Future Generation Computer Systems 56 (2016) 277–283

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Improving network performance on multicore systems: Impact of core affinities on high throughput flows

Nathan Hanford [a,*], Vishal Ahuja [a], Matthew Farrens [a], Dipak Ghosal [a], Mehmet Balman [b], Eric Pouyoul [b], Brian Tierney [b]

[a] Department of Computer Science, University of California, Davis, CA, United States

[b] Energy Sciences Network, Lawrence Berkeley National Laboratory, Berkeley, CA, United States

HIGHLIGHTS

- Affinity, or core binding, maps processes to cores in a multicore system.
- We characterized the effect of different receiving flow and application affinities.
- We used OProfile as an introspection tool to examine software bottlenecks.
- The location of the end-system bottleneck was dependent on the choice of affinity.
- There are multiple sources of end-system bottlenecks on commodity hardware.

ARTICLE INFO

Article history:
Received 20 February 2015
Received in revised form
15 August 2015
Accepted 11 September 2015
Available online 25 September 2015

Keywords:
Networks
End-system bottleneck
Traffic shaping
GridFTP
Flow control
Congestion avoidance

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 adaptor offloads and performance tuning have received a great deal of attention. Furthermore, several methods of multi-threading 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 various parameter settings do (or do not) work. In this paper, we build on previous research to track down the sources of the end-system bottleneck for high-speed TCP flows. We define protocol processing efficiency to be the amount of system resources (such as CPU and cache) used per unit of achieved throughput (in Gbps). The amount of various system resources consumed are measured using low-level system event counters. In a multicore end-system, affinitization, or core binding, is the decision regarding how the various tasks of network receive process including interrupt, network, and application processing are assigned to the different processor cores. We conclude that affinitization has a significant impact on protocol processing efficiency, and that the performance bottleneck of the network receive process changes significantly with different affinitization.

1. 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

[*] Corresponding author.
E-mail addresses: nhanford@ucdavis.edu (N. Hanford), vahuja@ucdavis.edu (V. Ahuja), mkfarrens@ucdavis.edu (M. Farrens), dghosal@ucdavis.edu (D. Ghosal), mbalman@lbl.gov (M. Balman), lomax@es.net (E. Pouyoul), bltierney@es.net (B. Tierney).

http://dx.doi.org/10.1016/j.future.2015.09.012
0167-739X/© 2015 Elsevier B.V. All rights reserved.

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. The New API for networks (NAPI) in the Linux network receive process allows the NIC to operate in two different contexts. First, there is the interrupt context (usually implemented with coalescing), in which the Network Interface Controller (NIC) interrupts the processor once it has received a certain number of packets. Then, the NIC transmits the packets to the processor via Direct Memory Access (DMA), and the NIC driver and the operating system (OS) kernel continue the protocol processing until the data is ready for the application [2–4]. Second, there is polling, where the kernel polls the NIC to see if there is any network data to receive. If such data exists, the kernel processes the data in accordance with the network and transport layer protocols in order to deliver the data to the Application through the sockets 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 [5,6] with detailed experimentation to stress-test each affinitization combination with a single, high-speed TCP flow. We perform end-system introspection, using tools such as Oprofile to understand the impact that the choice of affinity has on the receive-system efficiency. We conclude that there are three distinct affinitization performance scenarios, and that the performance bottleneck varies drastically within these scenarios. First, there is the scenario where the protocol processing and the application process are on the same core, which causes the processing capabilities of the single core to become the bottleneck. Second, there is the scenario where the flow receiving process and the application receiving the data are placed on different cores within the same socket. This configuration does not experience a performance bottleneck, but results in very high memory controller bandwidth utilization. Third, when the flow receiving process and the receiving application process are placed on different sockets (and thus, different cores), the bottleneck is most likely inter-socket communication.

The remaining part of the paper is organized as follows. In the next section, we discuss and summarize our prior research and give the motivation of this study. In Section 3, we describe the related work. In Section 4, we describe the experimental setup used to conduct this study. We discuss the results in Section 5. Finally, in Section 6, we give the conclusions and outline the future work.

2. Motivation

In a previous study, we conducted research into the effect of affinity on the end-system bottleneck [5], and concluded that affinitization has a significant impact on the end-to-end performance of high-speed flows. However, the study did not identify the precise location of the end-system bottleneck for the different affinitization scenarios and hence a clear understanding of why different affinitization leads to significantly different throughput performance. The goal of this research 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).

