Improving network stack

Why do we need to improve the network stack in today's operating systems ?

Luigi Rizzo
Università di Pisa
http://info.iet.unipi.it/~luigi/research.html

Work supported by H2020 project SSICLOPS

Summary

- Network stack overview
- Performance considerations
- Useful techniques
- Some case studies

Part 1: Network stack overviev

Network stack: definition

- code that lets applications talk to the network
- layered as it handles several different problems

Usually in-kernel for multiple reasons:

- access to shared resources
- security
- efficiency
- some convenience (but some rigidity, too)

Network stack tasks

- accumulate data (sockets, in and out)
- implement protocols (TCP, cong. control, flow control)
- manage resources (L4 ports, memory)
- interact with network (ARP, routes)
- talk to the hardware (device driver)

Operation triggered by

- user input
- network events
- timers

Layering comes naturally

Traffic characteristics and requirements

Outgoing (Upstream, app to network)

- trusted data (once in the kernel)
- controlled

Incoming (Downstream, network to app)

- untrusted
- uncontrolled

Outgoing path

- syscall
- buffering
- protocol processing
- routing
- (scheduling / virtual switching ...)
- device output

Outgoing path - code structure

Independent software layers

- sockets
- TCP/UDP/other transport protocol
- IP (network layer), MAC encapsulation
- device driver

Direct function calls

- immediate feedback, process to completion

Some critical sections

- mostly in the device driver, occasionally in higher layers (e.g. socket buffers)

Process to completion (or not)

"do at once all the processing required by a piece of data"

- good match with immediate calls into next layer
- maximises useful work

Not always possible

- window/output queue full, interrupt handler too long
- intermediate queues required to break the flow
- need balanced producers and consumers
- beware of livelock

Incoming path

- interrupt handling
- drain device
- validate traffic
- demultiplex
- protocol processing
- notify clients

Incoming path - code structure

Packets come at random times

- process to completion might be inappropriate
- better use a short interrupt handler, wakeup interrupt thread

Interrupt thread (NAPI)

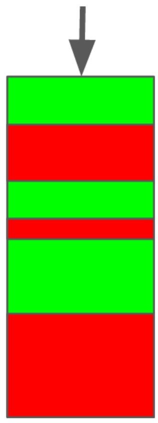
- most of the work, up to socket buffers
- notify client

Client thread

- finally consume data

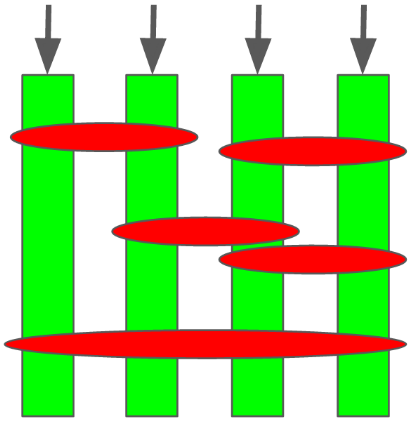
Uniprocessor OS

- common in the 90's, now called unikernels
- no concurrency issues: just disable interrupts
- efficient code
- cannot use multiple cores



Multiprocessor OS

- locks protect critical sections
- multiple critical sections while processing one packet
- lock contention may become significant
- better solutions (e.g. RCU) for read-mostly data
- memory latency may become critical at high speed



Network stack features

Existing network stacks mostly target user applications
- good support for TCP (client and server)
- big companies actively developing features
- hardware vendors eager to follow up

Ossification due to in-kernel implementation
- hard to update clients (too many, too varied)
- sometimes contrived workarounds (server side, QUIC, ...)

Network stack features (2)

Packet I/O not well supported

- niche application (compared to billions of phones and laptops)
- dedicated hardware can be more efficient

Cloud and virtualization change the scenario

- hardware based solution may not be viable anymore

Why do we need to improve the network stack ?

