NetApp®

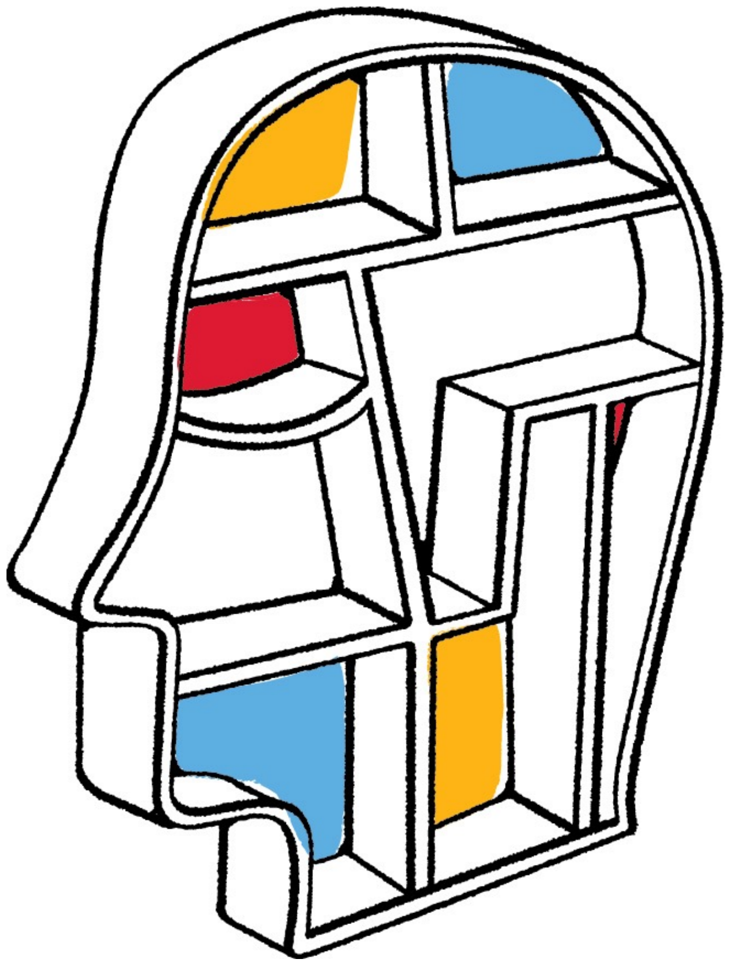
Go further, faster®

Speeding up Linux TCP/IP with a Fast Packet I/O Framework

Michio Honda

Advanced Technology Group, NetApp

michio@netapp.com



With acknowledge to Kenichi Yasukata, Douglas Santry and Lars Eggert

NetApp

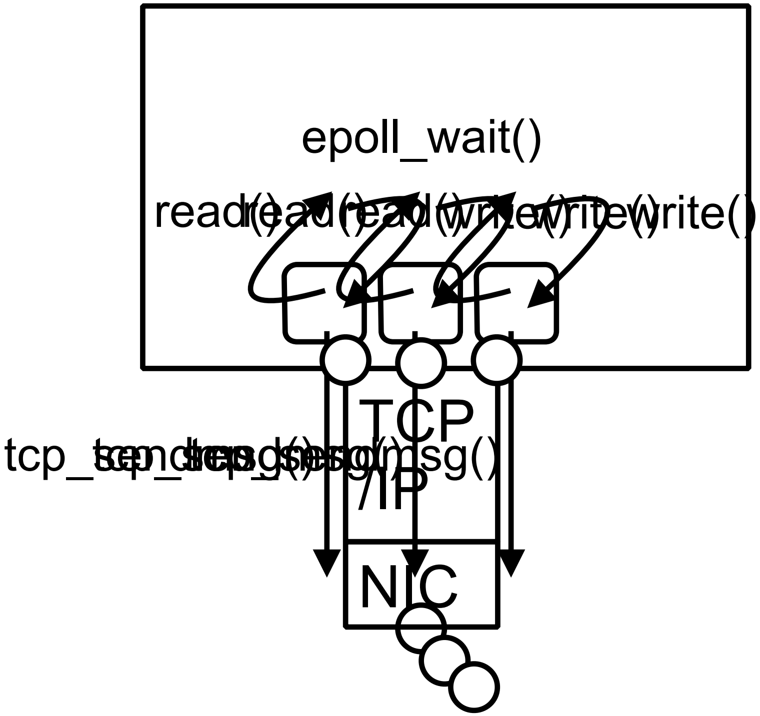
Motivation

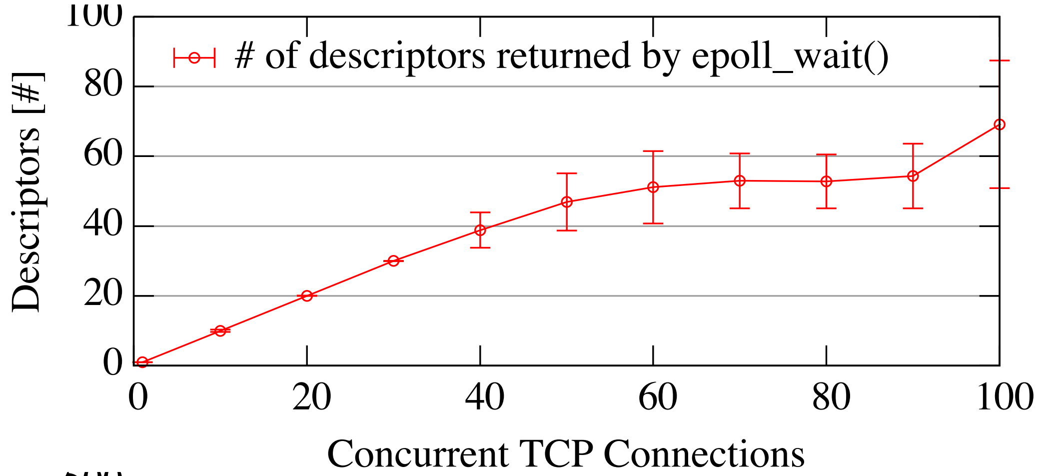
- Linux TCP/IP
  - State-of-the-art features
