Onload

User Guide

UG1586 (v1.2) July 31, 2023

Table of Contents

Chapter 1: What's New ... **9**
New Features in OpenOnload-8.1.0 ... 10
New Features in OpenOnload-8.0.2 ... 12
New Features in OpenOnload-8.0.1 ... 13
New Features in OpenOnload-8.0.0 ... 13
New Features in OpenOnload-7.1.3 ... 16
New Features in OpenOnload-7.1.2 ... 16
New Features in OpenOnload-7.1.1 ... 17
New Features in OpenOnload-7.1.0 ... 18
New Features in OpenOnload-7.0.0 ... 20
Change History ... 22

Chapter 2: X3 Low Latency Quickstart ... **23**
Software Installation ... 23
Test Setup ... 25
Reference System Specification ... 27
Latency Tests ... 28
Latency Test Results ... 31
Latency against Payload ... 32
Further Information ... 36

Chapter 3: Overview ... **37**
Contrasting with Conventional Networking ... 38

Chapter 4: Installation ... **41**
Onload Distributions ... 41
Hardware and Software Supported Platforms ... 42
Onload and Network Adapter Drivers ... 44
Removing an Existing Installation ... 46
Pre-install Notes ... 47
Building and Installing from a Tarball ... 48
Building and Installing from a DKMS Package ... 51

Building and Installing from a Source RPM.......... 53
Building and Installing from a Source DEB.......... 55
Onload Kernel Modules.......... 56
Configuring the Network Interfaces.......... 57
Installing Netperf and sfnettest.......... 57
Running Onload.......... 57
Testing the Onload Installation.......... 58
Applying an Onload Patch.......... 58
Kernel and OS Upgrades.......... 60

Chapter 5: Tuning Onload.......... 61
System Tuning.......... 61
Spinning, Polling and Interrupts.......... 64
Onload Deployment on NUMA Systems.......... 66
Interrupt Handling for the sfc Driver.......... 69
Performance Jitter.......... 76
Using Onload Tuning Profiles.......... 78
Benchmark Testing.......... 80
Application-Specific Tuning.......... 80
Worked Examples.......... 81

Chapter 6: Onload Functionality.......... 88
Onload Transparency.......... 88
Onload Stacks.......... 88
Virtual Network Interface (VNIC).......... 89
Functional Overview.......... 89
Onload with Mixed Network Adapters.......... 89
Maximum Number of Network Interfaces.......... 90
Allowlist and Denylist for Interfaces.......... 90
Onload Accelerated Process IDs.......... 91
File Descriptors, Stacks, and Sockets.......... 91
System Calls Intercepted by Onload.......... 92
Linux Sysctls.......... 92
Namespaces.......... 95
User-space Control Plane Server.......... 95
Changing Onload Control Plane Table Sizes.......... 98
SO_BINDTODEVICE.......... 100
Multiplexed I/O.......... 100
Wire Order Delivery.......... 103

Stack Sharing 105
Application Clustering 106
eXpress Data Path (XDP) 108
Zero-Copy API 110
Debug and Logging 110

Chapter 7: Timestamps 112
Introduction 112
Software Timestamps 112
Hardware Timestamps 113
Example Timestamping Applications 116

Chapter 8: Onload and TCP 119
TCP Operation 119
TCP Handshake, SYN and SYNACK 120
TCP SYN Cookies 120
TCP Socket Options 120
TCP Level Options 122
TCP File Descriptor Control 123
TCP Congestion Control 123
TCP SACK 124
TCP QUICKACK 125
TCP Delayed ACK 125
TCP Dynamic ACK 125
Limit Duplicate ACK Rate 126
Limit Challenge ACK Rate 126
TCP Loopback Acceleration 126
TCP Striping 128
TCP Connection Reset on RTO 128
ONLOAD_MSG_WARM 129
Listen/Accept Sockets 130
Socket Caching 130
Shared Local Ports 133
Scalable Filters 134
Transparent Reverse Proxy Modes 137
Transparent Reverse Proxy on Multiple CPUs 138
Performance in Lossy Network Environments 139
Initial Sequence Number Caching 140
Urgent Data Processing 141

TIMEWAIT Assassination.......... 141

Chapter 9: Onload and UDP.......... 142

UDP Operation.......... 142
Socket Options.......... 142
Source Specific Socket Options.......... 143
Onload Sockets vs. Kernel Sockets.......... 144
Send and Receive Paths for UDP Sockets.......... 144
Fragmented UDP.......... 145
User Level recvmmsg for UDP.......... 146
User-Level sendmmsg for UDP.......... 146
UDP sendfile.......... 146
Multicast Replication.......... 146
Multicast Operation and Stack Sharing.......... 147
Multicast Loopback.......... 150
Hardware Multicast Loopback.......... 150
IP_MULTICAST_ALL.......... 152

Chapter 10: Packet Buffers.......... 153

Network Adapter Buffer Table Mode.......... 153
Huge Pages.......... 154
How Onload Uses Packet Buffers.......... 155
Physical Addressing Mode.......... 158
Programmed I/O.......... 159
CTPIO.......... 161

Chapter 11: Interfaces.......... 168

Bonding, Link Aggregation and Failover.......... 168
Teaming.......... 169
VLANS.......... 170
MACVLAN.......... 171
IPVLAN.......... 171
Accelerated pipe().......... 172

Chapter 12: Onload and Virtualization.......... 174

Overview.......... 174
Onload and Linux KVM.......... 174
Onload and NIC Partitioning.......... 176
Onload in a Docker Container.......... 178

Chapter 13: Limitations **185**
Introduction 185
Changes to Behavior 186
Limits to Acceleration 191
Known Issues with Epoll 195
Configuration Issues 197

Chapter 14: Onload Change History **202**
Mapping Onload Versions 202
Features 202
Environment Variables 203
Module Options 204
Adapter Net Drivers 205

Appendix A: Parameter Reference **206**
Parameter List 206

Appendix B: Meta Options **280**
Environment Variables 280

Appendix C: Build Dependencies **282**
General 282
Red Hat Enterprise Linux 8.x 284

Appendix D: Onload Extensions API **285**
Common Components 286
Stacks API 293
Zero-Copy API 303
Receive Filtering API 313
Templated Sends API 315
Delegated Sends API 318

Appendix E: onload_stackdump **328**
General Use 328
List Onloaded Processes 329
List Onloaded Threads, Priority and Affinity 329
List Onload Environment Variables 329
TX PIO Counters 330
Send RST on a TCP Socket 330

Removing Zombie and Orphan Stacks.......... 330
Snapshot vs. Dynamic Views.......... 331
Monitoring Receive and Transmit Packet Buffers.......... 331
TCP Application Statistics.......... 333
The onload_stackdump LOTS Command.......... 335
Onload Stackdump Filters.......... 366
Remote Monitoring.......... 367

Appendix F: sfnettest.......... 370
sfnt-pingpong.......... 370
sfnt-stream.......... 373
Running Without Spinning.......... 376

Appendix G: onload_tcpdump.......... 377
Building onload_tcpdump.......... 377
Using onload_tcpdump.......... 377

Appendix H: ef_vi.......... 381
Components.......... 381
Compiling and Linking.......... 381
Documentation.......... 382

Appendix I: onload_iptables.......... 383
How it Works.......... 383
Features.......... 384
Rules.......... 384
Preview Firewall Rules.......... 385
Error Messages.......... 387

Appendix J: eflatency.......... 389
eflatency.......... 389

Appendix K: Management Information Base.......... 391
Host.......... 391
Container.......... 391
Namespaces.......... 391
List Available Options.......... 392
Tables.......... 392

Appendix L: X2 Low Latency Quickstart.......... 399

Software Installation....399
Test Setup....400
Reference System Specification....403
Latency Tests....404
Latency Test Results....408
Latency against Payload....408
Further Information....413

Appendix M: X2 Throughput Quickstart....414
Software Installation....414
Test Setup....415
Reference System Specification....416
Throughput....417
HTTP connections....418
Further Information....420

Appendix N: Additional Resources and Legal Notices....421
Finding Additional Documentation....421
Support Resources....422
References....422
Revision History....422
Please Read: Important Legal Notices....423

Chapter 1

What's New

This issue of the user guide identifies changes and new features introduced in the OpenOnload 8 release.

For a complete list of features and enhancements refer to the *Release Notes* and the *Release Change Log* available from the NIC Software and Drivers web page.

OpenOnload-8 includes the 5.3.12.1023 net driver.

Users should refer to `ReleaseNotes-sfc` in the distribution package for details of changes to the adapter driver.

Onload Version Numbering

Starting with version 7.0.0, OpenOnload version numbering has been revised to align with the EnterpriseOnload and net driver version numbering scheme.

OpenOnload-7.0.0 was the successor to onload-201811-u1.

Version numbers have four components. Their meaning is shown in the following table for the example of version 7.1.2.141:

Table 1: **Component Values and Meanings for Version Number 7.1.2.141**

Value	Meaning
7	Major version
1	Minor version - feature release
2	Minor version - bug fix update release
141	Build number

***Note*:** Use the NIC Software and Drivers web page to download supported Onload distributions, including EnterpriseOnload and OpenOnload.

Use the Onload GitHub web page for an unsupported preview of Onload work in progress.

The openonload.org website has been phased out.

New Features in OpenOnload-8.1.0

This is a feature release of Onload:

- It has extended support for the AMD Alveo™ X3522 low latency network adapters.
- It supports new Operating System and kernel releases.
- It includes features and fixes from OpenOnload 8.0.2.51 onwards.
- It refreshes the sfc net driver included in the package.

Linux Distribution Support

This package can be installed on:

- Red Hat Enterprise Linux 7.9
- Red Hat Enterprise Linux 8.4 - 8.8
- Red Hat Enterprise Linux 9.0 - 9.2
- SuSE Linux Enterprise Server 15 SP4 - SP5
- Canonical Ubuntu Server LTS 20.04 and 22.04
- Debian 11 “Bullseye”
- Debian 12 “Bookworm”
- Linux kernels 4.15 - 6.3

Drivers for X3522

Onload on the Alveo X3522 requires a compatible net driver version from https://www.xilinx.com/products/boards-and-kits/alveo/x3.html#software such as v1.5.2.0 at the time of writing. Older kernels such as on RHEL 7.9 and RHEL 8.4 also require the auxiliary bus driver to be installed (available from the same download page). See *Alveo X3522 Release Notes* and documentation.

Controlling Access to RX Queues on X3522

Onload 8.1 introduces a new mechanism for enabling users to reserve a hardware receive queue for a given singular application. This mechanism is currently only supported on X3522 network adapters.

This is via a new filter flag (EF_FILTER_FLAG_EXCLUSIVE_RXQ) that is applied via `ef_filter_spec_init()`. With this change, users can ensure the following properties for their application.

- Other applications will be unable to snoop on traffic filtered to this application.
- This application can guarantee that it will not receive any packets for which it did not explicitly add a filter.

In addition, this version of Onload introduces a new function, `ef_filter_spec_set_dest()`. This API enables the user to specify which hardware receive queue an application can listen on.

Due to the additional granularity afforded by the above granular controls, this version has necessitated the deprecation of the following broader ef_vi application flags:

- EF_VI_EFCT_UNIQUEUE
- EF_VI_RX_EXCLUSIVE

Further details of specific exclusivity conditions can be found under the EF_FILTER_FLAG_EXCLUSIVE_RXQ documentation in the *ef_vi User Guide* (SF-114063-CD).

Hardware Filter IDs on X3522

New features have been added to ef_vi to allow applications to reuse the result of the hardware packet parsing and filtering which an X3522 has performed on RX. The new `ef_vi_filter_query()` will return the ID which the NIC has allocated for a previously-added filter, which will match the `ef_event.rx_ref.filter_id` field returned by an `ef_vi_poll()` call.

***Note*:** ef_vi will not always use on-NIC filtering to satisfy a call to `ef_vi_filter_add()`, and so `ef_vi_filter_query()` has the ability to report that no filter ID is available. Applications should be written to be robust to this scenario.

IPv6 on X3522

When using IPv6 with an Onload accelerated socket on X3522 received IPv6 fragments will be dropped.

Transmit Path Warming on X3522

Onload 8.1 adds new API features to allow the X3522 transmit code path to be exercised without sending any data on the wire. This can potentially improve latency when transmitting after a period of inactivity, as recently executed code is more likely to be cached. This new functionality is available via both ef_vi and TCPDirect APIs.

- **ef_vi:** Use `efct_vi_start_transmit_warm()` and `efct_vi_stop_transmit_warm()` to start and stop warming. Call transmit functions while warming to exercise the code. These functions are only usable for EFCT (X3) NICs.

- **TCPDirect:** Use `zft_send_single_warm()` and `zfut_send_single_warm()`. These functions are not new, but Onload 8.1 extends them to be usable for any NIC architecture including EFCT (X3).

***Note*:** Transmit path warming is also supported forOnload applications via the ONLOAD_MSG_WARM socket option. This feature is hardware independent, and so has always been available to the X3522.

Deprecation of RHEL 7

Future feature releases of OpenOnload might not include support for Red Hat Enterprise Linux 7.x. Long Term Support for Onload 8.1 including on RHEL 7.9 will be available in the form of EnterpriseOnload 8.1. Please ask your sales representative for details.

New Features in OpenOnload-8.0.2

This is an update release of Onload that extends support to recent operating system updates and fixes various bugs.

See the accompanying `ChangeLog` for a list of bugs fixed.

Library Versioning and Backward Compatibility

Backward compatibility between Onload 8.0.1 or later and Onload 7 or earlier has been restored as follows:

- ef_vi:
    - Restored ABI compatibility.
    - Restored ability to link `libciul1.a` with `libonload_zf_static.a` into single executable.
- Onload extension library:
    - Restored API compatibility.
    - Restored ABI compatibility.
    - Reverted library major version number increment.

As a consequence of this change, backward compatibility between versions 8.0.1 and 8.0.0 of Onload has *not* been preserved and ef_vi applications or Onload applications using extensions built with the Onload 8.0.0 extension library need to be rebuilt with Onload 8.0.1 or later.

Checksum Validation and Non-TCP/UDP with ef_vi on X3522

Previous releases of Onloadwith X3522 support classified all non-TCP/UDP packets as discards. This release brings the behavior on X3522 in line with earlier network adapters, where by default other protocols generate a normal event, in this case an RX_REF event.

However, this change has implications for applications that rely on the hardware checksum validation. The hardware will only validate a checksum if it can successfully parse a packet as a protocol for which it does validation. If the protocol header is corrupted in such a way that the packet is not recognized as a given protocol, it might be reported as a valid packet of an unknown protocol that checksum validation is not performed for.

To allow an application to distinguish a packet that has been parsed and checksummed as a given protocol new discard types have been added. These allow the application to request that discard events are generated for any packets for protocols that are not expected by that application. For example, an application that expects to receive only UDP packets can set the discard mask for the VI to mark any packets of non-TCP/UDP L4 protocols as discards. This means that anything that is not a discard will have had the checksum validated successfully.

For more details refer to the documentation for `ef_vi_receive_set_discards()` and `ef_vi_rx_discard_err_flags` in the *ef_vi User Guide* (SF-114063-CD).

New Features in OpenOnload-8.0.1

This is an update release of Onload that improves support for the new X3-series network adapters from AMD, extends support to recent operating system updates, includes a new sfc net driver and fixes various bugs.

See the accompanying `ChangeLog` for a list of bugs fixed.

New Features in OpenOnload-8.0.0

This is a major feature release of Onload:

- It adds support for the new X3-series network adapters from AMD.
- It incorporates a significantly updated version of the `sfc` net driver for X2-series adapters.
- It supports major new Operating System releases including Red Hat Enterprise Linux 9.0.
- It includes features and fixes from OpenOnload-7.1.3.202 onwards.

TCPDirect is now supplied as a separate package.

Linux Distribution Support

This package can be installed on:

- Red Hat Enterprise Linux 7.8 - 7.9
- Red Hat Enterprise Linux 8.2 - 8.6
- Red Hat Enterprise Linux 9.0
- SuSE Linux Enterprise Server 15 SP1 - SP3
- Canonical Ubuntu Server LTS 20.04 and 22.04
- Canonical Ubuntu Server 21.10
- Debian 10 "Buster"
- Debian 11 "Bullseye"
- Linux kernels 4.15 - 5.15

Packaging, Source and Licensing Changes

The Onload control plane is now distributed as source code rather than a binary and built from source with Onload. The control plane, ef_vi and various other components are now provided with a BSD 2-Clause license.

TCPDirect has been moved into a separate package which should be installed with a corresponding version of Onload. For example, tcpdirect-8.0.0.x should be used with onload-8.0.0.y.

The table below summarizes the situation for each component:

Table 2: **Onload Packaging, Source, and Licensing**

	Supplied as	Licence	Package
Onload	Source	GPL-2.0	onload-8.0.0
Onload headers needed for ef_vi applications	Source	GPL-2.0 or BSD 2-Clause	onload-8.0.0
Onload extensions stub library	Source	BSD 2-Clause	onload-8.0.0
Control plane	Source	BSD 2-Clause	onload-8.0.0
User-modifiable scripts and examples	Source	BSD 2-Clause	onload-8.0.0
ef_vi	Source	BSD 2-Clause	onload-8.0.0
TCPDirect	Binary	Proprietary	tcpdirect-8.0.0

Use with X3-series Adapters

Applications are unaffected, Onload deals with underlying architectural differences, and the data visible to the application via the sockets API is unaffected.

The following architectural differences affect receive:

- Maximum of 256 filters.
- Filtering is by 3-tuple (remote IP address and port).
- No receive-side scaling.

The following architectural differences affect transmit:

- Maximum of 15 applications for unshared TX queues (16 queues).
- All transmission is by reliable CTPIO with stable performance.
- No hardware checksums.
- Frequent polling is essential to achieve reasonable packet rate due to small transmit buffer in hardware.

The following difference affect tuning:

- On X3522 the `xilinx_efct` net driver must perform regular work to keep the RX path supplied with buffers, including when using Onload/TCPDirect/ef_vi.
- Ensure that `xilinx_efct` interrupts are delivered to CPU cores that have sufficient capacity and can be handled in a timely fashion to avoid underruns.

For more detailed information about X3-series adapters and their differences from predecessors, see the *Alveo X3522 User Guide* (UG1523).

Installing for X3-series Adapters

The following components must be installed to use X3-series network adapters, in the given order of dependency:

- The auxiliary bus driver.
- The `xilinx_efct` net driver.
- OpenOnload-8.0.0.

Note: The auxiliary bus driver is supplied as a back-ported standalone component. It is only necessary for kernels that are built without `CONFIG_AUXILIARY_BUS`. The following supported distributions include built-in support and so do not need the driver:

- RedHat Enterprise Linux 8.5 and later.
- Canonical Ubuntu Server 20.10 and later.

For more detailed information about installing X3-series adapters and their drivers, see the *Alveo X3522 Installation Guide* (UG1522).

Python 3 Required

The Python scripts have been updated and now require a Python 3 interpreter.

Deprecation

This and successor releases do not support the acceleration of 32-bit applications.

The deprecated support for adding steering filters through the sfc_affinity driver and the sfcaffinity tool has been removed. sfc_affinity_config has been modified to use the sfc_resource driver.

New Features in OpenOnload-7.1.3

This is an update release of Onload that extends support to recent operating system updates and fixes various bugs.

See the accompanying `ChangeLog` for a list of bugs fixed.

Linux Distribution Support

Linux distribution support has been changed:

- Support recent Linux distributions (RHEL 8.5, Debian 11 "Bullseye").

See Hardware and Software Supported Platforms for a full list of supported OS and kernels.

Deprecation

TCP Urgent pointer handling is deprecated. EF_TCP_URG_MODE is set to 0 by default. This code will be removed completely in the next major feature release of Onload. Please contact AMD support if you need this feature.

New Features in OpenOnload-7.1.2

This is an update release of Onload that extends support to recent operating system updates and fixes various bugs.

See the accompanying `ChangeLog` for a list of bugs fixed.

Linux Distribution Support

Linux distribution support has been changed:

- Support recent Linux kernel versions (up to 5.12).
- Support recent Linux distributions (RHEL 8.4).

See Hardware and Software Supported Platforms for a full list of supported OS and kernels.

XDP/eBPF Filtering

This release fixes unnecessary high latency when XDP is enabled.

To support BPF filtering for recent kernels, it was necessary to give the control plane server the `CAP_SYS_ADMIN` capability. This is now disabled by default so the `cplane_track_xdp` module option must be provided, for example via the modprobe configuration.

New ef_vi Event EF_EVENT_TYPE_RESET

A new event, `EF_EVENT_TYPE_RESET` is issued by ef_vi to notify applications of fatal hardware restarts.

Deprecation

Support for accelerating 32-bit applications will be removed in the next major feature release of Onload, but 32-bit applications will continue to be supported by Onload-7.1.x bugfix releases.

New Features in OpenOnload-7.1.1

This is an update release of Onload that extends support to recent operating system updates, includes a new sfc net driver and fixes various bugs.

See the accompanying `ChangeLog` for a list of bugs fixed.

setuid/setgid

The `onload` binaries were previously installed as setuid/setgid by default. This is no longer the case. The previous behavior can be selected by passing the `--setuid` flag to `onload_install`.

