A System-Level Optimization Framework for High-Performance Networking

Thomas M. Benson
Sensors and Electromagnetic Applications Laboratory
Georgia Tech Research Institute
Atlanta, Georgia, USA

Abstract—**Processing data in a streaming fashion is involved in many applications, including radar, electro-optical and infrared imaging, and other scenarios in which real-time data is acquired from sensors. While the difficulty of processing such data in real-time or in an otherwise timely fashion is the topic of much research, transferring the data between machines or devices is also a key challenge. This work investigates the utilization of commodity high-speed networking products – in particular, 40 gigabit Ethernet – for supporting high-speed data transfer in streaming applications. For such systems, optimizing the system configuration to support data transfers approaching line rate is critical. This work demonstrates the use of an optimization framework to explore the impact of various system-level optimization settings, including kernel module settings, network interface driver-level settings, kernel buffer sizes, thread and CPU core scheduling, non-uniform memory access (NUMA) scheduling, interrupt handling, and CPU power-saving state management.**

I. Introduction

In many distributed systems, such as those developed for high-speed sensor processing, high-speed networking performance is critical. While application design in such scenarios is important, a significant amount of system-level optimization is also required, including network driver settings, system kernel configuration, interrupt processing, and CPU performance modes. For example, many high-performance networking hardware vendors publish performance tuning guides or support webpages that focus on system configuration for achieving high networking performance [1], [2].

Some related efforts focus on improving networking performance or packet processing rates by modifying how the system processes network traffic, typically via kernel bypass, specialized kernel modules, etc. For example, `netmap` [3] introduces an API for high-speed packet processing that includes new networking data structures and associated updates to network drivers. Other related research, such as RBUDP [4], considers bulk data transfer via high-speed networks over long distances ("long fat networks", or LFNs). Researchers have also previously addressed autotuning, often in the context of optimizing application performance on a specific hardware architecture, as is done with the Automatically Tuned Linear Algebra Software (ATLAS) [5]. This work, on the other hand, aims to optimize network performance achieved using traditional network transports, rather than requiring specialized kernel modules, new APIs, or direct hardware access. Autotuning provides guidance for specifying system parameters and does not modify the actual application under test.

Specific networking performance goals depend upon the application in question. For example, the goal may be to minimize latency, maximize throughput, or minimize packet loss. This work focuses primarily on the nominal design space for processing high-throughput streaming sensor data and thus point-to-point or switch-local connections, rather than longer paths through routers or LFNs, will be most common. Because interfacing with sensors often involves interfacing with a field programmable gate array (FPGA) on the sensor side, and FPGAs commonly use the user datagram protocol (UDP) to communicate data, this work focuses on UDP performance in addition to the otherwise more typical transmission control protocol (TCP). UDP does not include reliability mechanisms and thus may involve data loss; therefore, UDP data loss rates are considered in addition to throughput measurements.

Benchmarking is, in general, a challenging task, and network benchmarking is no exception. Many parameters and configuration options exist that can impact network performance, and many of these parameters are poorly documented. Furthermore, the application implementation can vary, and it interacts with the aforementioned system configuration. The approach presented in this paper is to hold the application under test constant and develop an autotuning framework to evaluate the impact of the system configuration. For the purposes of this work, the application under test is the popular `netperf` [6] utility. `netperf` is unique in that it is targeted toward maximizing measured network bandwidth and does not apply any processing to transferred data as would be typical for other applications. Thus, `netperf` is likely to register higher performance than a sensor processing application and is a reasonable upper bound for application performance. Ultimately, the autotuning framework provides a systematic approach for determining which parameters most significantly impact performance and tuning a system to maximize network performance associated with a given application.

II. Hardware Setup

The present work demonstrates a specific experimental setup, although the framework itself can be ported to other systems. The experimental setup comprises two servers running CentOS 6.5 with dual 40 gigabit Ethernet (40 GbE) network interface cards (NICs) and a 40 GbE switch; Table I provides the system specifications. Although the approach is general, the autotuning methodology modifies features and settings that may be specific to certain hardware or software versions.

