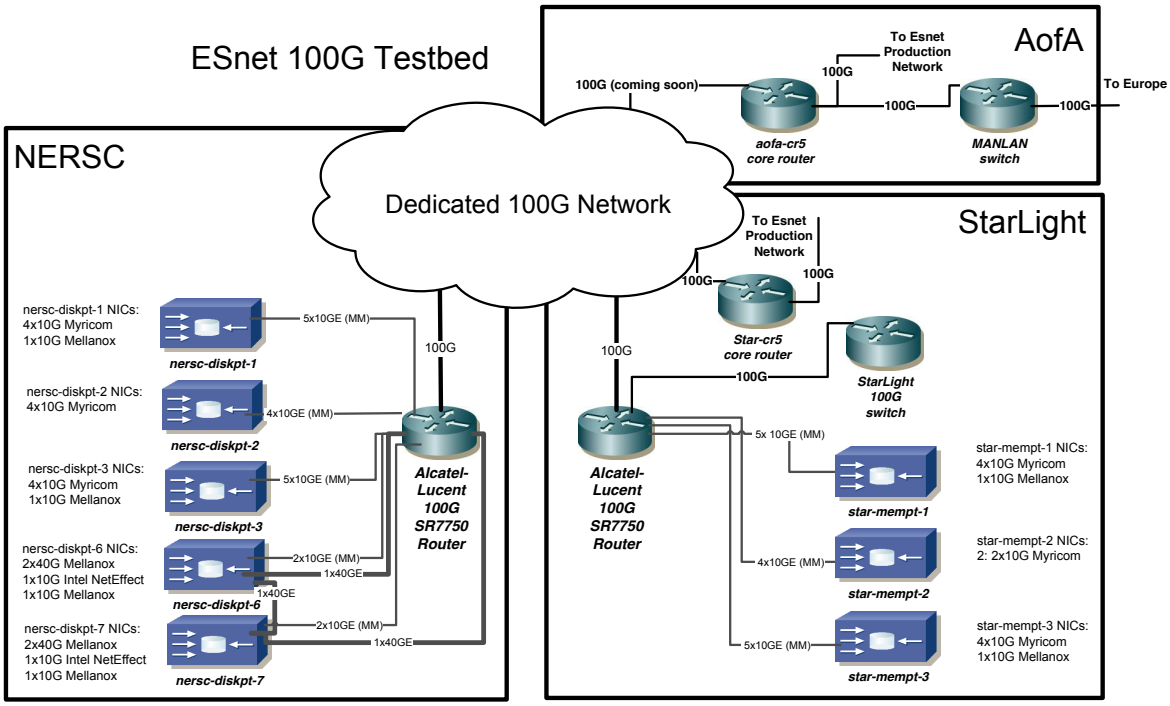

Figure 2: The ESNET 100G test-bed.

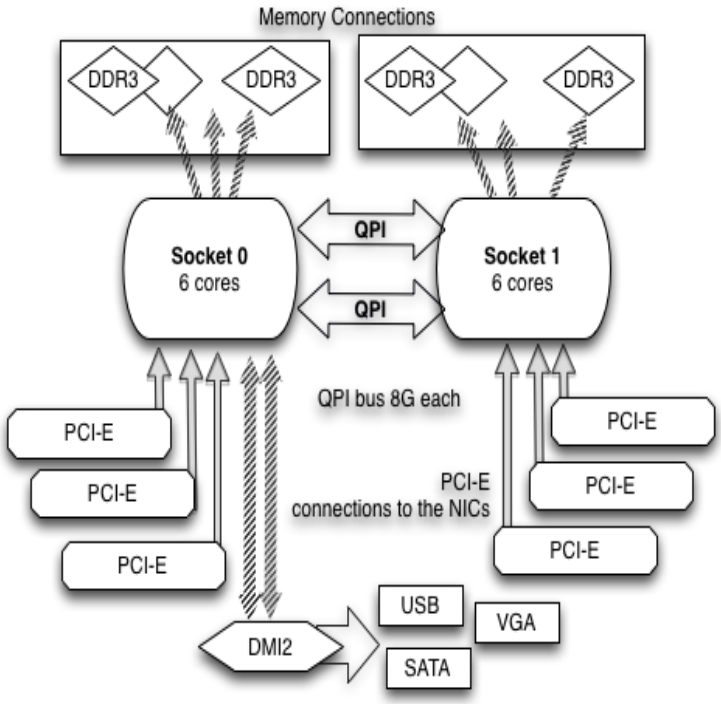

Figure 3: The end-system architecture [26].

