Problem. Solved.

A System-Level Optimization Framework For High-Performance Networking

Thomas M. Benson

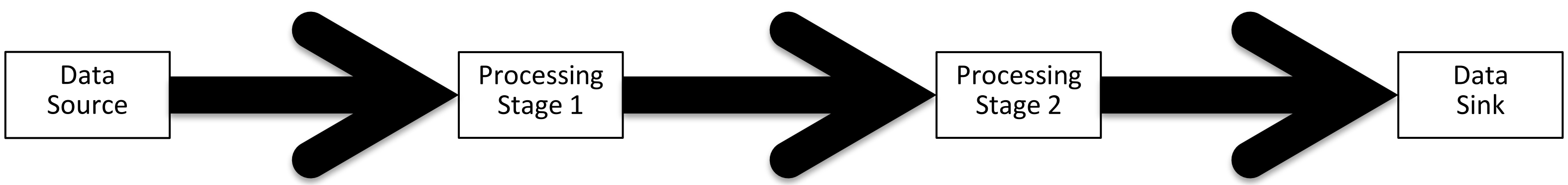
Georgia Tech Research Institute

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:



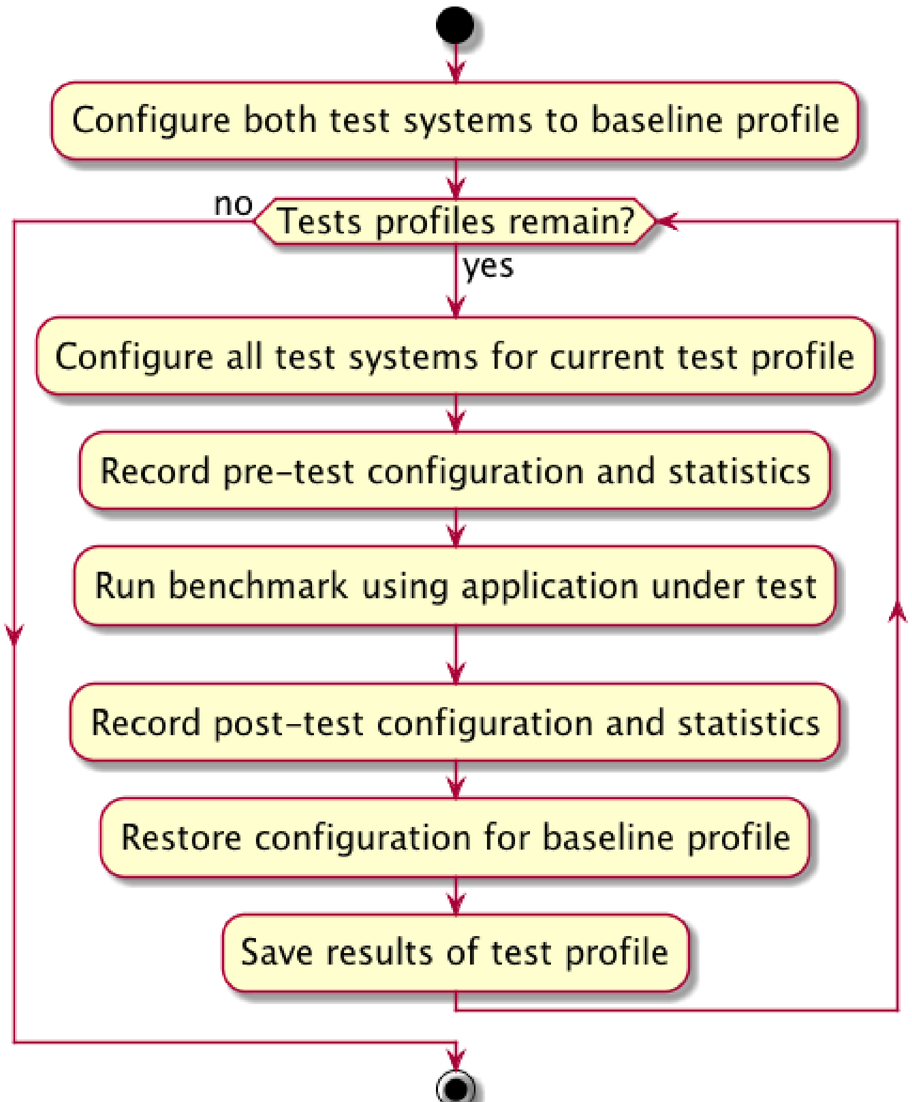
Very often, moving data is a key challenge for high-speed signal processing systems – especially when moving data between devices.