EF_TCP_COMBINE_SENDS_MODE

A new option controls how Onload fills packets in the TCP send buffer. In the default mode (set to 0) Onload will prefer to use all the space at the end of a previous packet before allocating a new one. When set to 1, Onload will prefer to allocate a new packet for each new send.

In all cases this is a hint rather than guaranteed behavior. There are conditions where the preference indicated by this option will not be possible, e.g. memory pressure might cause packets in the send queue to be combined. `MSG_MORE` and `TCP_CORK` can override this option when set. The zero-copy sends API can also use the segmentation provided by the caller's buffers.

For full control of message segmentation the delegated sends API can be used. Setting this option can affect the capacity of send buffers belonging to sockets in this stack and increase packet buffer usage. It can also reduce efficiency as packets will be allocated for each send call rather than being able to reuse one that is already available.

***Note*:** Setting this option is only recommended for those who have an explicit need to avoid combined or split sends.

Linux Distribution Support

Linux distribution support has been changed:

- Support recent Linux kernel versions (up to 5.9).
- Support recent Linux distributions (RHEL 7.9, RHEL 8.3, SLES 15 sp2).
- Deprecate older Linux distributions (RHEL 6.8, RHEL 7.5, RHEL 8.0, SLES 15, Ubuntu LTS 16.04).

See Hardware and Software Supported Platforms for a full list of supported OS and kernels.

New Features in OpenOnload-7.1.0

This release of Onload adds system-level interface blacklisting, provides improvements for XDP and TCPDirect, includes various bug fixes and refreshes the sfc net driver included in the package.

This release is the successor to the 7.0.0.176 version of OpenOnload and Cloud Onload and is provided in a single package which simplifies feature activation.

See the accompanying `ChangeLog` for a list of bugs fixed.

Feature Activation

Activation files are no longer used in this Onload release. Neither activation files nor NIC activation are required for most Onload features, including IPv6, XDP and UDP acceleration. TCPDirect and ultra low latency NIC features still require NIC activation.

The 'cloud' build profile is still required to enable IPv6 and XDP. This is to optimize performance for users who do not require these features and to ensure compatibility with older OSs.

For further information, please contact your sales representative.

TCPDirect

TCPDirect includes the following improvements:

- TCPDirect applications can now process incoming packet data during receipt.
- TCPDirect is now supported on full-feature firmware.

For full details, refer to the *TCPDirect User Guide* (SF-116303-CD).

System-level Interface Denylisting

In addition to the per-stack EF_INTERFACE_BLACKLIST and EF_INTERFACE_WHITELIST environment variables, a system-level configuration has been added. See Allowlist and Denylist for Interfaces.

Onload Remote Monitor Enhancement

The Onload Remote Monitor has been enhanced to add a new `orm_json_lib` library, which can be used by customers to build their own custom monitoring solutions. See Remote Monitoring.

eBPF/XDP

The implementation of XDP features is now delegated to the currently-running kernel. This expands the set of supported XDP features to those provided by the kernel but means that XDP is no longer supported on older OSs that do not provide the corresponding features natively.

For further details of Onload XDP support, see eXpress Data Path (XDP).

Modified Configuration Options

The EF_UDP_CONNECT_HANDOVER configuration option has been extended.

This option now also accepts a value of `2`, which will cause all UDP sockets to be handed over when calling `connect()`, regardless of whether the socket could have been accelerated. See EF_UDP_CONNECT_HANDOVER.

Linux Distribution Support

Linux distribution support has been changed:

- Support recent Linux kernel versions (up to 5.5).
- Support recent Linux distributions (RHEL 7.8, RHEL 8.2, SLES 12 sp5 and Ubuntu 20.04).
- Deprecate older Linux distributions (RHEL 7.4, SLES 12 sp3 and Ubuntu 19.04).

See Hardware and Software Supported Platforms for a full list of supported OS and kernels.

Deprecation

The ability to insert filters to steer kernel traffic using `sfcaffinity` is deprecated and will be removed in a future release. Instead `ethtool` should be used for this purpose.

New Features in OpenOnload-7.0.0

RHEL 8.x Dependencies

Users of Red Hat Enterprise Linux version 8.x should be aware of the build dependencies for Red Hat Enterprise Linux 8.x.

Activation Files and Build Profiles

Cloud Onload in Onload-7.0.0 requires:

- A Cloud Onload activation file to enable all Cloud Onload specific features
- Build with the cloud build profile

Without a Cloud Onload activation file, the default Onload activation file enables all standard Onload features.

In all cases an Onload AppFlex activation key is required on the adapter.

eXpress Data Path (XDP)

Support for XDP allows the user to insert eBPF programs very early into the packet receive path.

eBPF programs are executed before received packets reach the adapter driver or network stack. Dropped packets incur minimal CPU overhead and are not allocated socket memory buffers, so packet processing is very fast with almost no overheads.

For further details of Onload XDP support, see eXpress Data Path (XDP).

Equal Cost Multipath Routing

ECMP will attempt to distribute link load sharing by routing packets along multiple paths of equal cost.

IPv6 Acceleration

Acceleration of IPv6 traffic is added as a compile-time option to Onload-7.0.0. Refer to limitations of IPv6 Traffic.

Extensions API Timestamps

The EF_RX_TIMESTAMPING_ORDERING environment variable allows the user to select either NIC hardware timestamps or timestamps generated by an external equipment in cPacket trailer format.

For more information, refer to onload_ordered_epoll_wait.

The `onload_timestamping_request()` function supports received packet timestamps from multiple sources with sub-nanosecond resolution.

Timestamps can be generated by the network adapter and/or timestamps applied by an external equipment using the cPacket trailer format.

See onload_timestamping_request for details.

IPVLAN

Onload 7.0.0 will accelerate network traffic over layer 2 IPVLAN interfaces. For further details refer to IPVLAN.

Change History

Chapter 14: Onload Change History is updated with every revision of this document to include the latest Onload features, changes or additions to environment variables and changes or additions to Onload module options.

Chapter 2

X3 Low Latency Quickstart

This chapter demonstrates how to achieve very low latency coupled with minimum jitter on a system fitted with an X3 series network adapter and using the OpenOnload kernel-bypass network acceleration middleware.

The procedure will focus on the performance of the network adapter for TCP and UDP applications running on Linux, using the AMD supplied open source sfnettest network benchmark test tools, and also the industry-standard Netperf network benchmark application.

The results of these tests can be found in Latency Test Results, and Latency against Payload.

Note: Please read the supplied `ONLOAD_LICENSE` file regarding the disclosure of performance test results.

Software Installation

IMPORTANT! *Before installing Onload, you must ensure that the auxiliary bus driver, the xilinx_efct net driver and the correct firmware versions are installed. See the Alveo X3522 Installation Guide (UG1522).*

For example, for the reference system described later in this chapter:

```
[root@server-N]# ethtool -i <interface>
driver: xilinx_efct
version: 0.21.0.0
firmware-version: 1.11.1.1
```

Onload

Before Onload network and kernel drivers can be built and installed the system must support a build environment capable of compiling kernel modules. Refer to Appendix C: Build Dependencies for more details.

1. Download the `onload-<version>.tgz` file from the NIC Software and Drivers web page.
2. Unpack the tar file using the tar command:

   ```
   # tar -zxvf onload-<version>.tgz
   ```

3. Run the `onload_install` command from the `onload-<version>/scripts` subdirectory:

   ```
   # ./onload-<version>/scripts/onload_install
   ```

Refer to Driver Loading - NUMA Node to ensure that drivers are affinitized to a core on the correct NUMA node.

TCPDirect

Before TCPDirect can be built and installed:

- The system must support a build environment capable of compiling kernel modules. Refer to Appendix C: Build Dependencies for more details.
- The system must provide the libstdc++ package. This is an additional build dependency for TCPDirect, To add this package:

  ```
  # yum install libstdc++-static
  ```

- You must already have installed the following:
  - the auxiliary bus driver
  - the `xilinx_efct` network driver
  - Onload.

To build and install TCPDirect:

1. Set the ONLOAD_TREE environment variable to the path of the unpacked Onload build tree:

   ```
   # export ONLOAD_TREE=<path>/onload-<version>
   ```

2. Download the `tcpdirect-<version>.tgz` file from the NIC Software and Drivers web page.
3. Unpack the tar file using the tar command:

   ```
   # tar -zxvf tcpdirect-<version>.tgz
   ```

4. Enter the top directory of the unpacked TCPDirect tree:

   ```
   # cd tcpdirect-<version>
   ```

5. Run the `make` command:

   ```
   # make
   ```

Netperf

Netperf is available as a package for most OS distributions.

Netperf can also be downloaded from https://github.com/HewlettPackard/netperf

- Unpack the compressed zip file using the unzip command:

```
# unzip netperf-master.zip
```

- Refer to the `INSTALL` file within the distribution for instructions.

Following installation the `netperf` and `netserver` applications are typically located in the `/usr/local/bin` subdirectory.

Sfnettest

Download the `sfnettest` source from https://github.com/Xilinx-CNS/cns-sfnettest.

Unpack the downloaded source if necessary, for example:

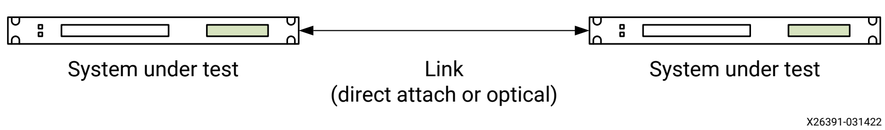
```
# unzip cns-sfnettest-master.tgz
```

Run the `make` utility from the `src` subdirectory to build the `sfnt-pingpong` and other test applications.

Test Setup

The following figure identifies the required physical configuration of two servers equipped with supported network adapters connected back-to-back.

Figure 1: **Test setup**



- Two servers are equipped with supported network adapters and connected with a single cable between the supported interfaces.
- The supported interfaces are configured with an IP address so that traffic can pass between them. Use `ping` to verify connection.
- Onload, sfnettest and netperf are installed on both machines.

If required, tests can be repeated with a switch on the link to measure the additional latency delta using a particular switch.

BIOS Settings

Make the following BIOS settings on both machines:

1. Enable Turbo Boost (sometimes called Turbo Mode).
2. Enable CStates.
3. Disable any of the following settings that are present:
   - Virtualization Technology (also called VT-d/VT-x)
   - IOMMU.

   These are similar in their effect, and typically only one will be present.

Pre-Test Configuration

The following configuration options are applicable to RHEL8 systems.

First, set some configuration options that decrease latency for Onload acceleration technologies. On both machines:

1. Add the following options to the kernel configuration line in `/boot/grub/grub.conf`:

   ```
   isolcpus=<comma separated cpu list> nohz=off iommu=off intel_iommu=off
   mce=ignore_ce nmi_watchdog=0
   ```

2. Stop the following services on the server:

   ```
   systemctl stop cpupower
   systemctl stop irqbalance
   systemctl stop firewalld
   ```

3. Allocate huge pages. For example, to configure 1024 huge pages:

   ```
   # sysctl -w vm.nr_hugepages=1024
   ```

   To make this change persistent, update `/etc/sysctl.conf`. For example:

   ```
   # echo "vm.nr_hugepages = 1024" >> /etc/sysctl.conf
   ```

   For more information refer to Allocating Huge Pages.

4. Consider the selection of the NUMA node, as this affects latency on a NUMA-aware system. Refer to Onload Deployment on NUMA Systems.
5. Disable interrupt moderation.

   ```
   # ethtool -C <interface> rx-usecs 0 adaptive-rx off
   ```

Now perform the following configuration to improve latency without Onload.

Note: These configuration changes have minimal effect on the performance of Onload.

1. Set interrupt affinity such that interrupts and the application are running on different CPU cores but on the same processor package.

   a. Use the following command to identify the interrupts used by the receive queues created for an interface:

   ```
   # cat /proc/interrupts | grep <interface>
   ```

   The output lists the IRQs. For example:

   ```
    46:      75744      68481      60581      90520   PCI-MSI 524288-
   edge      enp1s0f0np0-rx-0
    48:      77959     109946      51941      55480   PCI-MSI 524289-
   edge      enp1s0f0np0-rx-1
    50:      68880      70980      90226      65240   PCI-MSI 524290-
   edge      enp1s0f0np0-rx-2
    52:     117159      96120      47240      34807   PCI-MSI 524291-
   edge      enp1s0f0np0-rx-3
    54:          0          0          0          0   PCI-MSI 524292-
   edge      enp1s0f0np0-tx-4
   ```

   b. Direct the listed IRQs to unused CPU cores that are on the same processor package as the application. For example, to direct IRQs 34-38 to CPU core 2 (where cores are numbered from 0 upwards), using `bash`:

   ```
   # for irq in {46..54..2}
   > do
   > echo 04 > /proc/irq/$irq/smp_affinity
   > done
   ```

2. Set an appropriate tuned profile:

   - The tuned network-latency profile produces better kernel latency results:

     ```
     # tuned-adm profile network-latency
     ```

   - If available, the cpu-partitioning profile includes the network-latency profile, but also makes it easy to isolate cores that can be dedicated to interrupt handling or to an application. For example, to isolate cores 1-3:

     ```
     # echo "isolated_cores=1-3" \
          > /etc/tuned/cpu-partitioning-variables.conf
     # tuned-adm profile cpu-partitioning
     ```

3. Enable the kernel “busy poll” feature to disable interrupts and allow polling of the socket receive queue. The following values are recommended:

   ```
   # sysctl net.core.busy_poll=50 && sysctl net.core.busy_read=50
   ```

Reference System Specification

The following measurements were recorded on Intel® Ice Lake servers. The specification of the test systems is as follows:

- DELL PowerEdge R650 servers equipped with Intel Xeon Gold 6334 CPU @ 3.60 GHz, 64 GB RAM.
- BIOS configured as specified in BIOS Settings.
- AMD Alveo™ X3522 NIC (driver and firmware – see Software Installation).
- Direct attach cable linking the NICs:
  - 10 Gb cable for measurements at 10 Gb
- Red Hat Enterprise Linux 8.4 (x86_64 kernel, version 4.18.0-305.el8.x86_64).
- OS configured as specified in Pre-Test Configuration

  The `tuned` cpu-partitioning profile has been enabled, configured to isolate all cores except for core 0, to reduce jitter and remove outliers.
- OpenOnload distribution: 8.0.
- sfnettest version 1.5.0.
- netperf version 2.7.1.

It is expected that similar results will be achieved on any Intel based, PCIe Gen 4 server or compatible system.

Latency Tests

This section describes various latency tests.

X3 series network adapters use cut through PIO (CTPIO). Packets to be sent are streamed directly over the PCIe bus to the network port, bypassing the main adapter transmit datapath. For more information refer to CTPIO.

The tests are repeated for different CTPIO modes:

- cut-through CTPIO.
- store and forward CTPIO.

***Note*:** These different CTPIO modes require changes to the command lines, noted below.

The command lines given below use the `taskset` command to run the tests on core 1. Change this as necessary, to use an appropriate isolated core on your test system.

Layer 2 ef_vi Latency

ef_vi is a network layer 2 API.

ef_vi test applications can be found in:

```
onload-<version>/build/gnu_x86_64/tests/ef_vi
```

Run the `eflatency` UDP test application on both systems:

```
[sys-1]# taskset -c 1 eflatency <mode> -s <payload> pong <interface>
```

```
[sys-2]# taskset -c 1 eflatency <mode> -s <payload> ping <interface>
```

where:

- `<mode>` is omitted for cut-through CTPIO, or is `-p` for store and forward CTPIO
- `<payload>` is the payload size, in bytes
- `<interface>` is the interface to use.

The output gives various diagnostic information (ef_vi version, payload and frame length, number of iterations and warmups, and mode). It also identifies mean RTT, which is halved to give the mean ½ RTT latency.

Note: Appendix J: eflatency describes the `eflatency` application, command line options and provides example command lines.

TCPDirect Latency

TCPDirect test applications can be found in:

```
tcpdirect-<version>/build/gnu_x86_64/tests/zf_apps/static
```

Run the `zfudppingpong` application on both systems:

```
[sys-1]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zfudppingpong -s <payload> pong <sys-1_ip>:20000 <sys-2_ip>:20000
```

```
[sys-2]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zfudppingpong -s <payload> ping <sys-2_ip>:20000 <sys-1_ip>:20000
```

or run the `zftcppingpong` application on both systems:

```
[sys-1]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zftcppingpong -s <payload> pong <sys-1_ip>:20000
```

```
[sys-2]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zftcppingpong -s <payload> ping <sys-1_ip>:20000
```

where:

- `<interface>` is the interface to use

- `<mode>` is the CTPIO mode to use, which is `ct` for cut-through CTPIO, or `sf` for store and forward CTPIO
- `<payload>` is the payload size, in bytes
- `<sys-1_ip>` is the IP address of `sys-1`
- `<sys-2_ip>` is the IP address of `sys-2`.

The output identifies mean RTT, which is halved to give the mean ½ RTT latency.

Onload Latency with netperf

You can measure Onload latency with standard tools. This test identifies how to use `netperf`.

Run the `netserver` application on system-1:

```
[sys-1]# pkill -f netserver
[sys-1]# onload --profile=<profile> taskset -c 1 netserver
```

and the `netperf` application on system-2:

```
[sys-2]# onload --profile=<profile> taskset -c 1 \
    netperf -t <test> -H <sys-1_ip> -l 10 -- -r <payload>
```

where:

- `<profile>` is `latency-best` for cut-through CTPIO, or `latency` for store and forward CTPIO
- `<test>` is `UDP_RR` or `TCP_RR`, as appropriate
- `<payload>` is the payload size, in bytes
- `<sys-1_ip>` is the IP address of `sys-1`.

The output identifies the transaction rate per second, from which:

mean ½ RTT = (1 / transaction rate) / 2

Onload Latency with sfnt-pingpong

You can also measure Onload latency with the `sfnt-pingpong` application.

Note: The latencies measured with `sfnt-pingpong` are almost identical to the latencies measured with `netperf` in Onload Latency with netperf.

Run the `sfnt-pingpong` application on both systems:

```
[sys-1]# onload --profile=<profile> taskset -c 1 sfnt-pingpong
```

```
[sys-2]# onload --profile=<profile> taskset -c 1 sfnt-pingpong \
    --affinity "1;1" <protocol> <sys-1_ip>
```

where:

- `<profile>` is `latency-best` for cut-through CTPIO, or `latency` for store and forward CTPIO
- `<protocol>` is `udp` or `tcp`, as appropriate
- `<sys-1_ip>` is the IP address of `sys-1`.

The output identifies mean, minimum, median and maximum (nanosecond) ½ RTT latency for increasing packet sizes, including the 99% percentile and standard deviation for these results.

Latency Test Results

The table below shows the results of running the tests described in Latency Tests. The times given are ½ RTT latency for a 1 byte message.

Table 3: **½ RTT Latency for a 1 Byte Message**

Acceleration	Protocol	10 Gb	Notes	Description
ef_vi	UDP	797 ns	eflatency	Layer 2 ef_vi Latency
TCPDirect	UDP	830 ns	zfudppingpong	TCPDirect Latency
	TCP	846 ns	zftcppingpong	TCPDirect Latency
Onload	UDP	1040 ns	netperf	Onload Latency with netperf
		1060 ns	sfnt-pingpong	Onload Latency with sfnt-pingpong
	TCP	1140 ns	netperf	Onload Latency with netperf
		1150 ns	sfnt-pingpong	Onload Latency with sfnt-pingpong

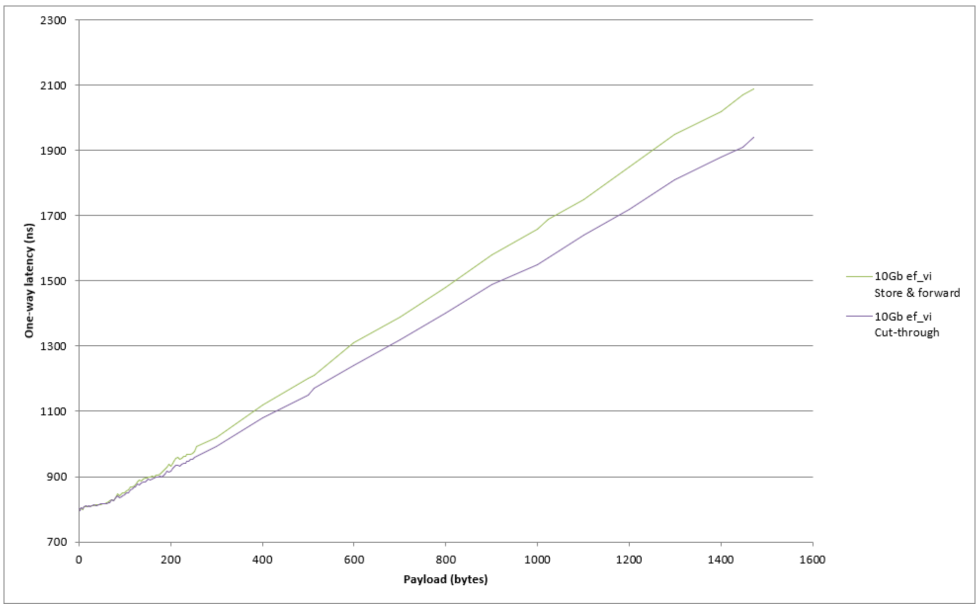
These tests have also been repeated with different payloads, to generate the graphs in Latency against Payload.

