2014 Fourth International Workshop on Network-Aware Data Management

Analysis of the Effect of Core Affinity on High-Throughput Flows

Nathan Hanford, Vishal Ahuja,
Matthew Farrens, and Dipak Ghosal
Department of Computer Science
University of California
Davis, CA
{nhanford, vahuja, mkfarrens, dghosal}@ucdavis.edu

Mehmet Balman, Eric Pouyoul, and Brian Tierney
Energy Sciences Network
Lawrence Berkeley National Laboratory
Berkeley, CA
mbalman@lbl.gov, lomax@es.net, bltierney@es.net

***Abstract*—Network throughput is scaling-up to higher data rates while end-system processors are scaling-out to multiple cores. In order to optimize high speed data transfer into multicore end-systems, techniques such as network adapter offloads and performance tuning have received a great deal of attention. Furthermore, several methods of multithreading the network receive process have been proposed. However, thus far attention has been focused on how to set the tuning parameters and which offloads to select for higher performance, and little has been done to understand why the settings do (or do not) work. In this paper we build on previous research to track down the source(s) of the end-system bottleneck for high-speed TCP flows. For the purposes of this paper, we consider protocol processing efficiency to be the amount of system resources used (such as CPU and cache) per unit of achieved throughout (in Gbps). The amount of various system resources consumed are measured using low-level system event counters. Affinitization, or core binding, is the decision about which processor cores on an end system are responsible for interrupt, network, and application processing. We conclude that affinitization has a significant impact on protocol processing efficiency, and that the performance bottleneck of the network receive process changes drastically with three distinct affinitization scenarios.**

I. Introduction

Due to a number of physical constraints, processor cores have hit a clock speed "wall". CPU clock frequencies are not expected to increase. On the other hand, the data rates in optical fiber networks have continued to increase, with the physical realities of scattering, absorption and dispersion being ameliorated by better optics and precision equipment [1]. Despite these advances at the physical layer, we are still limited with the capability of the system software for protocol processing. As a result, efficient protocol processing and adequate system level tuning are necessary to bring higher network throughput to the application layer.

TCP is a reliable, connection-oriented protocol which guarantees in-order delivery of data from a sender to a receiver, and in doing so, pushes the bulk of the protocol processing to the end-system. There is a certain amount of sophistication required to implement the functionalities of the TCP protocol, which are all instrumented in the end-system since it is an end-to-end protocol. As a result, most of the efficiencies that improve upon current TCP implementations fall into two categories: first, there are offloads which attempt to push TCP functions at (or along with) the lower layers of the protocol stack (usually hardware, firmware, or drivers) in order to achieve greater efficiency at the transport layer. Second, there are tuning parameters, which place more sophistication at the upper layers (software, systems, and systems management).

Within the category of tuning parameters, this work focuses on affinity. Affinity (or core binding) is fundamentally the decision regarding which resources to use on which processor in a networked multiprocessor system. Message Passing in the Linux network receive process in modern systems principally allows for two possibilities: First, there is interrupt processing (usually with coalescing), in which the NIC interrupts the processor once it has received a certain number of packets. Then, the NIC transmits the packets to the processor via DMA, and the NIC driver and the OS kernel continue the protocol processing until the data is ready for the application [2]. Second, there is NIC polling (known in Linux as the New API (NAPI)), where the kernel polls the NIC to see if there is any network data to receive. If such data exists, then the kernel processes the data in accordance with the transport layer protocol in order to deliver the data to the Application in accordance with the socket API. In either case, there are two types of affinity: 1) *Flow affinity*, which determines which core will be interrupted to process the network flow, and 2) *Application affinity*, which determines the core that will execute the application process that receives the network data. Flow affinity is set by modifying the hexadecimal core descriptor in /proc/irq/<irq#>/smp_affinity, while Application affinity can be set using *taskset* or similar tools. Thus, in a 12-core end-system, there are 144 possible combinations of Flow and Application affinity.

