FreeBSD Network Stack Optimizations for Modern Hardware

Robert N. M. Watson
FreeBSD Foundation

EuroBSDCon 2008

UNIVERSITY OF CAMBRIDGE

Introduction

- Hardware and operating system changes
- TCP input and output paths
- Hardware offload techniques for TCP
- Software optimizations

Changes in hardware

- Memory access dominates computation

  ➡ Cache-centered design

- Per-CPU performance growth slows

  ➡ Trend towards multi-processing

- Network speed growth outpaces CPUs

  ➡ Trend towards hardware offload (again)

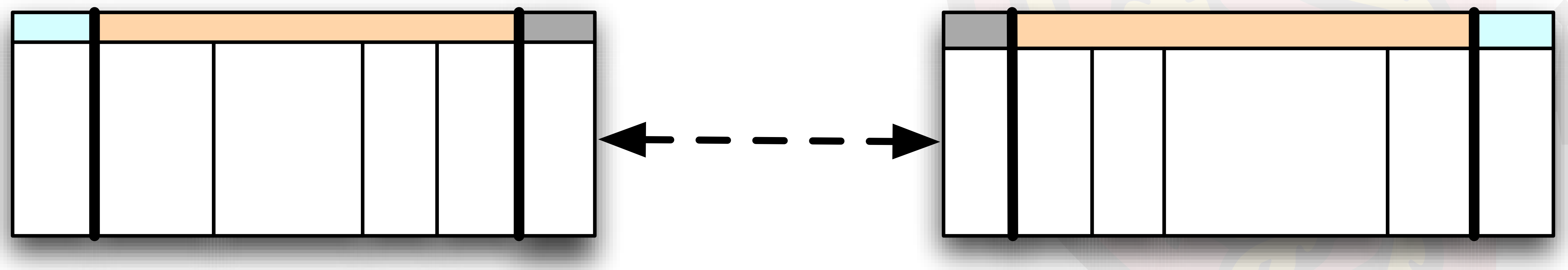
OS design responses

- Manage cache footprint and contention
  - Balance of instructions vs cache misses
- Develop techniques to exploit parallel processing with increasing core density
- Network stack support for offloaded protocol assist