Latency against Payload

Latency for UDP Payloads at 10 Gb

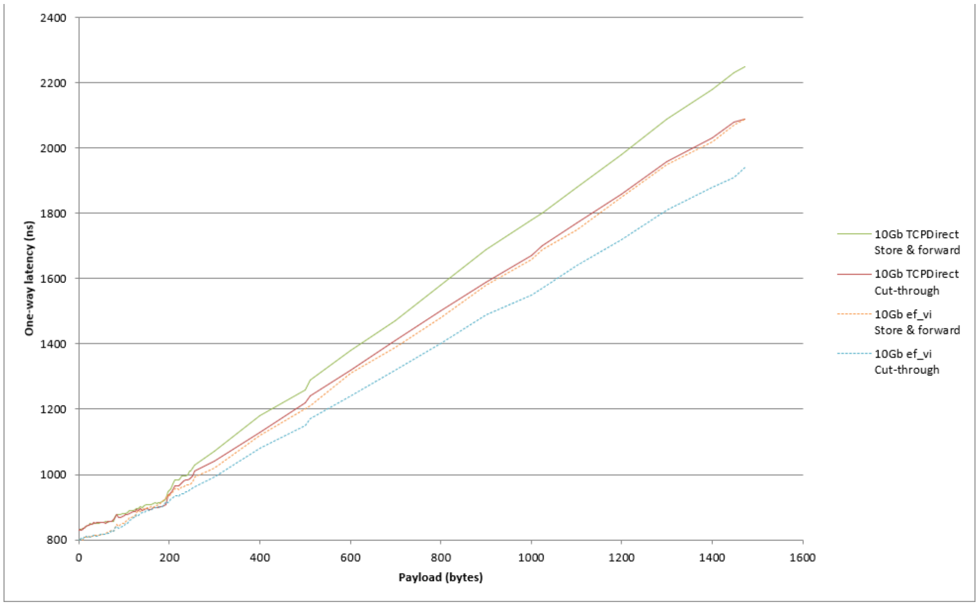
The following figure shows the latency for different UDP payloads using the ef_vi eflatency application (see Layer 2 ef_vi Latency).

Figure 2: **Latency for different UDP payloads at 10 Gb using ef_vi**



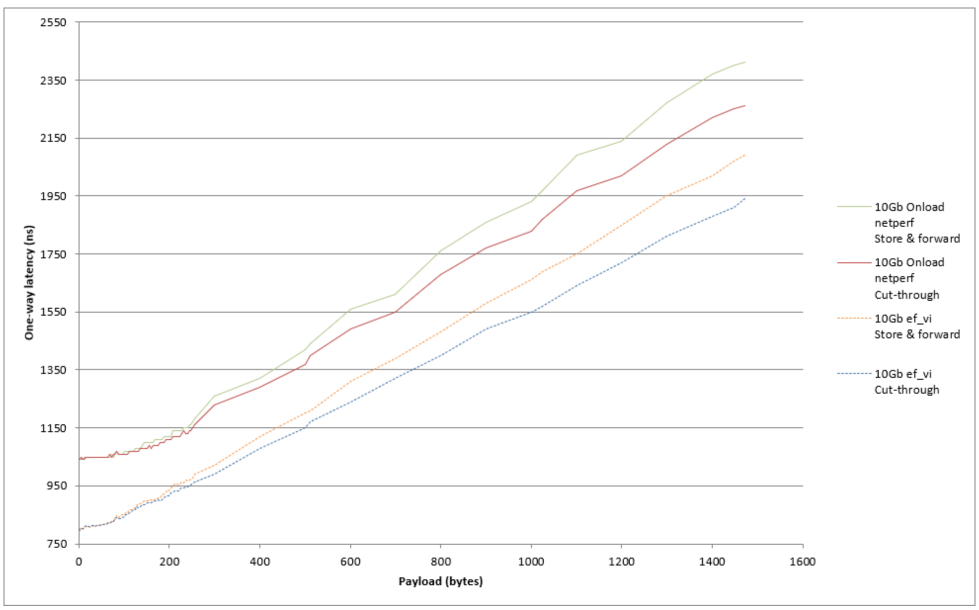
The following figure shows the latency for different UDP payloads using the TCPDirect zfudppingpong application (see TCPDirect Latency). The ef_vi latency is also shown as a baseline.

Figure 3: **Latency for different UDP payloads at 10 Gb using TCPDirect**



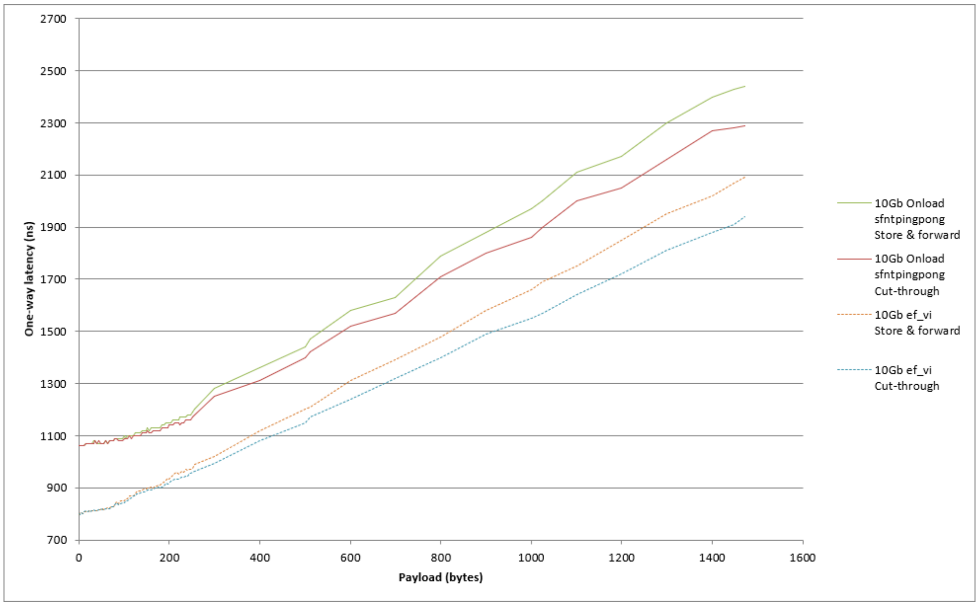
The following figure shows the latency for different UDP payloads using Onload with the netperf application (see Onload Latency with netperf). The ef_vi latency is also shown as a baseline.

Figure 4: **Latency for different UDP payloads at 10 Gb using Onload with netperf**



The following figure shows the latency for different UDP payloads using Onload with the sfnt-pingpong application (see Onload Latency with sfnt-pingpong). The ef_vi latency is also shown as a baseline.

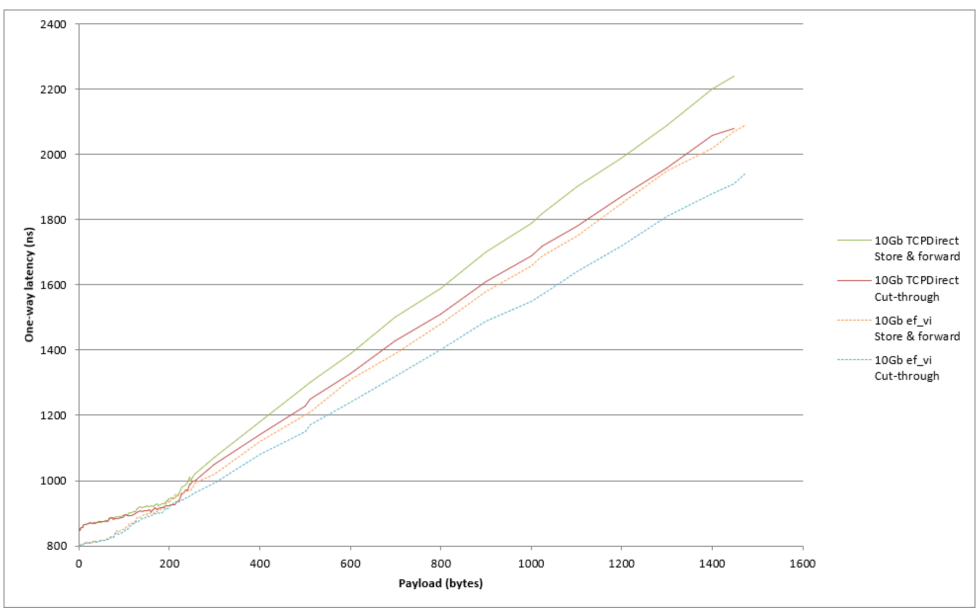
Figure 5: **Latency for different UDP payloads at 10 Gb using Onload with sfnt-pingpong**



Latency for TCP Payloads at 10 Gb

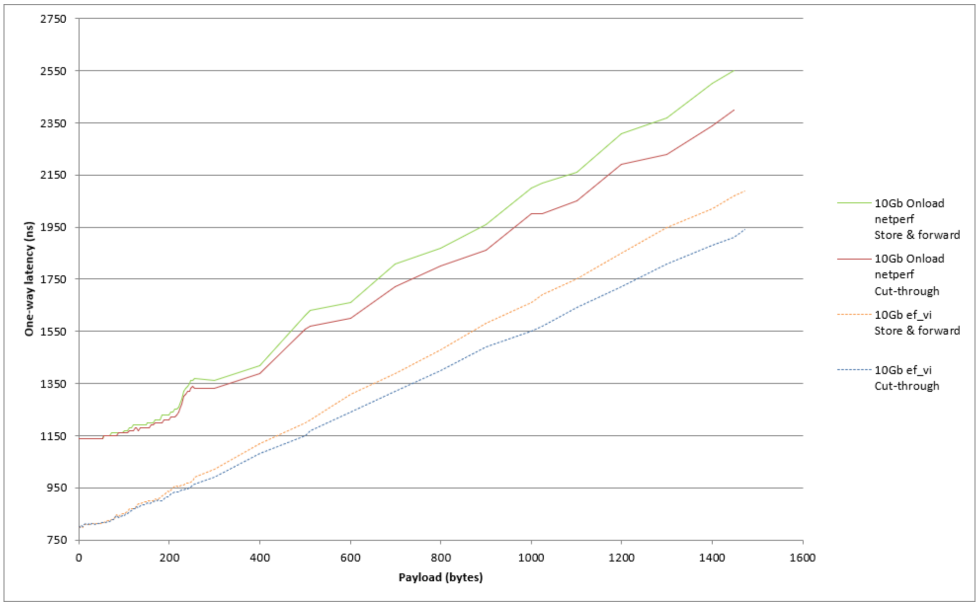
The following figure shows the latency for different TCP payloads using the TCPDirect zftcppingpong application (see TCPDirect Latency). The ef_vi latency is also shown as a baseline.

Figure 6: **Latency for different TCP payloads at 10 Gb using TCPDirect**



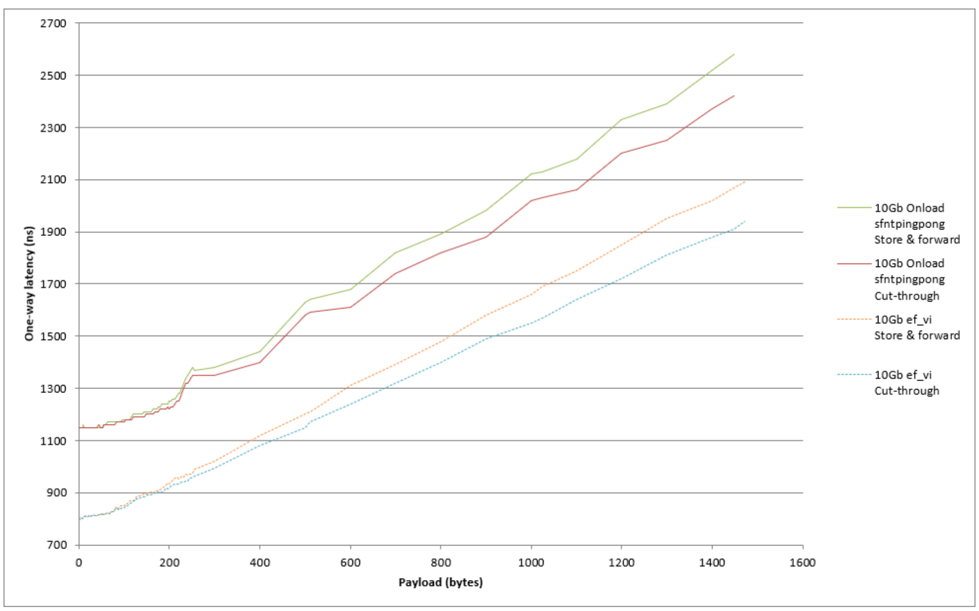
The following figure shows the latency for different TCP payloads using Onload with the netperf application (see Onload Latency with netperf). The ef_vi latency is also shown as a baseline.

Figure 7: **Latency for different TCP payloads at 10 Gb using Onload with netperf**



The following figure shows the latency for different TCP payloads using Onload with the sfnt-pingpong application (see Onload Latency with sfnt-pingpong). The ef_vi latency is also shown as a baseline.

Figure 8: **Latency for different TCP payloads at 10 Gb using Onload with sfnt-pingpong**



Further Information

For installation of X3522 adapters and performance tuning of the network driver when not using Onload refer to the *Alveo X3522 Installation Guide* (UG1522) and the *Alveo X3522 User Guide* (UG1523).

Questions regarding Solarflare products, Onload and this *User Guide* can be emailed to support-nic@amd.com.

Chapter 3

Overview

Onload is accelerated network middleware. It is an implementation of TCP and UDP over IP which is dynamically linked into the address space of user-mode applications, and granted direct (but safe) access to the network-adapter hardware. The result is that data can be transmitted to and received from the network directly by the application, without involvement of the operating system. This technique is known as 'kernel bypass'.

Kernel bypass avoids disruptive events such as system calls, context switches and interrupts and so increases the efficiency with which a processor can execute application code. This also directly reduces the host processing overhead, typically by a factor of two, leaving more CPU time available for application processing. This effect is most pronounced for applications which are network intensive, such as:

- Market-data and trading applications
- Computational fluid dynamics (CFD)
- HPC (High Performance Computing)
- HPMPI (High Performance Message Passing Interface), Onload is compatible with MPICH1 and 2, HPMPI, OpenMPI and SCALI
- Other physical models which are moderately parallelizable
- High-bandwidth video-streaming
- Web-caching, Load-balancing and Memcached applications
- Content Delivery Networks (CDN) and HTTP servers
- Other system hot-spots such as distributed lock managers or forced serialization points

The Onload library dynamically links with the application at runtime using the standard BSD sockets API, meaning that no modifications are required to the application being accelerated. Onload is the first and only product to offer full kernel bypass for POSIX socket-based applications over TCP/IP and UDP/IP protocols.

***Note*:** Throughout this user guide the term Onload refers to both OpenOnload and EnterpriseOnload unless otherwise stated.

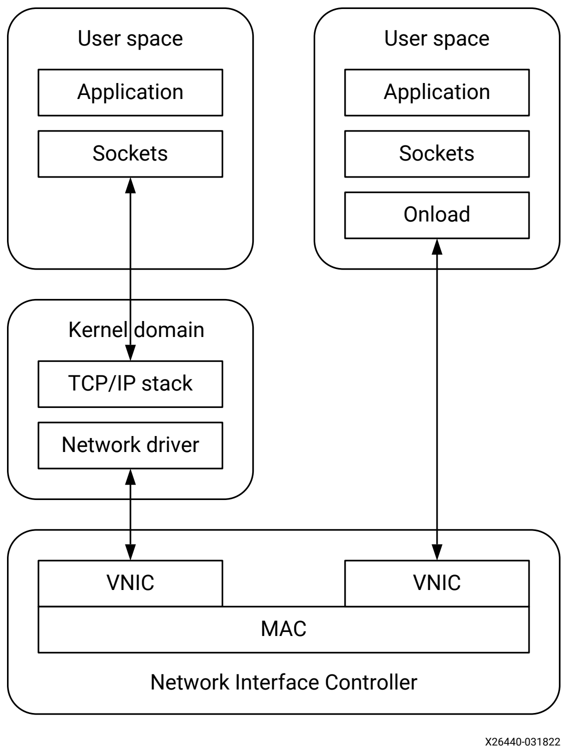
***Note*:** This guide should be read with the *Solarflare Server Adapter User Guide* (SF-103837-CD), which describes procedures for hardware and software installation of Solarflare network interfaces cards, network device drivers and related software.

Contrasting with Conventional Networking

When using conventional networking, an application calls on the OS kernel to send and receive data to and from the network. Transitioning from the application to the kernel is an expensive operation, and can be a significant performance barrier.

When an application accelerated using Onload needs to send or receive data, it need not access the operating system, but can directly access a partition on the network adapter. The two schemes are shown in the following figure.

Figure 9: **Contrast with Conventional Networking.**



An important feature of the conventional model is that applications do not get direct access to the networking hardware and so cannot compromise system integrity. Onload is able to preserve system integrity by partitioning the NIC at the hardware level into many, protected 'Virtual NICs' (VNIC). An application can be granted direct access to a VNIC without the ability to access the rest of the system (including other VNICs or memory that does not belong to the application). Thus Onload with a supported NIC allows optimum performance without compromising security or system integrity.

***Note*:** On X3-series adapters receive queues can be shared by more than one client, and so there must be a trust relationship between processes on the system. X3-series adapters should not be used in scenarios where strict POSIX isolation is necessary.

In summary, Onload can significantly reduce network processing overheads.

How Onload Increases Performance

Onload can significantly reduce the costs associated with networking by reducing CPU overheads and improving performance for latency, bandwidth and application scalability.

Overhead

Transitioning into and out of the kernel from a user-space application is a relatively expensive operation: the equivalent of hundreds or thousands of instructions. With conventional networking such a transition is required every time the application sends and receives data. With Onload, the TCP/IP processing can be done entirely within the user-process, eliminating expensive application/kernel transitions through system calls. In addition, the Onload TCP/IP stack is highly tuned, offering further overhead savings.

The overhead savings of Onload mean more of the CPU's computing power is available to the application to do useful work.

Latency

Conventionally, when a server application is ready to process a transaction it calls into the OS kernel to perform a 'receive' operation, where the kernel puts the calling thread 'to sleep' until a request arrives from the network. When such a request arrives, the network hardware 'interrupts' the kernel, which receives the request and 'wakes' the application.

All of this overhead takes CPU cycles as well as increasing cache and translation lookaside-buffer (TLB) footprint. With Onload, the application can remain at user level waiting for requests to arrive at the network adapter and process them directly. The elimination of a kernel-to-user transition, an interrupt, and a subsequent user-to-kernel transition can significantly reduce latency. In short, reduced overheads mean reduced latency.

Bandwidth

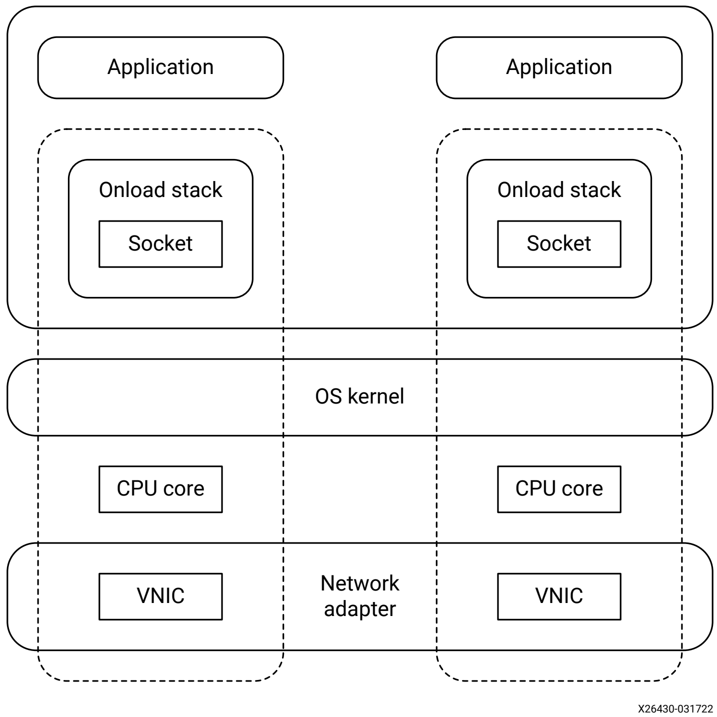
Because Onload imposes less overhead, it can process more bytes of network traffic every second. Along with specially tuned buffering and algorithms designed for 10 gigabit networks, Onload allows applications to achieve significantly improved bandwidth.

Scalability

Modern multi-core systems are capable of running many applications simultaneously. However, the advantages can be quickly lost when the multiple cores contend on a single resource, such as locks in a kernel network stack or device driver. These problems are compounded on modern systems with multiple caches across many CPU cores and Non-Uniform Memory Architectures.

