OpenOnload
A user-level network stack

Steve Pope
spope <at> solarflare.com

David Riddoch
driddoch <at> solarflare.com
Brief history of Local Area Networking

- 2.9 Mb/s  Ethernet Invented from May 1973

- 10GBASE-T Ethernet sampled August 2006
Local Area Networking - Parallel Universes

- HPC / R&D
- Ethernet (BASE-T) commodity
HPC/R&D => Bleeding edge network architectures

- Great performance because network access is made in the context of the application process
  - Reduces overheads, but requires user-safe hardware and special libraries
Issues of handling **protocol state** in the application context

- may persist longer than the application (send; exit)
- may require asynchronous advancement (TCP_ACK within TMO)
- may be destroyed (exec)
- may be shared (fork; exec) (SOL_SOCKET/SCM_RIGHT)
- may need to be synchronized (ARP, IP ROUTE)
- may be modified by a corrupt / malicious application

Solutions therefore tend to

- push protocol state / processing into the network
- implement strict programming models (RDMA, Matched send/recv)
OpenOnload Architecture
The OpenOnload architecture

- Network hardware provides a user-safe interface which can route Ethernet packets to an application context based on flow information contained within headers.
Protocol processing can take place both in the application and kernel context for a given flow.
The OpenOnload architecture

- Protocol state is shared between the kernel and application contexts through a protected shared memory communications channel.
OpenOnload Implementation
Performance metrics

- **Overhead**
  - Networking overheads take CPU time away from your application

- **Latency**
  - Holds your application up when it has nothing else to do
  - H/W + flight time + overhead

- **Bandwidth**
  - Dominates latency when messages are large
  - Limited by: algorithms, buffering and overhead

- **Scalability**
  - Determines how overhead grows as you add cores, memory, threads, sockets etc.
Anatomy of kernel-based networking

- Application
- libc
- Socket
- Network interface controller
- H/W driver
- Ethernet
- IP
- TCP / UDP

User-space
Kernel
A user-level architecture?
Direct & safe hardware access
Challenges for user-level networking
Some challenges: exec()
exec() wipes all user-level state
exec() wipes all user-level state
exec() wipes all user-level state
exec() wipes all user-level state
exec() wipes all user-level state
Some challenges: fork()
fork() duplicates your state!
fork() duplicates your state!
fork() duplicates your state!
Some other issues...

- File descriptors (including sockets) can be passed through a UNIX domain socket to another process

- When a process exits, any remaining TCP connections must be shutdown gracefully
  - and queued data must reach the receiver

- The process may not invoke the stack frequently
  - (because it is not scheduled, or just not doing any networking at the moment)
  - but we must still respond to network events in a timely fashion
This leads to some requirements...

- A user-level stack’s state must live outside the user-level process
  - in order to survive exec()
  - so that sockets survive exit()

- Sockets must be represented in the kernel’s fdtable
  - to avoid clashes in the descriptor space

- State must be shareable
  - to support fork()
  - and passing sockets to other processes
The OpenOnload architecture

- Stacks live in the kernel
- and are mapped into user-space
Onload and fork()...
Onload and fork()...just works
Onload and exec()
Onload and exec()
Onload and exec()
Onload and exec()
Onload and exec()
Onload and exec()
Pass-through file descriptors

- Application
  - libonload.so
  - stat()
- libc
- disk
- trusted stack state
  - k-sock
  - onload.ko
  - socket
  - shared stack state
  - H/W driver
  - Ethernet
  - IP
  - TCP / UDP
Pass-through file descriptors

- Application
  - libonload.so
  - libc

- Trusted stack state
  - k-sock
  - onload.ko

- Shared stack state
  - H/W driver
  - Ethernet
  - IP
  - TCP / UDP
The shared state

- Includes everything needed for the data-path:
  - sockets
  - packet buffers
  - v-nic state
  - demux table
  - timers (retransmit, zwin-probes, keep-alive etc.)
  - free resources
  - statistics
  - configuration
Address spaces