III. Autotuning Methodology

The autotuning framework implements the approach shown in Algorithm 1. Because the autotuning framework configures tuning profiles at run-time, the configuration elements under test must also be configured at run-time. Alternatively, settings

978-1-4799-6233-4/14/$31.00 © 2014 IEEE

TABLE I. EXPERIMENTAL SYSTEM CONFIGURATION. THE SYSTEM INCLUDES TWO IDENTICAL NODES (WITH RESPECT TO THE VALUES GIVEN BELOW) AND ONE SWITCH.

Parameter	Value
Processors	Dual-socket E5-2650 v2
Memory (RAM)	128 GiB (8 16 GiB DIMMs)
Operating System	CentOS 6.5
Network Card (NIC)	Mellanox ConnectX-3 (40 GbE mode)
NIC Driver Version	2.1.11
NIC Firmware Version	2.30.8000
Switch	Mellanox SX-1024

that require a reboot (e.g., those within the BIOS) require running the autotuning framework before and after manually adjusting the setting of interest, which would quickly become cumbersome. Fortunately, Linux supports modifying the vast majority of system parameters at run-time without requiring a system reboot. The following section describes the tunable parameters currently adjusted by the framework, including details for modifying the parameters at run-time.

Algorithm 1 General network autotuning framework.

```
1: Configure both systems to a baseline profile
2: for all Test profiles $t_k$ do
3:     Configure all systems for test profile $t_k$
4:     Record pre-test system statistics and configuration
5:     Run benchmark using the application under test
6:     Record post-test system statistics and configuration
7:     Restore configuration to baseline profile
8: end for
```

The current framework is implemented in Python. The mechanism for defining test profiles is to specify a series of 3-tuples where the entries indicate (1) the steps to be taken to adjust the baseline profile to the test profile, (2) the steps to be taken to revert the profile to the baseline profile, and (3) the name assigned to the test for later record-keeping. The steps for entries (1) and (2) are represented as strings invoked using `eval`. System statistics recorded before and after the test document both the configuration of the systems as well as dropped packet counts, interrupts handled per core, etc. These statistics allow investigating issues such as high dropped packet rates forensically after the conclusion of a test run.

The length of each benchmark execution directly impacts the total system tuning time and thus must be chosen to be both extensive enough to support drawing suitable conclusions while also allowing the full suite of tests to conclude in a reasonable amount of time. As an optimization, the baseline profile is constructed based on experience and previous testing such that it is expected to be nearly optimal. Therefore, each autotuning profile only modifies a single parameter or a small group of parameters to isolate the impact of the parameter rather than exhaustively testing all possibilities. This approach may only identify a local optimum rather than a global optimum. The approach could be easily extended to allow identifying sets of parameters that are expected to be correlated (i.e., the parameters need to be tuned concurrently to obtain optimal results) and these restricted sets could then be explored exhaustively. Of course, the full parameter space could be explored exhaustively as well (or, equivalently, all settings could be assumed to be correlated), but doing so would require a significant amount of time.

IV. TUNABLE PARAMETERS

Identifying system parameters that may impact networking performance is the first step in tuning performance. Once identified, we determine how to modify the candidate configuration parameter to incorporate it into the autotuning framework. The present work considers only run-time configurable parameters as they can be more naturally included in the framework. The following sections introduce the configurable parameters considered for the remainder of the paper.

A. NIC Kernel Module Parameters

Some network interface card (NIC) parameters must be specified at kernel module load time; these parameters vary by network card. In some cases the vendor documents the kernel module parameters, but they can also often be discovered by running `modinfo -p module_name` where `module_name` is the module name reported by `lsmod` in Linux. For the Mellanox drivers, we include the settings of `enable_sys_tune` and `high_speed_steer` in the autotuning framework. The method for modifying these values at run-time is to unload the kernel module, update the associated file in `/etc/modprobe.d/` to include the preferred configuration, and finally reload the module. The parameter settings are exposed through `/sys/module/module_name/parameters/` and can thus be queried to record configuration for record-keeping.