Onload results in the network adapter being partitioned and each partition being accessed by an independent copy of the TCP/IP stack. The result is that with Onload, doubling the cores really can result in doubled throughput as demonstrated by the following figure.

Figure 10: **Onload Partitioned Network Adapter**



Further Information

For detailed information refer to:

- Chapter 6: Onload Functionality.
- Chapter 8: Onload and TCP.
- Chapter 9: Onload and UDP.
- Chapter 12: Onload and Virtualization.

Chapter 4

Installation

This chapter describes how to install Onload.

Onload Distributions

Onload is available in the following distributions:

- “OpenOnload” is a free version of Onload available from the NIC Software and Drivers web page distributed as a source tarball under the GPLv2 license. OpenOnload is subject to a linear development cycle where major releases include the latest development features.
- “EnterpriseOnload” is a commercial enterprise version of Onload distributed as a source RPM under the GPLv2 license. EnterpriseOnload differs from OpenOnload in that it is offered as a mature commercial product that is downstream from OpenOnload having undergone a comprehensive software product test cycle resulting in tested, hardened and validated code.

These distributions are available in the following formats:

Table 4: **Available Formats for Distributions**

Distribution	Tarball	DKMS package	Source RPM	Source DEB
OpenOnload	✓	✓	✓	✓
EnterpriseOnload	—	—	✓	✓

- A *tarball* (called the “Release Package” on the download site) contains source for Onload and its drivers.

  A supplied script builds and installs Onload and its drivers from the source in the tarball. Another supplied script can uninstall Onload.
- A *DKMS package* requires that Dynamic Kernel Module Support (DKMS) framework is available.

  The DKMS framework builds and installs Onload and its drivers from the source in the DKMS package, and automatically rebuilds them if a new OS kernel is installed. The framework can also uninstall Onload.
- A *Source RPM* (or *SRPM*) requires that the RPM Package Manager is available.

The RPM Package Manager builds and installs Onload and its drivers from the source RPM. The Package Manager can also uninstall Onload.

- A *Source DEB* is a Debian package containing source, and requires that the Debian Package Management System is available.

  A tool from the Management System builds and installs Onload and its drivers from the source DEB. The tool can also uninstall Onload.

A flexible and broad range of support options is offered. Users should consult their reseller for details. For further details of the Enterprise Service and Support options that are available, contact support-nic@amd.com.

Cloud Build Profile

Users requiring IPv6 or XDP/eBPF filtering should build Onload with the cloud build profile. This enables additional cloud-specific features that target data centers providing cloud services:

```
# ./onload_install --build-profile cloud
```

Onload LICENSE Files

Users are advised to read the following license files in the Onload distribution:

- `LICENSE`
- `LICENSES-ALL`
- Any `*-LICENSE` files (older Onload distributions only).

Kubernetes Onload

"Kubernetes Onload" is a version of Onload for use with Red Hat OpenShift or Kubernetes/Calico available from the NIC Software and Drivers web page. It provides an Onload Operator that automates the deployment of Onload. It allows for the creation of pods with interfaces that can run accelerated Onload applications.

***Note*:** Installation of Kubernetes Onload is described within its distribution. See its *Release Note* for further details.

Hardware and Software Supported Platforms

Onload supports the following platforms.

Supported Network Adapters

Onload supports the following network adapters:

- AMD X3 series adapters.

  Refer to the *Alveo X3522 User Guide* (UG1523) for adapter details.
- Solarflare XtremeScale™ X2 series adapters.

  Refer to the *Solarflare Server Adapter User Guide* 'Product Specifications' for adapter details.
- Solarflare XtremeScale™ SFN8000 series adapters.

  Refer to the *Solarflare Server Adapter User Guide* 'Product Specifications' for adapter details.

Supported Processors

Onload can run on all Intel and AMD x86 processors, on 64 bit platforms.

Note: Older versions of Onload can also run on 32 bit platforms:

- Support for 32 bit kernels was removed in Onload 201805.
- Support for 32 bit userspace applications continues up to (and including) Onload-7.1.x bugfix releases, but has been removed in Onload 8.

Note: Support for AMD processors prior to Zen was removed in Onload 201811. Onload can still be installed and used on such systems. See Build and Install Onload.

Supported Operating Systems

The following table identifies supported operating systems/kernels:

Table 5: **OS/Kernel Support**

OS Version	Notes
Red Hat Enterprise Linux 7.9	—
Red Hat Enterprise Linux 8.4 - 8.8	—
Red Hat Enterprise Linux 9.0 - 9.2	—
SuSE Linux Enterprise Server 15 SP4 - SP5	—
Canonical Ubuntu Server LTS 20.04 and 22.04	—
Debian 11 "Bullseye"	—
Debian 12 "Bookworm"	—
Linux kernels 4.15 - 6.3	—

Notes:

1. AMD aim to support the OS current and previous major release at the point these are released (plus the latest long term support release if this is not already included). This includes all minor releases where the distributor has not yet declared end of life/support.

Whilst the Onload QA test cycle predominantly focuses on the Linux OS versions documented above, although not formally supported, AMD are not aware of any issues preventing Onload installation on other Linux variants such as CentOS and Fedora. Some versions of Ubuntu and Debian earlier than those listed above are also known to support Onload.

Onload and Network Adapter Drivers

The network adapter drivers that you must use with Onload depends on the adapter that you are using:

- If you are using an AMD X3 series adapter, you must use the xilinx_efct and auxiliary bus drivers. You must build and install these before you build Onload.

  See X3 Series Drivers.

- If you are using a Solarflare XtremeScale adapter, you must use the sfc driver that is supplied with Onload.

  See XtremeScale Drivers.

X3 Series Drivers

Drivers for X3 series adapters are available separately from Onload. The following are required:

- The auxiliary bus driver.

  This is provided by some recent Linux distributions, and is also available from AMD.

- The `xilinx_efct` network adapter driver.

  This is available from AMD.

> **IMPORTANT!** *You must install and build these drivers before you install Onload. Follow the instructions in the Installation Guide for your X3 series adapter.*

If you are not also using Solarflare XtremeScale adapters, omit the rest of this section and go to Removing an Existing Installation.

XtremeScale Drivers

Solarflare XtremeScale network adapters (X2-series and their predecessors) use the `sfc` network driver. This is generally available from three sources:

- Packaged in many Linux distributions such as Red Hat Enterprise Linux.

This is commonly known as 'in-tree', 'in-box', or 'boxed'. Depending on the OS version this driver might not support more recent Solarflare XtremeScale adapters, and also does not support any Onload features. *You must not use this driver with Onload.*

- Downloaded as source RPM from the NIC Software and Drivers web page.

  This method for getting the `sfc` driver is recommended when you require the latest driver for non-Onload use.

- Packaged in the OpenOnload/EnterpriseOnload distribution.

  This bundled `sfc` driver is installed along with the other Onload drivers. It supports the specific features of the Onload release. *You must always use this bundled driver with Onload.*

To ensure the bundled Onload driver is always loaded following system reboot, any other `sfc` drivers can be removed from the OS entirely. See Identifying and Removing Previously Installed Drivers.

Alternatively any Onload startup script should include the following command to reload the Onload drivers, including the bundled `sfc` driver:

```
# onload_tool reload
```

Identifying and Removing Previously Installed Drivers

You can identify any previously installed `sfc` drivers using the `ethtool -i` command:

- An 'in-tree' driver uses one of the following formats for its version number:
  - On older versions of Linux, the 'in-tree' driver outputs only Major and Minor revision numbers:

    ```
    # ethtool -i enp3s0f0
    driver: sfc
    version: 4.0
    ```

  - On recent versions of Linux, the 'in-tree' driver instead outputs the kernel version number:

    ```
    # ethtool -i enp3s0f0
    driver: sfc
    version: 4.18.0-372.9.1.el8.x86_64
    ```

- All other `sfc` drivers display detailed version information:

  ```
  # ethtool -i enp3s0f0
  driver: sfc
  version: 4.15.12.1008
  ```

***Note*:** For a list of the `sfc` driver included in each Onload release see Adapter Net Drivers.

To remove the previously installed driver (with Onload uninstalled or not yet installed):

```
# find /lib/modules/$(uname -r) -name 'sfc*.ko' | xargs rm -rf
# rmmod sfc
# update-initramfs -u -k <kernel version>
```

initramfs commands might differ on different Linux based OS, for example on CentOS 7 the following `dracut` command can be used:

```
# dracut -f
```

Removing an Existing Installation

When migrating between Onload versions or between Onload distributions (OpenOnload or EnterpriseOnload), a previously installed version or distribution must first be unloaded using the onload_tool unload command and then removed. Tarball installs can be removed with the `onload_uninstall` command.

```
# onload_tool unload
# onload_uninstall
```

In some specific cases it might be necessary to manually remove onload driver modules before upgrading to a more recent version. To do this, list the modules and remove each dependency before removing the modules:

```
# lsmod | grep onload
onload                 580599  3
sfc_char                47419  1 onload
                       162351  2 onload,sfc_char
sfc                    431807  4 sfc_resource,onload,sfc_char,sfc_affinity
onload_cplane          144142  3 onload
```

and:

```
# lsmod | grep sfc
sfc_chsfc_resourcear    47419  1 onload
sfc_resource           162351  2 onload,sfc_char
sfc_affinity            17948  1 sfc_resource
sfc                    431807  4 sfc_resource,onload,sfc_char,sfc_affinity
```

To remove modules:

```
# rmmod onload
# rmmod sfc_char
```

Repeat the rmmod command for each module.

CAUTION! *Attempts to unload or uninstall Onload and drivers when onload stacks are still present will result in the following type of warnings:*

```
# onload_tool unload
onload_tool: /sbin/modprobe -r onload
FATAL: Module onload is in use.
FATAL: Error running remove command for onload
onload_tool: ERROR: modprobe -r onload failed (0)
onload_tool: /sbin/modprobe -r sfc_char
FATAL: Module sfc_char is in use.
FATAL: Error running remove command for sfc_char
onload_tool: ERROR: modprobe -r sfc_char failed (0)
onload_tool: /sbin/modprobe -r sfc_resource
FATAL: Module sfc_resource is in use.
onload_tool: ERROR: modprobe -r sfc_resource failed (0)
onload_tool: /sbin/modprobe -r sfc_affinity
FATAL: Module sfc_affinity is in use.
FATAL: Error running remove command for sfc_affinity
onload_tool: ERROR: modprobe -r sfc_affinity failed (0)
onload_tool: /sbin/modprobe -r sfc
FATAL: Module sfc is in use.
onload_tool: ERROR: modprobe -r sfc failed (0)"
```

The user should check using onload_stackdump [-z] to ensure that all onload stacks have been terminated before the uninstall.

Removing RPMs

If Onload was installed from a source RPM, it might also be necessary to remove installed RPM packages:

```
rpm -qa | grep 'enterpriseonload' | xargs rpm -e
rpm -qa | grep 'cloudonload' | xargs rpm -e
rpm -qa | grep 'onload' | xargs rpm -e
rpm -qa | grep 'sfc' | xargs rpm -e
rpm -qa | grep 'sfutils' | xargs rpm -e
onload_uninstall
```

Pre-install Notes

Before installing, note the following:

- If Onload is to accelerate a 32-bit application on a 64-bit architecture, the 32-bit libc development headers should be installed before building Onload. Refer to Appendix C: Build Dependencies for install instructions.
- The Solarflare drivers are currently classified as unsupported in SLES 11 and 12, the certification process is underway. To overcome this add ‘`allow_unsupported_modules 1`’ to the following file:
    - For SLES 11, `/etc/modprobe.d/unsupported-modules`

  - For SLES 12, `/etc/modprobe.d/10-unsupported-modules.conf`

- Determine which Onload distribution and format you will be installing (see Onload Distributions). Then refer to the appropriate section from the following:
  - Building and Installing from a Tarball
  - Building and Installing from a DKMS Package
  - Building and Installing from a Source RPM
  - Building and Installing from a Source DEB.

Building and Installing from a Tarball

This section identifies the procedures to build and install Onload from a tarball. It uses OpenOnload as an example, but the same procedures apply to any other Onload distributions in this format.

Download and Untar Onload

1. Download the required tar file from the NIC Software and Drivers web page.

   The compressed tar file (.tgz) should be downloaded/copied to a directory on the machine on which it will be installed.

2. As root, unpack the tar file using the tar command.

   ```
   tar -zxvf onload-<version>.tgz
   ```

   This will unpack the tar file and, within the current directory, create a sub-directory called `onload-<version>` which contains other sub-directories including the `scripts` directory from which subsequent install commands can be run.

Build and Install Onload

***Note*:** Refer to Appendix C: Build Dependencies for details of build dependencies.

The following command will build and install Onload and required drivers in the system directories:

```
./onload_install [options]
```

Make sure that all required options are specified with the install command. For help use `./onload_install -h`

```
options:
  --newkernel <ver>          - Build and install drivers for new kernel
  --kernelver <ver>          - Specify kernel version for drivers
  --nobuild                  - Do not (re)compile
  --require32                - Fail if 32-bit binaries are not installed
  --setuid                   - Preload libraries are set-uid and set-gid
  --nosetuid                 - Preload libraries are not set-uid/gid
  --affinity                 - Include sfcaffinity
  --noaffinity               - Do not include sfcaffinity
  --debug                    - Build debug binaries
  --no-debug-info            - Omit debug info from binaries
  --strict                   - Compiler warnings are errors
  --require-optional-targets - Require optional targets
  --userfiles                - Only install user-level components
  --kernelfiles              - Only install kernel driver components
  --modprobe                 - Only install modprobe configuration
  --force                    - Force install if already installed
  --allow-unsupported-cpu    - Force install even when CPU is too old
  --verbose                  - Verbose logging of commands
  --test                     - Do not install; just print commands
  --listfiles                - Do not install; just list installed files
  --filter-engine            - Specify location of filter engine directory
                               by default - /opt/onload_filter_engine
  --no-filter-engine         - Don't include filter engine support even
                               if it's present in the system
  --build-profile            - Specify a build profile
  --no-initramfs             - Do not update initramfs
  --no-tcpdirect             - Do not install TCPDirect libraries
```

Successful installation will be indicated with the following output "`onload_install: Install complete`" – possibly followed by a warning that the sfc (net driver) driver is already installed.

Note: The `onload_install` script does not create RPMs.

- Some optional targets require additional packages (Optional targets are listed in Appendix C: Build Dependencies). By default, an Onload install continues if these targets cannot be built. If the --require-optional-targets option is specified, the install fails when any prerequisite for an optional target is missing:

  ```
  ./onload_install --require-optional-targets
  ```

- Installing on an unsupported CPU gives an error. This can also be overridden:

  ```
  ./onload_install --allow-unsupported-cpu
  ```

Cloud Build Profile

Users requiring IPv6 or XDP/eBPF filtering should build Onload with the cloud build profile. This enables additional cloud-specific features that target data centers providing cloud services:

```
# ./onload_install --build-profile cloud
```

Load the Onload Drivers

Following installation you must load the Onload drivers:

1. Load the network and Onload kernel drivers:

   - If you are using the `sfc` network driver (for 8000 series or X2 series adapters), just use the `onload_tool` command:

     ```
     # onload_tool reload
     ```

     This replaces any previously loaded `sfc` network driver with the `sfc` driver from the Onload distribution.

   - Otherwise you must reload the network driver yourself, and also the auxiliary driver. For example if you are using the `xilinx_efct` driver with an X3 series adapter:

     ```
     # modprobe -r xilinx_efct
     # modprobe -r auxiliary
     # modprobe auxiliary
     # modprobe xilinx_efct
     # onload_tool reload --onload-only
     ```

2. Confirm success:

   ```
   # onload
   OpenOnload <version>
   Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications,
   2002-2005 Level 5 Networks
   Built: <date> <time> (release)
   Kernel module: <version>
   ```

   If a `Kernel module` version is shown, this confirms that the Onload kernel module is installed and loaded.

Note: An alternative to the `onload_tool reload` command is to reboot the system to load Onload drivers.

Confirm Onload Installation

When the Onload installation is complete run the `onload` command to confirm installation of Onload software and kernel module:

```
# onload
```

This will display the Onload product banner and usage:

```
# onload
Onload b704114d6b 2022-02-08 master
Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications, 2002-2005
Level 5 Networks
Built: Feb  8 2022 17:19:25 (debug)
Build profile header: <ci/internal/transport_config_opt_extra.h>
Kernel module: b704114d6b 2022-02-08 master
```

```
usage:
  onload [options] <command> <command-args>
options:
  -p,--profile=<profile> -- comma sep list of config profile(s)
  --force-profiles       -- profile settings override environment
  --no-app-handler       -- do not use app-specific settings
  --app=<app-name>       -- identify application to run under onload
  --version              -- print version information
  -v                     -- verbose
  -h --help              -- this help message
```

Building a Source RPM from a Tarball

Alternatively, a source RPM can be built from the Onload tarball.

1. Download the required tarball from the NIC Software and Drivers web page.
2. As root, execute the following command:

```
rpmbuild -ts onload-<version>.tgz*
 x86_64 Wrote: /root/rpmbuild/SRPMS/onload-<version>.src.rpm
```

The output identifies the location of the source RPM. For instructions on installing this, see Building and Installing from a Source RPM.

Note: Use the `-ta` option to generate a binary RPM.

Building and Installing from a DKMS Package

This section identifies the procedures to build and install Onload from a DKMS package. It uses OpenOnload as an example, but the same procedures apply to any other Onload distributions in this format.

DKMS packages are available for OpenOnload from version 201811 onwards.

DKMS must be installed on the server. DKMS can be downloaded from http://linux.dell.com/dkms/ or from the OS distribution. To check this run the following command which will return nothing if DKMS is not installed:

```
# dkms --version
dkms: 2.2.0.3
```

Install on RHEL

To install on RHEL:

1. Install the Onload dkms package:

```
# rpm -i onload-dkms-<version>.noarch.rpm
```

2. Load the network and Onload kernel drivers:
   - If you are using the `sfc` network driver (for 8000 series or X2 series adapters), just use the `onload_tool` command:

     ```
     # onload_tool reload
     ```

     This replaces any previously loaded `sfc` network driver with the `sfc` driver from the Onload distribution.
   - Otherwise you must reload the network driver yourself, and also the auxiliary driver. For example if you are using the `xilinx_efct` driver with an X3 series adapter:

     ```
     # modprobe -r xilinx_efct
     # modprobe -r auxiliary
     # modprobe auxiliary
     # modprobe xilinx_efct
     # onload_tool reload --onload-only
     ```

3. Confirm success:

   ```
   # onload
   OpenOnload <version>
   Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications,
   2002-2005 Level 5 Networks
   Built: <date> <time> (release)
   Kernel module: <version>
   ```

   If a `Kernel module` version is shown, this confirms that the Onload kernel module is installed and loaded.

Install on Ubuntu

A method that can be used on Ubuntu is as follows:

1. Create a `.deb` package from the RPM:

   ```
   $ sudo alien -c onload-dkms-<version>.noarch.rpm
   ```

   You must use the `-c` option otherwise the driver binary will not be built.
2. Make sure the `.deb` package is created:

   ```
   $ ls
   onload-dkms_<version>_all.deb    onload-dkms-<version>.noarch.rpm
   ```

3. Install the `.deb` package (this takes a few minutes):

   ```
   $ sudo dpkg -i onload-dkms_<version>_all.deb
   ```

   This produces a lot of output while it builds the following components:

   `sfc`, `sfc_affinity.ko`, `sfc_char.ko`, `sfc_resource.ko`, and `onload.ko`.
4. Load the network and Onload kernel drivers:

- If you are using the `sfc` network driver (for 8000 series or X2 series adapters), just use the `onload_tool` command:

  ```
  $ sudo onload_tool reload
  ```

  This replaces any previously loaded `sfc` network driver with the `sfc` driver from the Onload distribution.

- Otherwise you must reload the network driver yourself, and also the auxiliary driver. For example if you are using the `xilinx_efct` driver with an X3 series adapter:

  ```
  $ sudo modprobe -r xilinx_efct
  $ sudo modprobe -r auxiliary
  $ sudo modprobe auxiliary
  $ sudo modprobe xilinx_efct
  $ sudo onload_tool reload --onload-only
  ```

5. Confirm success:

   ```
   $ onload
   OpenOnload <version>
   Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications,
   2002-2005 Level 5 Networks
   Built: <date> <time> (release)
   Kernel module: <version>
   ```

   If a `Kernel module` version is shown, this confirms that the Onload kernel module is installed and loaded.

Note: Instead of creating and installing a .deb package (1 - 3 above), you can use the alien command to install directly from the RPM:

```
$ sudo alien --scripts -i onload-dkms-<version>.noarch.rpm
```

This produces a lot of output similar to the previous method.

Building and Installing from a Source RPM

This section identifies the procedures to build and install Onload from a source RPM. It uses EnterpriseOnload as an example, but the same procedures apply to other Onload distributions in this format, such as OpenOnload.

Source RPMs can be built by the 'root' or 'non-root' user, but the user must have superuser privileges to install RPMs. Customers should contact their customer sales representative for access to Onload source RPM resources.

Build the RPMs

Note: Refer to Appendix C: Build Dependencies for details of build dependencies.