Parameter	Value
MTU	9000 bytes
SKB Size	Auto (up to 100MB)
RPS/RFS	Off
irqbalance	Off
netperf Version	2.6.0
Intel PCM Version	2.5.1
CentOS Linux Version	2.6.32-220.17.1.el6.x86_64
TCP congestion control algorithm	TCP Cubic
tcpdump Version	4.4.0
libpcap Version	1.4.0

Table 1: List of parameters.

QPI was not a limiting factor, since the maximum transfer rate is much faster than the memory transfer rate and the network throughput. It should also be noted that Simultaneous Multithreading (HyperThreading) was turned off for the duration of the tests. However, exhaustive tests with HyperThreading on this architecture are underway.

The preliminary tests were to confirm that the tuning suggestions and default configuration discussed by the ESnet staff were best under our testing scenario. All of the recommendations were confirmed to be beneficial, including turning off irqbalance and setting the MTU to 9000 bytes (Jumbo Frames). Using the affinitization recommendations of the ESnet staff, we were able to achieve a throughput of about 36 Gbps, a number close enough to the line rate for us to be convinced that the 4 Gbps not witnessed in the transfer was due to protocol processing and TCP overhead. Once this rate was reached, we ran different test lengths, to see if the throughput would continue increasing for longer tests. We found 180 seconds to be the point where the performance decrease due to TCP slow start becomes negligible, and running longer tests does not generally yield greater throughput numbers. Finally, we tested some of the parameters of our system using the Performance Counter Monitor. Most notable are the memory throughput figures noted above, which were obtained by several consecutive executions of the memoptest tool.

The main experiment took into account two types of affinity as independent variables: Application and Flow Affinity. The core binding of the receiving proces was application Affinity. The core binding of the NIC queues was flow affinity.0 We exhaustively tested each of the 144 combinations of Application Affinity and Flow Affinity on our SUT. We tracked the throughput and the L3 cache hit ratio for the core that the netperf receiving process was bound to using the Performance Counter Monitor. We tracked the L3 cache hit ratio because this number effectively tells us how little the application receiving core needed to access memory. These L3 cache hit ratios pertain only to cache requests made by the core with the application receiving process, so as to rule out cache misses on other cores due to other factors. The cores under test were also unloaded, so it can be safely assumed that the vast majority of these cache requests were due to the application receiving process. Finally, the entire exhaustive test was repeated, to ensure that the patterns were not anomalies. Furthermore, exhaustive preliminary throughput testing was done on an image running Fedora Core 17 with similar results.

5.3 Discussions of Results

Figure 4 and Figure 5 are based on the output of the exhaustive testing of every combination of affinity. The Flow Affinity lies across the horizontal axis, and is the core that all of the NIC queues have been bound to on the receiving system. The Application Affinity lies on the vertical axis, and is the core that the netperf receiving process has been bound to. Each bubble in the throughput graph represents the average throughput over a 180 second test. The area of the bubble corresponds to the throughput in gigabits per second. Each bubble in the L3 Cache Hit Ratio graph has an area corresponding to the L3 cache hit ratio of the core that the netperf receiving process resided on during the test. Generally, it can be assumed that the larger the L3 Cache Hit Ratio, the fewer times netperf had to go to memory to retrieve its data. (However, there could be a caveat to this, which we will discuss later.) The boxes represent the four quadrants of the graph showing the socket combination used. In the upper left quadrant, socket 0 has the Flow Affinity, while socket 1 has the Application Affinity. The lower-right quadrant represents the reverse. In the lower-left quadrant, socket 0 has both the Flow and Application Affinity. In the upper-right quadrant, socket 1 has the Flow and Application Affinity.

During the exhaustive testing of every combination of Application Affinity and Flow Affinity, we found that the conventional wisdom of affinitizing both application and flow to the same core produced poor throughput and mediocre L3 Cache Hit ratios. Due to the reported CPU utilization and the cache hit ratios, we believe that the low hit ratio may be due to both protocol processing overhead and cache pollution, with the L2 cache flooded with both application data and network processing data. The low throughput is most likely a result of competition for limited processing resources between the kernel- and user-space processes. This pattern can be observed in the charts below by observing the diagonals, where both affinities reside on the same core.