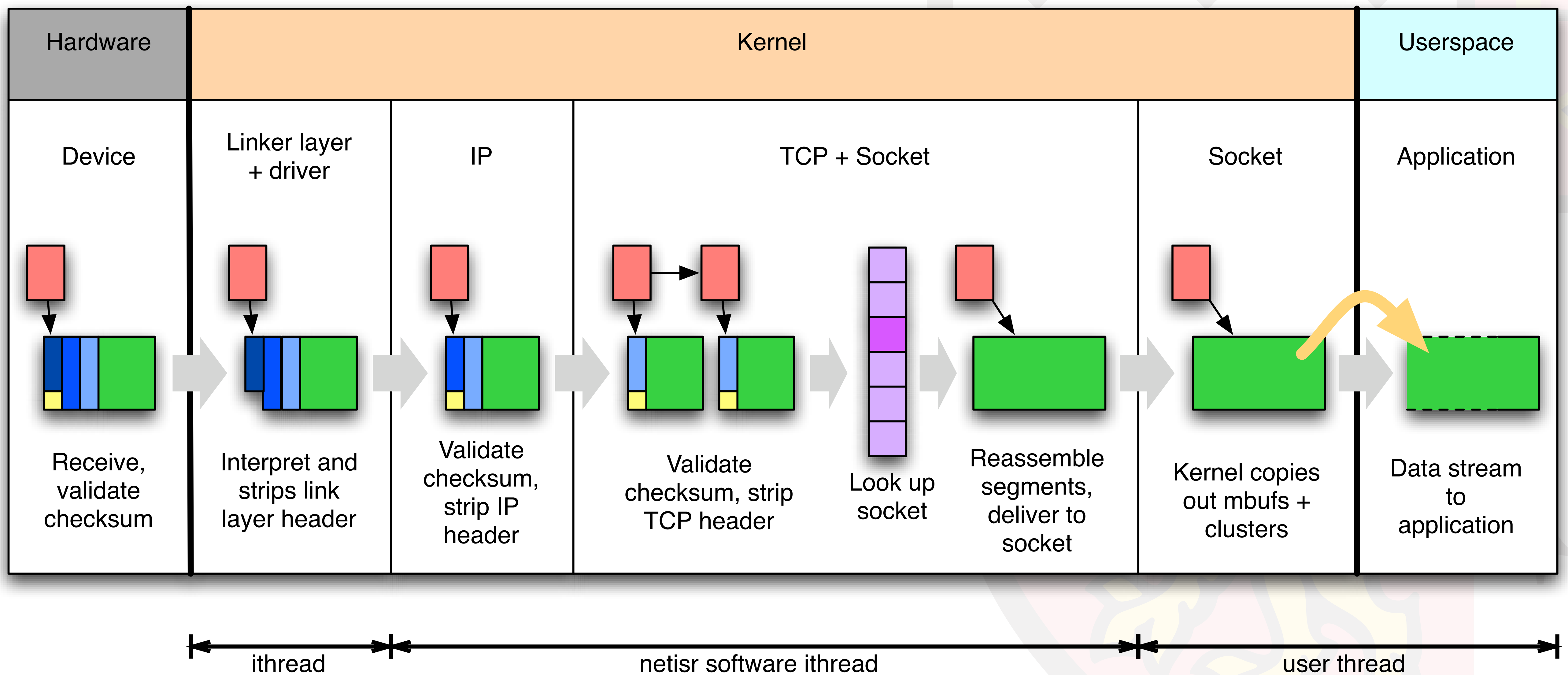
TCP input and output

TCP

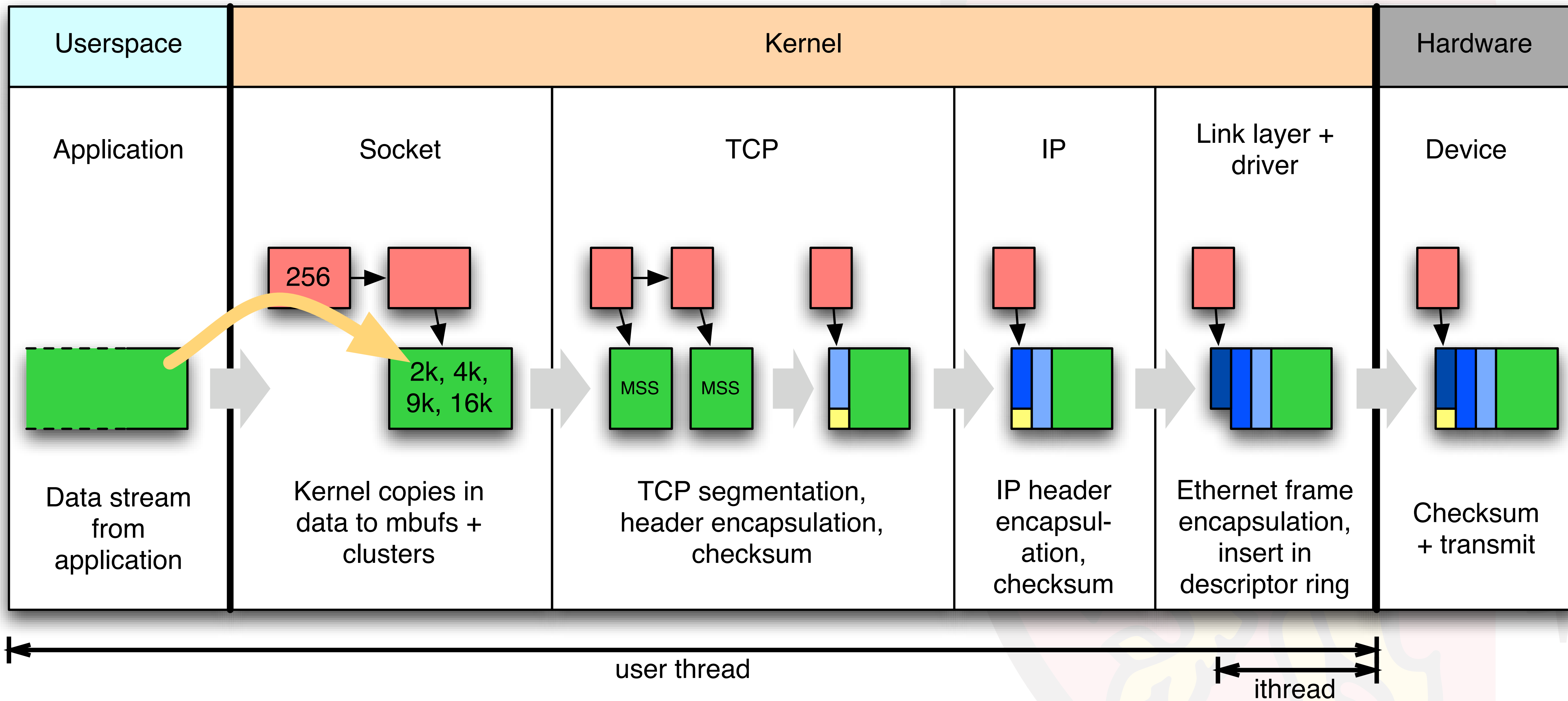
- Transmission Control Protocol (TCP)
- Stream protocol: reliable, ordered delivery
- Acknowledgement, retransmission, congestion control, data checksums



TCP input path



TCP output path



Hardware offload techniques for TCP

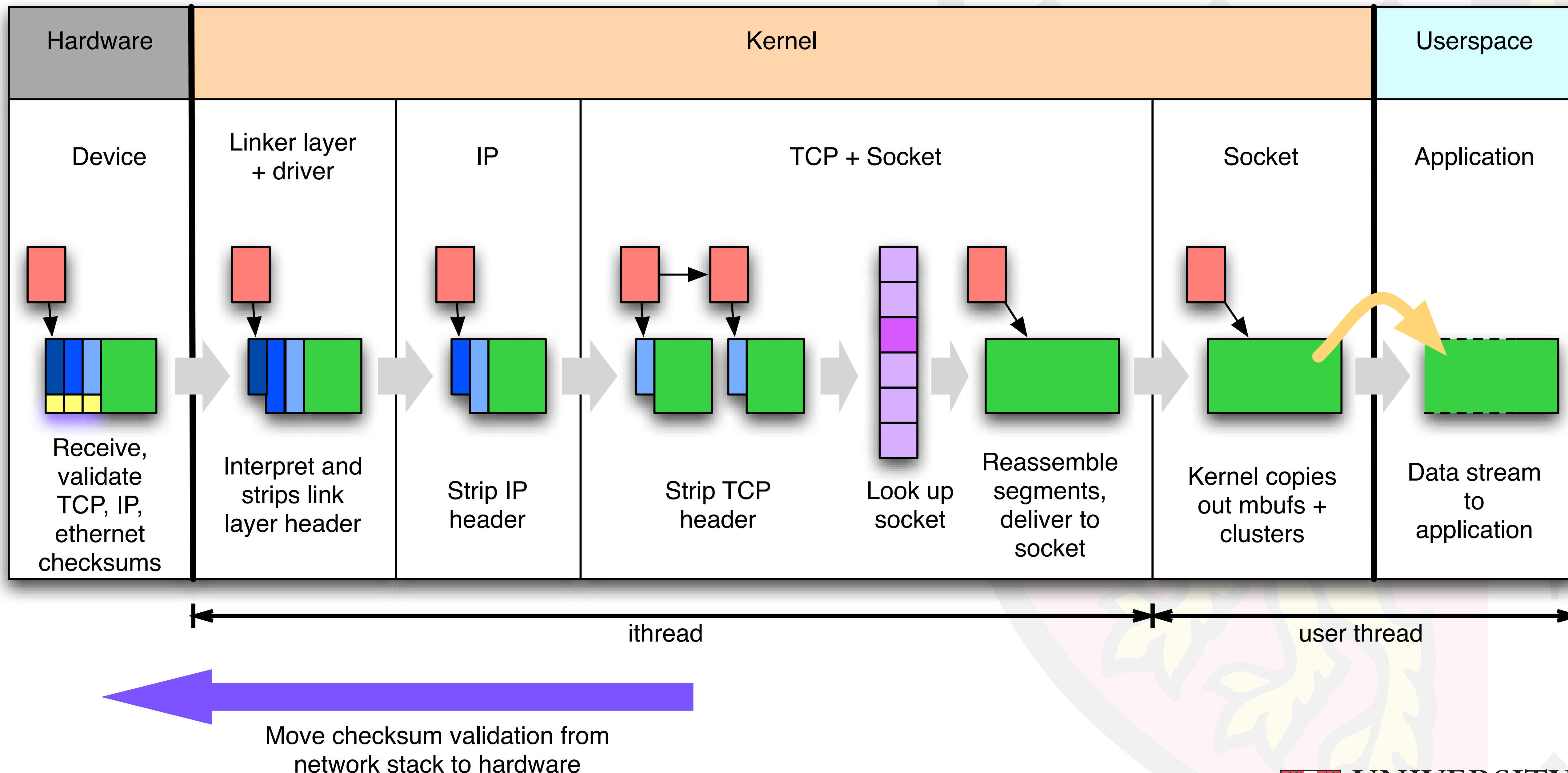
Hardware offload

- Reduce instructions and cache misses
  - Input and output checksum offload
  - TCP segmentation offload (TSO)
  - TCP large receive offload (LRO)
  - Full TCP offload (TOE)
- Not discussed: iSCSI offload, RDMA

TCP checksum offload

- Generate / validate TCP/UDP/IP checksums
- Reduce CPU instructions, cache misses
- Input path: hardware validates checksums
- Output path: hardware generates checksums