B. NIC `ethtool` Parameters

The `ethtool` command manages a large collection of NIC parameters. While `ethtool` has relatively standardized syntax between network cards, the parameters that can be modified and the specific values to which they can be set depend upon the card. However, high-end network adapters typically offer similar support (e.g., the receive ring size can often be increased, although the maximum varies with the card and driver). The `ethtool` settings can be easily modified and queried using the `ethtool` command; specific `ethtool` options of interest are described in the following sections.

1) Receive/Transmit Ring Size: The network driver ring size specifies how many frames can be buffered for transmission or reception; on receive, the frames are buffered pending being copied into kernel space. If the ring buffer is full and more packets arrive, then packets are dropped; thus, large receive ring buffers can decrease the number of dropped packets. The current ring size and the maximum possible value can be checked using `ethtool -g <interface>` and can also be modified via `ethtool -G`. The Mellanox ConnectX-3 card has default settings for both the receive and transmit ring size of 1024 with a maximum of 8192. Packets dropped by the NIC due to full ring buffers are typically reported in the `rx_dropped` and `tx_dropped` fields available via the NIC statistics (`ethtool -S`).

2) Interrupt Coalescence: Interrupts notify the kernel that action is needed, such as copying received packets from the receive ring into kernel space. Raising an interrupt for each packet can yield a high interrupt processing burden for the kernel and the associated CPU cores, so interrupt coalescence can be used to generate a single interrupt for multiple packets. Interrupt coalescence can be defined via `ethtool -C` in terms of the number of microseconds after having received a packet until an interrupt is generated, during which time more packets may have arrived. Furthermore, adaptive coalescing can be enabled, which attempts to adapt to traffic patterns.

3) Ethernet Flow Control: Ethernet flow control in the form of PAUSE packets can be enabled or disabled via `ethtool -A`. PAUSE packets can be sent by an overwhelmed device – either a NIC or switch – in order to pause the sender while the receiver processes packets. As a result, flow control can prevent data loss on the receiver side by pausing the sender when the receiver's receive buffer is nearly full. Note that many FPGA intellectual property (IP) cores that support Ethernet do not support PAUSE packets and thus PAUSE requests made by the receiver will be ignored by the FPGA Ethernet IP core.

C. `ifconfig` Parameters

Several parameters can be modified via the `ifconfig` utility, including the maximum transmission unit (MTU) and transmission queue length settings. The MTU defines the maximum payload that can be transmitted via Ethernet (i.e., the MTU limit includes the IP header, but not the Ethernet frame header); larger MTUs allow more data to be transferred per packet, which reduces overhead associated with interrupt processing and other steps. However, all components along the network path must support the larger MTU. Values exceeding 1500 bytes for the MTU are typically known as jumbo frames; it is conventional for the phrase jumbo frame to imply an MTU of at least 9000 bytes, but some equipment advertises jumbo frame support with a maximum MTU of less than 9000 bytes, so actual supported MTU sizes should be confirmed.

D. Linux Kernel Parameters

The Linux kernel exposes many parameters that impact the processing of UDP and TCP data. For example, default, minimum, and maximum buffer sizes can be specified for UDP and TCP socket connections. The trade-off for such settings is that kernel memory consumption will increase for each socket, which can be problematic if many sockets will be opened and serviced. However, the current focus is on largely isolated systems designed for processing a small number of high-speed data streams, so defining large buffer sizes is a reasonable optimization. These parameters can be inspected or modified via the `proc` file system or via the `sysctl` command. For the purposes of this work, each parameter is assigned only two sets of values to maintain a manageable optimization space; in particular, "small" and "large" buffer sizes are defined and shown in Table II. The values specified for "large" are likely larger than needed and could be decreased with minimal or no impact on performance; however, they are defined to be very large to maintain only two settings while minimizing the likelihood that even larger values would be beneficial.

TABLE II. KERNEL NETWORKING BUFFER PARAMETER SETTINGS. SIZES REPORTED ARE IN UNITS OF BYTES. ALL OF THE PARAMETERS ARE IN `/PROC/NET` AND THUS HAVE `NET` SUFFIXES IN `SYSCTL`.