In this paper, we extend our previous work [3], [4] with detailed experimentation to stress-test each of affinitization combinations with a single, high-speed TCP flow. We use end-system performance introspection to understand the effect that the choice of affinity has on the receive-system efficiency. We conclude that there are three different affinitization scenarios, and that the performance bottleneck varies drastically within these scenarios.

II. Related Work

There has been several studies evaluating performance of network I/O in multicore systems [5]–[8]. A major improvement that is enabled by default in most of the current kernels is

978-1-4799-7019-3/14 $31.00 © 2014 IEEE
DOI 10.1109/NDM.2014.10

NAPI [9]. NAPI is designed to solve the receive livelock problem [8] where most of the CPU cycles are spent for interrupt processing. When a machine enters to a livelock state since most of the cycles are spent in hard and soft interrupt contexts, it starts dropping packets. A NAPI enabled kernel switches to polling mode at high rates to save CPU cycles instead of operating in a purely interrupt-driven mode. Other related work characterizing the packet loss and performance over 10 Gbps WAN include [10], [11]. [12] focuses more on the architectural sources of latency rather than throughput of intra-datacenter links. Another method of improving the adverse effects of the end-system bottleneck involves re-thinking the hardware architecture of the end-system altogether. Transport-Friendly NICs, and even new server architectures have been proposed along these lines [13], [14]. Unfortunately, too few of these dramatic changes have found their way into the type of commodity end-systems that have been deployed for the purposes of these tests.

A. Linux Network Performance Tuning Knobs

Contemporary NICs support multiple receive and transmit descriptor queues. NICs apply a filter and send packets to different queues to distribute the load among multiple cores. In Receive-Side Scaling (RSS) [15], packets for each flow are sent to different receive queues and processed by different CPU cores. RSS is supported by the NIC. Receive Packet Steering (RPS) [16] is simply a software version of RSS done in host kernel. It selects the CPU core that will perform protocol processing for incoming set of packets. Receive Flow Steering (RFS) [17] is similar to RPS but it uses the TCP 4-tuple to make sure all packets of a flow go to the same queue and core. All these scaling techniques are designed to allow performance increase uniformly in multicore systems.

Common wisdom is to select cores that share the same lowest cache structure [1] when doing network processing [6], [18]. 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 daemon does a round-robin scheduling to distribute interrupt processing load among cores. However, it has adverse effects as shown by [19], [20]. We require a more informed approach and we need control over selecting cores for interrupt processing. In our experiments we disable irqbalance daemon.

Pause frames [21] allow Ethernet to prevent TCP with flow control, thus avoiding a multiplicative decrease in window size when only temporary buffering at the router or switch is necessary. In order to do this, Ethernet devices which support pause frames use a closed-loop process in each link in which the sending device is made aware of the need to buffer the transmission of frames until the receiver is able to process them.

Jumbo Frames are simply Ethernet frames that are larger than the IEEE standard 1500-byte MTU. In most cases, starting with Gigabit Ethernet, frame sizes can be up to 9000 bytes. This allows for better protocol efficiency by increasing the ratio of payload to header size for a frame. Although Ethernet speeds have now increased to 40 and 100 Gbps, this standard 9000-byte frame size has remained the same [22]. The reason for this is the various segmentation offloads. Large/Generic Segment Offload (LSO/GSO) and Large/Generic Receive Offload (LRO/GRO) work in conjunction with Ethernet implementations in contemporary routers and switches to send and receive very large frames in a single TCP flow, for which the only limiting factor is the negotiation of transfer rates and error rates.

[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.

B. Data Movement Techniques

Although this research is not directly concerned with the practice of moving large amounts of data across long distances, there have been several implementations of effective end-system applications which have helped leverage TCP efficiently [23]–[26]. The general idea behind these sophisticated applications is to use TCP striping, splitting transfers at the application layer into multiple TCP flows, and assigning the processing and reassembly of those flows to multiple processors [27]–[29]. Most of these applications are also capable of leveraging alternative transport-layer protocols based on UDP, such as RBUDP [30] and UDT [31].