TCP input checksum offload



TCP input checksum implementation

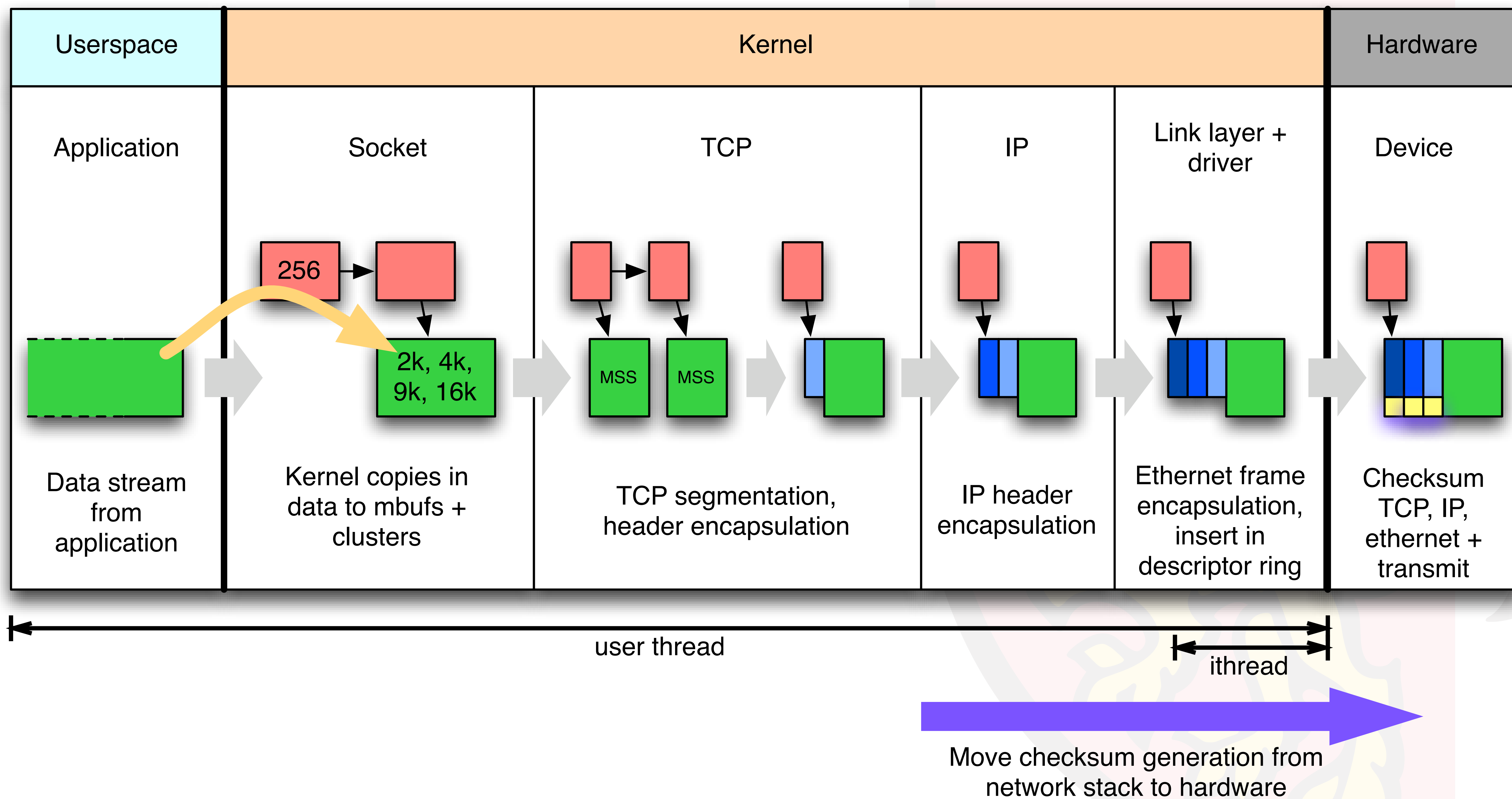
- Drivers declare support in capability flags on interface (if_hwassist)
- Checksum results (pass/fail) placed in receive descriptor entry by card
- Input routines check test for hardware-validated checksums, skipping software if possible

TCP input checksum implementation

tcp_input()

```
if (m->m_pkthdr.csum_flags & CSUM_DATA_VALID) {
        if (m->m_pkthdr.csum_flags & CSUM_PSEUDO_HDR)
                th->th_sum = m->m_pkthdr.csum_data;
        else
                th->th_sum = in_pseudo(ip->ip_src.s_addr,
                    ip->ip_dst.s_addr, htonl(
                    m->m_pkthdr.csum_data + ip->ip_len +
                    IPPROTO_TCP));
        th->th_sum ^= 0xffff;
} else {
        len = sizeof(struct ip) + tlen;
        ...
        th->th_sum = in_cksum(m, len);
}
if (th->th_sum)
        goto drop;
```

TCP output checksum offload



Output checksum implementation

- TCP layer doesn’t know if offload available on interface until routing decision made
  - Defer checksums to IP output routine
  - mbuf header flags track deferral state
- Device driver pushes higher layer state into hardware via transmit descriptors

Output checksum implementation

tcp_output()

```
m->m_pkthdr.csum_flags = CSUM_TCP;
m->m_pkthdr.csum_data = offsetof(struct tcphdr, th_sum);
th->th_sum = in_pseudo(ip->ip_src.s_addr, ip->ip_dst.s_addr,
    htons(sizeof(struct tcphdr) + IPPROTO_TCP + len + optlen));
```