Using cores on two different sockets also produced poor throughput and poor L3 cache hit ratios. This can be observed in the graphs by looking at the upper-left and lower-right quadrants of the graph. We believe that this is due to the use of memory as the first shared level of the memory hierarchy between the two sockets. In these instances, the throughput is limited to the throughput of the memory

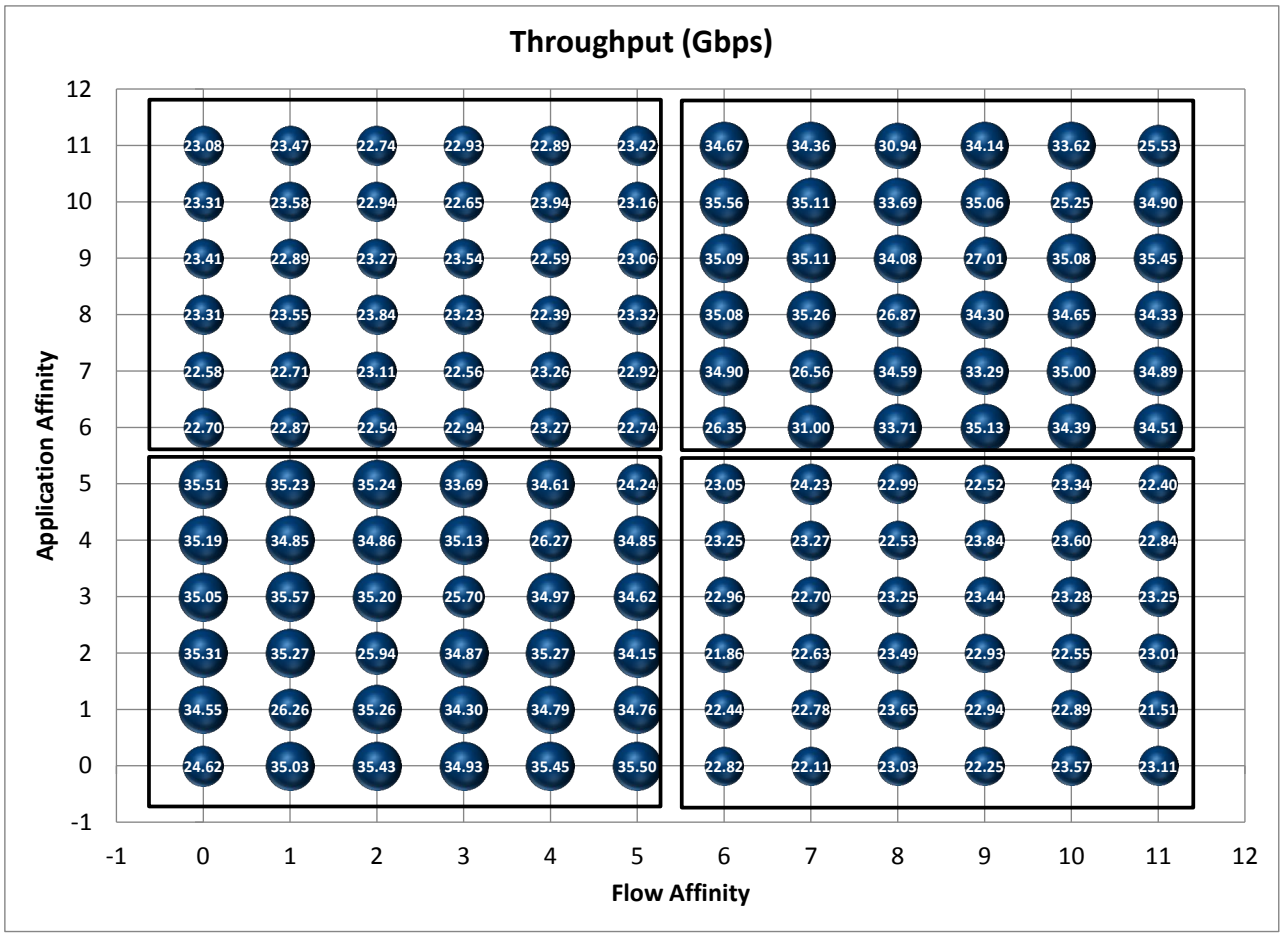

Figure 4: The achieved throughout as a function of the flow and application affinity.

as the network data must traverse memory from one socket to another after the initial protocol processing. This data does not flow across the QPI because the kernel and the application are responsible for placing it.