Parameter	Small	Large
core.wmem_max	262144	67108864
core.rmem_max	262144	67108864
core.wmem_default	262144	67108864
core.rmem_default	262144	67108864
core.optmem_max	20480	67108864
ipv4.tcp_mem	578181,770910,1156362	12385440,16513920,24770880
ipv4.tcp_rmem	4096,87380,6291456	4096,87380,16777216
ipv4.tcp_wmem	4096,16384,4194304	4096,65536,16777216

Many other settings exist for tuning TCP performance (e.g., `net.ipv4.tcp_timestamps`, `net.ipv4.tcp_sack`, etc.), but such settings are not included in this analysis for simplicity. However, the optimization framework presented in this work can easily accommodate such parameters.

E. IRQ Steering and Routing

Handling of interrupt requests (IRQs) is critical for high-performance networking because IRQs represent the typical mechanism by which the NIC notifies the kernel that data is available. Fully optimizing IRQ handling is a significant effort and thus is rather simplified for this work. In general, the NIC asserts interrupts when packets are available to retrieve and some CPU core will service the interrupt and transfer the packet from the receive ring into a kernel buffer. High-speed network nodes receive (or send) many packets per second and servicing IRQs can become a bottleneck. In extreme cases, interrupts can overwhelm the system and cause a situation known as receive livelock [7], which prevents the system from making forward progress.

There are many mechanisms for handling high IRQ rates. Higher-end network cards feature multiple receive queues and can assert different IRQs for each queue, which can then be load-balanced across available cores. This load balancing can be dynamic using, e.g., the `irqbalance` service, or IRQs can be statically configured to be delivered to specific cores via the `/proc/irq` file system. Processing interrupts on a single core provides cache benefits: the interrupt servicing routine is likely to be cache-hot (i.e., the servicing routine code and data structures are likely to be in the instruction and data caches). Furthermore, if the packets correspond to a single connection, then connection-related data structures are likely to be in cache on the core that processed the last packet. Thus, many NIC drivers apply receive hashing on the connection tuple (source and destination IP addresses and ports) to determine into which receive queue the packet should be stored and correspondingly what interrupt should be raised. The implication is that despite having multiple receive queues, all packets for a fixed connection (or, in the case of UDP, all packets with the same IP address and port tuples) will be forced into a single receive queue and thus result in the same IRQ being asserted repeatedly. The IRQs used for previous packets can be seen in either the `ethtool` statistics output or in the `/proc/interrupts` file.

While balancing packets and IRQs among CPU cores provides some benefits, it could also be problematic if those cores are dedicated for other purposes. As a result of all of the above, we primarily consider the scenario where all IRQs corresponding to an interface statically map to a single core on which no other user processes are explicitly run. This approach keeps the core cache-hot and also produces more repeatable results if subsequent test runs use different ports, which could then map to different IRQs and cores for other routing approaches.

F. CPU Performance Modes

Modern CPUs have many performance modes designed for either power savings or temporary boosts in performance. In particular, modern Intel CPUs feature P-states and C-states, which are both described in more detail in relation to Red Hat Enterprise Linux (RHEL) in the RHEL 6.5 Power

Management Guide [8]. P-states correspond to varying clock frequency and voltage pairs with correspondingly lower power consumption at "slower" P-states (P0 is the "fastest" P-state). Scaling governors control P-state behavior. The active scaling governor can be specified by writing the desired governor name to `cpuX/cpufreq/scaling_governor` in `/sys/devices/system/cpu/` where `X` is the CPU index. This work considers the `ondemand` and `performance` governors, which correspond to dynamically adjusting the P-state based on load and staying in state P0, respectively.

C-states, on the other hand, define the "depth" of sleep states allowed for CPU cores. Although C-states can be indirectly controlled via writing to `/dev/cpu_dma_latency`, this control is currently at the granularity of all cores and thus not benchmarked in this work due to the large increase in power consumption associated with preventing all cores from entering sleep states. However, as tools evolve to offer more fine-grained control over C-states, tuning C-states may become an important optimization.