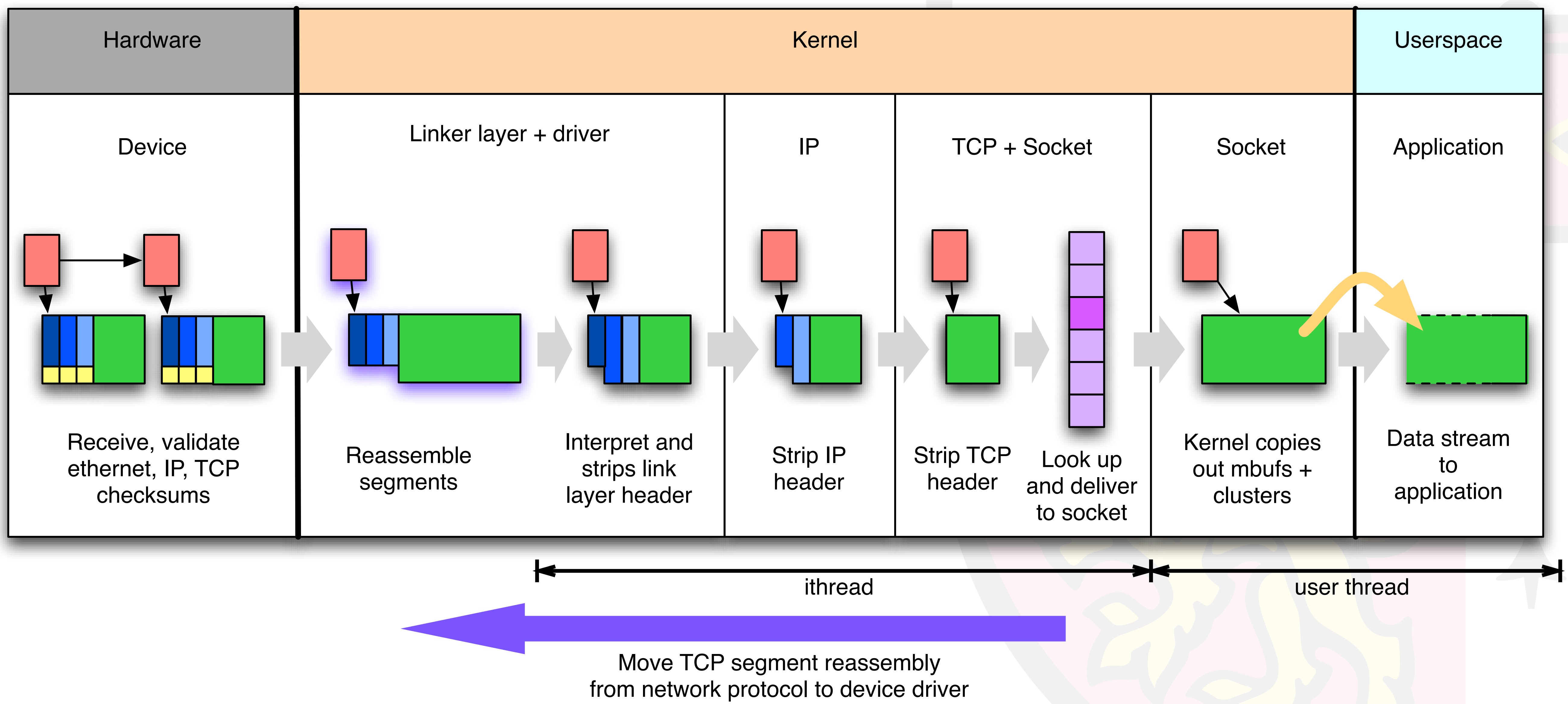
ip_output()

```
m->m_pkthdr.csum_flags |= CSUM_IP;
sw_csum = m->m_pkthdr.csum_flags & ~ifp->if_hwassist;
if (sw_csum & CSUM_DELAY_DATA) {
    in_delayed_cksum(m);
    sw_csum &= ~CSUM_DELAY_DATA;
}
m->m_pkthdr.csum_flags &= ifp->if_hwassist;
```

TCP large receive offload (LRO)

- TCP reassembles segments into streams
- Significant per-packet processing overhead for socket lookup, reassembly, delivery
- LRO: driver reassembles sequential packets to reduce number of "packets" processed
- Software implementation shared by drivers

TCP LRO



UNIVERSITY OF CAMBRIDGE

TCP LRO implementation

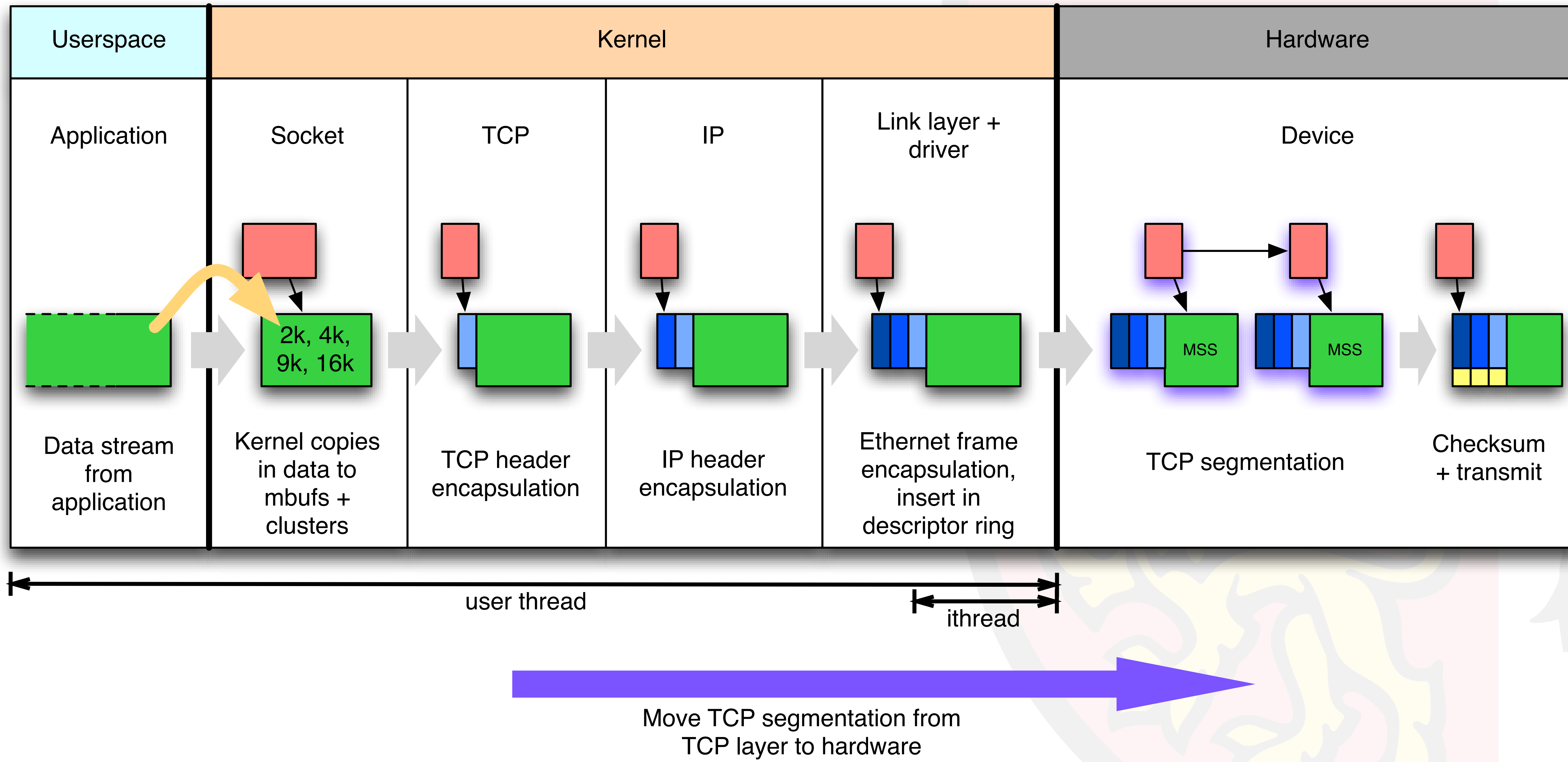
igb_rxeof()

```
while ((staterr & E1000_RXD_STAT_DD) &&
    (count != 0) && (ifp->if_drv_flags & IFF_DRV_RUNNING)) {
...
        if ((!lro->lro_cnt) || (tcp_lro_rx(lro, m, 0))) {
                /* Pass up to the stack */
                IGB_RX_UNLOCK(rxr);
                (*ifp->if_input)(ifp, m);
                IGB_RX_LOCK(rxr);
                i = rxr->next_to_check;
        }
...
while (!SLIST_EMPTY(&lro->lro_active)) {
        queued = SLIST_FIRST(&lro->lro_active);
        SLIST_REMOVE_HEAD(&lro->lro_active, next);
        tcp_lro_flush(lro, queued);
}
```

TCP segmentation offload (TSO)