    - Cope with all the network conditions and traffic patterns
      - FACK, FRTO, RACK, DSACK, Fast Open, DCTCP
    - Various security enhancements (e.g., RFC5961)
    - Out-of-tree: MPTCP, TcpCrypt
- User-space TCP/IP (e.g., Seastar)
  - Fast due to a dedicated NIC to an app (netmap, DPDK)
    - App-driven NIC I/O and network stack execution
    - Direct packet buffer access

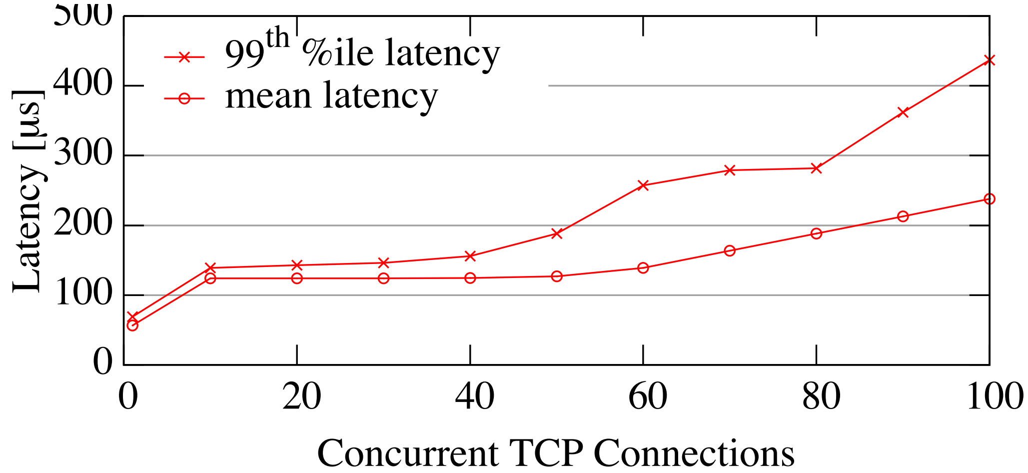
Integrating the best aspects of both of the worlds