- The shared state is mapped into the kernel and potentially multiple independent process address spaces
  - so all ‘pointers’ are indirect
  - packet buffers and sockets are identified by index
  - more generally fields are identified by offset
    - useful for complex data-structures such as timer wheels

- Untrusted user-space code may corrupt the state in arbitrary ways
  - processes that share a stack must trust one-another
    - if they can’t, then they mustn’t share a stack

- Code running in the kernel must be careful when accessing shared state
Untrustworthy children
Untrustworthy children
Untrustworthy children
Protecting the kernel

- Trusted state – kernel only
  - pointers to base of shared memory regions
  - various dimensions and masks
  - resources (wait-queues, demux filter etc.)

- Protecting memory accesses
  - pointers computed using inputs from shared state are made safe by bounds checking or masking

- Avoiding excessive resource consumption
  - loops are bounded (using bounds from trusted state)
Control plane

- ARP table, routing table and interface table
  - single copy shared by all stacks
  - mapped read-only into user-space
  - protected with generation counters
  - TCP sockets cache their route and layer-2 headers
    - validation is just a single compare of a generation counter in the ARP table (no bus-locked instructions)
    - any updates cause generation counters of potentially affected entries in the ARP table to be bumped
      - which pushes potentially affected parties to do a full lookup
  - synchronised with Linux via netlink sockets
Some problems with having a “parallel” stack

- We don’t initially know which interface a socket will use until it is bound or connected
  - it might not be an interface that is “onload enabled”
  - the interface can change

- Listening TCP sockets must be able to accept connections from any and all interfaces
  - not just those that are “onload enabled”
  - including the loopback interface, even if bound to a specific IP

- UDP sockets need to send and receive over multiple interfaces
  - including loopback
We need an O/S socket
We need an O/S socket
Handing over to the kernel stack
Handing over to the kernel stack

- libonload.so
- libc
- socket

trusted stack state
- onload.ko
  - H/W driver
  - Ethernet
  - IP
  - TCP / UDP

shared stack state
Handling network events

- The network controller sends and receives packets and delivers notifications *asynchronously*
- In the traditional architecture, an interrupt is raised which invokes the device driver, which processes notifications and passes received packets to the stack
- Received data is queued at a socket, or copied into a user buffer
- Threads that are blocked waiting for received data (or buffer space) may be woken
- The handling of network events is *asynchronous* with respect to the behaviour of the application
By default, interrupts are *not* enabled

Notifications are delivered to the notification queue, and ignored until...

```c
stack_poll() {
    reprime_hw_timer();
    handle_network_events();
    refill_rx_ring();
    refill_tx_ring();
    process_timers();
}
```
Polling the stack

- `stack_poll()` is invoked at user-level as needed to satisfy application operations on sockets:
  - in `recv()`, but only if the application’s buffer cannot be filled with data already queued
  - in `send()`, but only if further progress might be made
  - in `poll()` and `select()`, but only once for all of the sockets in a stack

- If a thread blocks on a socket, interrupts are enabled
- When an interrupt is raised the ISR invokes `stack_poll()`
Consequences...

- Improved batching *(sometimes)*
- More responsive *(at other times)*
- Improved temporal and spatial cache locality
  - The stack processes network events at exactly the right time: just before *(recv)* or after *(send)* the application has touched the associated data
  - The stack executes on the same processor core as the application thread, minimising cache line bouncing
When the application isn’t invoking us...