Using different cores on the same socket produced the best throughput. This can be observed in the upper-right and lower-left quadrants of the graph. Keeping both affinities on socket 1 (the socket that the NIC is physically connected to over the PCI bus) produced the best L3 cache hit ratios. Keeping both affinities on socket 0 produced mediocre L3 cache hit ratios. It would appear that these relatively low L3 cache hit ratios have no bearing on the throughput. However, it is possible that the L3 cache hit ratios are a form of statistical skew in this case, where the accesses over the QPI are unaccounted for. Unlike in the previous case, network data must move over the QPI in this case, since the NIC is physically connected to the other socket.

As far as cache performance is concerned, the assignment where the flow and application are affinitized to the same core is ideal, according to previous research. But if we consider the high incoming data rate, the cost of frequent context switching will be very high. The network stack process runs at a strictly higher priority than the user-space process, and as a result, the user-space process will not have enough CPU cycles to read the packets from its socket buffer, leading to packet loss. If hyperthreads are available, and the CPU can fetch instructions from multiple threads simultaneously, then this option may be ideal. Hyperthreads can reduce the cache misses beyond LLC because in the case of packet processing they will be accessing shared data, i.e. packets passed between the kernel process and the user process. If hyperthreads do not access shared data, then they would split all the shared caches in half, and yield no benefit. As an alternate, the third option can also lead to reduced LLC misses. The last option is certainly not desired because it will lead to additional cache misses.

Further evidence of a clear pattern in throughput and cache performance depending on affinity comes from the histograms of the throughput and L3 cache hit ratios shown in Figure 6 and Figure 7.

Two distinct peaks can be seen in the throughput histogram: one where the affinitization was sub-optimal, and one where the affinitization was optimal. Similarly, three peaks can be seen in the L3 cache hit ratio histogram. The lowest

