A System-Level Optimization Framework
For High-Performance Networking

Thomas M. Benson
Georgia Tech Research Institute
thomas.benson@gtri.gatech.edu
Why do we need high-performance networking?

Data flow diagrams are typically drawn like this:

But perhaps they should be drawn like this:

Very often, moving data is a key challenge for high-speed signal processing systems – especially when moving data between devices.
What is high performance?

• High performance in this context means near line rate
• So why not just buy some high-end networking equipment and be done with it?
  • Fine in theory – in practice, the system must be well tuned and the application carefully designed to perform near line rate, so just buying equipment may not be sufficient
• Initial tests with 40 GbE adapters yielded ~10 Gbps performance with stock configuration – what now?
Performance Autotuning Framework

1. Configure both test systems to baseline profile
2. Tests profiles remain?
   - no
   - yes
3. Configure all test systems for current test profile
4. Record pre-test configuration and statistics
5. Run benchmark using application under test
6. Record post-test configuration and statistics
7. Restore configuration for baseline profile
8. Save results of test profile

Flowchart diagram showing the steps of the autotuning framework.
tests = …
    ["set_cpu_scaling_governor(node,\"ondemand\")",
     "set_cpu_scaling_governor(node, \"performance\")",
     "cpu_scaling_governor_ondemand"],
...

for test in tests:
    for node in nodes: exec(test[0])
    newconfig = {}
    newconfig["before"] = get_current_config(interface, nodes)
    run_tcp_udp_tests(newconfig)
    newconfig["after"] = get_current_config(interface, nodes)
    test_results.append(newconfig)
    for node in nodes: exec(test[1])
To use the autotuning framework, we need

- Set of relevant system configuration settings
- Mechanism for modifying those settings at run-time (ideally)
  - Fortunately, Linux supports adjusting most parameters at run-time
  - Using, e.g., /proc filesystem, rmmod/modprobe, ifconfig, ethtool, sysctl, taskset/numactl
- Mechanism for gathering config/statistics for forensic analysis and recordkeeping
  - Output from application under test, netstat, commands above
Network Configuration Options - 1

- NIC kernel module parameters
  - Depends upon NIC: For Mellanox ConnectX-3, options are `enable_sys_tune` and `high_speed_steer`
- ethtool parameters
  - rx/tx ring size, interrupt coalescence, flow control
- ifconfig parameters
  - MTU, txqueuelen
- Linux kernel parameters
  - UPD/TCP read/write kernel buffer sizes, TCP timestamp and ack behavior, etc.
• IRQ steering and routing
  • Can statically configure IRQs to route to one or more cores, optionally with or without receive hashing
  • Can also use irqbalance daemon
• CPU performance modes
  • P-states: clock frequency/voltage pairs (modes: ondemand, performance)
  • C-states: correspond to “depth” of sleep states
• Turbo boosting
  • Boosts clock freq when under high CPU load and thermal overhead is available; can be disabled, but not forced on
• CPU/NUMA affinity
## Test Environment

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPUs</strong></td>
<td>Dual-socket E5-2650 v2</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>128 GiB (8 16 GiB DIMMs)</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>CentOS 6.5</td>
</tr>
<tr>
<td><strong>Network Card (NIC)</strong></td>
<td>Mellanox ConnectX-3 (40 GbE mode)</td>
</tr>
<tr>
<td><strong>NIC Driver Version</strong></td>
<td>2.1.11</td>
</tr>
<tr>
<td><strong>NIC Firmware Version</strong></td>
<td>2.30.8000</td>
</tr>
<tr>
<td><strong>Switch</strong></td>
<td>Mellanox SX-1024</td>
</tr>
</tbody>
</table>
### Baseline System Configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel module option enable_sys_tune</td>
<td>0 (off)</td>
</tr>
<tr>
<td>Kernel module option high_rate_steer</td>
<td>1 (on)</td>
</tr>
<tr>
<td>ethtool interrupt coalescence</td>
<td>Adaptive coalescence enabled</td>
</tr>
<tr>
<td>ethtool receive/transmit ring size</td>
<td>8,192</td>
</tr>
<tr>
<td>ethtool pause configuration</td>
<td>Enabled</td>
</tr>
<tr>
<td>ifconfig txqueuelen</td>
<td>10,000</td>
</tr>
<tr>
<td>ifconfig MTU</td>
<td>9,000</td>
</tr>
<tr>
<td>sysctl net.core.netdev_max_backlog</td>
<td>250,000</td>
</tr>
<tr>
<td>sysctl kernel buffer sizes</td>
<td>Varies (up to 16MiB for TCP, 64MiB for UDP)</td>
</tr>
<tr>
<td>IRQ routing</td>
<td>NIC IRQs to 2\textsuperscript{nd} core of NUMA-local socket</td>
</tr>
<tr>
<td>Scaling governor (P-states)</td>
<td>Performance</td>
</tr>
<tr>
<td>CPU core running benchmark</td>
<td>3\textsuperscript{rd} core of NUMA-local socket</td>
</tr>
</tbody>
</table>
## Test Configuration Profiles