As root:

```
rpmbuild --rebuild enterpriseonload-<version>.src.rpm
```

Or as a non-root user:

It is advised to use _topdir to ensure that RPMs are built into a directory to which the user has permissions. The directory structure must pre-exist for the rpmbuild command to succeed.

```
mkdir -p /tmp//myrpm/{SOURCES,BUILD,RPMS,SRPMS}
rpmbuild --define "_topdir /tmp/myrpm" \
         --rebuild enterpriseonload-<version>.src.rpm
```

Note: On some non-standard kernels the rpmbuild might fail because of build dependencies. In this event retry, adding the `--nodeps` option to the command line.

Building the source RPM will produce two binary RPM files which can be found in one of the following directories:

- `/usr/src/*/RPMS/`
- `_topdir/RPMS` (when built by a non-root user)
- `/tmp/myrpm/RPMS/x86_64/` (when _topdir was defined in the rpmbuild command line).

For example, the user-space components:

```
/usr/src/redhat/RPMS/x86_64/enterpriseonload-<version>.x86_64.rpm
```

and the kernel components:

```
/usr/src/redhat/RPMS/x86_64/enterpriseonload-kmod-2.6.18-92.el5-
<version>.x86_64.rpm
```

Install the Built RPMs

The Onload RPM and the kernel RPM must be installed for Onload to function correctly.

```
rpm -ivf enterpriseonload-<version>.x86_64.rpm
rpm -ivf enterpriseonload-kmod-2.6.18-92.el5-<version>.x86_64.rpm
```

Note: Onload is now installed but the kernel modules are not yet loaded.

Note: The `enterpriseonload-kmod` filename is specific to the kernel that it is built for.

Load the Onload Drivers

Following installation you must load the Onload drivers:

1. Load the network and Onload kernel drivers:

- If you are using the `sfc` network driver (for 8000 series or X2 series adapters), just use the `onload_tool` command:

  ```
  # onload_tool reload
  ```

  This replaces any previously loaded `sfc` network driver with the `sfc` driver from the Onload distribution.

- Otherwise you must reload the network driver yourself, and also the auxiliary driver. For example if you are using the `xilinx_efct` driver with an X3 series adapter:

  ```
  # modprobe -r xilinx_efct
  # modprobe -r auxiliary
  # modprobe auxiliary
  # modprobe xilinx_efct
  # onload_tool reload --onload-only
  ```

2. Confirm success:

   ```
   # onload
   OpenOnload <version>
   Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications,
   2002-2005 Level 5 Networks
   Built: <date> <time> (release)
   Kernel module: <version>
   ```

   If a `Kernel module` version is shown, this confirms that the Onload kernel module is installed and loaded.

Note: At this point Onload is loaded, but until the network interface has been configured and brought into service Onload will be unable to accelerate traffic.

Building and Installing from a Source DEB

This section identifies the procedures to build and install Onload from a source DEB. It uses EnterpriseOnload as an example, but the same procedures apply to other Onload distributions in this format, such as OpenOnload.

Debian source packages are available for EnterpriseOnload from version 4.0 onwards. Packages are named in the following format:

```
enterpriseonload_<version>-debiansource.tgz
```

1. Untar the source package:

   ```
   $ tar xf enterpriseonload_<version>-debiansource.tgz
   ```

2. Extract the source:

   ```
   $ dpkg-source -x enterpriseonload_<version>-1.dsc
   ```

3. Build the packages:

```
$ cd enterpriseonload-<version>
$ debuild -i -uc -us
```

4. Install the packages:

```
$ sudo dpkg -i ../enterpriseonload-user_<version>-1_amd64.deb
$ sudo dpkg -i ../enterpriseonload-source_<version>-1_all.deb
```

5. Build and install the modules:

```
$ sudo m-a a-i enterpriseonload
```

Onload Kernel Modules

To identify relevant drivers already installed on the server:

```
find /lib/modules/`uname -r` -type f -name '*.ko' -printf '%f\n' | grep -E
'sfc|onload'
```

Table 6: **Driver Names and Descriptions**

Driver Name	Description
sfc.ko	A Linux net driver provides the interface between the Linux network stack and the Solarflare network adapter.
sfc_char.ko	Provides low level access to the Solarflare network adapter virtualized resources. Supports direct access to the network adapter for applications that use the ef_vi user-level interface for maximum performance.
sfc_tune.ko	This is used to prevent the kernel during idle periods from putting the CPUs into a sleep state. Removed in openonload-201405.
sfc_affinity.ko	Used to direct traffic flow managed by a thread to the core the thread is running on, inserts packet filters that override the RSS behavior.
sfc_resource.ko	Manages the virtualization resources of the adapter and shares the resources between other drivers.
onload.ko	The kernel component of Onload.
onload_cplane.ko	The control plane component of Onload. User of Onload-201710 onwards refer to User-space Control Plane Server.

To unload any loaded drivers:

```
onload_tool unload
```

To remove the installed files of a previous Onload:

```
onload_uninstall
```

To load the Solarflare net driver (if not already loaded):

```
modprobe sfc
```

To reload drivers following upgrade or changed settings:

```
onload_tool reload
```

Note: The `onload_tool reload` command reloads drivers listed in the preceding table, but does not reload other network drivers such as the `xilinx_efct` driver used by X3 series adapters. To reload any unlisted drivers, use the `modprobe` command.

Configuring the Network Interfaces

Network interfaces should be configured according to the *Solarflare Server Adapter User's Guide*.

When the interface(s) have been configured, the `dmesg` command will display output similar to the following (one entry for each Solarflare interface):

```
sfc 0000:13:00.0: INFO:  eth2 Solarflare Communications NIC PCI(1924:803)
sfc 0000:13:00.1: INFO:  eth3 Solarflare Communications NIC PCI(1924:803)
```

Note: IP address configuration should be carried out using normal OS tools, such as system-config-network (Red Hat) or yast (SUSE).

Note: The `performance_profile` driver module option must **not** be set to `throughput` when using Onload.

Installing Netperf and sfnettest

Refer to the Appendix L: X2 Low Latency Quickstart for instructions to install the Netperf and sfnettest applications.

Running Onload

Once Onload has been installed there are different ways to accelerate applications.

- Prefixing the application command line with the Onload command will accelerate the application.

  ```
  # onload <app_name> [app_options]
  ```

- Exporting LD_PRELOAD to the environment will mean that all applications started in the same environment will be accelerated.

```
# export LD_PRELOAD=libonload.so
```

Testing the Onload Installation

The Appendix L: X2 Low Latency Quickstart demonstrates testing of Onload with the Netperf and sfnettest benchmark tools.

Applying an Onload Patch

Occasionally, AMD might issue a software 'patch' which is applied to Onload to resolve a specific bug or investigate a specific issue. The method of applying a patch is dependent on how Onload was installed.

Patching a Tarball Installation

This section describes how a patch should be applied when Onload was installed from a tarball. It uses OpenOnload as an example, but the same procedures apply to any other Onload distributions in this format.

1. Copy the patch to a directory on the server where Onload is already installed.
2. Go to the Onload directory:

   ```
   # cd onload-<version>
   ```

3. Apply the patch. For example:

   ```
   # patch -p1 < ~/<path>/<name of patch file>.patch
   ```

4. Uninstall the old Onload drivers:

   ```
   # onload_uninstall
   ```

5. Build and re-install the Onload drivers:

   ```
   # ./scripts/onload_install
   ```

6. Load the network and Onload kernel drivers:
   - If you are using the `sfc` network driver (for 8000 series or X2 series adapters), just use the `onload_tool` command:

     ```
     # onload_tool reload
     ```

This replaces any previously loaded `sfc` network driver with the `sfc` driver from the Onload distribution.

- Otherwise you must reload the network driver yourself, and also the auxiliary driver. For example if you are using the `xilinx_efct` driver with an X3 series adapter:

```
# modprobe -r xilinx_efct
# modprobe -r auxiliary
# modprobe auxiliary
# modprobe xilinx_efct
# onload_tool reload --onload-only
```

7. Confirm success:

```
# onload
OpenOnload <version>
Copyright 2019-2022 Xilinx, 2006-2019 Solarflare Communications,
2002-2005 Level 5 Networks
Built: <date> <time> (release)
Kernel module: <version>
```

If a `Kernel module` version is shown, this confirms that the Onload kernel module is installed and loaded.

Patching a Source RPM Installation

The following procedure describes how a patch should be applied when Onload was installed from a source RPM. It uses EnterpriseOnload as an example, but the same procedures apply to other Onload distributions in the RPM format, such as OpenOnload.

1. Copy the patch to the directory on the server where the Onload RPM package exists and carry out the following commands:

```
rpm2cpio enterpriseonload-<version>.src.rpm | cpio -id
tar -xzf enterpriseonload-<version>.tgz
cd enterpriseonload-<version>
patch -p1 < $PATCHNAME
```

2. This can now be installed directly from this directory:

```
./scripts/onload_install
```

3. Or it can be repackaged as a new source RPM:

```
cd ..
tar -czf enterpriseonload-<version>.tgz enterpriseonload-<version>
rpmbuild -ts enterpriseonload-<version>.tgz
```

4. The rpmbuild procedure will display a ‘Wrote’ line identifying the location of the source RPM. For example:

```
Wrote: /root/rpmbuild/SRPMS/enterpriseonload-<version>.el6.src.rpm
```

Kernel and OS Upgrades

An Onload installation is specific to a particular version of kernel and OS. If the kernel or OS is changed to a different version, Onload must be rebuilt for the new version, and then reinstalled:

1. Uninstall Onload. Refer to Removing an Existing Installation.
2. Install the new version of kernel or OS.
3. Rebuild Onload.
4. Install the newly built version of Onload.

Chapter 5

Tuning Onload

This chapter documents the available tuning options for Onload, and the expected results.

Most of the Onload configuration parameters, including tuning parameters, are set by environment variables exported into the accelerated applications environment. Environment variables can be identified throughout this manual as they begin with `EF_`. All environment variables are described in Appendix A: Parameter Reference or Appendix B: Meta Options in this guide. Examples throughout this guide assume the use of the *bash* or *sh* shells; other shells might use different methods to export variables into the applications environment.

- System Tuning describes tools and commands which can be used to tune the server and OS.
- Spinning, Polling and Interrupts describes how to perform standard heuristic tuning, which can help improve the application's performance. There are also benchmark examples running specific tests to demonstrate the improvements Onload can have on an application.
- Onload Deployment on NUMA Systems describes the selection of a NUMA node, the allocation of cache memory and the affinitization of drivers, processes and interrupts.
- Interrupt Handling for the sfc Driver describes how to tune interrupt handling on X2-series adapters (and earlier) that use the `sfc` driver.
- Performance Jitter helps you to reduce or eliminate jitter.
- Using Onload Tuning Profiles introduces you to tuning profiles that apply multiple settings.
- Benchmark Testing references some benchmarking procedures.
- Application-Specific Tuning introduces advanced tuning options driven from analysis of the Onload stack using `onload_stackdump`.
- Worked Examples demonstrate how to achieve the application tuning goals.

Note: Onload tuning and kernel driver tuning are subject to different requirements. This section describes the steps to tune Onload. For details on how to tune the Solarflare kernel driver, refer to the 'Performance Tuning on Linux' section of the *Solarflare Server Adapter User Guide*, available from the NIC Software and Drivers web page.

System Tuning

This section details steps to tune the server and operating system for lowest latency.

See also the system tuning described in Chapter 2: X3 Low Latency Quickstart

Sysjitter

The sysjitter utility measures the extent to which the system introduces jitter and so impacts on the user-level process. Sysjitter runs a thread on each processor core and when the thread is de-scheduled from the core it measures for how long. Sysjitter produces summary statistics for each processor core. The sysjitter utility can be downloaded from https://github.com/Xilinx-CNS/cns-sysjitter.

Sysjitter should be run on a system that is idle. When running on a system with cpusets enabled, run sysjitter as root.

Refer to the sysjitter README file for further information on building and running sysjitter.

The following is an example of the output from sysjitter on a single CPU socket server with four CPU cores.

```
./sysjitter --runtime 10 200 | column -t
core_i:           0           1           2           3
threshold(ns):    200         200         200         200
cpu_mhz:          3215        3215        3215        3215
runtime(ns):      9987653973  9987652245  9987652070  9987652027
runtime(s):       9.988       9.988       9.988       9.988
int_n:            10001       10130       10012       10001
int_n_per_sec:    1001.336    1014.252    1002.438    1001.336
int_min(ns):      1333        1247        1299        1446
int_median(ns):   1390        1330        1329        1470
int_mean(ns):     1424        1452        1452        1502
int_90(ns):       1437        1372        1357        1519
int_99(ns):       1619        5046        2392        1688
int_999(ns):      5065        22977       15604       3694
int_9999(ns):     31260       39017       184305      36419
int_99999(ns):    40613       45065       347097      49998
int_max(ns):      40613       45065       347097      49998
int_total(ns):    14244846    14719972    14541991    15031294
int_total(%):     0.143       0.147       0.146       0.150
```

The table below describes the output fields of the sysjitter utility.

Table 7: **Output Fields of the Sysjitter Utility**

Field	Description
`threshold (ns)`	Ignore any interrupts shorter than this period
`cpu_mhz`	CPU speed
`runtime (ns)`	Runtime of sysjitter - nanoseconds
`runtime (s)`	Runtime of sysjitter - seconds
`int_n`	Number of interruptions to the user thread
`int_n_per_sec`	Number of interruptions to the user thread per second
`int_min (ns)`	Minimum time taken away from the user thread due to an interruption

Table 7: **Output Fields of the Sysjitter Utility** *(cont'd)*

Field	Description
`int_median (ns)`	Median time taken away from the user thread due to an interruption
`int_mean (ns)`	Mean time taken away from the user thread due to an interruption
`int_90 (ns)`	90%percentile value
`int_99 (ns)`	99% percentile value
`int_999 (ns)`	99.9% percentile value
`int_9999 (ns)`	99.99% percentile value
`int_99999 (ns)`	99.999% percentile value
`int_max (ns)`	Max time taken away from the user thread
`int_total (ns)`	Total time spent not processing the user thread
`int_total (%)`	`int_total (ns)` as a percentage of total runtime

Timer (TSC) Stability

Onload uses the Time Stamp Counter (TSC) CPU registers to measure changes in time with very low overhead. Modern CPUs support an “invariant TSC”, which is synchronized across different CPUs and ticks at a constant rate regardless of the current CPU frequency and power saving mode. Onload relies on this to generate accurate time calculations when running across multiple CPUs. If run on a system which does not have an invariant TSC, Onload might calculate wildly inaccurate time values and this can, in extreme cases, lead to some connections becoming stuck.

Users should consult their server vendor documentation and OS documentation to ensure that servers can meet the invariant TSC requirement.

CPU Power Saving Mode

Modern processors have design features that enable a CPU core to drop into lowering power states when instructed by the operating system that the CPU core is idle. When the OS schedules work on the idle CPU core (or when other CPU cores or devices need to access data currently in the idle CPU core’s data cache) the CPU core is signaled to return to the fully-on power state. These changes in CPU core power states create additional network latency and jitter.

It is therefore recommended that customers wishing to achieve the lowest latency and lowest jitter disable the “C1E power state” or “CPU power saving mode” within the machine's BIOS.

Disabling the CPU power saving modes is required if the application is to realize low latency with low jitter.

Note: To ensure C states are not enabled, overriding the BIOS settings, it is recommended to put the line '`intel_idle.max_cstate=0 idle=poll`' (for Intel) or '`processor.max_cstate=0 idle=poll`' (for AMD) into the kernel command line `/boot/grub/grub.conf`. The settings will produce consistent results and are particularly useful when benchmarking. Allowing some cores to enable Turbo modes while others are idle can produce better latency in some servers. For this, use idle=mwait and enable C-states in the BIOS. Alternatively, on later Linux versions, the tuned service can be enabled and used with the network-latency profile. Users should refer to vendor documentation and experiment with C states for different applications.

Consult your system vendor and documentation for details concerning the disabling of C1E, C states or CPU power saving states.

Spinning, Polling and Interrupts

This section helps you choose when to use spinning for Onload, and whether to use a polling model or an interrupt-driven model.

Spinning (busy-wait)

Conventionally, when an application attempts to read from a socket and no data is available, the application will enter the OS kernel and block. When data becomes available, the network adapter will interrupt the CPU, allowing the kernel to reschedule the application to continue.

Blocking and interrupts are relatively expensive operations, and can adversely affect bandwidth, latency and CPU efficiency.

Onload can be configured to spin on the processor in user mode for up to a specified number of microseconds waiting for data from the network. If the spin period expires the processor will revert to conventional blocking behavior. Non-blocking sockets will always return immediately as these are unaffected by spinning.

Onload uses the EF_POLL_USEC environment variable to configure the length of the spin timeout.

```
export EF_POLL_USEC=100000
```

will set the busy-wait period to 100 milliseconds. See Processing at User-Level for a worked example, and Appendix B: Meta Options for more details.

Enabling Spinning

To enable spinning in Onload:

Set EF_POLL_USEC. This causes Onload to spin on the processor for up to the specified number of microseconds before blocking. This setting is used in TCP and UDP and also in `recv()`, `select()`, `pselect()` and `poll()`, `ppoll()` and `epoll_wait()`, `epoll_pwait()` and `onload_ordered_epoll_wait()`. Use the following command:

```
export EF_POLL_USEC=100000
```

***Note*:** If neither of the spinning options EF_POLL_USEC and EF_SPIN_USEC are set, Onload will resort to default interrupt driven behavior because the EF_INT_DRIVEN environment variable is enabled by default.

Setting the EF_POLL_USEC variable also sets the following environment variables.

```
EF_SPIN_USEC=EF_POLL_USEC
EF_SELECT_SPIN=1
EF_EPOLL_SPIN=1
EF_POLL_SPIN=1
EF_PKT_WAIT_SPIN=1
EF_TCP_SEND_SPIN=1
EF_UDP_RECV_SPIN=1
EF_UDP_SEND_SPIN=1
EF_TCP_RECV_SPIN=1
EF_BUZZ_USEC=MIN(EF_POLL_USEC, 100)
EF_SOCK_LOCK_BUZZ=1
EF_STACK_LOCK_BUZZ=1
```

Turn off adaptive moderation and set interrupt moderation to a high value (microseconds) to avoid flooding the system with interrupts. Use the following command:

```
/sbin/ethtool -C eth2 rx-usecs 60 adaptive-rx off
```

See Appendix B: Meta Options for more details.

When to Use Spinning

The optimal setting is dependent on the nature of the application. If an application is likely to find data soon after blocking, or the system does not have any other major tasks to perform, spinning can improve latency and bandwidth significantly.

In general, an application will benefit from spinning if the number of active threads is less than the number of available CPU cores. However, if the application has more active threads than available CPU cores, spinning can adversely affect application performance because a thread that is spinning (and therefore idle) takes CPU time away from another thread that could be doing work. If in doubt, it is advisable to try an application with a range of settings to discover the optimal value.

Polling vs. Interrupts

Interrupts are useful because they allow the CPU to do other useful work while simultaneously waiting for asynchronous events (such as the reception of packets from the network). The historical alternative to interrupts was for the CPU to periodically poll for asynchronous events and on single processor systems this could result in greater latency than would be observed with interrupts. Historically it was accepted that interrupts were “good for latency”.

On modern, multicore systems the tradeoffs are different. It is often possible to dedicate an entire CPU core to the processing of a single source of asynchronous events (such as network traffic). The CPU dedicated to processing network traffic can be spinning (also known as busy waiting), continuously polling for the arrival of packets. When a packet arrives, the CPU can begin processing it almost immediately.

Contrast the polling model to an interrupt-driven model. Here the CPU is likely in its “idle loop” when an interrupt occurs. The idle loop is interrupted, the interrupt handler executes, typically marking a worker task as runnable. The OS scheduler will then run and switches to the kernel thread that will process the incoming packet. There is typically a subsequent task switch to a user-mode thread where the real work of processing the event (for example acting on the packet payload) is performed. Depending on the system, it can take on the order of a microsecond to respond to an interrupt and switch to the appropriate thread context before beginning the real work of processing the event. A dedicated CPU spinning in a polling loop can begin processing the asynchronous event in a matter of nanoseconds.

It follows that spinning only becomes an option if a CPU core can be dedicated to the asynchronous event. If there are more threads awaiting events than CPU cores (that is if all CPU cores are oversubscribed to application worker threads), then spinning is not a viable option, (at least, not for all events). One thread will be spinning, polling for the event while another could be doing useful work. Spinning in such a scenario can lead to (dramatically) increased latencies. But if a CPU core can be dedicated to each thread that blocks waiting for network I/O, then spinning is the best method to achieve the lowest possible latency.

Onload Deployment on NUMA Systems

When deployed on NUMA systems, application load throughput and latency performance can be adversely affected unless due consideration is given to the selection of the NUMA node, the allocation of cache memory and the affinitization of drivers, processes and interrupts.

For best performance the accelerated application should always run on the NUMA node nearest to the network adapter. The correct allocation of memory is particularly important to ensure that packet buffers are allocated on the correct NUMA node to avoid unnecessary increases in QPI traffic and to avoid dropped packets.