Missing features

- better support for packet I/O and software switching
- more flexibility in adding features
- better support for virtualization

Part 2: network stack performance

Network stack performance

Defined by multiple metrics

- throughput
- latency
- efficiency
- scalability

Tradeoffs are unavoidable

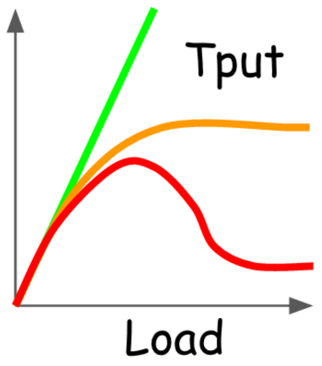
Performance metrics: throughput

Measured in bits per second or packet per second

- the latter is usually the relevant one
- may depend heavily on traffic patterns
- tradeoff with latency

Beware of livelock

- throughput drops as offered load increases
- can result from adversarial load, or poor design choices



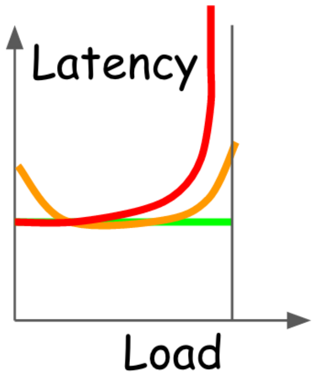
Performance metrics: latency

Time to traverse the stack

- also look at distribution, not just average/median
- main component is usually queueing delay (see bufferbloat)
- next come CPU (un)availability, timer and interrupt delays

Normally tradeoff with throughput, energy efficiency

- interrupt moderation helps throughput but increases latency
- busy wait reduces latency but kills efficiency



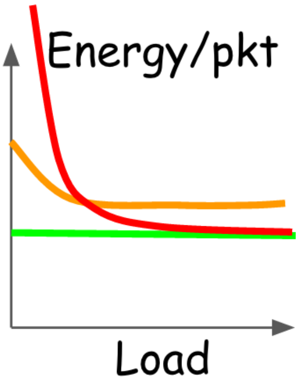
Performance metrics: efficiency

Work = useful packet processing + wait for data + cleanup

- Interrupt or busy wait
- interrupts and notifications are expensive
- busy wait improves throughput and latency
- optimal strategy varies with load

Even more surprises with pipelines

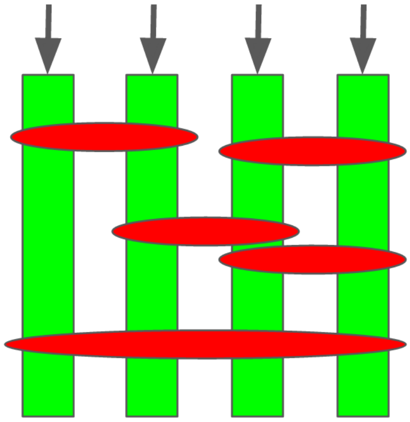
- unbalanced stages may worsen all metrics



Performance metrics: scalability

Locking is an easy way to protect shared data structures

- sprinkle locks on UP code base to make it SMP-capable
- fine-grained locking to increase parallelism
- can scale really poorly in multicore/multisocket systems
- often need ad-hoc solutions and code restructuring



Network stack performance status

Not (yet) problematic on the client side
- TCP heavily optimized (also with HW acceleration)
- 1-10 Gbit/s easy to achieve

Server side problematic at 40-100 Gbit/s
- both CPU and HW bottlenecks

Very poor packet I/O performance
- at least with standard APIs
- scheduling also problematic

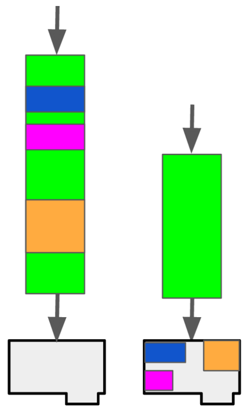
Performance improvement strategy

First, identify bottlenecks
- CPU, memory and link speeds progress in large steps and uncoordinated ways
- in most cases, throughput improves faster than latency
- latency sensitivity is harder to deal with
- CPU bound workloads can be addressed
- not much to do with HW bound workloads

CPU bound workloads: hardware offload

One client/core may be unable to saturate the link

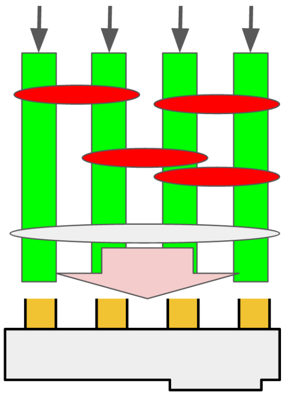
- use hardware offload for checksums, segmentation, encryption, filtering
- reduced CPU usage can improve throughput
- can be implemented with small code changes



CPU bound workloads: multiqueue

Still not fast enough (e.g. inbound processing):

- run multiple independent clients in parallel
- multiqueue NICs reduce contention in the device driver
- again, requires small code modifications
- does not accelerate individual clients