There are many valid arguments made in favor of the use of various NIC offloads [7]. 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 [8]. A number of reports provide details of the experiments that have led to their tuning suggestions. However, there is a significant gap in the understanding for these tuning suggestions and offloads.

In this paper, our methods focus on analyzing the parallelism of protocol processing within the end-system. We endeavor to demonstrate the variability of protocol processing efficiency depending on the spatial placement of protocol processing tasks. In this experimental study, we employ *iperf3* [9] 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 [10] 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.

3. Related work

There have been several studies that have evaluated the performance of network I/O in multicore systems [11–14]. A major improvement which is enabled by default in almost all the current kernels is NAPI [15,3]. NAPI is designed to solve the receive livelock problem [14] where most of the CPU cycles are spent in interrupt processing. When the kernel enters a livelock state it spends most of the available cycles in hard and soft interrupt contexts, and consequently, is unable to perform protocol processing of any more incoming data. As a result of the interrupt queues overflowing, packets would eventually be dropped during protocol processing. This would trigger TCP congestion avoidance, but the problem would soon repeat itself. A NAPI-enabled kernel switches to polling mode at high rates to save CPU cycles instead of operating in a purely interrupt-driven mode. Related studies that characterize packet loss and the resulting performance degradation over 10 Gbps Wide-Area Network (WAN) include [4,16,17]. The study in [18] 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. NICs optimized for specific transport protocols have been proposed along these lines [19]. End-system architectural reorganization has also been proposed in [20]. Unfortunately, too few of these changes have found their way into the type of commodity end-systems that have been deployed for the purposes of these tests.

3.1. Protocol processing parallelism

Since end-system processor architectures have been scaling out to multiple cores, rather than up in clock speed, systems designers have faced unique challenges in exploiting this parallelism for network-related processing. There have been several related, but essentially discrete methods to this end.

Receive-side scaling (RSS) is a NIC driver technology which allows multiqueue-enabled NICs to take advantage of the multiprocessing capabilities of an multicore end-system. Specifically, it allows the NIC driver to schedule the interrupt service routine (ISR) on a specific core [21]. This will ensure that the core which received the data is the same core that processes the interrupt. This effectively binds flow and interrupts affinity on most modern operating systems. NICs may implement RSS-compatibility in hardware or the driver. Receive Packet Steering (RPS) [22] is implemented in host kernel. It allows for the selection of the CPU core that will perform protocol processing for an incoming set of packets on a receiving end-system. Receive Flow Steering (RFS) [23] is extension of RPS which adds another layer of control aimed at ensuring that flow and application processing occur on the same logical core.

As commodity multicore systems continue to scale out to more multiple cores, the resource access time between cores is no longer uniform, due to the fact that core-to-core interconnects could no longer be implemented practically supporting uniform propagation speeds. Originally, this affected access to memory most drastically, hence, such systems were referred to as Non-Uniform Memory Access (NUMA) [24]. However, with today's high speed I/O and network devices rivaling memory throughput, this non-uniformity affects network and I/O data movement as well.

Common wisdom is to select cores that share the same lowest cache structure[1] when doing network processing [25,12]. 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.

The Linux *irqbalance* daemon does a round-robin scheduling to distribute interrupt processing load among cores. However, it has adverse effects as shown by [26,27]. We require a more informed approach and we need control over selecting cores for interrupt processing. In our experiments we disable the *irqbalance* daemon.

Pause frames [28] allow Ethernet to implement its own link-layer flow-control to avoid triggering TCP flow or congestion 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 Ethernet frames that are larger than the original IEEE standard 1500-byte Maximum Transmission Unit (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 [29]. 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.

3.2. 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 [10,30–32]. 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 [33–35]. Most of these applications are also capable of leveraging alternative transport-layer protocols based on UDP, such as RBUDP [36] and UDT [37].

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. One contemporary example is Remote Direct Memory Access (RDMA) [38] and its original physical layer, InfiniBand [39]. Typically, due to the addressing and routing requirements of these protocols, they require well-curated networks and have often been relegated to intra-datacenter traffic.

4. Experimental setup

We have employed two servers that are connected back-to-back with a Round-Trip Time (RTT) of less than 1 ms. In our previous experimental study [5] we had used the ESnet Testbed's 95 ms RTT fiber loop. While loop testing is important to analyze end-to-end TCP performance 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 the end-systems.