Useful Commands

- To identify NUMA nodes, socket memory and CPU core allocation:

  ```
  # numactl -H
  ```

- To identify the NUMA node local to an adapter:

  ```
  # cat /sys/class/net/<interface>/device/numa_node
  ```

- To identify memory allocation and use on a particular NUMA node:

  ```
  # cat /sys/devices/system/node/node<N>/numastat
  ```

- To identify NUMA node mapping to cores, use one of the following:

  ```
  # numactl --hardware# cat /sys/devices/system/node/node<N>/cpulist
  ```

Driver Loading - NUMA Node

When loading, the Onload module will create a variety of common data structures. To ensure that these are created on the NUMA node nearest to the network adapter, `onload_tool reload` should be affinitized to a core on the correct NUMA node.

```
# numactl --cpunodebind=1 onload_tool reload
```

When there is more than one supported adapter in the same server, on different NUMA nodes, the user must select one node over the other when loading the driver, but also make sure that interrupt IRQs are affinitized to the correct local CPU node for each adapter.

onload_tool reload is single threaded, so running with "cpunodebind=0,1", for example, means the command could run on either node which is not identifiable by the user until after the command has completed.

Memory Policy

To guarantee that memory is appropriately allocated - and to ensure that memory allocations do not fail, a memory policy that binds to a specific NUMA node should be selected. When no policy is specified the system will generally use a default policy allocating memory on the node on which a process is executing.

Application Processing

The majority of processing by Onload occurs in the context of the Onloadaccelerated application. Various methods can be used to affinitize the Onload accelerated process; `numactl`, `taskset` or `cpusets` or the CPU affinity can be set programatically.

Workqueues

An Onloaded application will create two *shared* workqueues and one *per-stack* workqueue. The implementation of the workqueue differs between Linux kernels - and so does the method used to affinitize workqueues.

On more recent Linux kernels (3.10+) the Onload work queues will be initially affinitized to the node on which they are created. Therefore if the driver load is affinitized and the Onloaded application affinitized to the correct node, Onload stacks will be created on the correct node and there will be no further work required.

Specifying a cpumask via sysfs for a workqueue is NOT recommended as this can break ordering requirements.

On older Linux kernels dedicated workqueue threads are created - and these can be affinitized using taskset or cpusets. Identify the two workqueues shared by all Onload stacks:

```
onload-wqueue
sfc_vi
```

Identify the per-stack workqueue which has a name in the format `onload-wq<stack id>` (for example onload-wq:1 for stack 1).

Use the `onload_stackdump` command to identify Onload stacks and the PID of the process that created the stack:

```
# onload_stackdump
#stack-id stack-name      pids
0         -               106913
```

Use the Linux `pidof` command to identify the PIDs for Onload workqueues:

```
# pidof onload-wq:0 sfc_vi onload-wqueue
106930 105409 105431
```

It is recommended that the shared workqueues are affinitized immediately after the driver is loaded and the per-stack queue immediately after stack creation.

Interrupts

When Onload is being used in an interrupt-driven mode (see Interrupt Handling - Using Onload) interrupts should affinitized to the same NUMA node running the Onload application, but not on the same CPU core as the application.

When Onload is spinning (busy-wait) there will be few (if any) interrupts, so it is not a real concern where these are handled.

Verification

The onload_stackdump lots command is used to verify that allocations occur on the required NUMA node:

```
# onload_stackdump lots | grep numa
numa nodes: creation=0 load=0
numa node masks: packet alloc=1 sock alloc=1 interrupt=1
```

The load parameter identifies the node where the adapter driver has been loaded. The creation parameter identifies the node allocating memory for the Onload stack. The `numa node masks` identify which NUMA nodes allocate memory for packets and for sockets, and the nodes on which interrupts have actually occurred. A mask value of 1 identifies node 0, a value of 2 identifies node 1, a value of 3 identifies both nodes 0 and 1 etc.

For most purposes it is best when load and creation identify the same node which is also the node local to the network adapter. To identify the local node use the following:

```
# cat /sys/class/net/<interface>/device/numa_node
```

The cpu affinity of individual Onloaded threads can be identified with the following command:

```
# onload_stackdump threads
```

Interrupt Handling for the sfc Driver

This section describes interrupt handling on X2-series adapters (and earlier) that use the `sfc` driver.

For details of interrupt handling with X3-series adapters, refer to the *Alveo X3522 User Guide* (UG1523).

Default Behavior

Using the value identified from the `rss_cpus` option, the Solarflare NET driver will create a number of receive (and transmit) queues (termed an “RSS channel”) for each physical interface. By default the driver creates one RSS channel per CPU core detected in the server up to a maximum of 32.

The `rss_cpus` sfc driver module option can be set in a user created file <sfc.conf> in the `/etc/modprobe.d` directory. The driver must be reloaded before the option becomes effective. For example, `rss_cpus` can be set to an integer value:

```
options sfc rss_cpus=4
```

In the above example four receive queues are created per interface. The default value is `rss_cpus=cores`. Other available options are `rss_cpus=<int>`, `rss_cpus=hyperthreads` and `rss_cpus=packages`.

Note: If the sfc driver module parameter '`rss_numa_local`' is enabled, RSS will be restricted to use cores/hyperthreads on the NUMA node local to the network adapter.

Affinitizing RSS Channels to CPUs

As described in the previous section, the default behavior of the Solarflare network driver is to create one RSS channel per CPU core. At load time the driver affinitizes the interrupt associated with each RSS channel to a separate CPU core so the interrupt load is evenly distributed over the available CPU cores.

Note: These initial interrupt affinities will be disrupted and changed if the Linux IRQ balancer daemon is running. To stop the IRQ balancer use the following command: `# service irqbalance stop`

In the following example, we have a server with two dual-port network adapters (total of four network interfaces), installed in a server with two CPU sockets with eight cores per socket (hyperthreading is disabled).

If we set `rss_cpus=4`, each interface will create four RSS channels. The driver takes care to spread the affinitized interrupts evenly over the CPU topology, that is evenly between the two CPU sockets and evenly over shared L2/L3 caches.

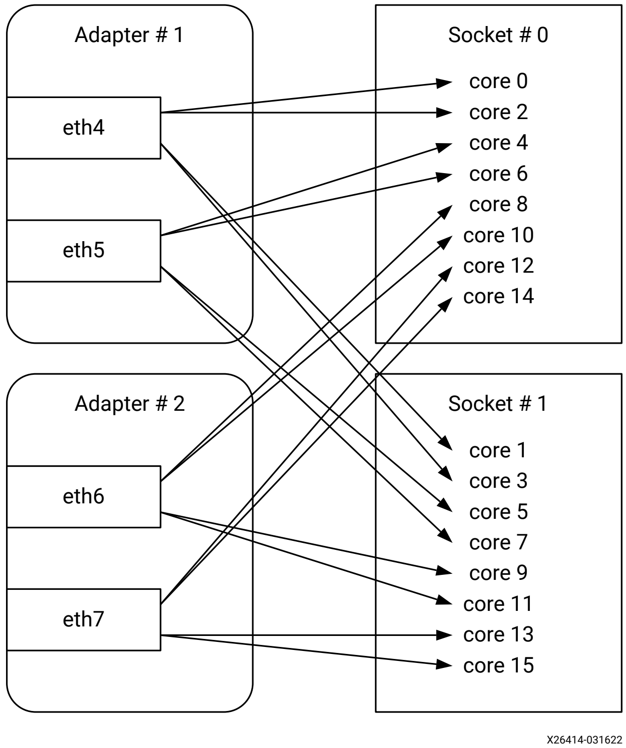
The driver also attempts to spread the interrupt load of the multiple network interfaces by using different CPU cores for different interfaces:

Table 8: **Example RSS Channel Mapping**

Interface	Number of RX Queues	Map to Cores
1	4	0,1,2,3
2	4	4,5,6,7
3	4	8,9,10,11
4	4	12,13,14,15

With four receive queues created per interface this results, on this machine, to the first network interface mapping to the four lowest number CPU cores, that is two cores from each CPU socket as illustrated below. The next network interface uses the next four CPUs until each CPU core is loaded with a single RSS channel – as illustrated in the following figure.

Figure 11: **Mapping RSS Channels to CPU cores.**



To identify the mapping of receive queues to CPU cores, use the following command:

```
# cat /proc/interrupts | grep eth4
106: 19  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  IR-PCI-MSI-edge eth4-0
107:  0 11  0  0  0  0  0  0  0  0  0  0  0  0  0  0  IR-PCI-MSI-edge eth4-1
108:  0  0 10  0  0  0  0  0  0  0  0  0  0  0  0  0  IR-PCI-MSI-edge eth4-2
109:  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  IR-PCI-MSI-edge eth4-3
```

Observe that each receive queue has an assigned IRQ. Receive queue eth4-0 is served by IRQ 106, eth4-1 by IRQ 107 etc.

sfcaffinity_config

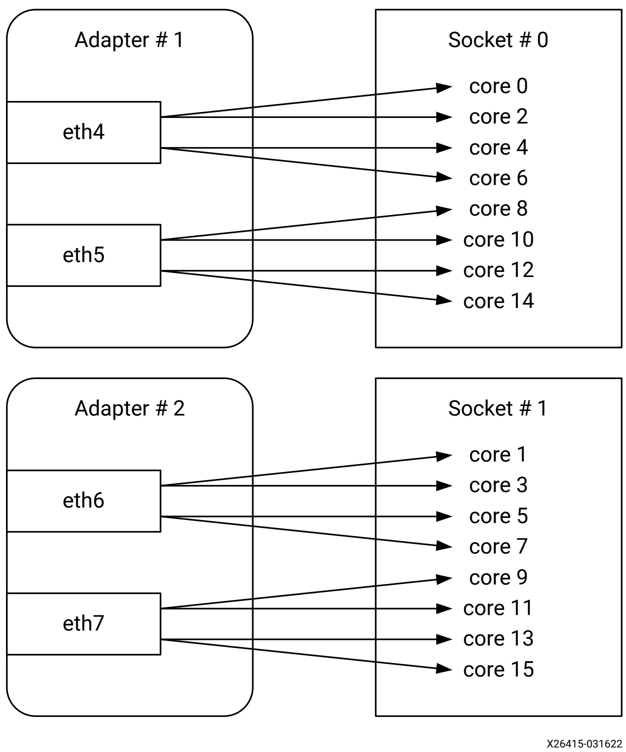
The OpenOnload distribution also includes the `sfcaffinity_config` script which can also be used to affinitize RSS channel interrupts. `sfcaffinity_config` has a number of command line options but a common way of running it is with the `auto` command:

```
# sfcaffinity_config auto
```

Auto instructs `sfcaffinity_config` to set interrupts affinities to evenly spread the RSS channels over the available CPU cores. Using the above scenario as an example, where `rss_cpus` has been set to 4, the command will affinitize the interrupt associated with each receive queue evenly over the CPU topology – in this case the first four CPU cores.

```
sfcaffinity_config: INFO: eth4: Spreading 4 interrupts evenly over 2 shared
caches
sfcaffinity_config: INFO: eth4: bind rxq 0 (irq 106) to core 1
sfcaffinity_config: INFO: eth4: bind rxq 1 (irq 107) to core 0
sfcaffinity_config: INFO: eth4: bind rxq 2 (irq 108) to core 3
sfcaffinity_config: INFO: eth4: bind rxq 3 (irq 109) to core 2
sfcaffinity_config: INFO: eth4: configure sfc_affinity n_rxqs=4
cpu_to_rxq=1,0,3,2,1,0,3,2,1,0,3,2,1,0,3,2
```

Figure 12: **Mapping with sfcaffinity_config auto**



In this example, after running the `sfcaffinity_config auto` command, interrupts for the four receive queues from the four interfaces are now all directed to the same four cores 0,1,2,3 as illustrated by the preceding figure.

***Note*:** Running the `sfcaffinity_config auto` command also disables the kernel `irqbalance` service to prevent interrupts being redirected by the kernel to other cores.

Using the irqbalance Service

If you want to keep using the `irqbalance` service, do not use the `sfcaffinity_config auto` command. Configure the `irqbalance` service using the `/etc/sysconfig/irqbalance` file:

- To prevent the network adapter interrupts from being redirected by `irqbalance`, append instances of the `--banirq` option. to the IRQBALANCE_ARGS environment variable. For example, to exclude interrupts 106-109 inclusive:

```
IRQBALANCE_ARGS="--banirq=106 --banirq=107 --banirq=108 --banirq=109"
```

- To exclude irqbalance from redirecting any interrupts to specific CPUs, include them in the IRQBALANCE_BANNED_CPUS bitmask. For example, to exclude CPUs 1 and 2, set it to 3 (bits 1 and 2 are set):

```
IRQBALANCE_BANNED_CPUS=3
```

Note: If this bitmask is not set, recent versions of `irqblance` do not use CPUs that are listed in the `isolcpus` kernel configuration parameter.

You can then manually configure the affinity of any excluded interrupts.

Restrict RSS to Local NUMA Node

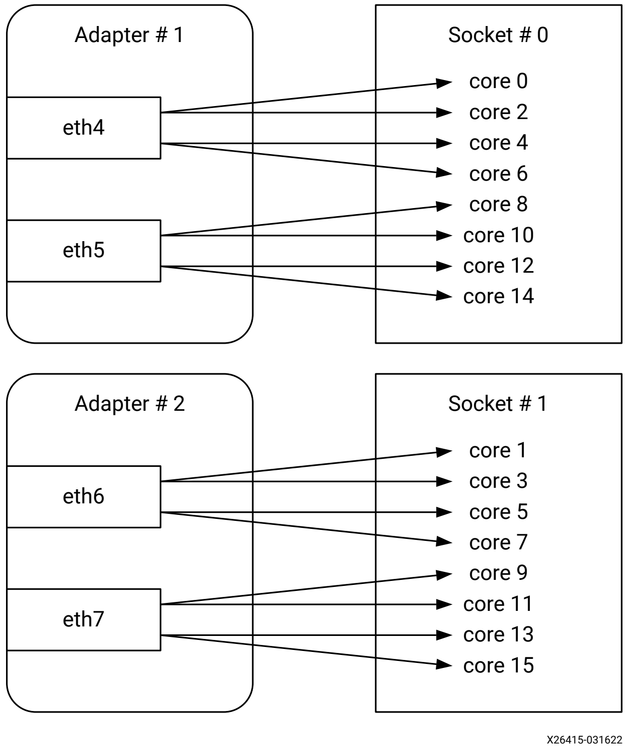
The sfc driver module parameter `rss_numa_local` will restrict RSS to only use CPU cores or hypterthreads (if hyperthreading is enabled) on the NUMA node local to the network adapter.

`rss_numa_local` does NOT restrict the number of RSS channels created by the driver – it instead works by restricting the RSS spreading so only the channels on the local NUMA node will receive kernel driver traffic.

In the default case (where `rss_cpus=cores`), one RSS channel is created per CPU core. However, the driver adjusts the RSS settings such that only the RSS channels affinitized to the local CPU socket receive traffic. It therefore has no effect on the Onload allocation and use of receive queues and interrupts.

The following figure identifies the receive queue interrupts spread when `rss_cpus=4` and `rss_numa_local=1`. In this machine adapter 1 is attached to the PCIe bus on socket #0 with adapter #2 attached to the PCIe bus on socket #1.

Figure 13: **Mapping with rss_numa_local**



Restrict RSS Receive Queues

The `ethtool -X` command can also be used to restrict the receive queues accessible by RSS. In the following example `rss_cpus=4` and `ethtool -x` identifies the four receive queues per interface:

```
# ethtool -x eth4

RX flow hash indirection table for eth4 with 4 RX ring(s):
    0:      0      1      2      3      0      1      2      3
    8:      0      1      2      3      0      1      2      3
   16:      0      1      2      3      0      1      2      3
   24:      0      1      2      3      0      1      2      3
   32:      0      1      2      3      0      1      2      3
   40:      0      1      2      3      0      1      2      3
   48:      0      1      2      3      0      1      2      3
   56:      0      1      2      3      0      1      2      3
   64:      0      1      2      3      0      1      2      3
   72:      0      1      2      3      0      1      2      3
   80:      0      1      2      3      0      1      2      3
   88:      0      1      2      3      0      1      2      3
   96:      0      1      2      3      0      1      2      3
  104:      0      1      2      3      0      1      2      3
  112:      0      1      2      3      0      1      2      3
  120:      0      1      2      3      0      1      2      3
```

To restrict RSS to spread receive flows evenly over the first two receive queues, use `ethtool -X`:

```
# ethtool -X eth4 equal 2

RX flow hash indirection table for eth4 with 4 RX ring(s):
    0:      0     1     0     1     0     1     0     1
    8:      0     1     0     1     0     1     0     1
   16:      0     1     0     1     0     1     0     1
   24:      0     1     0     1     0     1     0     1
   32:      0     1     0     1     0     1     0     1
   40:      0     1     0     1     0     1     0     1
   48:      0     1     0     1     0     1     0     1
   56:      0     1     0     1     0     1     0     1
   64:      0     1     0     1     0     1     0     1
   72:      0     1     0     1     0     1     0     1
   80:      0     1     0     1     0     1     0     1
   88:      0     1     0     1     0     1     0     1
   96:      0     1     0     1     0     1     0     1
  104:      0     1     0     1     0     1     0     1
  112:      0     1     0     1     0     1     0     1
  120:      0     1     0     1     0     1     0     1
```

Interrupt Handling - Using Onload

A thread accelerated by Onload will either be interrupt driven or it will be spinning.

When the thread is interrupt driven, a thread which calls into Onload to read from its receive queue and for which there are no received packets to be processed, will ‘sleep’ until an interrupt(s) from the kernel informs it that there is more work to do.

When a thread is spinning, it is busy waiting on its receive queue until packets are received - in which case the packets are retrieved and the thread returns immediately to the receive queue, or until the spin period expires. If the spin period expires the thread will relinquish the CPU core and ‘sleep’ until an interrupt from the kernel informs it that further packets have been received. If the spin period is set greater than the packet inter-arrival rate, the spinning thread can continue to spin and retrieve packets without interrupts occurring. Even when spinning, an application might experience a few interrupts.

As a general rule, when spinning, only a few interrupts will be expected so performance is typically insensitive as to which CPU core processes the interrupts.

However, when Onload is interrupt driven performance can be sensitive to where the interrupts are handled and will typically benefit to be on the same CPU socket as the application thread handling the socket I/O. The method required depends on the setting of the EF_PACKET_BUFFER_MODE environment variable:

- If EF_PACKET_BUFFER_MODE=0 or `2`, an Onload stack will use one or more of the interrupts assigned to the NET driver receive queues. The CPU core handling the interrupts is defined by the RSS mapping of receive queues to CPU cores:

- If `sfcaffinity_config` has been used to affinitize RSS channel interrupts, the interrupt handling core for the stack can be set using the EF_IRQ_CORE environment variable.

  It is only possible for interrupts to be handled on the requested core if a NET driver interrupt is assigned to the selected core.

  See Affinitizing RSS Channels to CPUs.

- Otherwise, the interrupt handling core for the stack can be set using the EF_IRQ_CHANNEL environment variable. Onload interrupts are handled by the same core assigned to the NET driver receive channel.

- If EF_PACKET_BUFFER_MODE=1 or `3`, the onload stack creates dedicated interrupts. The interrupt handling core for the stack can be set using the EF_IRQ_CORE environment variable.

For more information about these environment variables, see:

- EF_IRQ_CHANNEL
- EF_IRQ_CORE
- EF_PACKET_BUFFER_MODE.

When Onload is using a NET driver RSS channel for its source of interrupts, it can be useful to dedicate this channel to Onload and prevent the driver from using this channel for RSS traffic. See Restrict RSS to Local NUMA Node and Restrict RSS Receive Queues for methods of how to achieve this.

Performance Jitter

On any system reducing or eliminating jitter is key to gaining optimum performance, however the causes of jitter leading to poor performance can be difficult to define and difficult to remedy. The following section identifies some key points that should be considered.

- A first step towards reducing jitter should be to consider the configuration settings specified in the Appendix L: X2 Low Latency Quickstart - this includes the disabling of the irqbalance service, interrupt moderation settings and measures to prevent CPU cores switching to power saving modes.
- Use isolcpus to isolate CPU cores that the application - or at least the critical threads of the application will use and prevent OS housekeeping tasks and other non-critical tasks from running on these cores.
- Set an application thread running on one core and the interrupts for that thread on a separate core - but on the same physical CPU package. Even when spinning, interrupts can still occur, for example, if the application fails to call into the Onload stack for extended periods because it is busy doing other work.

- Ideally each spinning thread will be allocated a separate core so that, in the event that it blocks or is de-scheduled, it will not prevent other important threads from doing work. A common cause of jitter is more than one spinning thread sharing the same CPU core. Jitter spikes might indicate that one thread is being held off the CPU core by another thread.

  You can detect this using the scheduling statistics from `cat /proc/<pid>/sched` for the application threads. The `nr_involuntary_switches` counter records the number of times the process was descheduled, for example because of an interrupt handler or another task running on the same CPU core.

- When `EF_STACK_LOCK_BUZZ=1`, threads will spin for the EF_BUZZ_USEC period while they wait to acquire the stack lock. Lock buzzing can lead to unfairness between threads competing for a lock, and so result in resource starvation for one. Occurrences of this are counted in the 'stack_lock_buzz' counter. EF_STACK_LOCK_BUZZ is enabled by default when EF_POLL_USEC (spinning) is enabled.