Problems

- Request-response traffic with:
  - Small messages/packets at high rates
- Concurrent TCP connections
  - Queueing delays







rx-usecs 1 (default), 1024 B response message

NetApp

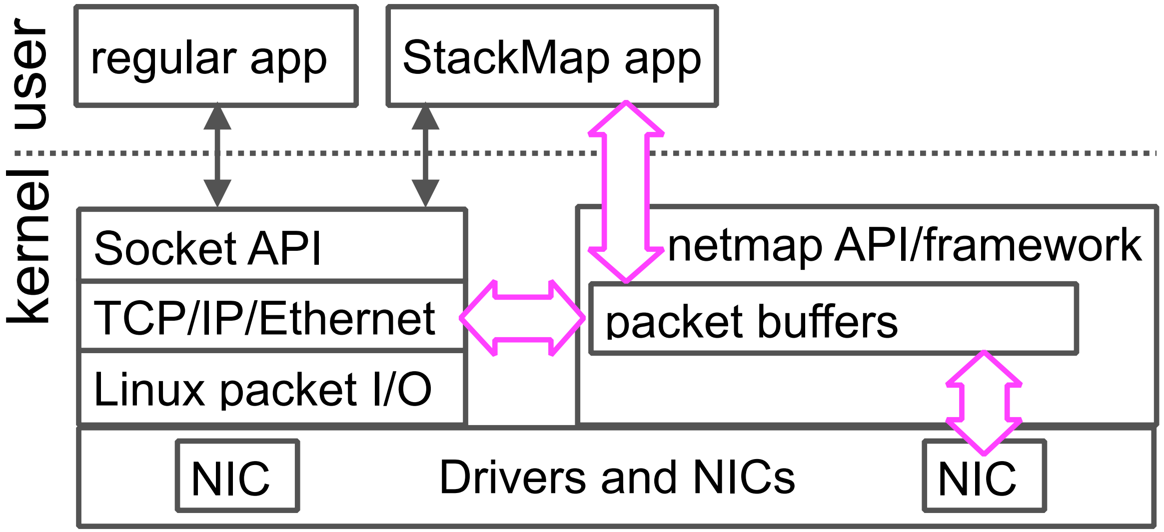
Design Principles

- Dedicate a NIC to a privileged app
  - Similar to fast user-space TCP/IPs
- Use TCP/IP stack in the kernel
  - Regular apps must be able to run on other NICs
  - When the privileged app crashes, the system and the other apps must survive