There have long been protocols designed for moving large amounts of data quickly and reliably within closed networks, either within datacenters, or between datacenters. Lately, such protocols have focused on moving data directly from the memory of one system into the memory of another. Two examples include RDMA [32] and InfiniBand [33]. The commonality of these protocols is their reliance on relatively sophisticated, robust, and reliable intermediary network equipment between the end-systems.

Previously, we conducted research into the effect of affinity on the end-system bottleneck [3], and concluded that affinitization has a significant impact on the end-to-end performance of high-speed flows. That research left off with the question of precisely where the end-system bottleneck lies in these different affinitization scenarios. Our current research goal is to identify the location of the end-system bottleneck in these different affinitization scenarios, and evaluate whether or not these issues have been resolved in newer implementations of the Linux kernel (previous work was carried out on a Linux 2.6 kernel).

III. Experimental Setup

There are many valid arguments made in favor of the use of various NIC offloads. NIC manufacturers typically offer many suggestions on how to tune the system in order to get the most out of their high-performance hardware. A valuable resource for Linux tuning parameters, obtained from careful experimentation on ESnet's 100 Gbps testbed, is available from [34]. ESnet has given a number of presentations detailing the experiments that have led to their tuning suggestions - however, light is rarely shed on the empirical rationale for these tuning suggestions and offloads.

In this paper, our methods focus on improving the multithreading of protocol processing, or pushing the protocol

processing to different parts of the protocol stack. We endeavor to demonstrate the importance of spatial locality of reference in the processing of data flows across multiple flows in an end system. As such, these experiments employ *iperf3* [35] to generate a stress test which consists of pushing the network I/O limit of the end-system using a single, very high-speed TCP flow. This is not a practical scenario; an application such as GridFTP [23] delivers faster, more predictable performance by using multiple flows, and such a tool should be carefully leveraged in practice when moving large amounts of data across long distances. However, it is important to understand the limitations of data transmission in the end-system, which can best be accomplished using a single flow.

To that end, we have employed two servers that are connected with <1ms RTT, as opposed to our previous experiments [3] which used the ESnet Testbed's 95ms RTT fiber loop. While loop testing is important to see TCP's bandwidth behavior over long distances, our goal here was to place stress on, and analyze, the performance efficiency of receiver end-system.

Both of the systems in these experiments were running Fedora Core 20 with the 3.13 kernel, as opposed to our previous experiments, which used CentOS6 running a 2.6 kernel. The use of one of the latest Linux kernels assures that the latest advancements in kernel networking design are employed in this system.

The benchmark application used to generate the TCP flows was iperf3. Again, to ensure that the stress was placed on the end-system, the transfers were performed in zero-copy mode, which utilizes of the TCP sendfile system call to avoid unnecessary copies into and out of memory on the sending system.

The systems under test were modeled after prototypes for ESnet Data Transfer Nodes (DTNs). The goal of these systems is to serve as an intermediary to transfer large amounts of data from high-performance computers (HPCs) to the consuming end-systems [36]. In practice, they must be able to take in large amounts of data as quickly as possible through InfiniBand host-bus adapters, transfer the data to local disks or large amounts of local memory, and then serve the data over high-speed (100 Gbps) WAN to similar receiving systems. They make use of PCI-Express Generation 3 connected to Intel Sandy Bridge processors. There are two hexa-core processors per end-system[2]. Due to the design of Sandy Bridge processors, each socket is directly connected to its own PCI-Express bus, meaning that certain PCI-Express slots are directly physically linked to a single socket. This limitation is overcome with the addition of low-level inter-socket communication provided by the Quick Path Interconnect (QPI). This architecture is seen in Figure 1.

The testbed used was the ESnet 100 Gbps testbed [37], which is host to a variety of commodity hardware based end-systems connected to a dedicated transcontinental 100 Gbps network. The testbed is open to any researcher, and provides the added benefit of yielding repeatable results, since the entire testbed is reservable. This guarantees that there is no competing traffic. For the purposes of these experiments, it allowed us to ensure that the bottleneck was in the end-systems, and not the network.