- If a multi-thread application is doing lots of socket operations, stack lock contention will lead to send/receive performance jitter. In such cases improved performance can be had when each contending thread has its own stack. This can be managed with EF_STACK_PER_THREAD which creates a separate Onload stack for the sockets created by each thread. For an example see Minimizing Lock Contention.

  If separate stacks are not an option then it might be beneficial to reduce the EF_BUZZ_USEC period or to disable stack lock buzzing altogether.

- It is always important that threads that need to communicate with each other are running on the same CPU package so that these threads can share a memory cache.

  See Onload Deployment on NUMA Systems for more information.

- Jitter can also be introduced when some sockets are accelerated and others are not. Onload will ensure that accelerated sockets are given priority over non-accelerated sockets, although this delay will only be in the region of a few microseconds - not milliseconds, the penalty will always be on the side of the non-accelerated sockets. The environment variables EF_POLL_FAST_USEC and EF_POLL_NONBLOCK_FAST_USEC can be configured to manage the extent of priority of accelerated sockets over non-accelerated sockets.

- If traffic is sparse, spinning will deliver the same latency benefits, but the user should ensure that the spin timeout period, configured using the EF_POLL_USEC variable, is sufficiently long to ensure the thread is still spinning when traffic is received.

  See Spinning, Polling and Interrupts for more information.

- When applications only need to send and receive occasionally it might be beneficial to implement a keepalive - heartbeat mechanism between peers. This has the effect of retaining the process data in the CPU memory cache. Calling send or receive after a delay can result in the call taking measurably longer, due to the cache effects, than if this is called in a tight loop.

- Some adapters support warming the send path without actually transmitting data. This can similarly retain data in cache and so reduce jitter.

- On some servers BIOS settings such as power and utilization monitoring can cause unnecessary jitter by performing monitoring tasks on all CPU cores. The user should check the BIOS and decide if periodic tasks (and the related SMIs) can be disabled.
- The sysjitter utility can be used to identify and measure jitter on all cores of an idle system - refer to Sysjitter for details.

Using Onload Tuning Profiles

Environment variables set in the application user-space can be used to configure and control aspects of the accelerated application's performance. These variables can be exported using the Linux export command. For example:

```
export EF_POLL_USEC=100000
```

Onload supports tuning profile script files which are used to group environment variables within a single file to be called from the Onload command line.

The `latency` profile sets the `EF_POLL_USEC=100000` setting the busy-wait spin timeout to 100 milliseconds. The profile also disables TCP faststart for new or idle connections where additional TCP ACKs will add latency to the receive path. To use the profile include it on the `onload` command line. For example:

```
onload --profile=latency netperf -H onload2-sfc -t TCP_RR
```

Following Onload installation, the profiles that it provides are located in the following directory - this directory will be deleted by the `onload_uninstall` command:

```
/usr/libexec/onload/profiles
```

User-defined environment variables can be written to a user-defined profile script file (having a .opf extension) and stored in any directory on the server. The full path to the file should then be specified on the onload command line. For example:

```
onload --profile=/tmp/myprofile.opf netperf -H onload2-sfc -t TCP_RR
```

As an example the latency profile, provided by the Onload distribution is shown below:

```
# Onload low latency profile.
# Enable polling / spinning. When the application makes a blocking call
# such as recv() or poll(), this causes Onload to busy wait for up to 100ms
# before blocking.
onload_set EF_POLL_USEC=100000
# Disable FASTSTART when connection is new or has been idle for a while.
# The additional acks it causes add latency on the receive path.
onload_set EF_TCP_FASTSTART_INIT 0
onload_set EF_TCP_FASTSTART_IDLE 0
```

For a complete list of environment variables refer to Appendix A: Parameter Reference.

The latency-best Profile

The `latency-best` profile targets the lowest possible latency for a given release of Onload. This means that:

- Some features used in the profile might be experimental.
- The combination of features in the profile might not be suitable for all deployments, or all types of traffic.
- The profile is subject to change between releases.

  As new low-latency features become available, they will be added to the profile.

Consequently, the following is recommended:

- Always create a renamed copy of the profile, and use the copy.

  This will avoid the profile unexpectedly changing when you update Onload, and potentially breaking your applications.
- Always test and tune your copy of the profile in a non-production environment, before deploying it.

  This will avoid issues caused by combinations of settings that are inappropriate for your production systems.

IMPORTANT! *If you do not follow the above recommendations, and directly use the latency-best profile in a production environment, you might experience issues either now, or when upgrading Onload in the future.*

The nginx_reverse_proxy Profile

Onload's clustering capability has been extended to provide better support for the NGINX application's master process pattern and hot restart operation by correctly associating Onload stacks and application worker processes. To do so, the following settings are used:

```
EF_SCALABLE_FILTERS_ENABLE:          2
EF_CLUSTER_HOT_RESTART:              1
```

See EF_CLUSTER_HOT_RESTART, and EF_SCALABLE_FILTERS_ENABLE.

A new `nginx_reverse_proxy` profile has an example set of relevant configurations, including further settings. This profile searches for the NGINX configuration file, and then uses the settings from that file to make the correct Onload settings. Similar techniques can be used to make profiles that target other applications.

***Note*:** Use of this profile is not compatible with use of the onload extensions stackname API.

Benchmark Testing

Benchmark procedures using Onload, netperf and sfnt-pingpong are described in the Chapter 2: X3 Low Latency Quickstart.

Application-Specific Tuning

Advanced tuning requires closer examination of the application performance.

Onload includes a diagnostic application called `onload_stackdump`, which can be used to monitor Onload performance and to set tuning options.

The following worked examples demonstrate the use of `onload_stackdump` to examine aspects of the system performance, and hence to determine which environment variables should be set to tune the application. The process should address these tuning objectives.

- To have as much processing at user-level as possible.

  See Processing at User-Level.
- To have as few interrupts as possible.

  See As Few Interrupts as Possible.
- To eliminate drops.

  See Eliminating Drops.
- To minimize lock contention.

  See Minimizing Lock Contention.

For further examples and use of `onload_stackdump` refer to Appendix E: onload_stackdump.

Monitoring Using onload_stackdump

To use `onload_stackdump`, enter the following command:

```
onload_stackdump [command]
```

To list available commands and view documentation for `onload_stackdump` enter the following commands:

```
onload_stackdump doc
onload_stackdump -h
```

A specific stack number can also be provided on the `onload_stackdump` command line.

Worked Examples

This section contains some worked examples for tuning Onload.

Reducing Jitter from Page Faults

The Onload environment variable EF_PREFAULT_PACKETS will cause the user process to 'touch' the specified number of packet buffers when an Onload stack is created. This means that memory for these packet buffers is pre-allocated and memory-mapped into the user-process address space.

Pre-allocation is advised to prevent latency jitter caused by the allocation and memory-mapping overheads.

When deciding how many packets to prefault, the user should look at the alloc value when the `onload_stackdump packets` command is run. The alloc value is a high water mark identifying the maximum the number of packets being used by the stack at any singular point. Setting EF_PREFAULT_PACKETS to at least this value is recommended.

```
onload_stackdump packets$ onload_stackdump packets
ci_netif_pkt_dump_all: id=0
  pkt_sets: pkt_size=2048 set_size=1024 max=32 alloc=2
  pkt_set[0]: free=544
  pkt_set[1]: free=446 current
  pkt_bufs: max=32768 alloc=2048 free=990 async=0
  pkt_bufs: rx=1058 rx_ring=992 rx_queued=2 pressure_pool=64
  pkt_bufs: tx=0 tx_ring=0 tx_oflow=0
  pkt_bufs: in_loopback=0 in_sock=0
    994: 0x200 Rx
   n_zero_refs=1054 n_freepkts=1 estimated_free_nonb=1053
   free_nonb=0 nonb_pkt_pool=ffffffffffffffff
```

Note: It is not possible to prefault a number of packets exceeding the current value of EF_MAX_PACKETS.

When deciding how many packets to prefault the user should consider that Onload must allocate from the EF_MAX_PACKET pool, a number of packet buffers per receive ring per interface. Once these have been allocated, any remainder can be prefaulted.

Users who require to prefault the maximum possible number of available packets can set EF_PREFAULT_PACKETS and EF_MAX_PACKETS to the same value:

```
EF_PREFAULT_PACKETS=64000 EF_MAX_PACKETS=64000 onload <myapplication>...
```

Users can alternatively set EF_PREALLOC_PACKETS to allocate EF_MAX_PACKETS packet buffers during stack creation:

```
EF_PREALLOC_PACKETS=1 EF_MAX_PACKETS=64000 onload <myapplication>...
```

Refer to Appendix A: Parameter Reference for details of these variables.

CAUTION! *Prefaulting packet buffers for one stack will reduce the number of available buffers available for others. Users should consider that over allocation to one stack might mean spare (redundant) packet buffer capacity that could be better allocated elsewhere.*

Processing at User-Level

Many applications can achieve better performance when most processing occurs at user-level rather than kernel-level. To identify how an application is performing, enter the following command:

```
onload_stackdump lots | grep polls
```

Table 9: **Onload_stackdump Polling Counters**

Counter	Description
`k_polls`	Number of times the socket event queue was polled from the kernel.
`u_polls`	Number of times the socket event queue was polled from user space.
`periodic_polls`	Number of times a periodic timer has polled for events.
`interrupt_polls`	Number of times an interrupt polled for network events.
`deferred_polls`	Number of times poll has been deferred to the stack lock holder.
`timeout_interrupt_polls`	Number of times timeout interrupts polled for network events.

```
$ onload_stackdump lots | grep poll k_polls: 673 u_polls: 41
```

The output identifies many more `k_polls` than `u_polls` indicating that the stack is operating mainly at kernel-level and might not be achieving optimal performance. A possible cause is that application logic is taking longer than EF_POLL_USEC.

Solution

Terminate the application and set the EF_POLL_USEC parameter to `100000`. Restart the application and re-run `onload_stackdump`:

```
export EF_POLL_USEC=100000 onload_stackdump lots | grep polls$
onload_stackdump lots | grep polls k_polls: 673 u_polls: 1289
```

The output identifies that the number of `u_polls` is far greater than the number of `k_polls` indicating that the stack is now operating mainly at user-level.

For more information see Spinning, Polling and Interrupts.

As Few Interrupts as Possible

A tuned application will reach a balance between the number/rate of interrupts processed and the amount of real work that gets done, for example by processing multiple packets per interrupt rather than one. Even spinning applications can benefit from the occasional interrupt. For example when a spinning thread has been de-scheduled from a CPU, a timeout interrupt will prod the thread back to action after 250 µs.

```
# onload_stackdump lots | grep ^interrupt
```

Table 10: **Onload_stackdump Interrupt Counters**

Counter	Description
Interrupts	Total number of interrupts received for the stack.
Interrupt polls	Number of times the stack is polled - invoked by interrupt.
Interrupt evs	Number of events processed when invoked by an interrupt.
Interrupt wakes	Number of times the application is woken by interrupt.
Interrupt primes	Number of times interrupts are re-enabled (after spinning or polling the stack).
Interrupt no events	Number of stack polls for which there was no event to recover.
Interrupt lock contends	The application polled the stack and has the lock before an interrupt fired.
Interrupt budget limited	Number of times, when handling a poll in an interrupt, the poll was stopped when the NAPI budget was reached. Any remaining events are then processed on the stack workqueue.

Solution

If an application is observed taking lots of interrupts it might be beneficial to increase the spin time with the EF_POLL_USEC variable or setting a high interrupt moderation value for the net driver using `ethtool`. You should also ensure that the application CPU cores are isolated to avoid descheduling. If it is not possible to isolate the cores, consider switching to interrupt mode.

The number of interrupts on the system can also be identified from `/proc/interrupts`.

Eliminating Drops

The performance of networks is impacted by any packet loss. This is especially pronounced for reliable data transfer protocols that are built on top of unicast or multicast UDP sockets.

First check to see if packets have been dropped by the network adapter before reaching the Onload stack. Use `ethtool` to collect stats directly from the network adapter:

```
# ethtool -S enps0f0 | grep -E 'drop|discard'
```

Table 11: **Ethtool Drop Counters**

Counter	Description
`rx_noskb_drops`	Number of packets dropped when there are no further socket buffers to use.
`port_rx_nodesc_drops`	Number of packets dropped when there are no further descriptors in the rx ring buffer to receive them.
`port_rx_dp_di_dropped_packets`	Number of packets dropped because filters indicate the packets should be dropped - this can happen when packets do not match any filter or the matched filter indicates the packet should be dropped.
`port_rx_dp_q_disabled_packets`	Number of packets sent to a queue which does not exist. A small number might be observed following initialization or teardown, a larger number or incrementing number might indicate a mismatch between the size of a VI set and the actual number of VIs.
`port_rx_pm_discard_bb_overflow`	Number of packets discarded due to packet memory buffer overflow.
`port_rx_pm_discard_vfifo_full`	Count of the number of packets dropped because of a lack of main packet memory on the adapter to receive the packet into.
`port_rx_pm_discard_mapping`	Number of packets dropped because they have an 802.1p priority level configured to be dropped.

```
# ethtool -S enps0f0 | grep drop
     rx_noskb_drops: 0
     port_rx_nodesc_drops: 0
     port_rx_dp_di_dropped_packets: 681618610
```

Solution

The most common cause for this is the application being descheduled. You can detect this using the scheduling statistics from `cat /proc/<pid>/sched` for the application. The `nr_involuntary_switches` counter records the number of times the process was descheduled, for example because of an interrupt handler or another task running on the same CPU core.You should ensure that the application CPU cores are isolated to avoid descheduling. If it is not possible to isolate the cores, consider switching to interrupt mode.

If packet loss is observed at the network level due to a lack of receive buffering try increasing the size of the receive descriptor queue size via EF_RXQ_SIZE. If packet drops are observed at the socket level consult the application documentation. It might also be worth experimenting with socket buffer sizes (see EF_UDP_RCVBUF). Setting the EF_EVS_PER_POLL variable to a higher value can also improve efficiency. Refer to Appendix A: Parameter Reference for descriptions of these variables.

Minimizing Lock Contention

Lock contention can greatly affect performance. When threads share a stack, a thread holding the stack lock will prevent another thread from doing useful work. Applications with fewer threads might be able to create a stack per thread (see EF_STACK_PER_THREAD and Stacks API).

Use `onload_stackdump` to identify instances of lock contention:

```
# onload_stackdump lots | egrep "(lock_)|(sleep)"
```

Table 12: **Onload_stackdump Lock and Sleep Counters**

Counter	Description
`periodic_lock_contends`	Number of times periodic timer could not get the stack lock.
`interrupt_lock_contends`	Number of times the interrupt handler could not get the stack lock because it is already held by user level or other context.
`timeout_interrupt_lock_contends`	Number of times timeout interrupts could not lock the stack.
`sock_sleeps`	Number of times a thread has blocked on a single socket.
`sock_sleep_primes`	Number of times select/poll/epoll enabled interrupts.
`unlock_slow`	Number of times the slow path was taken to unlock the stack lock.
`unlock_slow_pkt_waiter`	Number of times packet memory shortage provoked the unlock slow path.
`unlock_slow_socket_list`	Number of times the deferred socket list provoked the unlock slow path.
`unlock_slow_need_prime`	Number of times interrupt priming provoked the unlock slow path.
`unlock_slow_wake`	Number of times the unlock slow path was taken to wake threads.
`unlock_slow_swf_update`	Number of times the unlock slow path was taken to update sw filters.
`unlock_slow_close`	Number of times the unlock slow path was taken to close sockets/pipes.
`unlock_slow_syscall`	Number of times a syscall was needed on the unlock slow path.
`lock_wakes`	Number of times a thread is woken when blocked on the stack lock.
`stack_lock_buzz`	Number of times a thread has spun waiting for the stack lock.
`sock_lock_sleeps`	Number of times a thread has slept waiting for a sock lock.
`sock_lock_buzz`	Number of times a thread has spun waiting for a sock lock.
`tcp_send_ni_lock_contends`	Number of times TCP `sendmsg()` contended the stack lock
`udp_send_ni_lock_contends`	Number of times UDP `sendmsg()` contended the stack lock
`getsockopt_ni_lock_contends`	Number of times `getsockopt()` contended the stack lock.
`setsockopt_ni_lock_contends`	Number of times `setsockopt()` contended the stack lock.

Table 12: **Onload_stackdump Lock and Sleep Counters** *(cont'd)*

Counter	Description
`lock_dropped_icmps`	Number of dropped ICMP messages not processed due to contention.

Solution

Performance will be improved when stack contention is kept to a minimum. When threads share a stack it is preferable for a thread to spin rather than sleep when waiting for a stack lock. The EF_BUZZ_USEC value can be increased to reduce 'sleeps'. Where possible use stacks per thread.

Stack Contention - Deferred Work

When multiple threads share an Onload stack, the ability for one thread to defer sending tasks to another thread that is currently holding the stack lock, can mitigate the effects of lock contention. When sending data, contention occurs when one thread calls `send()`, while another thread holds the stack lock. The task of sending the data can be deferred to the lock holder - freeing the deferring thread to continue with other work. However a `send()` which also processes a lot of deferred work will take longer to execute - preventing other threads from getting the stack lock.

A thread which calls send() when the stack EF_DEFER_WORK_LIMIT has been reached cannot defer further work to the lock holder, but is forced to block and wait for the stack lock. The `defer_work_limited` counter identifies the number of these occurrences.

onload_stackdump per-socket counters will indicate the level of deferred work on each socket within a stack. For example:

```
TCP 2:10 lcl=172.16.20.123:4112 rmt=172.16.20.88:4112 ESTABLISHED  snd:
limited rwnd=17 cwnd=129 nagle=0 more=0 app=103905  tx: defer=48799 nomac=0
warm=0 warm_aborted=0
```

onload_stackdump per-stack counters also indicate the level of lock contention:

- `deferred_work` - the number packets sent using the deferred mechanism.
- `defer_work_limited` - the number of times a sending thread is prevented from deferring to the stack lock holder because the EF_DEFER_WORK_LIMIT has been reached.
- `deferred_polls` - a thread is prevented from polling the stack when another thread has the stack lock. The poll is deferred to the lock holder. The lock holder will place any ready received data on the correct socket queues and wake other threads if there is work to be done.

Solutions

To reduce the level of stack lock contention, the following actions are recommended:

- For affected stacks, reduce the number of threads performing network I/O.

- Applications with fewer threads can use a stack for each thread - see EF_STACK_PER_THREAD.
- Bind critical sockets to selected stacks - see Stacks API.
- For TCP connections, use `onload_move_fd()` to place sockets accepted from a listening socket into multiple stacks.

For more information see Minimizing Lock Contention.

Chapter 6

Onload Functionality

This chapter provides detailed information about specific aspects of Onload operation and functionality.

Onload Transparency

Onload provides significantly improved performance without the need to rewrite or recompile the user application, whilst retaining complete interoperability with the standard TCP and UDP protocols.

In the regular kernel TCP/IP architecture an application is dynamically linked to the libc library. This OS library provides support for the standard BSD sockets API via a set of 'wrapper' functions with real processing occurring at the kernel-level. Onload also supports the standard BSD sockets API. However, in contrast to the kernel TCP/IP, Onload moves protocol processing out of the kernel-space and into the user-level Onload library itself.

As a networking application invokes the standard socket API function calls such as `socket()`, `read()`, `write()` etc, these are intercepted by the Onload library making use of the LD_PRELOAD mechanism on Linux. From each function call, Onload will examine the file descriptor identifying those sockets using a supported interface - which are processed by the Onload stack, whilst those not using a supported interface are transparently passed to the kernel stack.

Onload Stacks

An Onload 'stack' is an instance of a TCP/IP stack. The stack includes transmit and receive buffers, open connections and the associated port numbers and stack options. Each stack has associated with it one or more Virtual NICs (typically one per physical port that stack is using).

In normal usage, each accelerated process will have its own Onload stack shared by all connections created by the process. It is also possible for multiple processes to share a single Onload stack instance (refer to Stack Sharing), and for a single application to have more than one Onload stack. Refer to Appendix D: Onload Extensions API.

***Note*:** An Onload stack can only exist in a single network namespace and cannot be shared by different network namespaces.

Virtual Network Interface (VNIC)

SFN8000 series and X2 series Solarflare network adapters support 1024 transmit queues, 1024 receive queues, 1024 event queues and 1024 timer resources per network port. A VNIC (virtual network interface) consists of one unique instance of each of these resources which allows a VNIC client such as the Onload stack, an isolated and safe mechanism of sending and receiving network traffic. Received packets are steered to the correct VNIC by means of IP/MAC filter tables on the network adapter and/or Receive Side Scaling (RSS). An Onload stack allocates one VNIC per Solarflare network port so it has a dedicated send and receive channel from user mode.

Following a reset of the Solarflare network adapter driver, all virtual interface resources including Onload stacks and sockets will be re-instated. The reset operation will be transparent to the application, but traffic will be lost during the reset.

Functional Overview

When establishing its first socket, an application is allocated an Onload stack which allocates the required VNICs.