G. Turbo Boosting

Although not explicitly controlled via the autotuning framework, the E5-2650 v2 CPUs include support for turbo boosting that increases the clock frequency beyond the nominal 2.6 GHz for CPU cores under load when thermal overhead is available. The actual frequencies being used in a given time interval can be inspected with the `turbostat` tool and we have observed that the cores running `netperf` typically run between 3 and 3.4 GHz, with the higher clock frequencies being used when power management is permitted to lower the clocks and voltage of the other cores effectively.

Furthermore, turbo boosting does seem to substantially increase performance for UDP with the consequence that the autotuning framework can yield the best results in cases that do not typically correspond to higher performance, such as using the `ondemand` governor instead of the `performance` governor. However, this may not be a realistic benchmarking scenario because in a nominal streaming processing case, the other CPU cores will be busy processing data. Therefore, we run the autotuning framework in two scenarios. In the first, no additional processes (beyond those needed by the system) are run on the cores not running `netperf` or handling IRQs; in the second, a busy-wait loop is run on the cores not being used for `netperf`. The "busy" scenario thus decreases the thermal overhead available for turbo boosting the cores on which the benchmark is running, which is likely a more realistic scenario than having all of the non-benchmarked CPU cores in the system idle.

H. CPU and NUMA affinity

The particular core on which a process runs can be important for several reasons. For one, the core should ideally be otherwise unused when running a benchmarking application. Furthermore, most modern multi-socket systems feature non-uniform memory access (NUMA) whereby memory is "local" to a specific socket and accesses to that memory from other sockets have higher latency. Thus, ideally the core running the benchmark should be NUMA-local to the data associated with the benchmark. In the case of communicating with an attached PCIe device, such as a network card or GPU, the benchmark core should also be NUMA-local to that device, as should the interrupt processing core. Finally, in the case of processing network traffic, the benchmark application can either be on the same core processing interrupts or another core on the same socket; we use the latter configuration for these tests, although at lower data rates a single CPU core can handle both tasks.

V. Results and Conclusions

The results included herein are not meant to be exhaustive, but rather to highlight some of the more interesting trends observed throughout the course of this work. Table III contains the baseline configuration parameters used by the autotuning framework; these values have already been relatively optimized based on previous experience. The `netperf` utility comprises the application under test and it is executed ten times for twenty seconds each execution for both UDP and TCP. For UDP, the transmit side initially presented the primary bottleneck and thus the test results included in this section run two transmit processes on one node with each transmit process sending data to the same receive process. The TCP tests utilize only a single process for each of transmit and receive.

TABLE III. Baseline system configuration used by the autotuning framework.

Parameter	Value
Kernel module option `enable_sys_tune`	0 (off)
Kernel module option `high_rate_steer`	1 (on)
`ethtool` interrupt coalescence	Adaptive coalescence enabled
`ethtool` receive/transmit ring size	8192
`ethtool` pause configuration	Enabled
`ifconfig` txqueuelen	10000
`ifconfig` MTU	9000
`net.core.netdev_max_backlog`	250000
`sysctl` buffer sizes	Large (see Table II)
IRQ routing	Map NIC IRQs to 2nd core of NUMA-local socket
CPU scaling governor (for P-states)	performance
CPU core running netserver for receive	3rd core of NUMA-local socket

Figures 1 and 2 depict TCP and UDP results, respectively. As described in Section IV-G, the tests are run with the hosting systems in both an "idle" and "busy" scenario to determine the impact of turbo boosting and other power management features. The results are shown for twelve test profiles (P1-P12) in addition to the baseline profile (P0). Each profile modifies a single setting or a small set of configuration items relative to the baseline profile; Table IV describes the test profiles.

Several conclusions can be drawn from these results. First, test profile P1 indicates that a large MTU is critical for high-performance for both TCP and UDP. Disabling jumbo frame support and using the default 1500 byte MTU decreased TCP and UDP throughput to approximately 15 Gbps and 8 Gbps, respectively. Furthermore, the UDP packet loss rate for test profile P1 exceeded 50%. Second, test profile P3 indicates that "pinning" the benchmark to a specific core (i.e., explicitly placing the process on a core via `numactl` or similar) is important for high performance – not doing so decreased performance by a factor of nearly two.