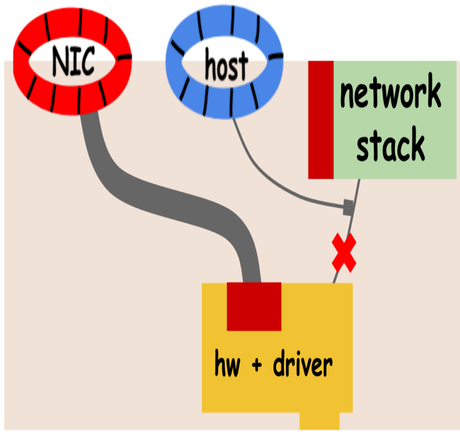
CPU bound workloads: simplify the stack

Some workloads need faster APIs

- simplify the stack, see netmap or XDP
- exploit batching and zero copy
- provide efficient APIs

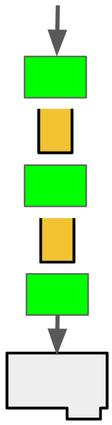
Multiple variants of this concept

- simplification is the key feature, not userspace processing



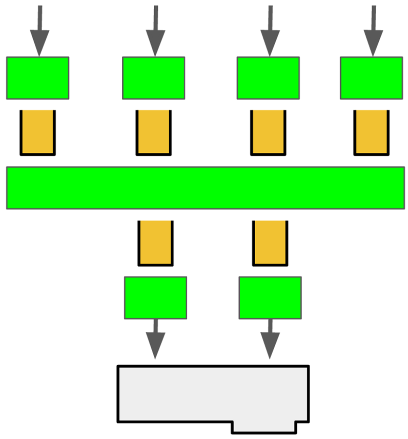
CPU bound workloads: pipelined structure

- split processing vertically
- insert non-blocking mailboxes between stages
- can exploit parallelism even for a single client



CPU bound workloads: dedicated cores

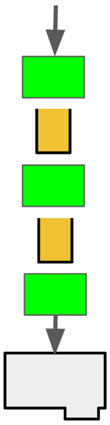
- pipeline stages can have different number of workers
- some can be used for inherently sequential functions, such as scheduling
- equivalent to having dedicated (co)processors
- helps addressing lock contention



CPU bound workloads: reduce synchronisation cost

Mailboxes and queues require expensive notifications
- hw and sw interrupts must find and reach the target thread
- interact with scheduler, Inter Processor Interrupts
- can easily take microseconds

Mitigation techniques
- interrupt moderation: rate limit
- batching: amortize
- busy wait/short sleep: shift load on consumer



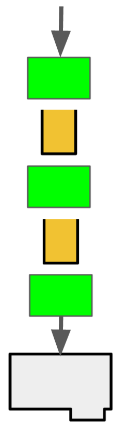
CPU bound workloads: introduce batching

Impractical to rewrite the stack to support batching

- introduce MORE_FLAG in metadata
- default off, each enqueue calls `notify()`
- incrementally deployable, suppress `notify()` if more data known to come

Initially proposed in QEMU networking (Maffione 2013)

- dismissed as "not used in Linux" (N.I.H.)
- rediscovered as XMIT_MORE in Linux



CPU bound workloads: better algorithms

- problems have finite size, look at constants in addition to asymptotic complexity
- be aware of and exploit hardware features (caches, memory, special instructions)
- look at approximate solutions

CPU bound workloads: examples of better algorithms

- DXR (lookups, finite size problem);
- poptrie (special CPU instruction)
- huge pages (reduce TLB misses)
- QFQ (O(1) scheduling thanks to approximate timestamps)
- RCU (implicit coordination)
- RouteBricks (lock removal via dedicated paths)
- PSPAT (centralised scheduler thread)

CPU bound workloads: VM networking

Expensive VM exits kill packet I/O performance

- virtio mitigates exits with mailboxes and helper threads
- passthrough moves device driver to the guest
- virtual passthrough gives hardware independence/zero copy

HW bound workloads

Eventually, hardware will become the bottleneck

- PCIe bandwidth
- PCIe transaction rate (NIC's limited)
- low performance NICs (most cannot do line rate)

Buy better hardware!

- or, make good use of existing one
- find good operating region, rate limit HW access

Why do we need to improve the network stack ?

Poor performance

- we have several good solutions
- use them!