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 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 [40]. In practice, they must be able to take in large amounts of data as quickly as possible through InfiniBand host-bus adaptors, 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 shown in Fig. 1.

The testbed used was the ESnet 100 Gbps testbed [41], 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.

[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.

[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".

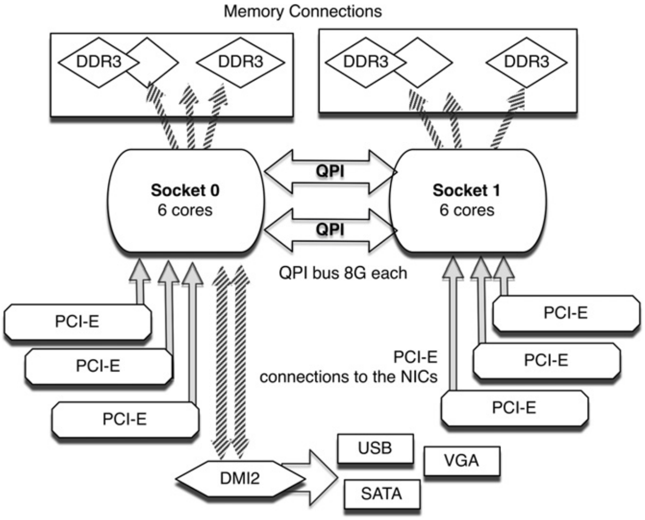


Fig. 1. Block diagram of the end-system I/O architecture.

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 [42]. 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 modified receiving system parameters used in the experiments:

- Flow affinity: Cores 0 through 11
- Application affinity: Cores 0 through 11
- Total number of tests: $12 \times 12 = 144$.

In order to analyze both the performance and overall performance efficiency of the transfers in the different affinitization scenarios, we measured the throughput, the instructions retired (both system-wide and by the kernel) and user processes, the last level cache references, the L2 cache accesses, the memory transactions retired, and the off-core requests. In order for any Oprofile comparisons to be meaningful, the total instructions retired by the processor core needs to be counted [42]. The remaining counter results can be considered subsets of the instructions retired. We are particularly interested in the memory hierarchy for these experiments, because memory access allows us to at least infer how data is being moved around through the processor and which links are used. We also focused on the memory controller as a possible bottleneck, and we found the count of cache transactions at layers 2 and 3 helpful for this, as well as the number of primary memory transactions. (Note that, theoretically, primary memory transactions need only occur when the last level cache is either filled or missed.) Offcore requests were also monitored, but did not appear to pick up the critical low-level PCI data being transmitted between the sockets on the QPI. Unfortunately, those counters have not been made available by our BIOS manufacturer.

Table 1
List of system parameters.

Parameter	Value
RTT	<1 ms
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

4.1. List of system parameters

A summary of the experimental environment is listed in Table 1. The sending and receiving end systems were identical.

4.2. 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.

5. Results

Both our current and previous work [5] 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 over-subscribed 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.

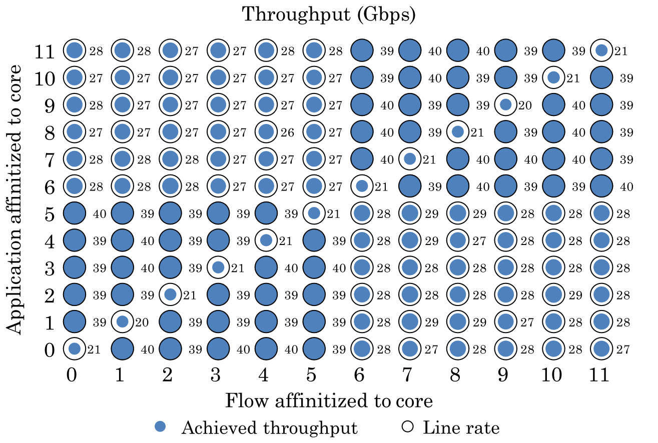


Fig. 2. The throughput results of all 144 tests arranged in a matrix by affinity choice. The empty circles represent the line rate of the NICs (40 Gbps). The size of the filled-in circles corresponds to the achieved throughput.

5.1. *Interpretation of results*

In order to convey the results of the 144 tests effectively, we elected to create a matrix as seen in Fig. 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 Fig. 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.

5.2. *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.

Fig. 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 Fig. 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.

5.3. *Impact of the 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 Figs. 4–6. 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 (Fig. 3).