- TCP transmit processing converts data stream into a series of packets
- Per-packet overhead is significant
- Pass large chunks of data to card, let it perform segmentation based on MSS

TSO


Userspace
Kernel
Hardware
Application
Socket
TCP
IP
Link layer + driver
Device
2k, 4k, 9k, 16k
MSS
MSS
Data stream from application
Kernel copies in data to mbufs + clusters
TCP header encapsulation
IP header encapsulation
Ethernet frame encapsulation, insert in descriptor ring
TCP segmentation
Checksum + transmit
user thread
ithread
Move TCP segmentation from TCP layer to hardware

TSO implementation

- TCP layer detects TSO support, generates segments exceeding interface MTU
- TCP tags packet with CSUM_TSO and MSS so device can implement segmentation
- If TSO is disabled/route changes, ip_output() returns error, MSS recalculated
- Device driver tags TSO data in descriptor

TSO implementation

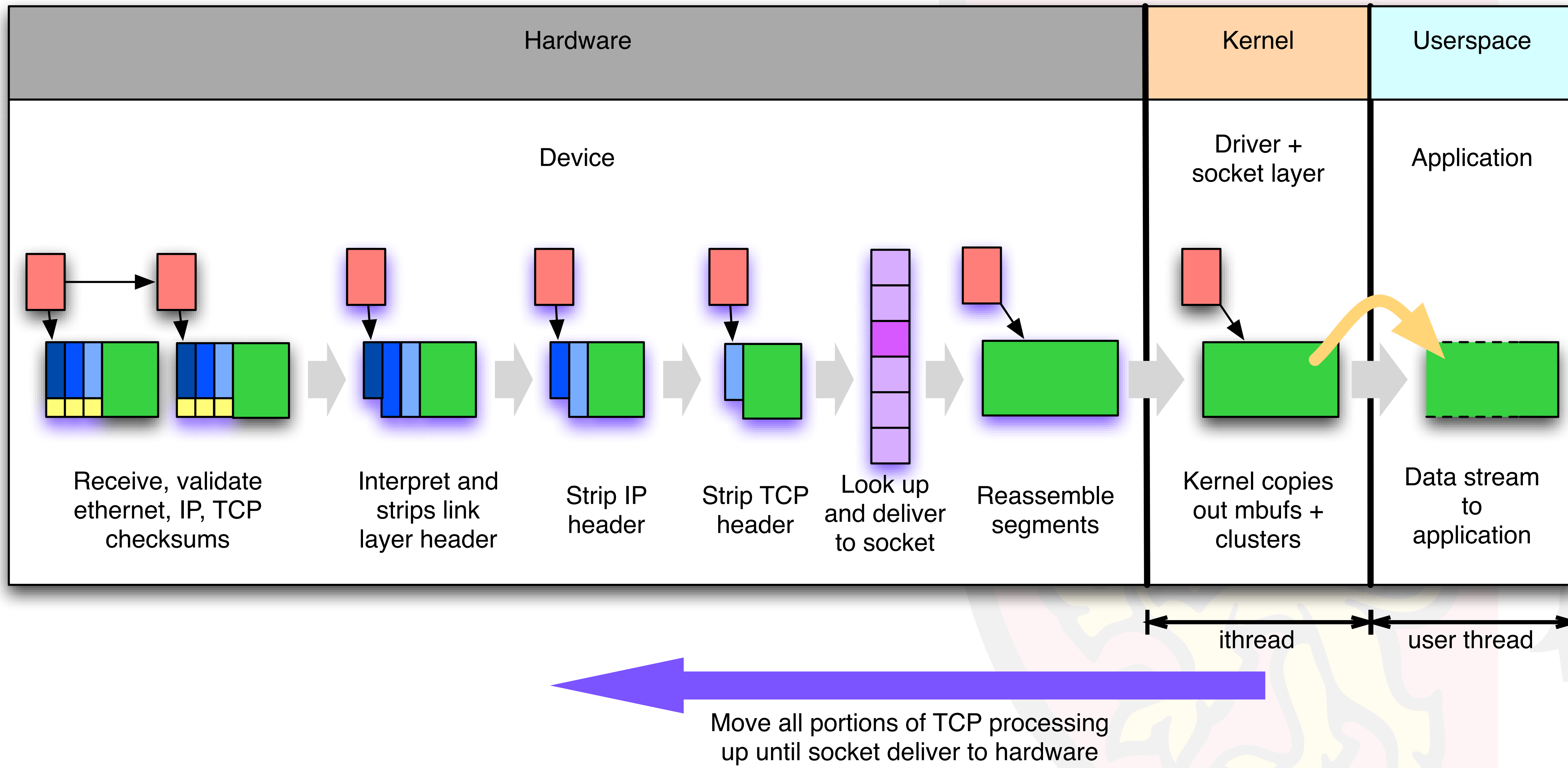
tcp_output()

```
if (len > tp->t_maxseg) {
        if ((tp->t_flags & TF_TSO) && tcp_do_tso &&
            ((tp->t_flags & TF_SIGNATURE) == 0) &&
            tp->rcv_numsacks == 0 && sack_rxmit == 0 &&
            tp->t_inpcb->inp_options == NULL &&
            tp->t_inpcb->in6p_options == NULL && ipsec_optlen == 0)
                tso = 1;
        else {
                len = tp->t_maxseg;
                sendalot = 1;
                tso = 0;
        }
}
...
if (tso) {
        m->m_pkthdr.csum_flags = CSUM_TSO;
        m->m_pkthdr.tso_segsz = tp->t_maxopd - optlen;
}
```

TCP Offload Engine (TOE)