Third, it is clear that the benchmarks perform very differently in the "idle" and "busy" cases where the CPU cores unrelated to the benchmark have light and heavy processing loads, respectively. For UDP, packet loss increases substantially for the "busy" case, which seems reasonable because the lack of turbo boost on the CPU cores responsible for receiving data and processing interrupts significantly decreases the clock

TABLE IV. TEST PROFILES IN TERMS OF DEVIATION FROM THE BASELINE DESCRIBED IN TABLE III.

Profile	Difference relative to baseline
P0	none (baseline profile)
P1	MTU set to 1500
P2	scaling governor set to `ondemand`
P3	`netperf` not pinned to a specific core
P4	kernel module option `high_rate_steer` disabled
P5	kernel module option `enable_sys_tune` enabled
P6	coalescence disabled via `ethtool`
P7	Ethernet pause (flow control) disabled via `ethtool`
P8	Receive/transmit ring size set to 1024
P9	Map NIC IRQs to all cores on NUMA-local socket instead of single core
P10	`netdev_max_backlog` set to 1000
P11	`sysctl` buffer sizes set to small (c.f. Table II)
P12	`txqueuelen` set to 1000

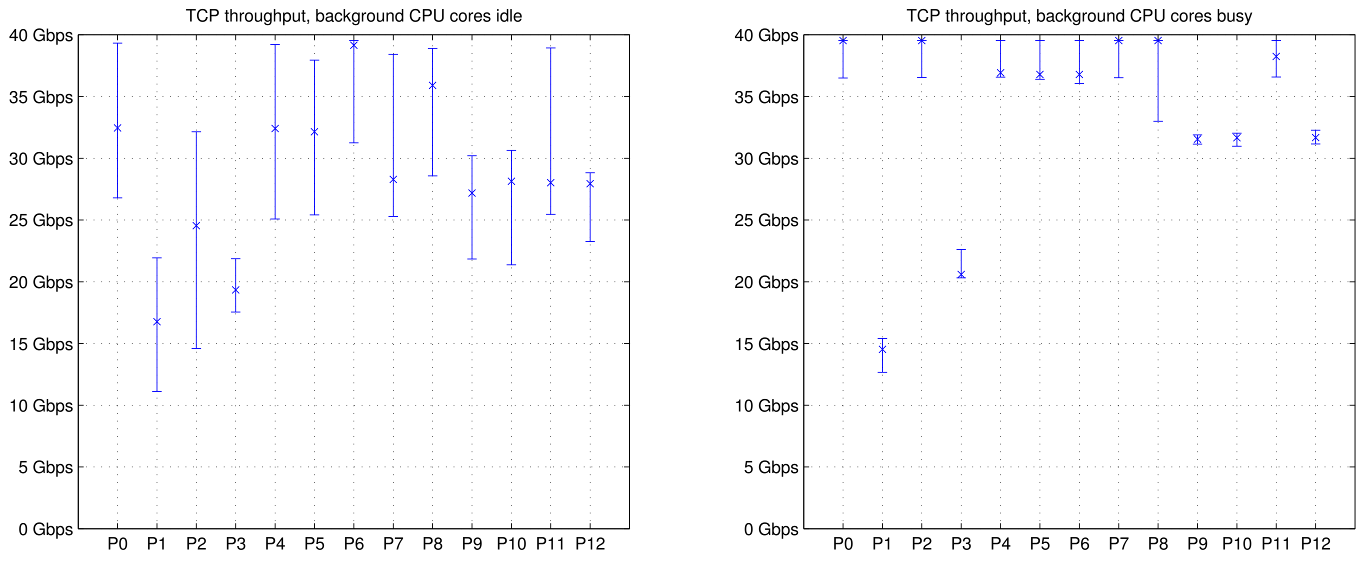

Fig. 1. TCP throughput results as a function of test profile. The error bars indicate the minimum and maximum throughput for ten tests; the cross indicates the median throughput. In the left plot, the CPU cores not being used for the test itself were idle whereas in the right plot they were running a busy wait operation.