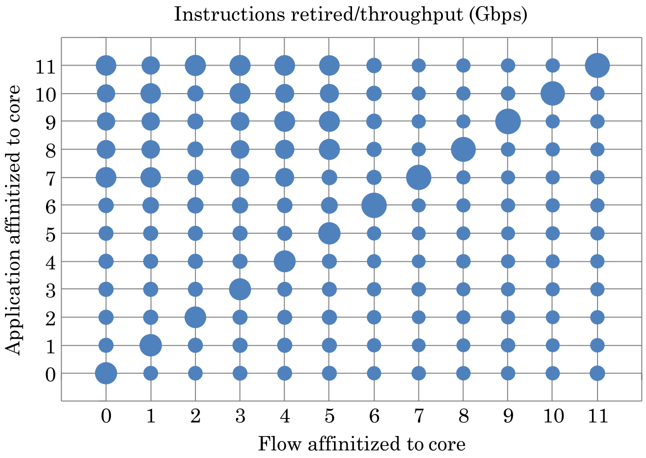


Fig. 3. The processing efficiency of the 144 tests; larger circles represent poorer efficiency.

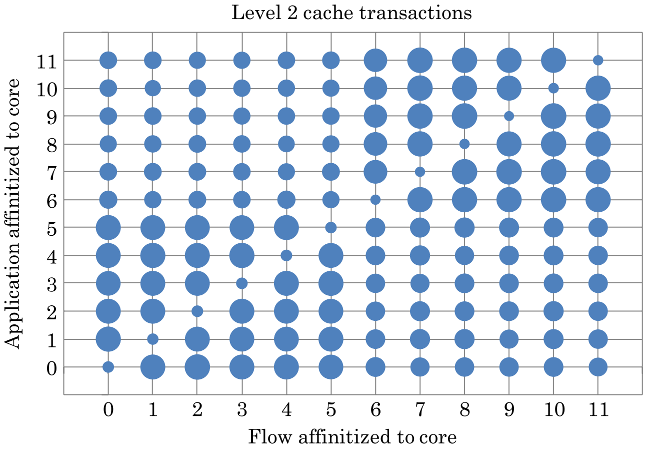


Fig. 4. The fraction of instructions retired/throughput dedicated to Level 2 cache transactions (as measured by counter l2_trans) for all 144 tests.

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.

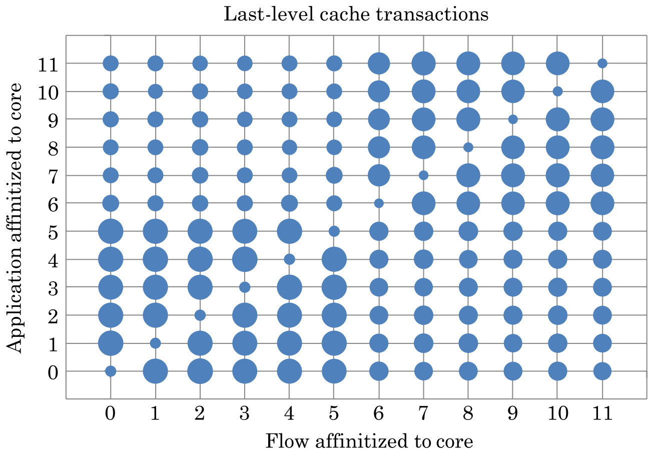


Fig. 5. The fraction of instructions retired/throughput dedicated to Last level cache transactions (as measured by counter LLC_TRANS) for all 144 tests.

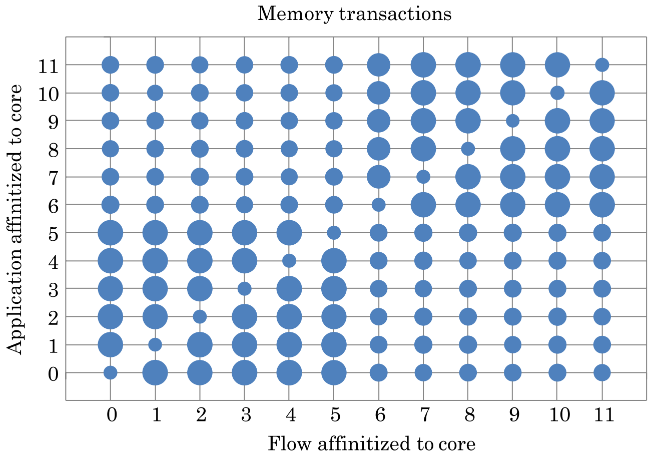


Fig. 6. The fraction of instructions retired/throughput dedicated to memory transactions (as measured by counter mem_trans_retired) for all 144 tests.