NetApp

StackMap Overview

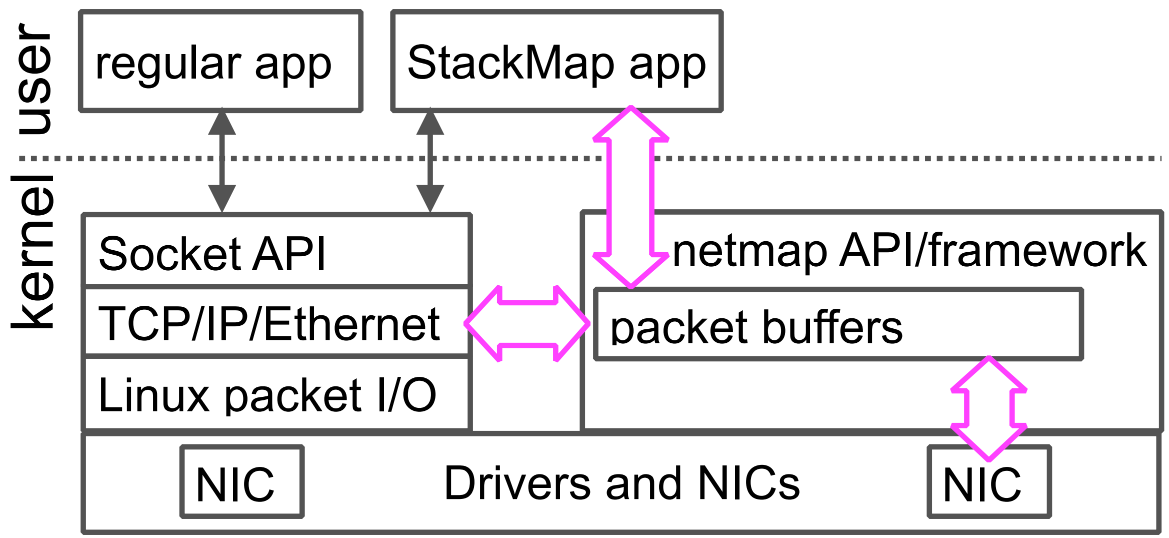
- App registers a NIC
- Socket API for control
  - socket(), bind(), listen() etc
- netmap API for *datapath (alters read()/write())*



NetApp

StackMap Datapath

- Packet buffers are mapped to NIC rings, app and pre-allocated skbuffs
- App triggers NIC I/O via netmap API syscall
- The syscall processes data/packets in TCP/IP
  - before (TX) or after (RX) NIC I/O