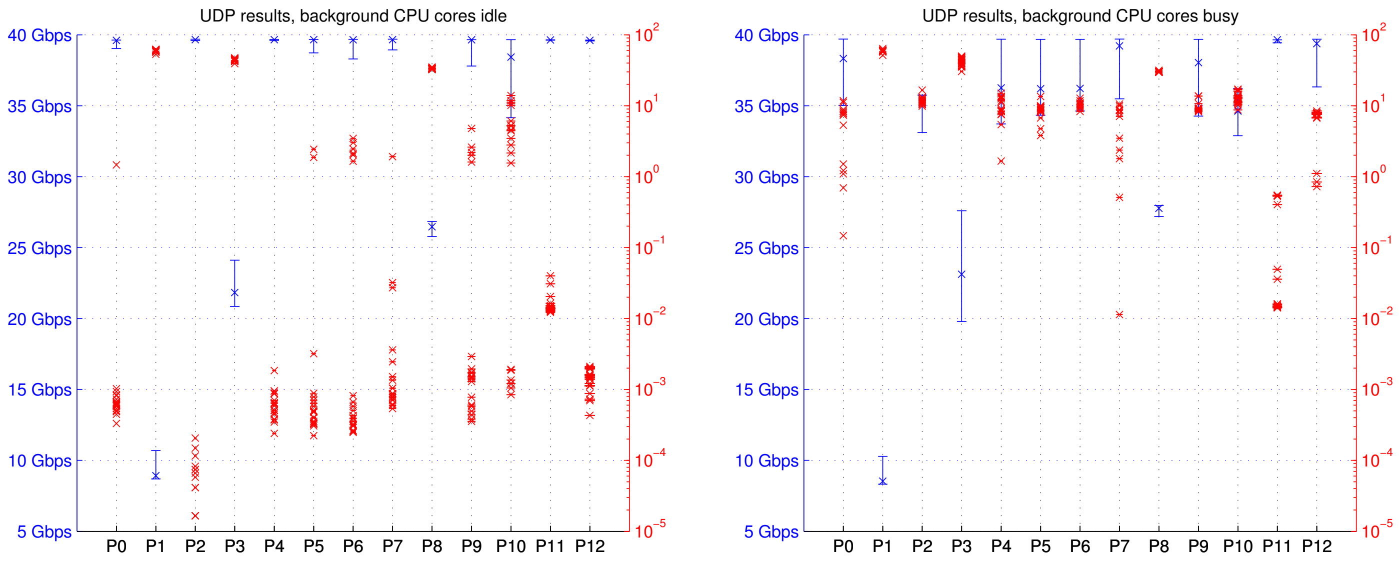

Fig. 2. UDP results as a function of test profile. Blue indicates throughput and red indicates packet loss as a percentage on a log10 axis. The error bars indicate the minimum and maximum throughput for ten tests; the cross indicates the median throughput. For packet loss, discrete points are included rather than error bars due to the presence of some zeros. In the left plot, the CPU cores not being used for the test itself were idle whereas in the right plot they were running a busy wait operation.

frequency associated with those operations. Furthermore, note that test profile P2 for the idle case indicates that using the `ondemand` scaling governor in the case of a largely idle system, which will enable very aggressive turbo boosting, decreases UDP packet loss. These observations imply that UDP performance is strongly dependent upon the clock frequency of the CPU cores processing the data and interrupts. On the other hand, TCP performance generally increased and became more predictable on an otherwise busy system. While this seems contradictory, latency heavily impacts TCP throughput and TCP is less CPU-bound than UDP due to network card offload engines. Thus, it is possible that a busy system will prevent components from going into power-saving sleep states as frequently, which could potentially reduce latency and thus benefit the TCP congestion control mechanisms. However, this is a speculative explanation; the TCP behavior on an otherwise busy system warrants further investigation.