<table>
<thead>
<tr>
<th>Testing Profiles</th>
<th>Difference relative to baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>none (baseline profile)</td>
</tr>
<tr>
<td>P1</td>
<td>MTU set to 1500</td>
</tr>
<tr>
<td>P2</td>
<td>scaling governor set to ondemand</td>
</tr>
<tr>
<td>P3</td>
<td>netperf not pinned to a specific core</td>
</tr>
<tr>
<td>P4</td>
<td>kernel module option high_rate_steer disabled</td>
</tr>
<tr>
<td>P5</td>
<td>kernel module option enable_sys_tune enabled</td>
</tr>
<tr>
<td>P6</td>
<td>coalescence disabled via ethtool</td>
</tr>
<tr>
<td>P7</td>
<td>Ethernet pause (flow control) disabled via ethtool</td>
</tr>
<tr>
<td>P8</td>
<td>Receive/transmit ring size set to 1024</td>
</tr>
<tr>
<td>P9</td>
<td>Map NIC IRQs to all cores on NUMA-local socket</td>
</tr>
<tr>
<td>P10</td>
<td>netdev_max_backlog set to 1000</td>
</tr>
<tr>
<td>P11</td>
<td>sysctl buffer sizes set to small</td>
</tr>
<tr>
<td>P12</td>
<td>txqueuelen set to 1000</td>
</tr>
</tbody>
</table>
TCP Results

TCP throughput, background CPU cores idle

TCP throughput, background CPU cores busy
UDP Results

UDP results, background CPU cores idle

UDP results, background CPU cores busy
**Performance Observations**

- Large MTU critical for both TCP/UDP for 40 GbE
- Manually controlling CPU affinities still important
- Interactions between busy/idle cores, turboboost, etc.
  - TCP benefits from busy background cores – leads to more predictability in P/C states?
  - UDP benefits from Turboboost, but that only works if other cores are not busy
- Interrupt control important for both TCP/UDP
- `netdev_max_backlog` and `txqueuelen` important
  - Both for TCP, `netdev_max_backlog` for UDP
- `rx` ring size needs to be large for UDP, but not TCP
- Basically always some UDP packet loss – would need to rate-limit application below 40 Gbps to minimize
Conclusions

- Tuning required for high-performance at 40 Gbps
  - System defaults yielded ~ 10 Gbps performance
- P-state and busy/idle results indicate that clock rate strongly correlated to networking performance
  - Given that single core scalar performance has stagnated, what does this mean for 100 Gbps networking?
- Not shown here, but RDMA (RoCE or IB) achieves near line-rate performance at low CPU utilization
  - Use RDMA if possible, but it requires support on both send/receive sides and support is less ubiquitous than IP
  - Same is true over 56 GbE, which these adapters support
- Application design also very important – netperf has benefit of not having to actually use transferred data
Potential Future Work

- Port this framework to RHEL/CentOS 7
- Hyperthreading, CPU isolation
- CPU C-states
  - Some testing already, but with odd results; ideally would have core-level control
- RDMA and rsockets
  - Impacts kernel module loading/unloading routines
  - Requires RMDE-enabled application – using OpenMPI benchmarks as surrogate RDMA application
- Low-latency sockets / device polling
  - New feature in RHEL 7 and newer kernels
  - May be particularly useful for UDP/IP
- Sophisticated interrupt management (i.e. leverage rx/tx queues)
- Compare to kernel bypass technologies (Mellanox VMA, PF_RING, etc.)
Questions?