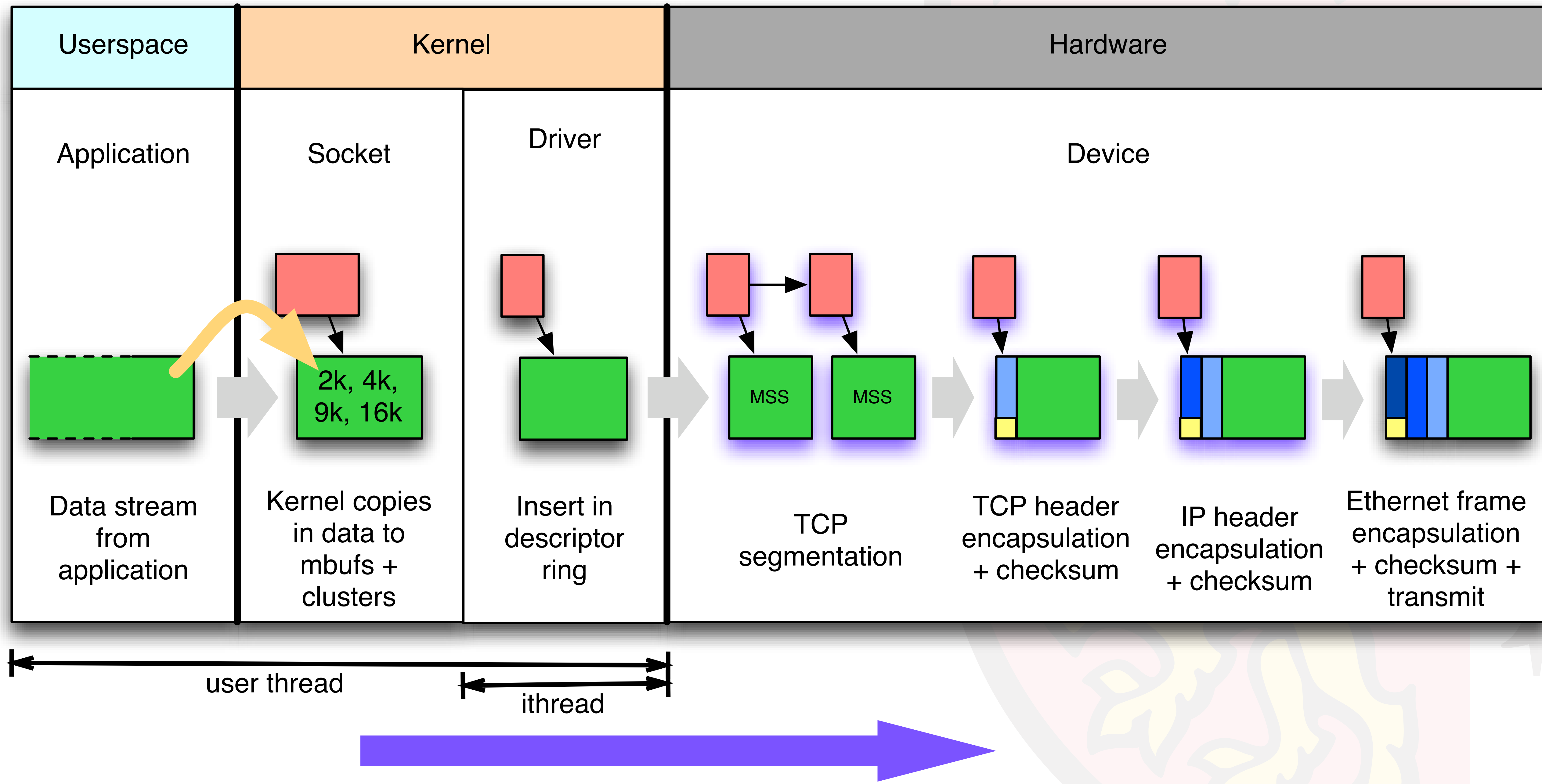
- Hardware implements most or all of TCP
  - TCP state machine, bindings
  - Segmentation reassembly
  - Acknowledgement, pacing
- Sockets deliver directly to driver/hardware
- Driver delivers directly to sockets

TOE input



Hardware
Kernel
Userspace
Device
Driver + socket layer
Application
Receive, validate ethernet, IP, TCP checksums
Interpret and strips link layer header
Strip IP header
Strip TCP header
Look up and deliver to socket
Reassemble segments
Kernel copies out mbufs + clusters
Data stream to application
ithread
user thread
Move all portions of TCP processing up until socket deliver to hardware

TOE output



Move all portions of TCP processing after socket send into hardware

Side effects of offload

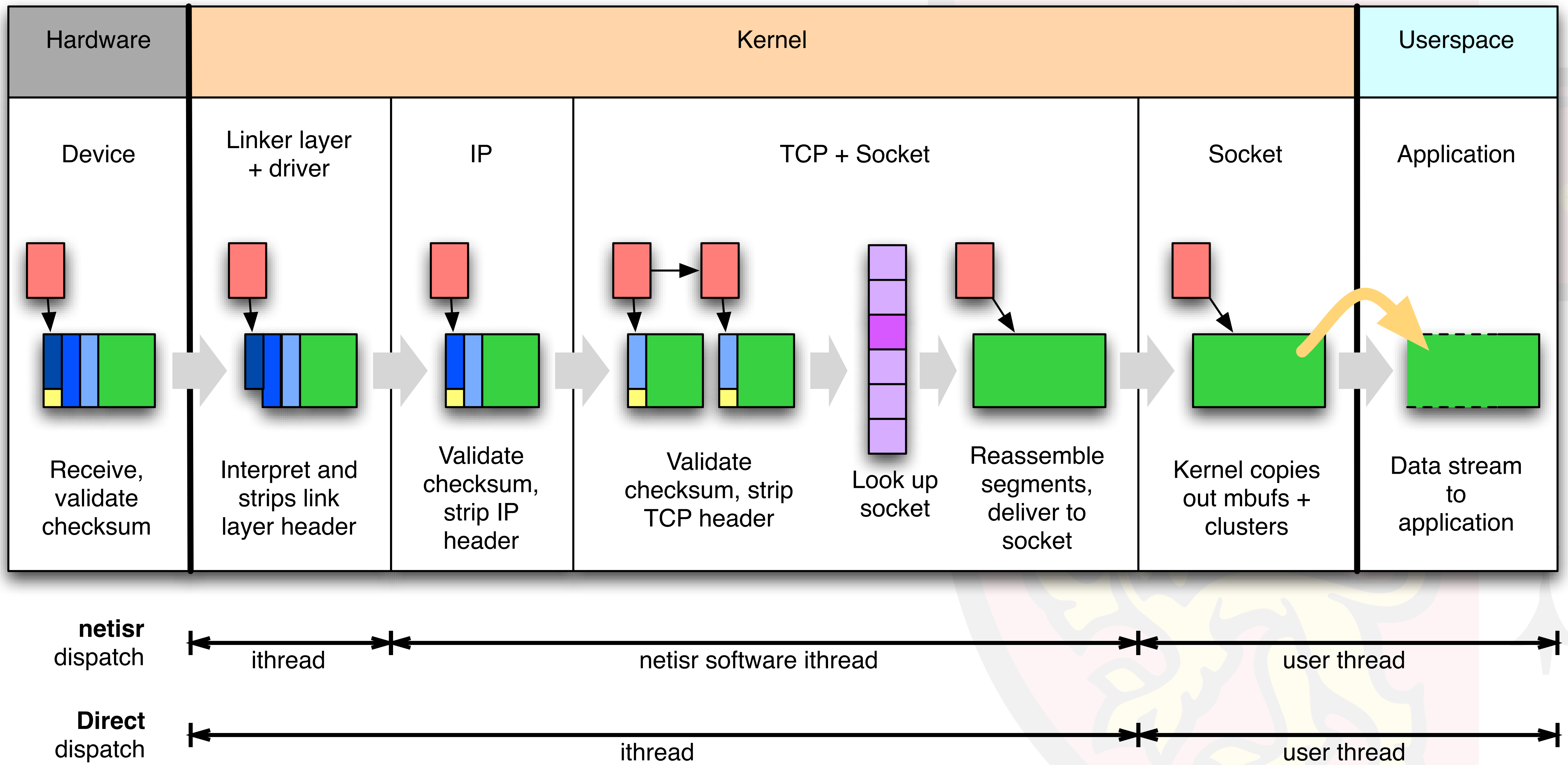
- "Layering violations" not invisible to users
- Hardware bugs harder to work around
- Stack instrumentation below socket
  - BPF, firewalls, traffic management, etc
- TCP protocol behavior
- Not all TOEs equal: SYN, TIMEWAIT, etc.