Part 3: Overview of existing solutions

Various kernel/network stack/layer bypass

- DPDK, netmap, XDP

Fast switch fabric

- mswitch, custom DPDK-based tools

Virtualization support

- virtual passthrough

Custom applications

- PSPAT packet scheduling

Bypass techniques

Motivation: network stack inadequate for packet I/O

- full bypass: DPDK
- network stack bypass: netmap
- integrated filtering: XDP

Full bypass: UIO and DPDK

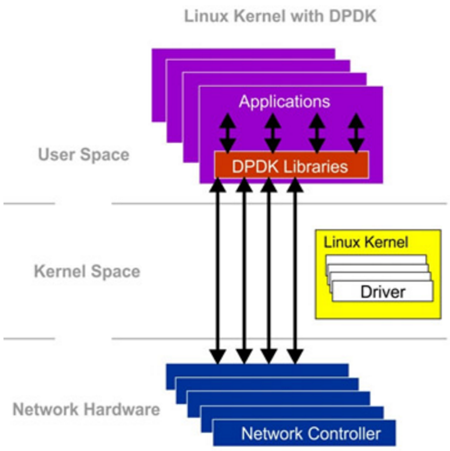
Take the entire device driver to userspace.

PROS

- convenient workaround for lack of kernel support
- only need UIO to export the PCI device to userspace
- userspace device driver needs to do all device programming

CONS

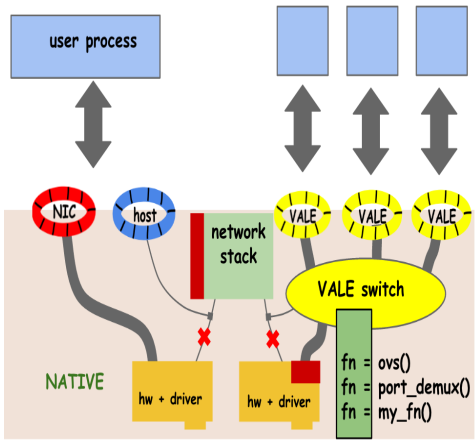
- no interrupt or event support
- reinjection via socket-like API
- hard to share resources



Network stack bypass: netmap

Keep driver in hardware, provide user API for I/O and synchronisation

- complete ecosystem, not just physical device
- useful for programmable switches, userspace protocol demultiplexing



Integrated filtering: XDP

- device driver RX hook just before creating the sbk
- call an **eBPF** program to determine packet's fate
- included in Linux

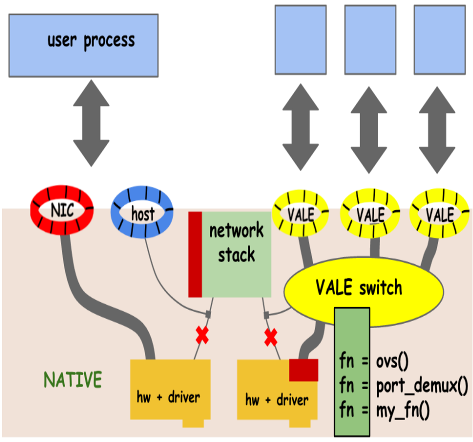
Currently mostly a proof of concept

- poor support for generic packet processing or userspace I/O
- needs in-kernel development

Fast switch fabric

VALE/mSwitch provide a fast programmable in-kernel dataplane

- show that high speed software dataplanes are possible
- useful for L2 as well as protocol demultiplexing
- enabler for embedding protocols into user applications



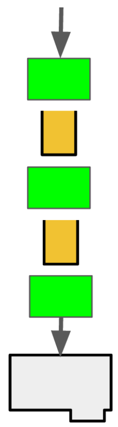
Fast VM networking

Various techniques to amortize VM exits

- virtio and vhost-net is a first start, shared host/vm queue with helper thread
- hardware passthrough removes data copies but binds VM to hardware
- virtual passthrough (possibly using the netmap API) gives complete hardware independence

Pipeline performance

- pipeline is a common pattern in networking software
- balance between stages is critical for performance at high load
- always full or always empty queue requires frequent expensive notifications
- same as livelock, important to understand the phenomenon and remedies



PSPAT, software scheduling

First block in the network path for VMs
- there are legitimate users with high PPS
- need to protect the virtual switch and the rest of the stack