[2]Herein, these six-core packages will be referred to as "sockets" and the individual multi-instruction multi-data (MIMD) cores will be referred to as "cores."

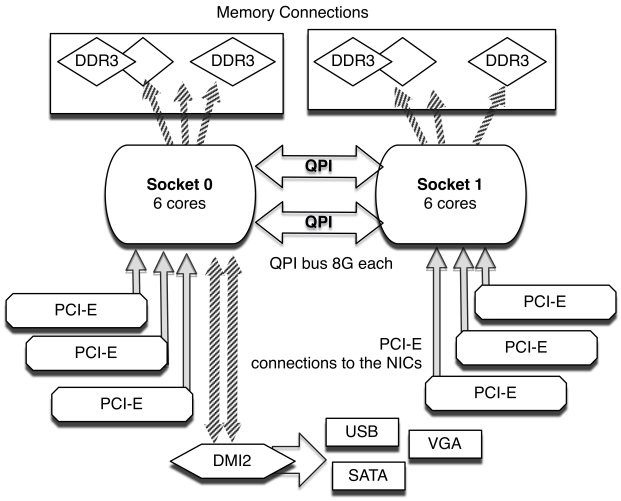

Fig. 1: Block diagram of the end-system I/O architecture

The Linux performance profiler used was *Oprofile*. Oprofile is a system profiler for Linux which enables lightweight, but highly introspective monitoring of system hardware counters [38]. Oprofile's new *ocount* and *operf* tools were used to monitor counters of various events on the receiving system. Oprofile's low overhead and ability to do detailed Linux kernel introspection proved critical in these experiments, due to the need to monitor a possibly oversubscribed receiver. The overhead of operf was able to be effectively measured through the introspection, and it was found that this overhead was always at least one order of magnitude less than the counter results from the monitored processes. Oprofile was chosen over Intel's Performance Counter Monitor (PCM) for these experiments due to the number of counters available and the introspection capability. However, PCM is capable of reporting power consumption, which could be useful in future tests.

The following are the independent variables used in the experiments:

- Flow affinity: Cores 0 through 11
- Application affinity: Cores 0 through 11
- Total Number of Tests: 12x12=144

The following are the dependent variables used in the experiments

- Throughput
- Instructions Retired (both System-Wide and with Introspection into the kernel and user processes)
- Last-Level Cache References (System-Wide and Introspection)
- L2 Cache Accesses
- Memory Transactions Retired
- Offcore Requests

A. List of Environmental Parameters

A summary of the experimental environment is listed in Table I. The sending and receiving end systems were identical.

Parameter	Value
RTT	<1ms
Router	ALU S7750 100 Gbps
Motherboard	PCI Gen3
Processors	2 hexa-core Intel Sandy Bridge
Processor Model	Intel Xeon E5-2667
On-Chip QPI	8 GT/s
NIC	Mellanox ConnectX-3 EN
NIC Speed	40 Gbps
Operating System	Fedora Core 20 Kernel 3.13
irqbalance	Disabled
TCP Implementation	HTCP
Hardware Counter Monitor	Oprofile 0.9.9
Test Application	iperf3 3.0.2

TABLE I: List of environmental parameters.

IV. Experimental Approach

Before each experimental run of 144 tests, a script would recheck a variety of system network settings and tuning parameters, to ensure that subsequent runs of the experiment were consistent. The system was configured using ESnet's tuning recommendations for the highest possible TCP performance. Preliminary tests were conducted to ensure that no anomalies were causing variable bandwidth results. The sending system was set to an optimal affinity configuration and its affinity settings were not changed. An iperf3 server was started on the sender and left running. Then, on the receiver, a nested for-loop shell script modified the settings in /proc/irq/<irq# of all rx queues>/smp_affinity such that all the receive queues were sent to the same core. The inner for loop would run operf while conducting an iperf3 reverse TCP zero-copy test, binding the receiver to a given core, and then report the results. In this manner, all combinations of Flow and Application affinity were tested. The experiment was run several times to ensure consistency.