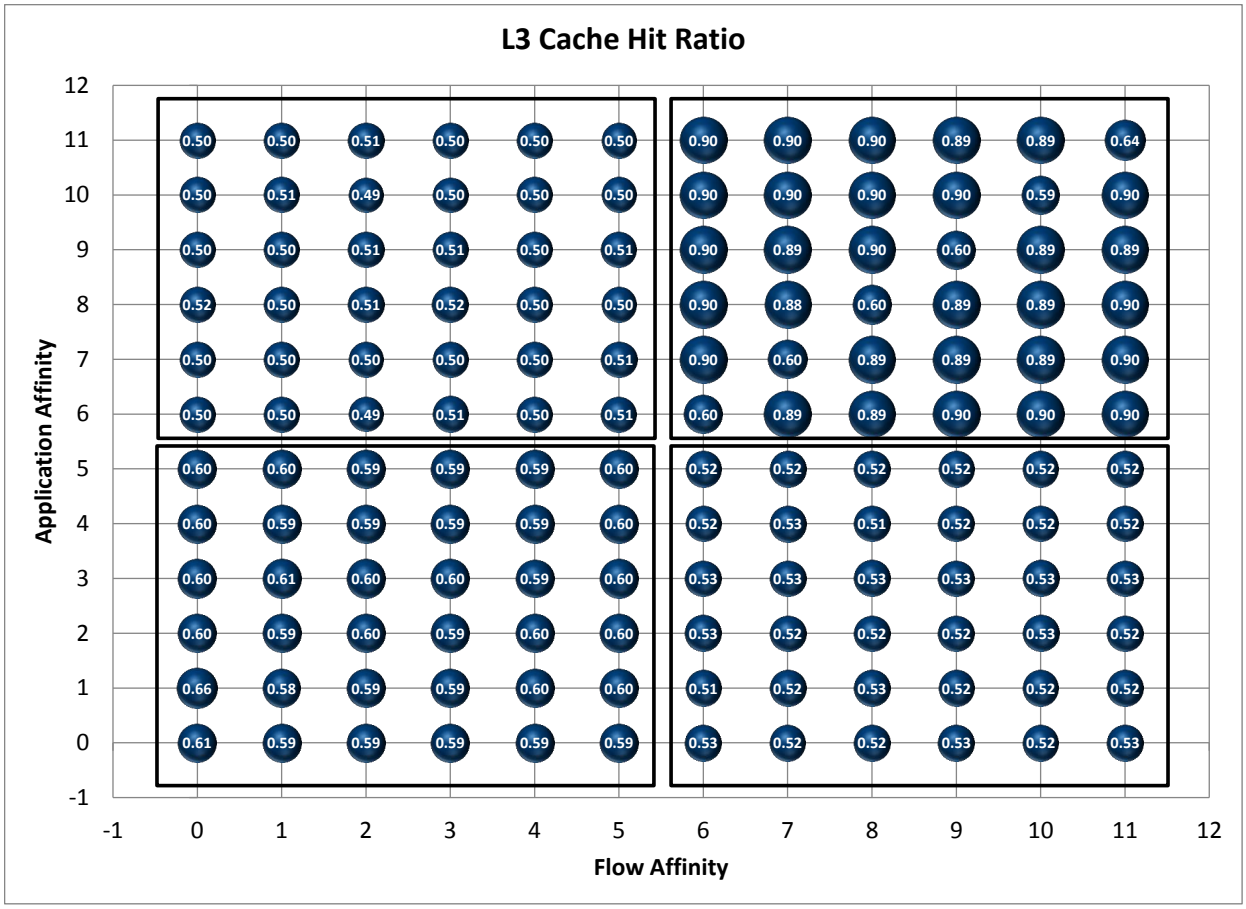

Figure 5: The L3 cache hit ratio for the core that services the application for different flow and application affinity.

peak represents configurations where the network data was passed through memory. The middle peak represents configurations where both processes shared a core, and when both processes were affinitized to socket 0. The highest peak represents configurations where both processes were affinitized to socket 1. Evidence of a pattern comes from the fact that these very similar throughput and cache hit ratio numbers came up many times during testing.

6. CONCLUSION

One of the most important conclusions of this research is that any system attempting to direct a large transfer to a particular core needs to pay attention to which core and which socket the application receiving the data resides on. It needs to steer the flow to a different core on the same socket in order to optimize the protocol processing overhead of the transfer.

There are many options for future research that we intend to pursue. As mentioned above, our first priority is to explain the L3 cache hit ratio mismatch by running kernel-introspective experiments. Also, we will be performing these same experiments with Hyper-Threading turned on in order to see the effect on priority and cache utilization. We would also like to perform these experiments on different architectures, including architectures from other vendors. This would allow us to gather data on the differences in high-speed network performance between architectures. Furthermore, we will be performing these experiments with multiple 40 Gbps NICs, and therefore multiple flows. We will also perform experiments that mix disk I/O activity, scientific computing activity, and high-speed network transfers to find optimal configurations using more practical benchmarks. Another area of further research is to simulate different application characteristics, including short data transfers and streaming data. Our overall goal is to develop a tool based on our current introspective tools that will automatically configure end-systems for optimal high-speed network transfers.

7. ACKNOWLEDGEMENTS

This research used resources of the ESnet Testbed, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

8. REFERENCES

[1] V. Ahuja, M. Farrens, and D. Ghosal. Cache-aware affinitization on commodity multicores for high-speed network flows. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and*

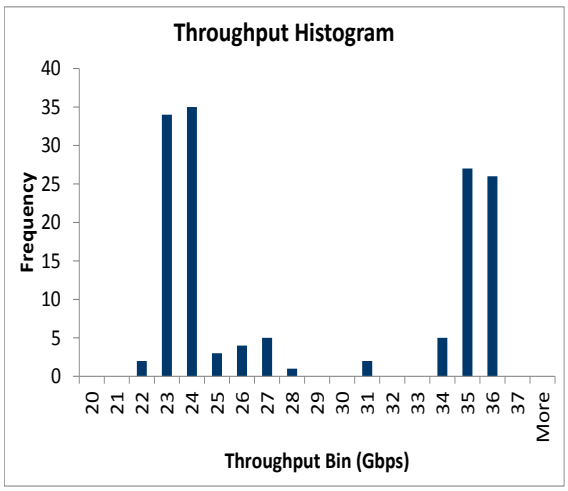

Figure 6: Histogram of the different throughput values.