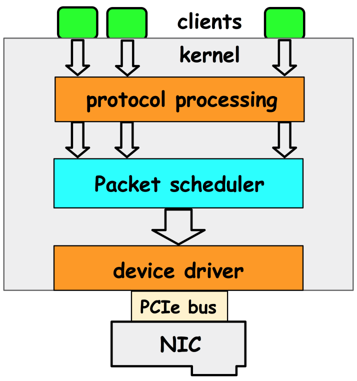
Hardware does not always give perfect isolation
- the bus (PCIe) can be a bottleneck
- scheduling after the bottleneck is ineffective

Traditional Software Packet Scheduler

PROS
- no hardware dependencies
- large choice of algorithms

CONS
- heavy lock contention in accessing the scheduler
- under congestion I/O becomes serialized
- scalability can be problematic

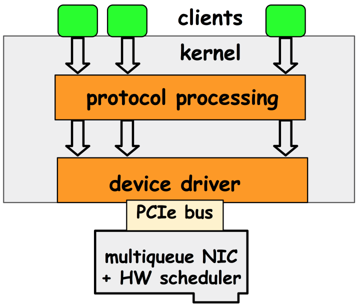
TC delivers 2 Mpps, decreasing with number of clients



Hardware Packet Schedulers

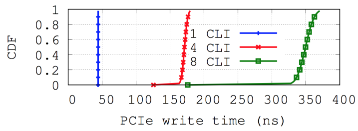
PROS
- fully parallel down to the NIC
- reduced system load due to HW offloading

CONS
- limited choice of algorithms
- the bus is still a point of contention.



PCIe access issues

- PCIe arbitration is round robin, **not programmable**
- PCIe service rate is limited by the NIC
- PCIe bus can saturate as well



Latency distributions in µs on I7

CLI	Notes	Percentile min	10	50	90	99
1	HW	5.7	5.8	6.0	6.1	6.4
1	TC	5.5	5.7	5.9	6.1	6.6
1	PSPAT	6.3	6.8	7.2	7.7	8.2
5	HW (PCIe congestion)	9.8	117.0	125.0	137.0	152.0
5	TC @ 10G .812 Mpps	6.6	8.5	12.6	16.6	18.6
5	PSPAT @ 10G .823 Mpps	6.4	7.3	9.0	11.1	12.2

Dilemma

SW flexible but slow, HW not as good as we would like

1. **Denial:** we don't need fast schedulers
   - what about NFV ?
2. **Faith:** hardware will get better
   - what about existing hardware ?
3. **Various approximate solutions**
   - trivial schedulers (FIFO, DRR: fast but poor delay guarantees)
   - active queue management (RED, CODEL: rely on **everyone** behaving)
   - bounded number of queues: rely on quiet neighbours

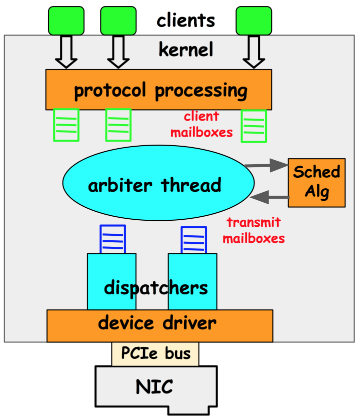
PSPAT: Packet Scheduling with PArallel Transfers

Decouple scheduling and transmission

- a dedicated arbiter thread runs the SA (sequential)
- traffic is released at link rate to the device driver
- possibly one or more threads perform transmission in parallel

Results

- reduced contention, increased parallelism
- large speedup compared to TC
- the architecture permits a worst case analysis



PSPAT implementation

Two versions

- in-kernel, for complete compatibility with TC:
  - intercept traffic in `__dev_queue_xmit()`,
  - deliver to `dev_hard_start_xmit()`
  - reuses Linux QDISC code
  - kernel module to implement mailboxes and threads
- userspace, for fast prototyping and optimized performance
  - supports userspace networking (netmap, DPDK...)
  - can use fast scheduling code from dummynet

Performance analysis

Metrics:

- throughput and latency

Platforms:

- I7 with 40G NIC and Linux 4.7 (in-kernel PSPAT)
- dual Xeon E5-2640 (userspace)

Sources (one per core, pinned):

- UDP sockets (not very fast)
- pkt-gen (the netmap version), very fast
- Linux pktgen bypasses __dev_queue_xmit()