Software structure optimizations

Direct dispatch

- Stack input processing in three contexts
  - interrupt thread (device driver, link layer)
  - netisr thread (protocol, socket deliver)
  - user/kernel thread (socket copy out)
- netisr limits parallelism, so **directly dispatch** protocol from interrupt context

Direct dispatch



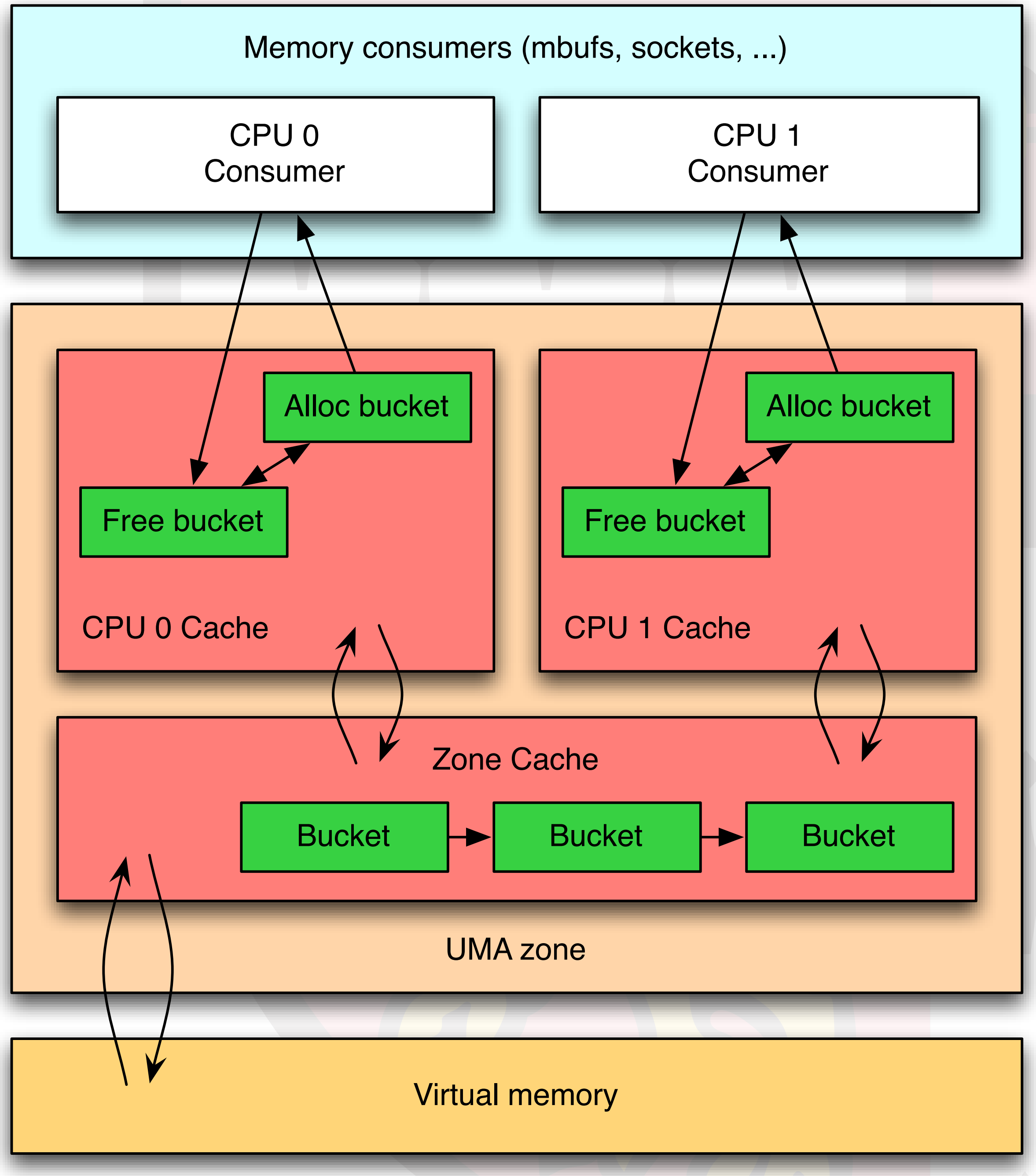
UNIVERSITY OF CAMBRIDGE

SMPng Project

- FreeBSD 3.x introduced SMP support
  - Userspace on multiple processors
  - Kernel on a single processor at a time
- FreeBSD 5.x introduced SMPng
  - Fine-grained kernel locking, parallelism
- 6.x, 7.x increase maturity and performance

UMA - Universal Memory Allocator

- Kernel slab allocator (Bonwick 1994)
- Mature object life cycle: amortize initialization and destruction costs
- Per-CPU caches: encourage locality by allocating memory from CPU where last freed, avoid lock contention



UNIVERSITY OF CAMBRIDGE

Multi-processor network stack

- Apply SMPng architecture to network stack
- Identify and lock key data structures
- Create new opportunities for parallelism
- Project essentially complete, although optimization continues
- All protocols and network device drivers run without the Giant lock

From mutual exclusion to read-write locking

- Initial locking strategy largely mutexes
  - "Mutual exclusion"
- Many data structures require **stability** rather than **mutability** in most uses
- rwlock primitive optimized in FreeBSD 7.1
  - Many mutexes are becoming rwlocks

UDP receive and transmit optimization

- FreeBSD 7.1: fully parallel UDP receive and transmit at socket and protocol layers
- rwlocks for mostly read-only state: connection lists and connections
- Specialized socket send/receive functions avoid stream socket overhead
- Limited by routing, hardware queues