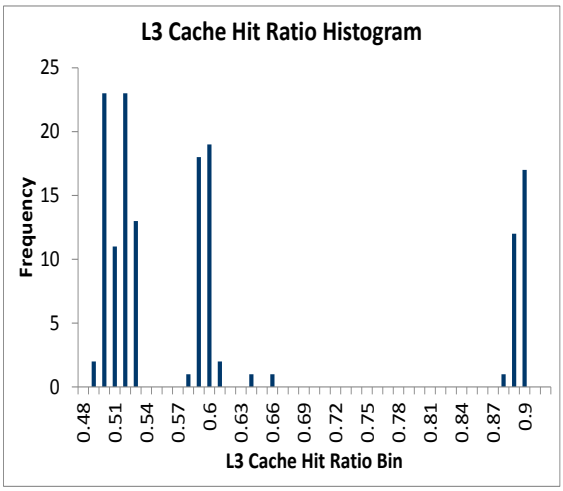

Figure 7: Histogram of the different cache hit ratio values.

communications systems, pages 39–48. ACM, 2012.

[2] M. Balakrishnan. *Reliable Communication for Datacenters*. PhD thesis, Cornell University, 2009.

[3] I. S. Bridge. Sandy bridge architecture, `http://en.wikipedia.org/wiki/SandyBridge/`.

[4] A. Foong, J. Fung, D. Newell, S. Abraham, P. Irelan, and A. Lopez-Estrada. Architectural characterization of processor affinity in network processing. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 207–218. IEEE, 2005.

[5] T. Herbert. rfs: receive flow steering, september 2010. `http://lwn.net/Articles/381955/`.

[6] T. Herbert. rps: receive packet steering, september 2010. `http://lwn.net/Articles/361440/`.

[7] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 50–59. IEEE Computer Society, 2005.

[8] S. Jana, A. Pande, A. Chan, and P. Mohapatra. Network characterization and perceptual evaluation of skype mobile videos. In *22nd International Conference on Computer Communications and Networks (ICCCN)*, 2013.

[9] H. Jang and H. Jin. Miami: Multi-core aware processor affinity for tcp/ip over multiple network interfaces. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pages 73–82. IEEE, 2009.

[10] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of direct cache access on multi-core systems and 10gbe. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 341 –352, feb. 2009.

[11] E. León, K. Ferreira, and A. Maccabe. Reducing the impact of the memorywall for i/o using cache injection. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 143–150. IEEE, 2007.

[12] E. León, R. Riesen, K. Ferreira, and A. Maccabe. Cache injection for parallel applications. *proc. HDPC'11*, pages 15–26, 2011.

[13] G. Liao, D. Guo, L. Bhuyan, and S. King. Software techniques to improve virtualized i/o performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 161–170. ACM, 2008.

[14] G. Liao, X. Zhu, and L. Bnuyan. A new server i/o architecture for high speed networks. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 255–265. IEEE, 2011.

[15] T. Marian. *Operating systems abstractions for software packet processing in datacenters*. PhD thesis, Cornell University, 2011.

[16] T. Marian, D. Freedman, K. Birman, and H. Weatherspoon. Empirical characterization of uncongested optical lambda networks and 10gbe commodity endpoints. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2010.

[17] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15(3):217–252, 1997.

[18] G. Narayanaswamy, P. Balaji, and W. Feng. Impact of network sharing in multi-core architectures. In *Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on*, pages 1–6. IEEE, 2008.

[19] E. S. Network. Esnet, `http://www.es.net/`.

[20] S. Networking. Eliminating the Receive Processing Bottleneckâ Introducing RSS. *Microsoft WinHEC (April 2004)*, 2004.

[21] V. Omwando, A. Pande, Y. Zeng, and P. Mohapatra. Evaluating perceptual video quality in 802.11n wlan with mobile clients. In *The 8th ACM International Workshop on Wireless Network Testbeds,*

Experimental Evaluation and Characterization (ACM WiNTECH) 2013, pages –, 2013.

[22] A. Pande and J. Zambreno. Efficient translation of algorithmic kernels on large-scale multi-cores. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 915–920. IEEE, 2009.

[23] RSS. Scaling in the Linux Networking Stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

[24] J. Salim. When napi comes to town. In *Linux 2005 Conf*, 2005.

[25] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack: highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[26] SuperMicro. Supermicro x9dr3-f user's manual,`http://www.supermicro.com/products/motherboard/xeon/c600/x9dr3-f.cfm`.

[27] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler. The linux networking architecture. *Design and Implementation of Network Protocols in the Linux Kernel*, 2005.

[28] W. Wu, P. DeMar, and M. Crawford. A transport-friendly nic for multicore/multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4):607–615, 2012.