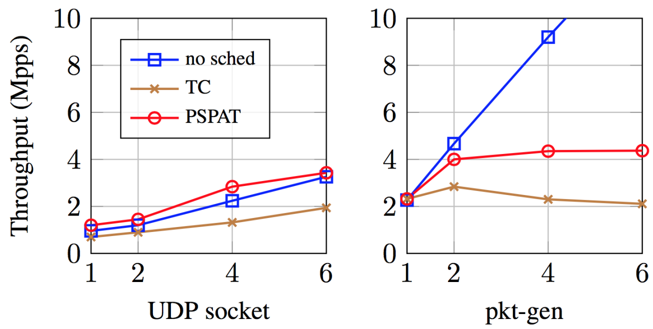
Packet schedulers:

- none (HW), TC, PSPAT

Throughput measurements

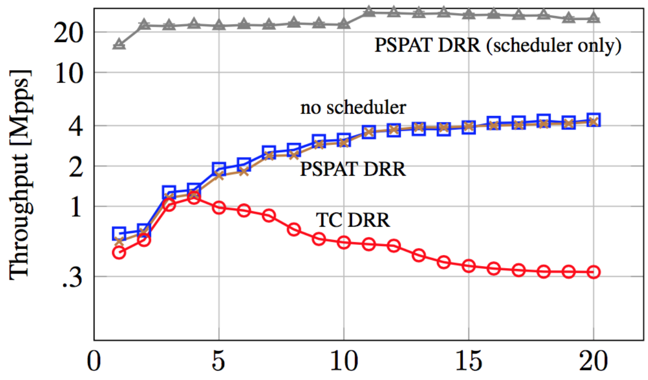
- Clients send as fast as possible
- variable number of clients
- schedulers use QFQ (DRR is marginally faster)
- TC and PSPAT rates higher than scheduler's capacity
- measurements in PPS as that is the relevant metric

Throughput with regular UDP (I7)

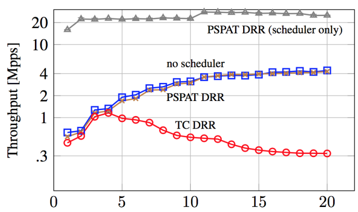


- 2x speedup (limited by the qdisc code)

Throughput for userspace PSPAT (Xeon)



High speed I/O



	Clients			
Configuration	1	2	4	6
netmap pipes	15.00	17.00	36.00	35.90
scheduler only	25.00	37.00	36.00	35.90

- Scheduling decisions alone are extremely fast
- using netmap we are in 15-20 Mpps territory

One way latency measurements

Experiments with different link rates and number of clients

- one client has **weight=100**, sends at half the reserved bandwidth
- other clients have **weight=1**, send as fast as possible

Theory says latency is proportional to MSS/RATE

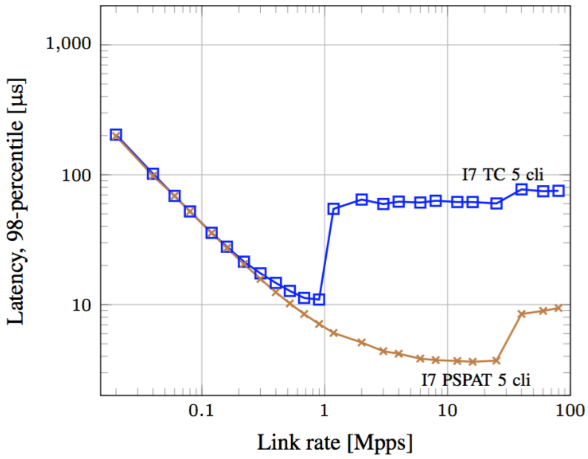
One way latency measurements (1)

Latency distributions in μs on I7

CLI	Notes	Percentile				
		min	10	50	90	99
1	HW	5.7	5.8	6.0	6.1	6.4
1	TC	5.5	5.7	5.9	6.1	6.6
1	PSPAT	6.3	6.8	7.2	7.7	8.2
5	HW (PCIe congestion)	9.8	117.0	125.0	137.0	152.0
5	TC @ 10G .812 Mpps	6.6	8.5	12.6	16.6	18.6
5	PSPAT @ 10G .823 Mpps	6.4	7.3	9.0	11.1	12.2

- No big surprises for PSPAT:
  a couple of extra us due to rate-limited scans and handoffs
- Note the huge effect of congestion on the PCIe bus

Latency versus rate, I7 + linux 4.5



Conclusions

Network stacks missing in four areas

- packet I/O
- switching performance
- VM support
- agile protocol replacement

There are useful solutions to improve all of these areas

http://info.iet.unipi.it/~luigi/research.html