Read-mostly locks

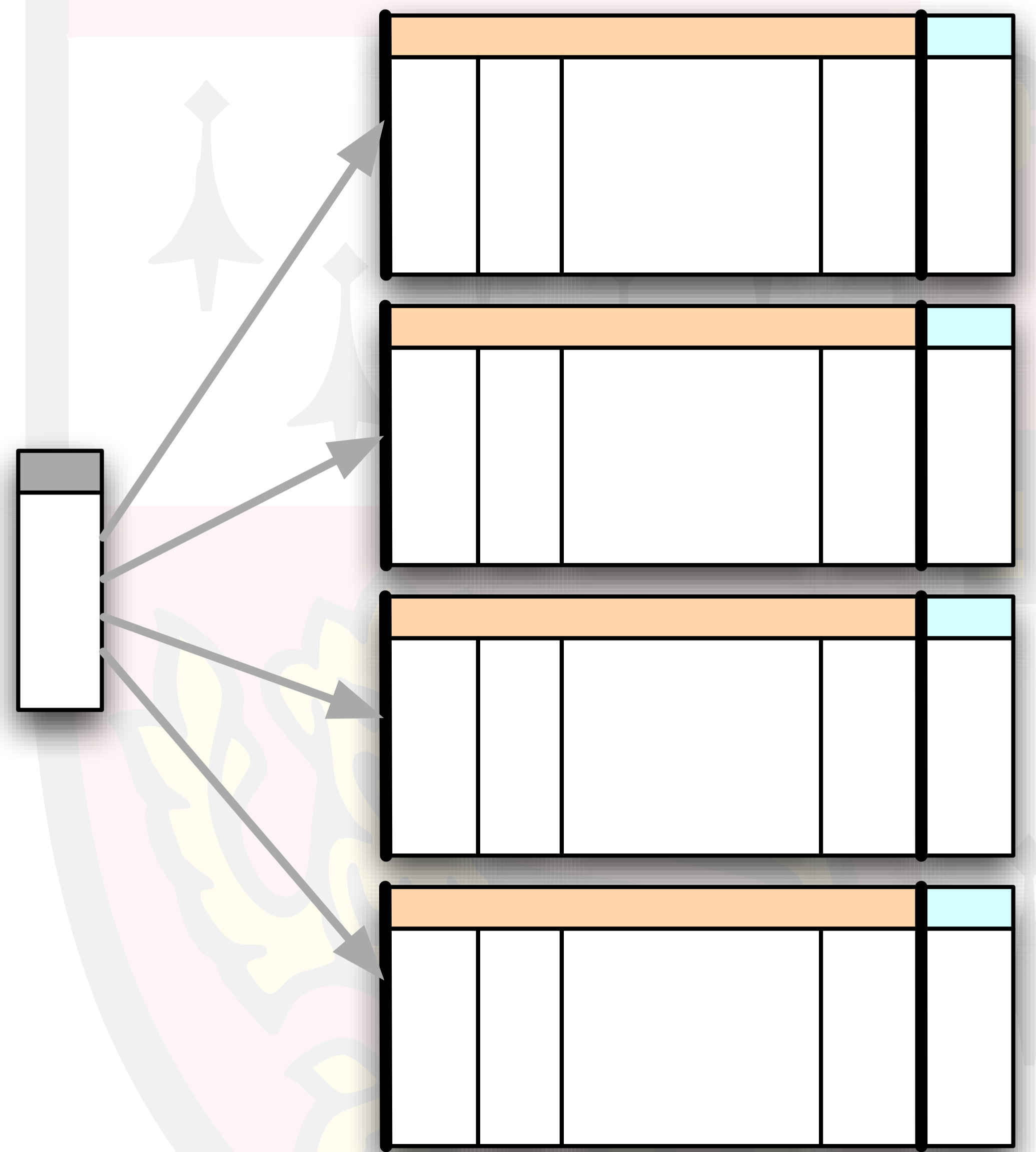
- New primitive in FreeBSD 8.x
  - Synchronization optimized for reads
  - Address/connection lists, firewall rules, ..
- Read acquire with non-atomic instruction
- Write synchronizes CPUs using inter-processor interrupts (IPIs)

Hardware devices as a source of contention

- Locking not just for data structures
  - I/O to hardware also needs serialization for non-atomic sequences
- Ethernet cards one (or more) receivers, transmitters
- Serialization leads to contention, lack of software parallelism

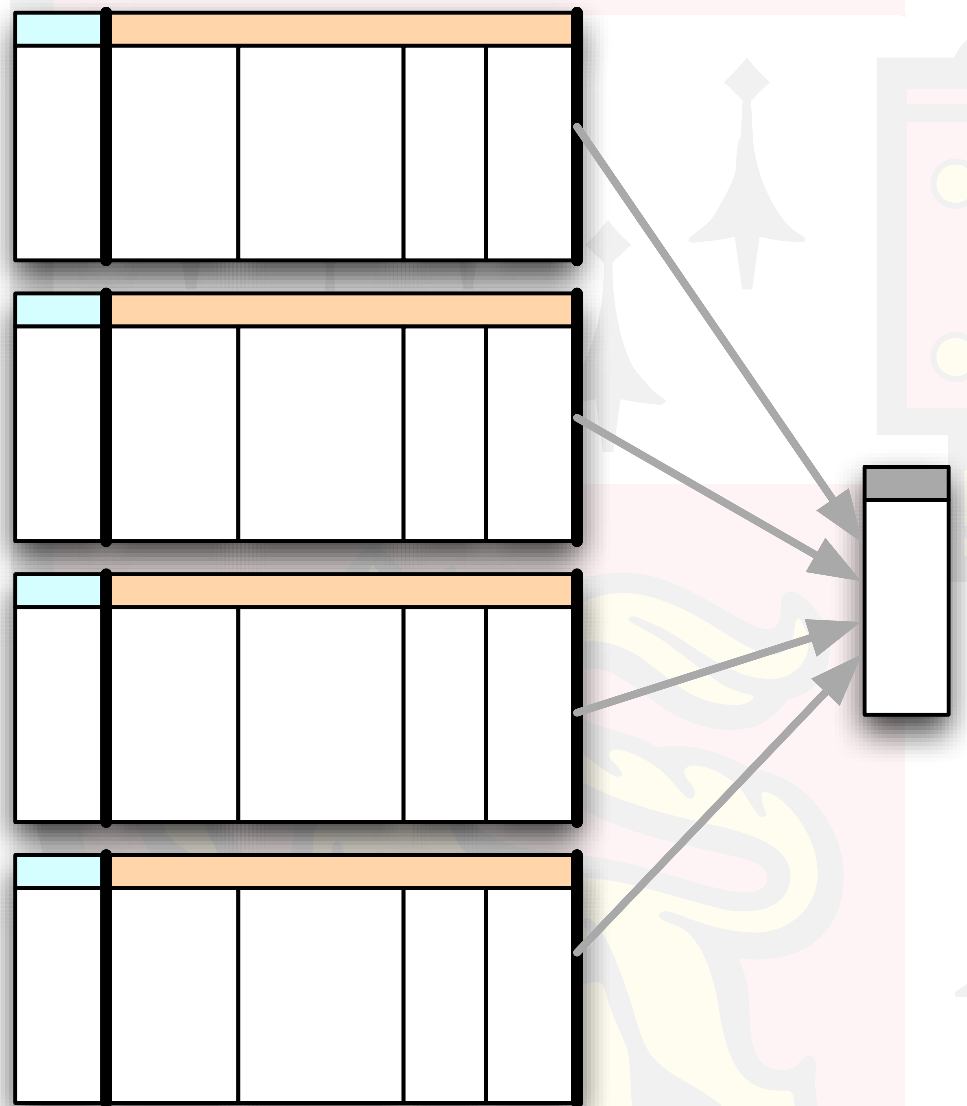
Multiple input queues

- Parallelism for hardware input interface
- Device delivers flows into independent queues using hashing scheme to maintain
- Each queue assigned an ithread to process input queue, allowing parallel processing of input



Multiple output queues

- Hardware exposes multiple independent output queues
- OS assigns work based on flows to maintain ordering relative to flow
- Moderates outgoing queue lock contention



Where next?

- Continue to optimize locking primitives
- Improve granularity on route locking
- Multiple output queue support (8.0)
- Weak CPU affinity on connections (8.0)
- Hashed locks on global structures (8.0)
- Zero-copy BPF (7.2), sockets (today)

Conclusion

- Hardware changes motivate significant operating system changes
  - Cache-centric design
  - Parallelism
  - Hardware offload
- Progression over FreeBSD versions as use of techniques introduced and refined