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

Georgia Tech Research Institute



Autotuning Framework Example

```
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])
```

Autotuning Framework Requirements

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.

Network Configuration Options - 2

Georgia Tech Research Institute

- 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

Georgia Tech Research Institute

CPUs	Dual-socket E5-2650 v2
Memory	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

Baseline System Configuration

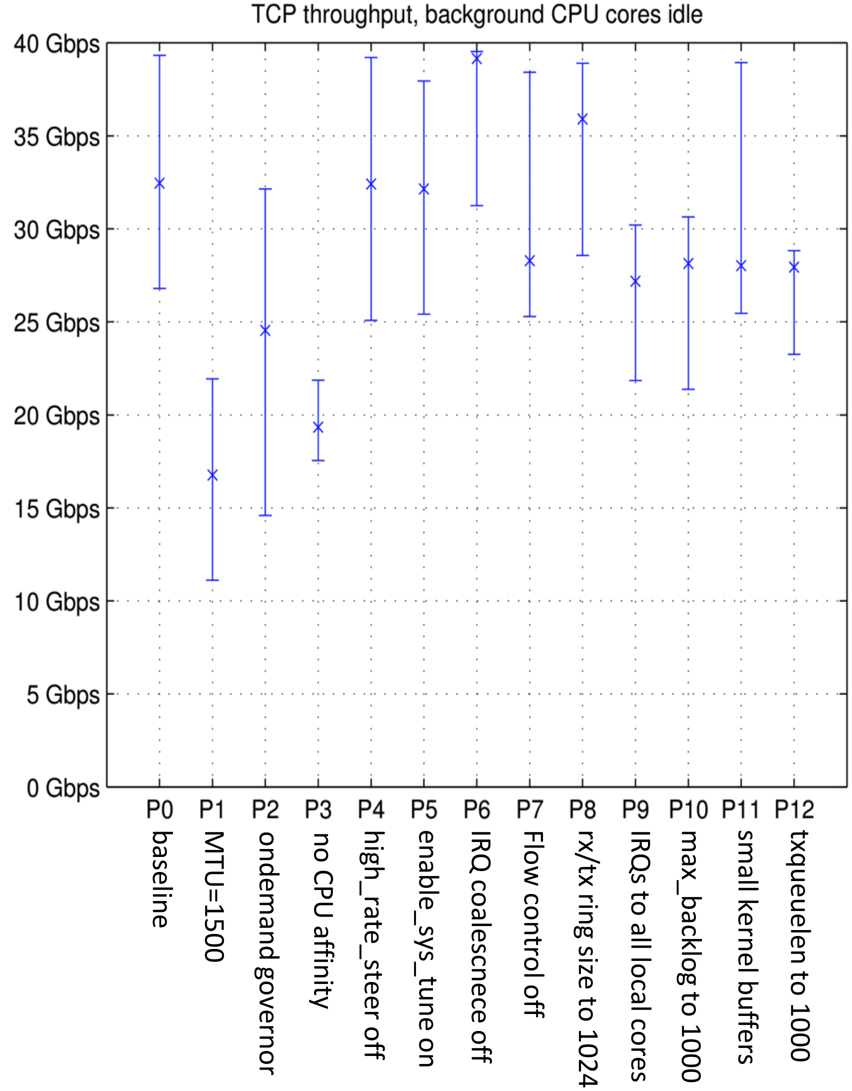
Georgia Tech Research Institute