V. Results

Both our current and previous work [3] concluded that there exists three different performance categories, corresponding to the following affinitization scenarios: 1) Same Socket Same Core (i.e., both Flow and Application affinitized to the same core), which reaches a throughput of around 20 Gbps; 2) Different Sockets (thus Different Cores) which reaches a throughput of around 28 Gbps; and 3) Same Socket Different Cores, which reaches a throughput of around 39 Gbps. While changing the OS (from CentOS running a 2.6 kernel to Fedora running a 3.13 kernel) and updating the NIC driver improved the overall performance, the relative performance for the three affinitization settings remained the same.

Our previous work showed that there was a correlation between low throughput and cache misses on the oversubscribed receiver at the system level. This work picks up at that point, using ocount to monitor the hardware counters at the system level and operf for kernel introspection, to closely examine the source of the bottleneck in the receiver.

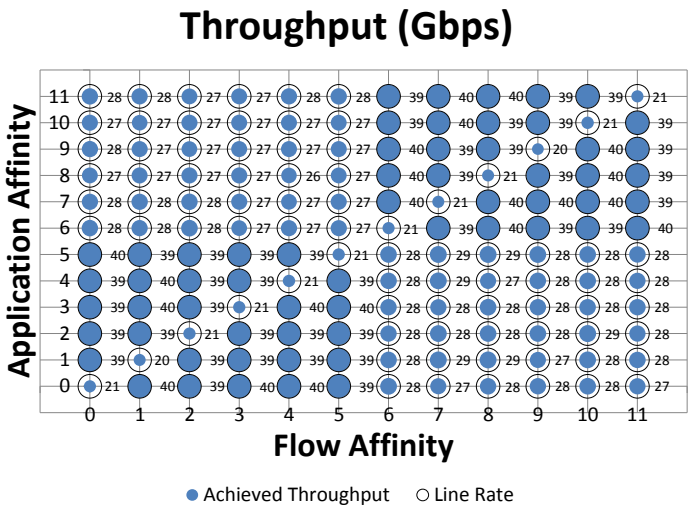

Fig. 2: The throughput results of all 144 tests arranged in a matrix by affinity choice. The empty circles represent the line rate of the NICs (40 Gbps). The size of the filled-in circles corresponds to the achieved throughput.

A. Interpretation of Results

In order to convey the results of the 144 tests effectively, we elected to create a matrix as seen in Figure 2. The Application affinity lies on the y-axis, while the Flow affinity lies along the x-axis. The numbers 0 through 11 on both axes represent the physical cores, as they appear to the operating system. Therefore, cores 0 through 5 lie on socket 0, and cores 6 through 11 lie on socket 1. As a matrix, the chart has two important properties:

1) The diagonal that appears in figure 2 represents the case where the Application and the Flow are affinitized to the same core.
2) The chart may be viewed in quadrants, where each quadrant represents the four possible combinations of affinity to the two sockets. In other words, all of the points where the Application affinity core is greater than 5 but the Flow affinity core is less than 6 represent the case where the Application is affinitized to socket 1 and the Flow is affinitized to socket 0, etc.

B. Overall Flow and Application Processing Efficiency

In the following figures we introduce Oprofile hardware counter results. When interpreting these results, it is important to note that hardware counters, on their own, convey little information. For example, one could simply look at the total number of Instructions Retired during the iperf3 transfer, but this would not take into account the amount of data that was actually transferred. The goal here is to view the efficiency, so the number of instructions retired has been divided by the

throughput of the transfer (in Gbps). This allows normalization of the results because the length of each test was identical.

Figure 3 shows the number of instructions retired per gigabyte per second of data transfer. The diagonal in this figure shows the processing inefficiency when both the Flow and Application are affinitized to the same core. In this case, not only is the throughput poor, as seen in Figure 2, but the processing efficiency is also much worse than the other cases. The case where the Flow is affinitized to socket 0 but the Application is affinitized to socket 1 shows that more instructions are required to move data from socket 0 to socket 1.