Fourth, test profile P9 indicates that tightly controlling interrupt servicing behavior is important for both TCP and UDP. In particular, allowing IRQs to be delivered to any core on the socket that is NUMA-local to the network interface card decreased performance relative to always pushing the IRQs to a single core. Load balancing IRQs between a few cores may be required when the interrupts overwhelm a single core, but that did not seem to be a bottleneck for the present work.

Fifth, the `netdev_max_backlog` and `txqueuelen` settings have a strong impact on TCP performance as can be seen in test profiles P10 and P12, respectively. At default settings, each of those parameters degrade performance by over 5 Gbps relative to the baseline. The former value is also important for UDP performance, but `txqueuelen` is less so. UDP performance in these tests has been receive-side limited, so it is reasonable that the transmit queue length has little effect on total performance.

Sixth, as evidenced by test profile P8, increasing the receive ring size is critical for UDP, but has little impact for TCP. In particular, reverting the receive ring size to the default setting increased the UDP dropped packet rate to approximately 50% and decreased the throughput by over 10 Gbps.

Finally, while relatively high throughput using UDP is achieved for many of the test profiles, the packet loss rates vary by several orders of magnitude and generally are quite high. Packet loss could be addressed by adding a reliability layer, but adding such a layer may significantly impact performance. Furthermore, in some cases, it may be difficult or impossible to modify either the sender or receiver, so adding a reliability layer may not be an option. Thus, if very low data loss is required, then significant tuning will likely be necessary and the throughput may need to be decreased relative to peak in order to reduce the packet loss rates to an acceptable level. On the other hand, TCP offers reliable data transfer and exhibits excellent performance in many of the test profiles.

Although not included as a separate test profile in this work, system default configurations typically differ substantially from those presented herein as the baseline configuration. In general, the "low" values for the configuration items correspond to typical default settings. For example, the initial TCP throughput tests on this same hardware with no tuning yielded less than 10 Gbps whereas several of the test profiles exceeded 35 Gbps. Therefore, it is clear that system-level tuning is typically required to achieve high performance on leading-edge networking hardware. The autotuning framework presented in this work provides a methodology for systematically evaluating system performance and recording the results for later interpretation and analysis. The results demonstrate that although the baseline profile from Table III is reasonable for both TCP and UDP, options exist for improving performance in certain situations. However, many such options improve performance for some scenarios while degrading performance in others, such as profile P8 performing well with TCP, but substantially degrading UDP performance.

VI. Future Work

While this evaluation considered many system-level tuning parameters, yet more parameters exist. For example, CPU isolation can be used to prevent the Linux kernel from scheduling processes on a given CPU core, although the core is still available for explicitly placed processes. This analysis also did not explore hyperthreading, which allows multiple logical cores to be associated with a single physical core, although no workloads were explicitly assigned to a hyperthread in the present work. In addition, more configuration is available via `ethtool`, such as modifying the offload configuration. Some options exist for bypassing the kernel networking stack, which can reduce the CPU burden associated with high packet rates. Furthermore, new features are becoming available, such as low-latency sockets or socket polling, which is included in the release candidate for Red Hat Enterprise Linux 7 and newer versions of the Linux kernel. Low-latency sockets (LLS) incorporates polling to reduce the need for using interrupts (and the latency associated with doing so) for applications where low latency is critical or packets arrive very frequently. Having an autotuning infrastructure in place enables more rapid evaluation of the impact of these options by incorporating the new configuration options into the autotuning framework.

References

[1] "Performance tuning guidelines for Mellanox network adapters," Mellanox Technologies, Tech. Rep., available as of this writing at http://www.mellanox.com/related-docs/prod_software/Performance_Tuning_Guide_for_Mellanox_Network_Adapters.pdf.

[2] Myricom. (2014, May) Myri10ge documentation and FAQ. [Online]. Available: https://www.myricom.com/software/myri10ge.html

[3] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

[4] 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*, 2002, pp. 317–324.

[5] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005.

[6] R. Jones. Netperf. [Online]. Available: http://www.netperf.org/netperf/

[7] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, Aug. 1997.

[8] "Red Hat Enterprise Linux 6.5 power management guide," Red Hat, Tech. Rep., 2013, available as of this writing at https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Power_Management_Guide/.