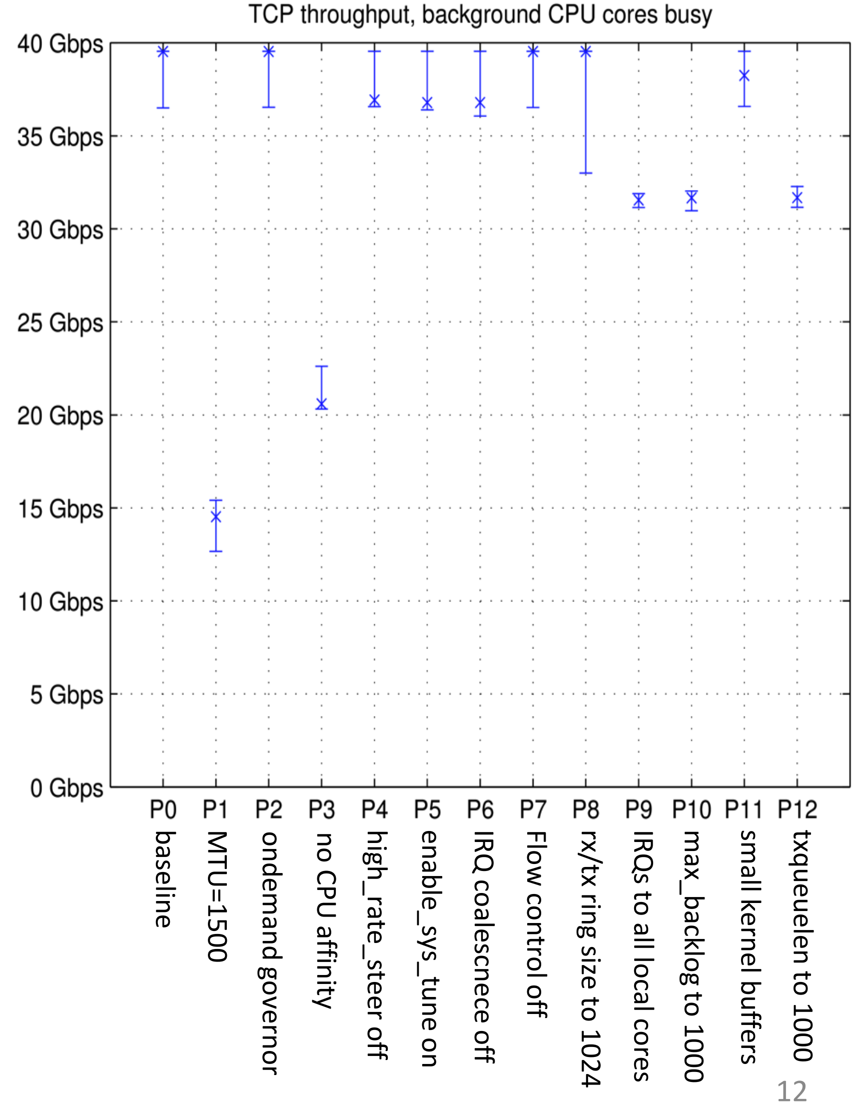
Kernel module option enable_sys_tune	0 (off)
Kernel module option high_rate_steer	1 (on)
ethtool interrupt coalescence	Adaptive coalescence enabled
ethtool receive/transmit ring size	8,192
ethtool pause configuration	Enabled
ifconfig txqueuelen	10,000
ifconfig MTU	9,000
sysctl net.core.netdev_max_backlog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

Test Configuration Profiles

Testing Profiles	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
P10	netdev_max_backlog set to 1000
P11	sysctl buffer sizes set to small
P12	txqueuelen set to 1000