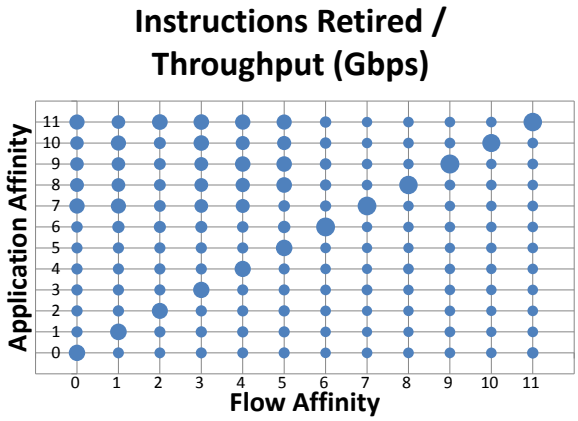

Fig. 3: The processing efficiency of the 144 tests; Larger circles represent poorer efficiency.

C. Memory Hierarchy

With the exception of the diagonal case mentioned above (where the Application and Flow were affinitized to the same core), user copies (copy_user_generic_string) dominated the resource consumption in the end system. From a system-wide perspective, this was demonstrated by a large percentage of instructions that were dedicated to accessing the memory hierarchy, as shown in Figures 4, 5 and 6. It should be noted that the titles of these figures have been abbreviated. Again, the raw counter output has little meaning here, so the counter data has been divided by the Instructions Retired/Throughput (Figure 3).

In the cases where the Application and Flow are affinitized to different cores, but are on the same sockets, memory hierarchy transactions appear to dominate the total instructions retired. However, these transfers were so close to line rate that the memory hierarchy was most likely not an actual bottleneck. Preliminary investigation shows that for the cases where the Flow and Application affinity are on different sockets, the bottleneck is possibly due to LOCK prefixes as the consuming core waits for coherency.

However, in the cases where the Application and Flow were affinitized to different sockets, a notably smaller fraction of instructions retired are dedicated to memory hierarchy transactions, despite the fact that user copies continue to dominate CPU utilization. Many different counters have been monitored and analyzed in an attempt to find the bottleneck in this case, including hardware interrupts and cycles due to LOCK prefixes, but none showed any correlation to this affinitization scenario.

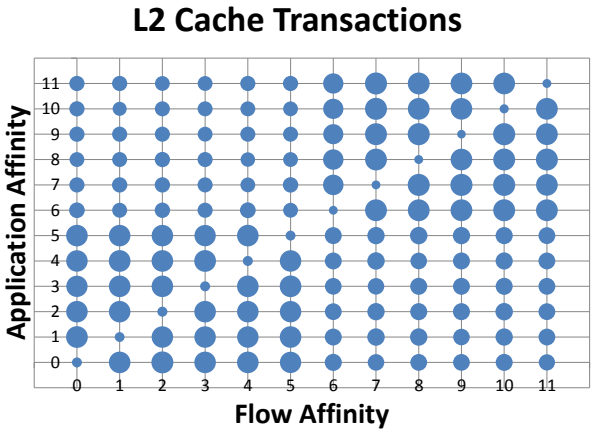

Fig. 4: The fraction of Instructions Retired/Throughput dedicated to Level 2 Cache Transactions (as measured by counter l2_trans) for all 144 tests.