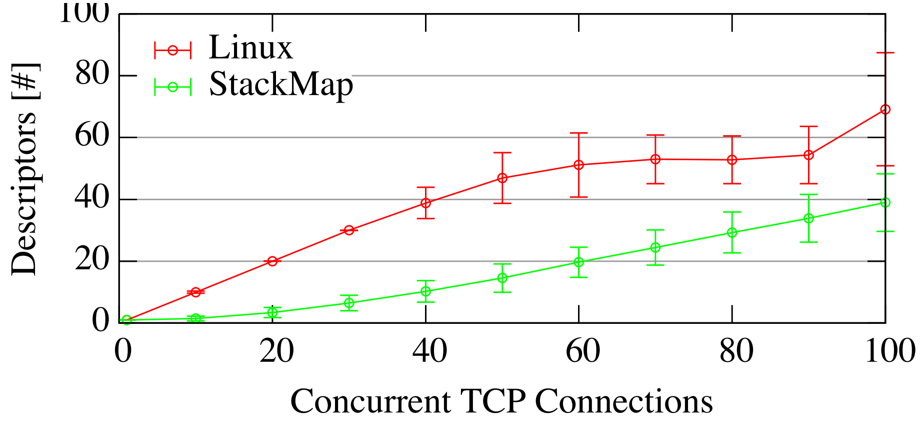
NetApp

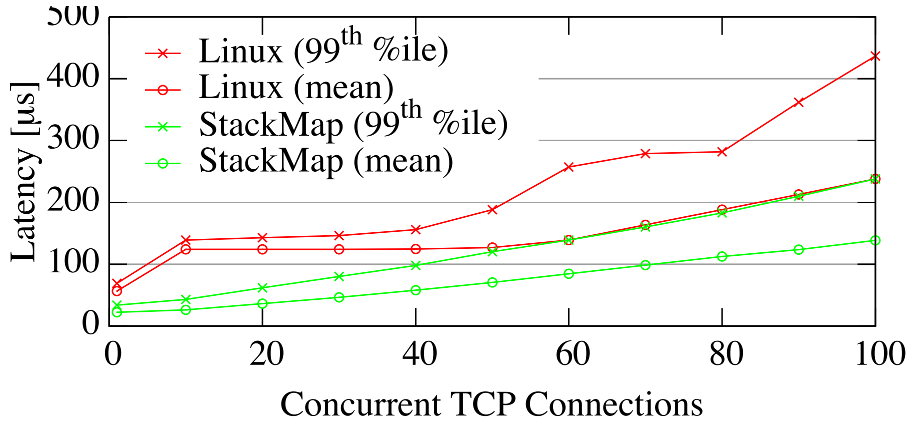
Experimental Results

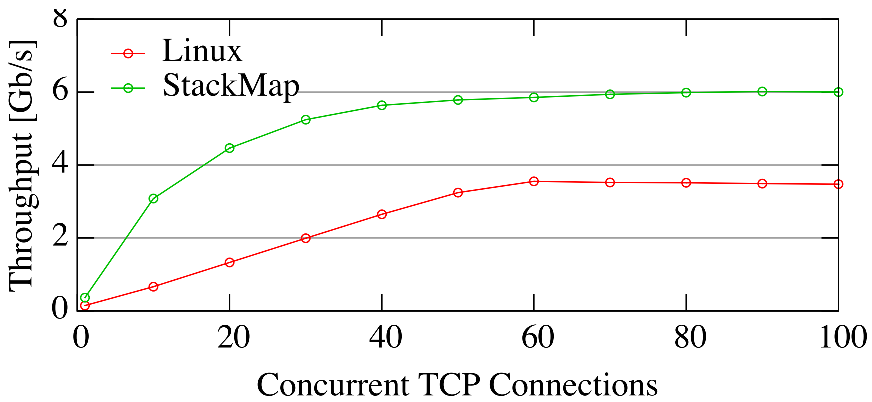
- Implementation
  - Linux 4.2 with 188 LoC changes
  - netmap with 68 LoC changes
  - A new kernel module with 2200 LoC
- Setup
  - Two machines with
    - Xeon E5-2680 v2 (2.8 Ghz)
    - Intel 83599 10 GbE NIC
  - Server:
    - Linux (rx-usecs 1) or StackMap
  - Client:
    - Linux with **wrk** HTTP benchmark tool