5.4. 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.

6. 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 [20], 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 switches between interrupt coalescing and polling 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 [25].

Acknowledgments

This research used resources of the ESnet Testbed, which was 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 grants CNS-0917315 and CNS-1528087.

References

[1] G. Keiser, Optical Fiber Communications, John Wiley & Sons, Inc, 2003.
[2] C. Benvenuti, Understanding Linux Network Internals, O'Reilly Media, 2005.
[3] W. Wu, M. Crawford, M. Bowden, The performance analysis of linux networking packet receiving, Comput. Commun. 30 (5) (2007) 1044–1057. Advances in Computer Communications Networks.
[4] W. Wu, M. Crawford, Potential performance bottleneck in linux tcp, Int. J. Commun. Syst. 20 (11) (2007) 1263–1283.
[5] N. Hanford, V. Ahuja, M. Balman, M.K. Farrens, D. Ghosal, E. Pouyoul, 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, ACM, New York, NY, USA, 2013, pp. 1:1–1:10.
[6] N. Hanford, V. Ahuja, M. Balman, M.K. Farrens, D. Ghosal, E. Pouyoul, B. Tierney, Impact of the end-system and affinities on the throughput of high-speed flows, 2014.
[7] A. Currid, Tcp offload to the rescue, Queue 2 (2004) 58–65.
[8] ESnet, Linux tuning, http://fasterdata.es.net/host-tuning/linux.
[9] ESnet, iperf3, http://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf-and-iperf3/.
[10] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster, The globus striped gridftp framework and server, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 2005, p. 54.
[11] A. Pande, J. Zambreno, Efficient translation of algorithmic kernels on large-scale multi-cores, in: International Conference on Computational Science and Engineering, 2009, vol. 2, CSE’09, IEEE, 2009, pp. 915–920.
[12] A. Foong, J. Fung, D. Newell, An in-depth analysis of the impact of processor affinity on network performance, in: 12th IEEE International Conference on Networks, 2004. (ICON 2004). vol. 1, Proceedings, vol. 1, Nov 2004, pp. 244–250.
[13] M. Faulkner, A. Brampton, S. Pink, Evaluating the performance of network protocol processing on multi-core systems, in: International Conference on Advanced Information Networking and Applications, 2009. AINA ’09, May 2009, pp. 16–23.
[14] J. Mogul, K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel, ACM Trans. Comput. Syst. 15 (3) (1997) 217–252.
[15] J. Salim, When napi comes to town, in Linux 2005 Conf, 2005.
[16] T. Marian, D. Freedman, K. Birman, H. Weatherspoon, Empirical characterization of uncongested optical lambda networks and 10gbe commodity endpoints, in: 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2010, pp. 575–584.
[17] T. Marian, Operating systems abstractions for software packet processing in datacenters (Ph.D. thesis), Cornell University, 2011.

[18] S. Larsen, P. Sarangam, R. Huggahalli, S. Kulkarni, Architectural breakdown of end-to-end latency in a tcp/ip network, Int. J. Parallel Program. 37 (6) (2009) 556–571.

[19] W. Wu, P. DeMar, M. Crawford, A transport-friendly nic for multicore/multiprocessor systems, IEEE Trans. Parallel Distrib. Syst. 23 (4) (2012) 607–615.

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

[21] B.E. Veal, A. Foong, Adaptive receive side scaling, June 29 2007. US Patent App. 11/771,250.

[22] T. Herbert, rps: receive packet steering, September 2010. http://lwn.net/Articles/361440/.

[23] T. Herbert, rfs: receive flow steering, September 2010. http://lwn.net/Articles/381955/.

[24] J.L. Hennessy, D.A. Patterson, Computer Architecture — A Quantitative Approach, fifth ed., Morgan Kaufmann, 2012.

[25] V. Ahuja, M. Farrens, 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, ACM, 2012, pp. 39–48.

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

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

[28] B. Weller, 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.

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

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

[31] M. Balman, 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.

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

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

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

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

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

[37] Y. Gu, R.L. Grossman, Udt: Udp-based data transfer for high-speed wide area networks, Comput. Netw. 51 (7) (2007) 1777–1799. Protocols for Fast, Long-Distance Networks.