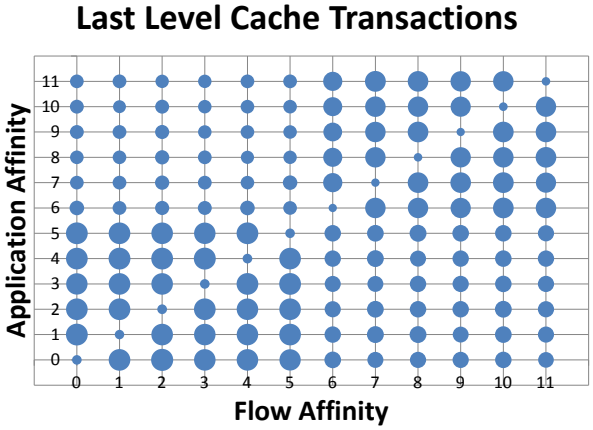

Fig. 5: The fraction of Instructions Retired/Throughput dedicated to Last Level Cache Transactions (as measured by counter LLC_TRANS) for all 144 tests.

D. The NIC Driver CPU Utilization Bottleneck

Interestingly, in the diagonal case, transactions involving the memory hierarchy represent a relatively small fraction of the overall instructions retired. In these cases, introspection shows us that the NIC driver (mlx4_en) is the primary consumer of system resources. The exact source of the bottleneck in this case will be investigated in the future using driver introspection.

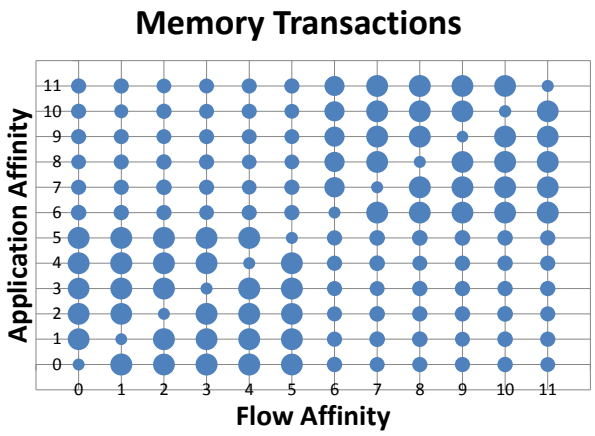

Fig. 6: The fraction of Instructions Retired/Throughput dedicated to Memory Transactions (as measured by counter mem_trans_retired) for all 144 tests.

VI. Conclusion and Future Work

One of the most important results of the clock speed wall is that the line between intra-system and inter-system communication is rapidly blurring. For one processor core to communicate with another, data must traverse an intra-system (on-chip) network. For large-scale data replication and coherency, data must traverse a WAN. How are these networks meaningfully different? WAN data transfer performance continues to become less of a limiting factor, and networks are becoming more reliable and more easily reconfigurable. At the same time, intra-system networks are becoming more complex (due to scale-out systems and virtualization), and perhaps less reliable (as energy conservation occasionally demands that parts of a chip could be slowed down, or turned off altogether). When discussing affinitization, it becomes obvious that despite these changes, distance and locality still matter, whether the network is "large" or "small". In the future, the most efficient solution may be not only to integrate a NIC onto the processor die [14], but perhaps even integrate the functionality with existing I/O structures, such as the North Bridge. However, the feasibility of doing so may be years away.

More tangibly, this research has concluded that moving more components of a system onto a chip (in this case, the PCI north bridge) needs to be done carefully, or it could result in sub-optimal performance across sockets. This provides an important backdrop upon which to perform end-system-centric throughput and latency tests, with attention to the fact that architectural latency sources for end-to-end TCP flows could vary drastically on high-throughput, high-performance hardware.

In the meantime, other NICs and other NIC drivers are being tested in similar ways to see if results are similar, and if generalizations can be made. The relatively recent advancement in NIC drivers that automatically switch between interrupt coalescing and NAPI is also being studied. In addition, results for practical, multi-stream TCP, and UDT GridFTP transfers are being examined along these lines. A future goal should be to implement a lightweight middleware tool that could optimize affinitization on a larger scale, extending the work that has been carried out on the Cache Aware Affinitization Daemon [18].

Acknowledgments

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. This research was also supported by NSF grant CNS-0917315.

References

[1] G. Keiser, *Optical Fiber Communications*. John Wiley & Sons, Inc., 2003.

[2] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly Media, 2005.