Basic Performance

Serving 1024 B HTTP OK
with a single CPU core



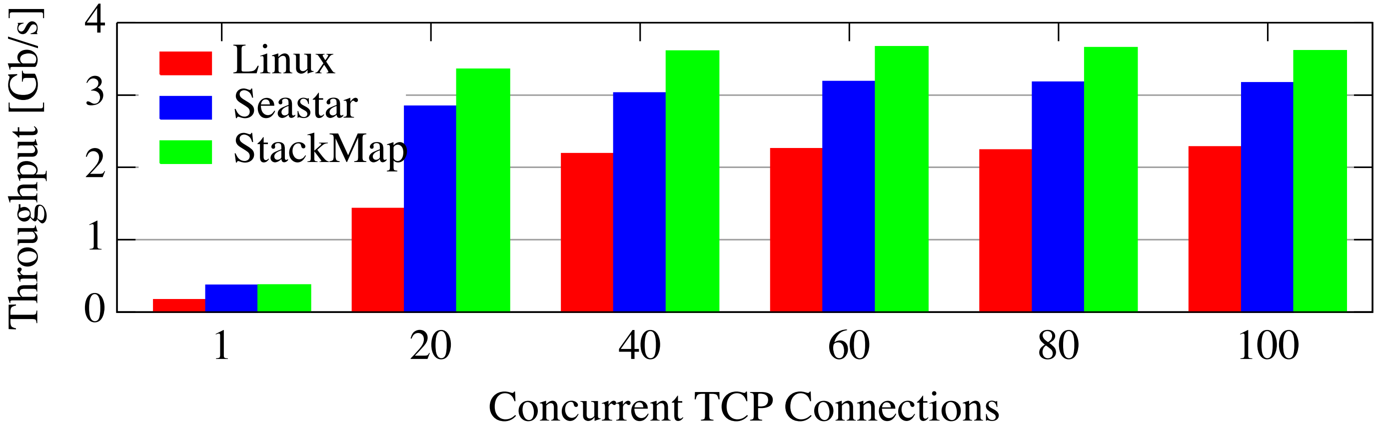




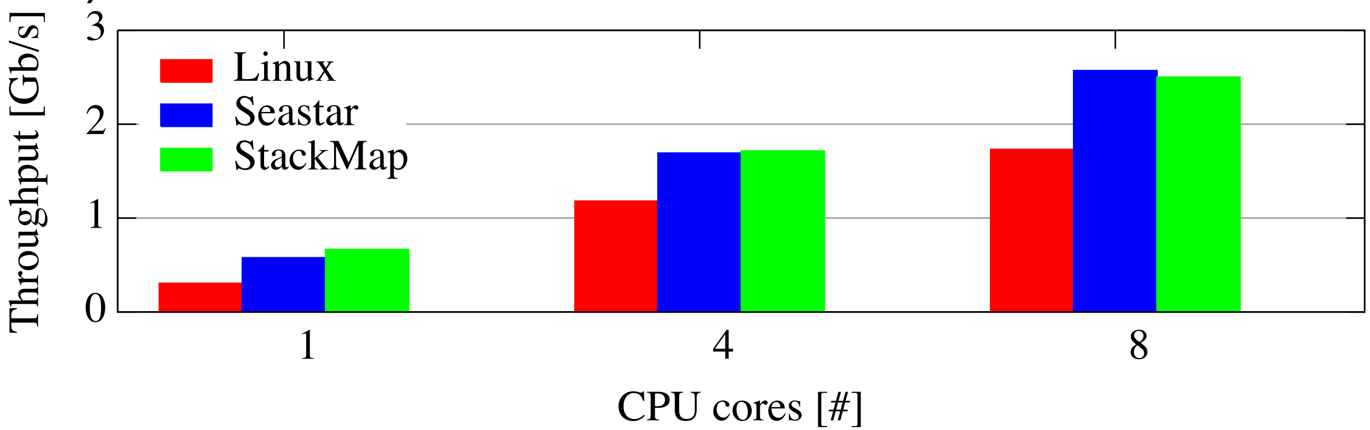
NetApp

Memcached Performance

Memcached with 10% set and 90 % get (1024 B objects, single CPU core)



Memcached with 10% set and 90 % get (64 B objects, 60 concurrent TCP connections)



NetApp

Conclusion

- Linux TCP/IP **protocol processing** is fast
- We can bring the most of techniques in user-space TCPs into Linux TCP/IP
- What makes StackMap fast?
  - all the advantages of the netmap **framework**
    - syscall batching
    - memory allocator
      - static but flexible packet buffer pool whose buffers can be dynamically linked to a NIC ring (without dma_(un)map_single())
    - I/O batching (more aggressive than xmit_more)
  - no skb (de)allocation, no vfs layer
  - synchronous execution of app and protocol processing

NetApp

Base Latency

- Single HTTP request (97B) and response (1024B) latency
  - Linux: rx-usec 0 with epoll_wait(timeout=0)
    - 23.05 us 43.64 MB
  - Linux: rx-usec 0 with epoll_wait(timeout=-1)
    - 25.54 us 39.49 MB
  - Linux: rx-usec 1 with epoll_wait(timeout=0)
    - 56.60 us 18.11 MB
  - Linux: rx-usec 1 with epoll_wait(timeout=-1)
    - 56.67 us 18.11 MB
  - Linux: rx-usec 1 with net.core.busy_poll=50 (poll())
    - 23.02 us 43.76 MB
  - StackMap (NIC polling)
    - 21.94 us (45.80 MB/s)