- (the system is busy, or the app is blocked elsewhere)
- We must still respond promptly to network events
  - we don’t want our RX ring to empty (packets will be discarded)
  - we don’t want our TX ring to empty (the link may not be fully utilised)
  - we don’t want the other end to retransmit

- Solution: A count-down timer on the NIC
  - reset periodically in stack_poll()
  - raises an interrupt if not reset for too long
  - ISR invokes stack_poll()
Concurrency control

- Mutual exclusion locks are expensive
  - even in the absence of contention
- Contention at user-level is especially expensive
  - spinlocks don’t work well when the lock holder is de-scheduled
- Classic trade-off between:
  - minimising locking overhead (few locks please)
  - minimising contention (fine grained locking please)
Concurrency control

- Classic tradeoff – no, not that simple
  - mutual exclusion is not the only option
  - lock-free algorithms
  - data structures constructed with atomic operations

- However, atomic operations are also expensive
  - 100s of cycles
Concurrency control: our approach

- Do the most common stuff under one lock
  - everything in stack_poll()

- Try to ensure operations invoked frequently by applications do not contend that lock

- Minimise total locking
  - single lock per operation in common case

- Minimise other bus-locked (atomic) operations
Onload locking model
Protecting the receive path
Protecting the receive path
Protecting the receive path

- Stack
- Head
- Extract
- Tail
- Socket
Protecting the receive path
Protecting the receive path

stack  head  tail

socket  extract
Protecting the receive path
Protecting the receive path

- stack
- head
- tail
- socket
- extract
Protecting the receive path

- Stack
- Head
- Extract
- Tail
- Socket
Protecting the receive path

stack

socket

head

extract

tail
Protecting the transmit path

- stack
- send_q
- pre_q
- socket
- retransmit_q

1 → 2
Protecting the transmit path

- Stack
- retransmit_q
- send_q → 1
- pre_q → 2
- socket

Slide 71
Protecting the transmit path

Diagram:
- Stack
- Retransmit queue
- Send queue
- Pre queue
- Socket

Diagram shows the flow from stack to retransmit queue, then through send queue, pre queue, and finally to socket.
Protecting the transmit path

- stack
- send_q
- pre_q
- socket
- retransmit_q

Diagram showing the flow between different components of the transmit path.
Protecting the transmit path

- stack
- send_q
- pre_q
- socket
- retransmit_q

Diagram elements numbered from 1 to 4.
Protecting the transmit path

- stack
- retransmit_q
- send_q
- pre_q
- socket
Protecting the transmit path

Diagram:
- stack
- send_q
- pre_q
- socket
- retransmit_q

1. Inputs from stack and send_q
2. Processed by pre_q
3. Sends data to retransmit_q
4. Outputs to socket
Protecting the transmit path

```
stack

send_q 3 2

retransmit_q

pre_q

socket
```

Slide 77
Protecting the transmit path

Stack

Send_q

Retransmit_q

Pre_q

Socket
Protecting the transmit path

- **stack**
- **send_q**
- **pre_q**
- **socket**
- **retransmit_q**
Protecting the transmit path
Debugging and monitoring