[3] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney, "Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows," in *Proceedings of the Third International Workshop on Network-Aware Data Management*, NDM '13, (New York, NY, USA), pp. 1:1–1:10, ACM, 2013.

[4] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney, "Impact of the end-system and affinities on the throughput of high-speed flows." poster - Proceedings of The Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) ANCS14, 2014.

[5] 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*, vol. 2, pp. 915–920, IEEE, 2009.

[6] A. Foong, J. Fung, and D. Newell, "An in-depth analysis of the impact of processor affinity on network performance," in *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, vol. 1, pp. 244–250 vol.1, Nov 2004.

[7] M. Faulkner, A. Brampton, and S. Pink, "Evaluating the performance of network protocol processing on multi-core systems," in *Advanced Information Networking and Applications, 2009. AINA '09. International Conference on*, pp. 16–23, May 2009.

[8] J. Mogul and K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 3, pp. 217–252, 1997.

[9] J. Salim, "When napi comes to town," in *Linux 2005 Conf*, 2005.

[10] 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*, pp. 575–584, IEEE, 2010.

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

[12] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural breakdown of end-to-end latency in a tcp/ip network," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009.

[13] W. Wu, P. DeMar, and M. Crawford, "A transport-friendly nic for multicore/multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 4, pp. 607–615, 2012.

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

[15] S. Networking, "Eliminating the receive processing bottleneckintroducing rss," *Microsoft WinHEC (April 2004)*, 2004.

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

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

[18] 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 communications systems*, pp. 39–48, ACM, 2012.

[19] 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*, pp. 207–218, IEEE, 2005.

[20] 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*, pp. 1–6, IEEE, 2008.

[21] B. Weller and S. Simon, "Closed loop method and apparatus for throttling the transmit rate of an ethernet media access controller," Aug. 26 2008. US Patent 7,417,949.

[22] M. Mathis, "Raising the internet mtu," *http://www.psc.edu/mathis/MTU*, 2009.

[23] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 54, IEEE Computer Society, 2005.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o.," in *OSDI*, pp. 135–148, 2012.

[25] M. Balman and T. Kosar, "Data scheduling for large scale distributed applications," in *Proceedings of the 9th International Conference on Enterprise Information Systems Doctoral Symposium (DCEIS 2007)*, DCEIS 2007, 2007.

[26] M. Balman, *Data Placement in Distributed Systems: Failure Awareness and Dynamic Adaptation in Data Scheduling*. VDM Verlag, 2009.

[27] M. Balman and T. Kosar, "Dynamic adaptation of parallelism level in data transfer scheduling," in *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, pp. 872–877, March 2009.

[28] M. Balman, E. Pouyoul, Y. Yao, E. W. Bethel, B. Loring, M. Prabhat, J. Shalf, A. Sim, and B. L. Tierney, "Experiences with 100gbps network applications," in *Proceedings of the Fifth International Workshop on Data-Intensive Distributed Computing*, DIDC '12, (New York, NY, USA), pp. 33–42, ACM, 2012.

[29] M. Balman, "Memznet: Memory-mapped zero-copy network channel for moving large datasets over 100gbps network," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, IEEE Computer Society, 2012.

[30] E. He, J. Leigh, O. Yu, and T. Defanti, "Reliable blast udp : predictable high performance bulk data transfer," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 317 – 324, 2002.

[31] Y. Gu and R. L. Grossman, "Udt: Udp-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777 – 1799, 2007. Protocols for Fast, Long-Distance Networks.

[32] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler, "An rdma protocol specification," tech. rep., IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[33] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[34] ESnet, "Linux tuning, http://fasterdata.es.net/host-tuning/linux."

[35] ESnet, "iperf3, http://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf-and-iperf3/."

[36] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 85:1–85:10, ACM, 2013.

[37] "Esnet 100gbps testbed." http://www.es.net/RandD/100g-testbed.

[38] J. Levon and P. Elie, "Oprofile: A system profiler for linux." http://oprofile.sf.net, 2004.