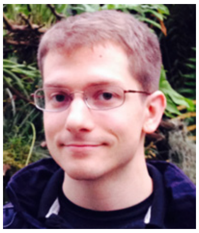
[38] R. Recio, P. Culley, D. Garcia, J. Hilland, B. Metzler, A Remote Direct Memory Access Protocol Specification, tech. rep., IETF RFC 5040, 2007, http://dx.doi.org/10.17487/RFC5040.

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

[40] E. Dart, L. Rotman, B. Tierney, M. Hester, 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, ACM, New York, NY, USA, 2013, pp. 85:1–85:10.

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

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



Nathan Hanford is a Ph.D. candidate and graduate student researcher for the Graduate Group in Computer Science at the University of California, Davis. His research interests involve leveraging commodity end-system hardware over very high-speed interconnects and networks. To this end, he is interested in system-level software and lightweight middleware which enable meaningful end-system and node-level awareness. He is also interested in the HPC and cloud-computing applications of commodity hardware, and its implications for SDN. He has previously worked as a summer student in ESnet's Advanced Network Technologies Group, where he conducted research on Data Transfer Nodes for moving large amounts of scientific data. He holds a Bachelor of Science in Engineering in Computer Science and Engineering from the University of Connecticut.

Vishal Ahuja is a Senior Data Engineer at Target. Previously he worked as a Post-doctoral Fellow for the Department of Computer Science at the University of California, Davis. His interests focus on creatively leveraging emerging commodity hardware towards efficient and predictable distributed systems. Towards that end, he has developed several tools and optimizations for end-to-end data transfers along with message broker implementations. He holds a Ph.D. in Computer Science from the University of California, Davis.

Matthew K. Farrens is a Professor at the University of California, Davis. He is interested in all aspects of computer architecture, but primarily in the architecture and design of high-performance single-chip processors with an emphasis on the interconnection/communication layer. He is also interested in high-speed scientific processing, in particular in exploring issues related to the memory system, and in instruction level parallelism. He holds a Ph.D. in Electrical and Computer Engineering from the University of Wisconsin, Madison.

Dipak Ghosal is a Professor at the University of California, Davis. His research interests include high-speed networks, wireless networks, vehicular ad hoc networks, parallel and distributed systems, timing channels, and the performance evaluation of computer and communication systems. He holds a Ph.D. in Computer Science from the University of Louisiana, Lafayette.

Mehmet Balman is a Senior Performance Engineer at VMware Inc., working on hyper-converged storage and virtualized solutions. He is also a Guest Scientist at Lawrence Berkeley National Laboratory. He has previously worked as a Researcher/Engineer in the Computational Research Division at Lawrence Berkeley National Laboratory, where his work particularly dealt with performance problems in high-bandwidth networks, efficient data transfer mechanisms and data streaming, high-performance network protocols, network virtualization, and data transfer scheduling for large-scale applications. He holds a Ph.D. in Computer Science from Louisiana State University.

Eric Pouyoul is a Senior System Engineer in ESnet's Advanced Network Technologies Group at Lawrence Berkeley National Laboratory (LBNL). His interests include all aspects of high performance big data movement, networking, hardware, software and distributed systems. He has been ESnet Lead for designing Data Transfer Nodes (DTN) as defined in the Science DMZ architecture as well as ESnet's work in Software Defined Networking (OpenFlow). He joined ESnet in 2009 and his 25 years prior experience includes real-time operating system, supercomputing and distributed systems.

Brian L. Tierney is a Staff Scientist and Group Leader of the ESnet Advanced Network Technologies Group at Lawrence Berkeley National Laboratory, and is PI of ESnet's 100G Network Testbed Project. His research interests include high-performance networking and network protocols; distributed system performance monitoring and analysis; network tuning issues; and the application of distributed computing to problems in science and engineering. He has been the PI for several DOE research projects in network and Grid monitoring systems for data intensive distributed computing. He was the Principal Designer of the Distributed Parallel Storage System (DPSS), where many of the ideas for GridFTP originated. He also designed the first version of the NetLogger Toolkit, and worked on the Bro Intrusion Detection System. He was co-chair of the 2nd International Workshop on Protocols for Long Distance Networks (PFLDnet) in 2004. He holds an M.S. in Computer Science from San Francisco State University, and a B.A. in Physics from the University of Iowa.