- Application
- libonload.so
- libc

- k-sock
- socket

trusted stack state
onload.ko

shared stack state

- H/W driver
- Ethernet
- IP
- TCP / UDP
stackdump: attaching

Application

libonload.so

libc

stackdump

trusted stack state

onload.ko

k-sock

socket

shared stack state

H/W driver
Ethernet
IP
TCP / UDP
stackdump: attaching

stackdump

trusted stack state

onload.ko

Application

libonload.so

H/W driver
Ethernet
IP
TCP / UDP

socket

k-sock

shared stack state

libc

libc

libc

libc
stackdump: attaching
stackdump: example output

```
ci_netif_dump: id=10
  lock=10000000 UNLOCKED  nics=1 primed=0
  sock_bufs: max=1024 n_allocated=12
  pkt_bufs: max_mtu=1876 size=2048 max=32768
  alloc=1344 free=704 async=89

ci_netif_dump_vi: id=10 intf=0 vi_instance=74
  evq: cap=2048 current=11ba3d8 is_32_evs=0 is_ev=1
  txq: cap=511 spc=498 level=13 removed=1543110
  rxq: cap=511 spc=6 level=505 removed=1868567
  time: netif=f12a576 poll=f12a576 now=f12a57d
  (diff=0.012sec)
```
TCP 10:11 lcl=172.16.129.26:42235 rmt=172.16.129.27:65534 ESTABLISHED
lock: 10000000 UNLOCKED
sleep_seq=221174 wake_rq= flags=
s_flags: NONB NDLY NODELAY FILTER_BOUND
rcvbuf=129940 sndbuf=131072 rx_errno=0 tx_errno=0 so_error=0
tcpflags: TSO WSCL SACK ESTAB NONBCON
snd: up=7849ee3f una-nxt-max=7849ee3f-784a8e6f-784be42b enq=784a8e6f
   send=0(0) max=91 pre=0 inflight=41008(29) wnd=128492
cwnd=396144+0 used=0 sssthresh=65535 bytes_acked=7005032 Open
rcv: nxt-max=c5441644-c54611d8 current=c54611d8 FAST
   rob_n=0 recv1_n=11 recv2_n=0 wnd_adv=129940 cur=129940 usr=0
async: rx_put=-1 rx_get=-1 tx_head=-1
eff_mss=1448 smss=1460 amss=1460 used_buffers=40 uid=511 wscl s=1 r=1
srtt=01 rttvar=000 rto=110 retrans=0 dups=0 zwins=0 rto=0 frec=0
timers: rto(198ms[f12a5e4]) zwin(91ms[f12a5a9])
Performance
Some performance results

- Test platform: typical commodity server
  - Intel clovertown 2.3 GHz quad-core xeon (x1)
    1.3 GHz FSB, 2 Gb RAM
  - Intel 5000X chipset
  - Solarflare Solarstorm SFC4000 (B) controller, CX4
  - Back-to-back
  - RedHat Enterprise 5 (2.6.18-8.el5)
Performance: Latency and overhead

- TCP ping-pong with 4 byte payload
- 70 byte frame: 14+20+20+12+4

<table>
<thead>
<tr>
<th></th>
<th>½ round-trip latency (microseconds)</th>
<th>CPU overhead (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>4.2</td>
<td>--</td>
</tr>
<tr>
<td>Kernel</td>
<td>11.2</td>
<td>7.0</td>
</tr>
<tr>
<td>Onload</td>
<td>5.3</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Performance: Streaming bandwidth

![Graph showing streaming bandwidth performance comparison between Onload and Kernel K2 opus.](image)
Performance: Streaming bandwidth

- Onload (1 cpu)
- Kernel (1 cpu)
Performance: Streaming bandwidth

![Graph](image-url)
Performance: UDP transmit

- Message rate:
  - 4 byte UDP payload (46 byte frame)

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>Onload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sender</td>
<td>473,000</td>
<td>2,030,000</td>
</tr>
</tbody>
</table>
Performance: UDP transmit

- Message rate:
  - 4 byte UDP payload (46 byte frame)

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>Onload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sender</td>
<td>473,000</td>
<td>2,030,000</td>
</tr>
<tr>
<td>2 senders</td>
<td>532,000</td>
<td>3,880,000</td>
</tr>
</tbody>
</table>
Performance: UDP receive

![Graph showing UDP receive performance](image)
Performance: MPI pallas – kernel stack

- Default settings: decent bandwidth
- Interrupt moderation off: decent latency
Performance: MPI pallas

- Onload vs. kernel (default settings)
Scalability

- Scales across applications
  - completely separate stacks, no cross talk

- Scales with cores
  - stack per core (or set of cores)
  - use receive-side-scaling to direct packets to stacks
  - scales with threads

- A whole other talk...
Conclusions

- **Ethernet compatibility**
  - Simple, Robust, Cheap, Easy to manage

- **Sockets/POSIX compatibility**
  - Support existing applications / programming models

- **TCP/IP compatibility**
  - Internetworking with the existing compute environment
  - Single-ended acceleration

- Improved performance!
OpenOnload Open Source

- OpenOnload available as Open Source (GPLv2)
  - Please contact us if you’re interested

- Compatible with x86 (ia32, amd64/emt64)

- Currently supports SMC10GPCIe-XFP and SMC10GPCIe-10BT NICs
  - Could support other user-accessible network interfaces

- Very interested in user feedback
  - On the technology and project directions