FreeBSD Network Stack Optimizations for Modern Hardware

Robert N. M. Watson
FreeBSD Foundation

EuroBSDCon 2008
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

1. **Device**: Receive, validate checksum
2. **Linker layer + driver**: Interpret and strips link layer header
3. **IP**: Validate checksum, strip IP header
4. **TCP + Socket**: Validate checksum, strip TCP header, Look up socket, Reassemble segments, deliver to socket
5. **Socket**: Kernel copies out mbufs + clusters
6. **Application**: Data stream to application

**Threads**:
- **ithread**
- **netisr software ithread**
- **user thread**
TCP output path

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Kernel</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Socket</td>
<td>Link layer + driver</td>
</tr>
<tr>
<td>Data stream from application</td>
<td>TCP</td>
<td>Ethernet frame encapsulation,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>insert in descriptor ring</td>
</tr>
<tr>
<td></td>
<td>TCP segmentation, header encapsulation, checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP</td>
<td>Checksum + transmit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- TCP segments: MSS, MSS
- Kernel copies data to mbufs + clusters
- Device

User thread

ithread
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

Move checksum validation from network stack to hardware
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()

```c
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

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Kernel</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Socket</td>
<td>TCP</td>
</tr>
<tr>
<td>Data stream from application</td>
<td>Kernel copies in data to mbufs + clusters</td>
<td>TCP segmentation, header encapsulation</td>
</tr>
</tbody>
</table>

- **2k, 4k, 9k, 16k**
- **256**
- **MSS**
- **MSS**
- **Device**

**User thread**

**iThread**

Move checksum generation from network stack to hardware

[Diagram showing the flow from application to device with various stages of processing.]
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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Kernel</th>
<th>Userspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Linker layer + driver</td>
<td>Application</td>
</tr>
<tr>
<td>Receive, validate ethernet, IP, TCP checksums</td>
<td>Reassemble segments</td>
<td>Data stream to application</td>
</tr>
<tr>
<td></td>
<td>Interpret and strips link layer header</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strip IP header</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strip TCP header</td>
<td>Look up and deliver to socket</td>
</tr>
<tr>
<td></td>
<td>Kernel copies out mbufs + clusters</td>
<td></td>
</tr>
</tbody>
</table>

Move TCP segment reassembly from network protocol to device driver
TCP LRO implementation

igb_rxeof()

```c
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
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

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
if (len > tp->t_maxseg) {
    if ((tp->t_flags & TF_TSO) && tcp_do_tso &&
        ((tp->t_flags & TF_SIGNATURE) == 0) &&
        tp->rcv_num_sacks == 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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Kernel</th>
<th>Userspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive, validate ethernet, IP, TCP checksums</td>
<td>Driver + socket layer</td>
<td>Data stream to application</td>
</tr>
<tr>
<td>Interpret and strips link layer header</td>
<td>Kernel copies out mbufs + clusters</td>
<td></td>
</tr>
<tr>
<td>Strip IP header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strip TCP header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Look up and deliver to socket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reassemble segments</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Move all portions of TCP processing up until socket deliver to hardware**
TOE output

```
Userspace          Kernel                Hardware

Application    ---    Socket               Driver               Device

Data stream     2k, 4k, from application  Kernel copies  TCP       Ethernet frame
                 9k, 16k               in data to encapsulation + encapsulation +
                                  mbufs + checksum + checksum +
                                  clusters                          transmit

User thread

Kernel thread

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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Kernel</th>
<th>Userspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Linker layer + driver</td>
<td>Receive, validate checksum</td>
</tr>
<tr>
<td></td>
<td>IP</td>
<td>Validate checksum, strip IP header</td>
</tr>
<tr>
<td></td>
<td>TCP + Socket</td>
<td>Validate checksum, strip TCP header</td>
</tr>
<tr>
<td></td>
<td>Socket</td>
<td>Look up socket</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reassemble segments, deliver to socket</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel copies out mbufs + clusters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data stream to application</td>
</tr>
</tbody>
</table>

**Direct dispatch**

**netisr dispatch**

**ithread**

**netisr software ithread**

**user thread**

**Direct dispatch**

**ithread**

**user thread**
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
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 ichannel 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