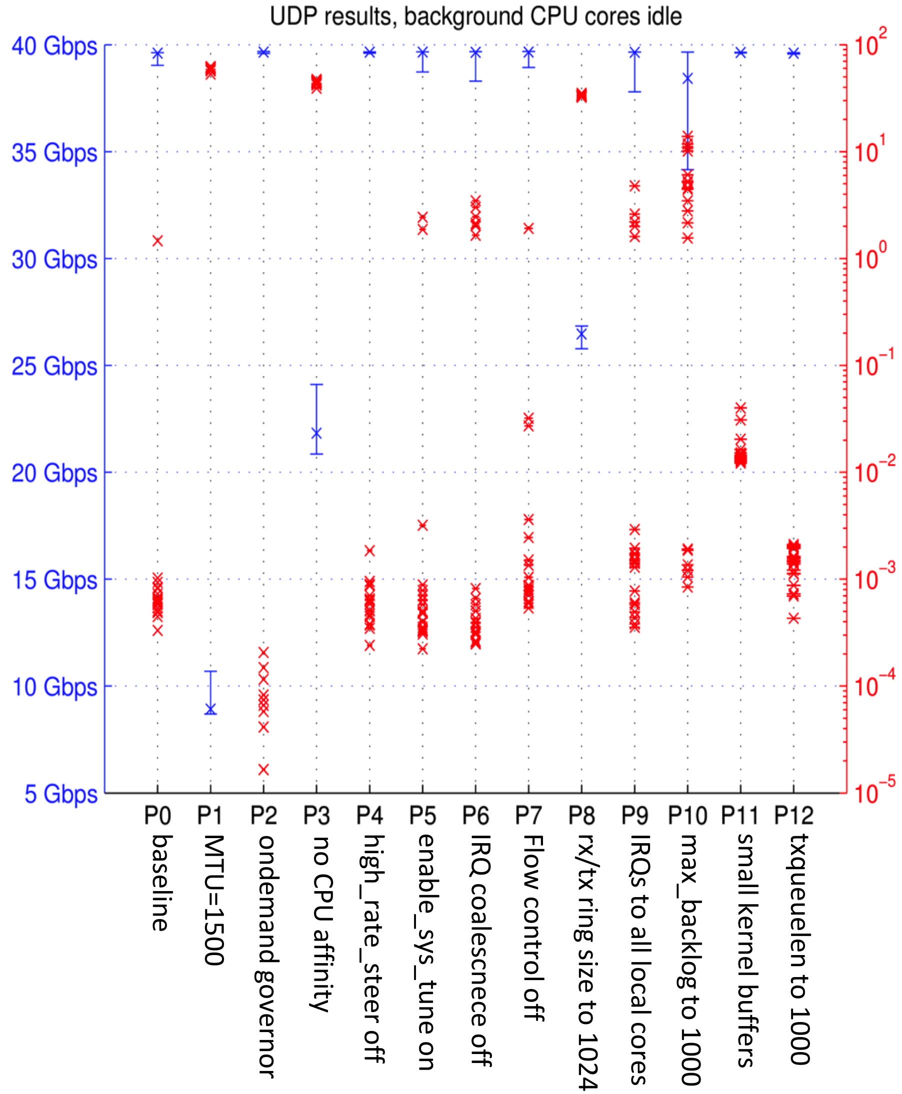
TCP Results

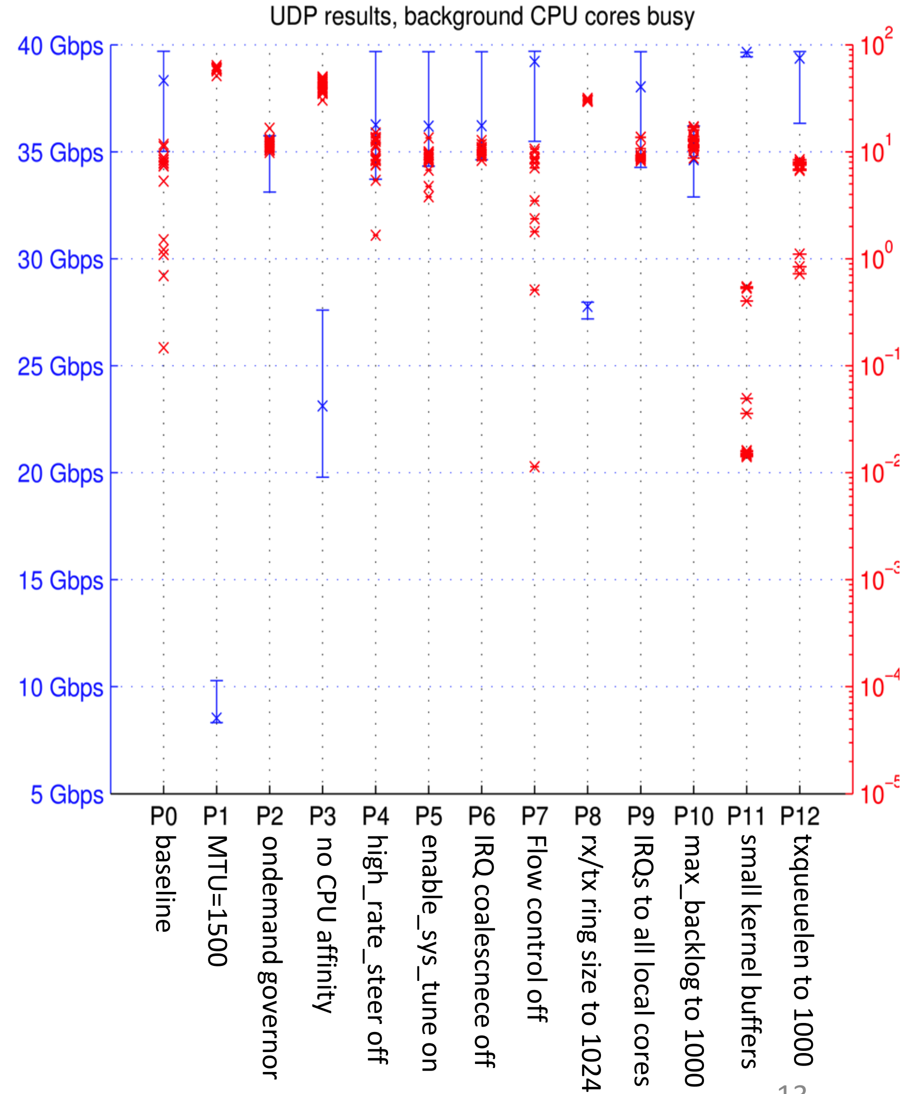

TCP throughput, background CPU cores idle
40 Gbps
35 Gbps
30 Gbps
25 Gbps
20 Gbps
15 Gbps
10 Gbps
5 Gbps
0 Gbps
P0 baseline
P1 MTU=1500
P2 ondemand governor
P3 no CPU affinity
P4 high_rate_steer off
P5 enable_sys_tune on
P6 IRQ coalescnece off
P7 Flow control off
P8 rx/tx ring size to 1024
P9 IRQs to all local cores
P10 max_backlog to 1000
P11 small kernel buffers
P12 txqueuelen to 1000


TCP throughput, background CPU cores busy
40 Gbps
35 Gbps
30 Gbps
25 Gbps
20 Gbps
15 Gbps
10 Gbps
5 Gbps
0 Gbps
P0 baseline
P1 MTU=1500
P2 ondemand governor
P3 no CPU affinity
P4 high_rate_steer off
P5 enable_sys_tune on
P6 IRQ coalescnece off
P7 Flow control off
P8 rx/tx ring size to 1024
P9 IRQs to all local cores
P10 max_backlog to 1000
P11 small kernel buffers
P12 txqueuelen to 1000

UDP Results

Georgia Tech Research Institute





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 RMDA-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.)

Thank You!

Questions?