When a packet arrives, IP filtering in the adapter identifies the socket and the data is written to the next available receive buffer in the corresponding Onload stack. The adapter then writes an event to an "event queue" managed by Onload. If the application is regularly making socket calls, Onload is regularly polling this event queue, and then processing events directly rather than interrupts are the normal means by which an application is able to rendezvous with its data.

User-level processing significantly reduces kernel/user-level context switching and interrupts are only required when the application blocks - because when the application is making socket calls, Onload is busy processing the event queue picking up new network events.

Onload with Mixed Network Adapters

A server might be equipped with supported network interfaces and unsupported network interfaces. When an application is accelerated, Onload reads the Linux kernel routing tables to identify which network interface is required to make a connection. If an unsupported interface is required to reach a destination Onload will pass the connection to the kernel TCP/IP stack. No additional configuration is required to achieve this as Onload does this automatically.

Maximum Number of Network Interfaces

A maximum of 32 network interfaces capable of acceleration can be registered with the Onload driver. Third party interfaces that cannot be accelerated do not count towards this limit.

Further limits are set by values in the `src/include/ci/internal/transport_config_opt.h` header file within the source code:

- The maximum number of hardware ports in the system is set by the `CI_CFG_MAX_HWPORTS` value. The default for this value is 8, and it can be increased to a maximum of 32.
- The maximum number of network interfaces per stack is set by the `CI_CFG_MAX_INTERFACES` value. The default for this value is 8, and it can be increased to a maximum of 16.

  If this value is less than the number of interfaces that the driver provides:

  - The interfaces can be distributed between stacks using blacklisting or namespacing. See Allowlist and Denylist for Interfaces, and Namespaces.
  - If there remain more interfaces visible to an Onload stack than it can support, then the higher interfaces will not be accelerated.

Following changes to these values it is necessary to rebuild and reinstall Onload.

Allowlist and Denylist for Interfaces

The user can, on a system-wide or a per-stack basis, specify interfaces that can be used by the Onload stack, or prevent interfaces being used by the stack:

- The system-level configuration Files in `/proc/driver/sfc_resource/<intf-name>/enable` can be written with “`1`” or “`0`” to enable or disable a network interface for all applications on the system:

  This feature is available from Onload-7.1.0 onwards.
- The per-stack environment variables EF_INTERFACE_BLACKLIST and EF_INTERFACE_WHITELIST are space-separated lists of interfaces. The network interfaces can be identified as the real interface such as p1p1 or eth1, or as a higher-order interface such as a VLAN, MACVLAN, team or bond. When the Onload stack is created interface names will be resolved to identify the underlying and adapter interface.

  This feature is available from OpenOnload-201710 onwards.

By default, all interfaces are enabled when the driver is loaded. To disable all interfaces initially, pass the “enable_accel_by_default=0” module parameter to `sfc_resource`.

All interfaces identified in the allowlist will be accelerated by Onload, however the denylist takes precedence such that an interface appearing in both lists will not be accelerated by Onload.

If an interface is in the denylist:

- Onloaded applications will not accelerate sockets using the interface (the socket will be handled by the kernel).
- ef_vi applications will fail the `ef_pd_alloc()` call at startup.

Onload Accelerated Process IDs

To identify processes accelerated by Onload use the `onload_fuser` command:

```
# onload_fuser -v
9886 ping
```

Only processes that have created an Onload stack are present. Processes which are loaded under Onload, but have not created any sockets are not present. The `onload_stackdump` command can also list accelerated processes. See List Onloaded Processes for details.

File Descriptors, Stacks, and Sockets

For an Onloaded process it is possible to identify the file descriptors, Onload stacks and sockets being accelerated by Onload. Use the `/proc/<PID>/fd` file - supplying the PID of the accelerated process. For example:

```
# ls -l /proc/9886/fd
total 0
lrwx------ 1 root root 64 May 14 14:09 0 -> /dev/pts/0
lrwx------ 1 root root 64 May 14 14:09 1 -> /dev/pts/0
lrwx------ 1 root root 64 May 14 14:09 2 -> /dev/pts/0
lrwx------ 1 root root 64 May 14 14:09 3 -> onload:[tcp:6:3]
lrwx------ 1 root root 64 May 14 14:09 4 -> /dev/pts/0
lrwx------ 1 root root 64 May 14 14:09 5 -> /dev/onload
lrwx------ 1 root root 64 May 14 14:09 6 -> onload:[udp:6:2]
```

Accelerated file descriptors are listed as symbolic links to `/dev/onload`. Accelerated sockets are described in [`protocol:stack:socket`] format.

System Calls Intercepted by Onload

System calls intercepted by the Onload library are listed in the following file:

```
[onload]/src/include/onload/declare_syscalls.h.tmpl
```

Linux Sysctls

The Linux directory `/proc/sys/net/ipv4` contains default settings which tune different parts of the IPv4 networking stack. In many cases Onload takes its default settings from the values in this directory. In some cases the default can be overridden, for a specified processes or socket, using socket options or with Onload environment variables. The following sections identify the default Linux values and how Onload tuning parameters can override the Linux settings.

Refer to the Appendix A: Parameter Reference for details of environment variables.

tcp_slow_start_after_idle

Controls congestion window validation as per RFC2861.

Onload value

“off” by default in Onload, as it's not usually useful in modern switched networks.

Comments

#define CI_CFG_CONGESTION_WINDOW_VALIDATION in `transport_config_opt.h`.

Recompile after changing.

tcp_congestion_control

Determines what congestion control algorithm is used by TCP. Valid settings include `reno`, `bic` and `cubic`.

Onload value

No direct equivalent - see TCP Congestion Control.

Comments

See EF_CONG_AVOID_SCALE_BACK.

tcp_adv_win_scale

Defines how quickly the TCP window will advance.

Onload value

No direct equivalent - see TCP Congestion Control.

Comments

See EF_TCP_ADV_WIN_SCALE_MAX.

tcp_rmem

The default size of sockets' receive buffers (in bytes)

Onload value

Defaults to the currently active Linux settings, but is ignored on TCP accepted sockets. Refer to EF_TCP_RCVBUF_ESTABLISHED_DEFAULT.

Comments

Can be overriden with the `SO_RCVBUF` socket option.

Can be set with EF_TCP_RCVBUF.

tcp_wmem

The default size of sockets' send buffers (in bytes).

Onload value

Defaults to the currently active Linux settings.

Comments

EF_TCP_SNDBUF (see EF_TCP_SNDBUF) overrides `SO_SNDBUF` which overrides `tcp_wmem`.

tcp_dsack

Allows TCP to send duplicate SACKS.

Onload value

Uses the currently active Linux settings.

Comments

—

tcp_fack

Enables forward acknowledgment algorithm.

Onload value

Enabled.

Comments

—

tcp_sack

Enable TCP selective acknowledgments, as per RFC2018.

Onload value

Enabled by default. Onload uses the currently active Linux setting.

Comments

Clear bit 2 of EF_TCP_SYN_OPTS (see EF_TCP_SYN_OPTS) to disable.

tcp_max_syn_backlog

The maximum size of a listening socket's backlog.

Onload value

Set with EF_TCP_BACKLOG_MAX (see EF_TCP_BACKLOG_MAX).

Comments

—

tcp_synack_retries

The maximum number of retries of SYN-ACKs.

Onload value

Set with EF_RETRANSMIT_THRESHOLD_SYNACK (see EF_RETRANSMIT_THRESHOLD_SYNACK).

Comments

Default value 5.

Namespaces

Onload includes support for all Linux namespace types. Network namespace support has been primarily implemented for use with Onload in a Docker container, but Onload will support all Linux namespace types in host and container environments. Onload will create a control_plane instance per namespace in which an Onload stack is created.

An Onload stack can exist in only one network namespace. The stack cannot be moved between network namespaces and cannot be shared by multiple network namespaces.

Note: Onload stacks cannot be shared by different network namespaces.

The following are not supported:

- multiple interfaces with the same IP address
- multiple interfaces with the same MAC address.

User-space Control Plane Server

Starting from the onload-201710 release, Onload deploys a user-space control plane daemon.

A single `onload_cp_server` process is created per network namespace in which there is an active onload_stack.

The `onload_cp_server` process is spawned when the first Onload stack is created in a namespace, and stack creation will wait until the process becomes ready - this might result in a noticeable delay. Onload also spawns a control plane server for the default (main) network namespace at load time, thus avoiding the delay for the majority of use-cases.

The `onload_cp_server` exits after a “grace period” when the last stack in the namespace has been destroyed. A new stack, created in the same namespace, before the grace period expires, can use the existing `onload_cp_server` avoiding the stack creation delay. The grace period, in seconds, can be managed - see options below.

The compiler architecture of the `onload_cp_server` must match that of the host kernel.

See also the Management Information Base in Appendix K: Management Information Base.

Onload Options for the Control Plane Server

The following Onload options configure the user-space Control Plane Server. For the latest details of options use the following command:

```
# modinfo onload
```

Table 13: **Onload Options for the Control Plane Server**

Option	Description
cplane_init_timeout (int)	Time in seconds to wait for the control plane to initialize when creating a stack. This initialization requires that the user-level control plane process be spawned if one is not already running for the current network namespace. If this parameter is zero, stack-creation will fail immediately if the control plane is not ready. If it is negative, stack-creation will block indefinitely in wait for the control plane.
cplane_spawn_server (bool)	If true, control plane server processes are spawned on-demand. Typically this occurs when a stack is created in a network namespace in which there are no other stacks.
cplane_server_path	Sets the path to the `onload_cp_server` binary. Defaults to `/sbin/onload_cp_server` if empty.
cplane_server_params	Set additional parameters for the `onload_cp_server` server when it is spawned on-demand.
cplane_server_grace_timeout	Time in seconds to wait before killing the control plane server after the last user has gone (that is, after the last Onload stack in this namespace has been destroyed). It is used with `cplane_spawn_server = Y` only.
cplane_route_request_limit (int)	Queue depth limit for route resolution requests.
cplane_route_request_timeout_ms	Time out value for route resolution requests.
ONLOAD_CPLANE_USER	Add this option in `/etc/sysconfig/onload` file to restrict access to users. The server retains CAP_NET_ADMIN capability.
cplane_server_uid	Restrict cplane access to specific user.
cplane_server_gid	Restrict cplane access to specific user group.
cplane_use_prefsrc_as_local	If true, use a preferred source of any accelerated route in the same way as an address assigned to accelerated interface. This setting allows the acceleration of unbound connections via accelerated routes when the preferred source is assigned to another network interface. See also oof_use_all_local_ip_addresses module parameter.

Parameters for onload_cp_server

The `onload_cp_server` process has parameters that can be passed to it on startup. These parameters are usually specified by setting the `cplane_server_params` module option. For example, you might add the following line to a file in the `/etc/modprobe.d` directory:

```
options onload cplane_server_params="--llap-max=100 --fwd-max=512"
```

These parameters can also be given with the environment variable CI_OPTS:

Table 14: **Parameters for onload_cp_server**

Parameter	Description	Default
-s --dump	Interval between table dump, in seconds	3
--no-listen	Do not listen for netlink updates	false
--fwd-refresh	Interval between fwd cache housekeeping, in seconds	5
-t --time-to-live	Time-to-live for forward cache entries, in seconds	300
-K --log-to-kmsg	Log to `/dev/kmsg` (with -D only)	false
--affinity	CPU mask to set the cp_server affinity to. Limited to 64 CPUs.	-1
-l --llap-max	Maximum number of network interfaces (including “lo”)	32
-b --bond-max	Maximum number of bond/team interfaces and their ports	64
-m --mac-max	Maximum number of ARP entries in the system (will be rounded up to a power of 2)	1024
-f --fwd-max	Maximum number of remote addresses used by Onload (will be rounded up to a power of 2). The default value is usually sufficient; but might need to be increased for complex routing setups.	1024
-r --fwd-req-max	Ignored	—
--bond-base-period	Interval between background bond-state polls, in milliseconds	100
--bond-peak-period	Interval between peak-rate bond-state polls, in milliseconds	10
--bond-peak-polls	Number of peak-rate bond polls before reverting to background rate	20
--bond-3ad-period	interval between re-dumping bond-3ad slave state, in milliseconds	100

The following additional parameters are automatically set when the Onload drivers launch the `onload_cp_server` process.

***Note*:** The default values shown in the following table are for when the process is not started by the Onload drivers.)

Table 15: **Additional Parameters for onload_cp_server**

Parameter	Description	Default
--network-namespace-file	Path to a file specifying the network namespace to manage	—
-D --daemonise	Daemonise at start and log to syslog	false
--bootstrap	Manage the namespace even if there are no clients	false
-h --hwport-max	Maximum number of hardware ports When Onload is starting the onload_cp_server process, set this via CI_CFG_MAX_HWPORTS.	8
-i --ipif-max	Maximum number of local IP addresses (on all interfaces) When Onload is starting the onload_cp_server process, set this via CI_CFG_MAX_LOCAL_IPADDRS	64
--force-bonding-netlink	Only use netlink (and not ioctls) for determining bonding state	false
--no-ipv6	Disable IPv6 support	false

For the latest details of parameters use the following command:

```
# onload_cp_server --help
```

Changing Onload Control Plane Table Sizes

The default sizes of the tables used by the Onload Control Plane are normally sufficient for the majority of applications. The table sizes can be changed, but creating larger tables might impact application performance.

The procedure for doing so changed in the onload-201710 release.

***Note*:** Following changes Onload should be restarted using the reload command: `onload_tool reload`.

Changing Table Sizes for Onload-201710 and Later

From onload-201710 onwards, the sizes of Onload Control Plane tables are set by parameters for the `onload_cp_server` process. These are listed in Parameters for onload_cp_server.

If non-default values are needed, the user should create a file in the `/etc/modprobe.d` directory. The file must have the `.conf` extension. The `onload_cp_server` parameters can be added to the file in a single line, space-separated, in the following format:

```
options onload cplane_server_params="--mac-max=512 --fwd-max=256"
```

Changing Table Sizes before Onload-201710

Releases of Onload prior to onload-201710 support the following runtime configurable options which determine the size of control plane tables:

Table 16: **Obsolete Options for Changing Table Sizes**

Option	Description	Default
max_layer2_interfaces	Sets the maximum number of network interfaces, including physical interfaces, VLANs and bonds, supported in Onload's control plane. Table: mib_llap (also referred to in messages as the "local address table").	50
max_local_addrs	Sets the maximum number of local network addresses supported in Onload's control plane. Table: mib_ipif	256
max_neighs	Sets the maximum number of rows in the Onload ARP/neighbor table. The value is rounded up to a power of two. Table: mib_mac	1024
max_routes	Sets the maximum number of entries in the Onload route table. The default size is usually sufficient; but might need to be increased for complex routing setups. The minimum size needed can be calculated as ((local IP addresses +1) * remote IP addresses); or determined by: `ip link show | wc -l` Table: mib_fwd	256

If non-default values are needed, the user should create a file in the `/etc/modprobe.d` directory. The file must have the `.conf` extension. Onload options can be added to the file, a single option per line, in the following format:

```
options onload_cplane max_neighs=512
options onload_cplane max_routes=256
```

SO_BINDTODEVICE

In response to the `setsockopt()` function call with `SO_BINDTODEVICE`, sockets identifying unsupported interfaces will be handled by the kernel and all sockets identifying supported interfaces will be handled by Onload. All sends from a socket are sent via the bound interface. Only traffic received over the bound interface will be delivered to the socket.

Multiplexed I/O

Linux supports three common methods for handling multiplexed I/O operation; `poll()`, `select()` and the `epoll` set of functions.

The general behavior of the `poll()`, `select()` and `epoll_wait()` functions with Onload is as follows:

- If there are operations ready on any file descriptors, `poll()`, `select()` and `epoll_wait()` will return immediately. Refer to Poll, ppoll, Select, pselect and Epoll for specific behavior details.
- If there are no file descriptors ready and spinning is not enabled, calls to `poll()`, `select()` and `epoll_wait()` will enter the kernel and block.
- In the cases of `poll()`and `select()`, when the set contains file descriptors that are not accelerated sockets, there is a slight latency overhead as Onload must make a system call to determine the readiness of these sockets. There is no such cost when using `epoll_wait()` and a system call is only needed when non-Onload descriptors become ready.

  To ensure that non-accelerated (kernel) file descriptors are checked when there are no events ready on accelerated (onload) descriptors, disable the following options:

  - EF_SELECT_FAST and EF_POLL_FAST - setting both to zero.
  - EF_POLL_FAST_USEC and EF_SELECT_FAST_USEC - setting both to zero.
- If there are no file descriptors ready and spinning is enabled, Onload will spin to ensure that accelerated sockets are polled a specified number of times before unaccelerated sockets are examined. This reduces the overhead incurred when Onload has to call into the kernel and reduces latency on accelerated sockets.

The following subsections discuss the use of these I/O functions and Onload environment variables that can be used to manipulate behavior of the I/O operation.

Poll, ppoll

The `poll()`, `ppoll()` file descriptor set can consist of both accelerated and non-accelerated file descriptors. The environment variable EF_UL_POLL enables/disables acceleration of the `poll()`, `ppoll()` function calls. Onload supports the following options for the EF_UL_POLL variable:

Table 17: **Options for the EF_UL_POLL Variable**

Value	Behavior
0	Disable acceleration at user-level. Calls to `poll()`, `ppoll()` are handled by the kernel. Spinning cannot be enabled.
1	Enable acceleration at user-level. Calls to `poll()`, `ppoll()` are processed at user level. Spinning can be enabled and interrupts are avoided until an application blocks.

Additional environment variables can be employed to control the `poll()`, `ppoll()` functions and to give priority to accelerated sockets over non-accelerated sockets and other file descriptors. Refer to EF_POLL_FAST, EF_POLL_FAST_USEC and EF_POLL_SPIN.

Select, pselect

The `select()`, `pselect()` file descriptor set can consist of both accelerated and non-accelerated file descriptors. The environment variable EF_UL_SELECT enables/disables acceleration of the `select()`, `pselect()` function calls. Onload supports the following options for the EF_UL_SELECT variable:

Table 18: **Options for the EF_UL_SELECT Variable**

Value	Epoll Behavior
0	Disable acceleration at user-level. Calls to `select()`, `pselect()` are handled by the kernel. Spinning cannot be enabled.
1	Enable acceleration at user-level. Calls to `select()`, `pselect()` are processed at user-level. Spinning can be enabled and interrupts are avoided until an application blocks.

Additional environment variables can be employed to control the `select()`, `pselect()` functions and to give priority to accelerated sockets over non-accelerated sockets and other file descriptors. Refer to EF_SELECT_FAST and EF_SELECT_SPIN.

Epoll

The epoll set of functions, `epoll_create()`, `epoll_ctl()`, `epoll_wait()`, `epoll_pwait()`, are accelerated in the same way as poll and select. The environment variable EF_UL_EPOLL enables/disables epoll acceleration. Refer to the release change log for enhancements and changes to epoll behavior.

Using Onload an epoll set can consist of both Onload file descriptors and kernel file descriptors. Onload supports the following options for the EF_UL_EPOLL environment variable:

Table 19: **Options for the EF_UL_EPOLL Variable**

Value	Epoll Behavior
0	Accelerated epoll is disabled and `epoll_ctl()`, `epoll_wait()` and `epoll_pwait()` function calls are processed in the kernel. Other functions calls such as `send()` and `recv()` are still accelerated. Interrupt avoidance does not function and spinning cannot be enabled. If a socket is handed over to the kernel stack after it has been added to an epoll set, it will be dropped from the epoll set. `onload_ordered_epoll_wait()` is not supported.
1	Function calls to `epoll_ctl()`, `epoll_wait()`, `epoll_pwait()` are processed at user level. Delivers best latency except when the number of accelerated file descriptors in the epoll set is very large. This option also gives the best acceleration of `epoll_ctl()` calls. Spinning can be enabled and interrupts are avoided until an application blocks. CPU overhead and latency increase with the number of file descriptors in the epoll set. `onload_ordered_epoll_wait()` is supported.
2	Calls to `epoll_ctl()`, `epoll_wait()`, `epoll_pwait()` are processed in the kernel. Delivers best performance for large numbers of accelerated file descriptors. Spinning can be enabled and interrupts are avoided until an application blocks. CPU overhead and latency are independent of the number of file descriptors in the epoll set. `onload_ordered_epoll_wait()` is not supported.
3	Function calls to `epoll_ctl()`, `epoll_wait()`, `epoll_pwait()` are processed at user level. Delivers best acceleration latency for `epoll_ctl()` calls and scales well when the number of accelerated file descriptors in the epoll set is very large - and all sockets are in the same stack. The cost of the `epoll_wait()` is independent of the number of accelerated file descriptors in the set and depends only on the number of descriptors that become ready. The benefits will be less if sockets exist in different Onload stacks: • From Onload 201805 onwards, each socket can be in up to four epoll sets at a time, provided that each epoll set is in a different process • Otherwise, each socket can be in at most one epoll set at a time. In such cases the recommendation is to use EF_UL_EPOLL`=2`. EF_UL_EPOLL`=3` does not allow monitoring the readiness of the epoll file descriptors from another `epoll`/`poll`/`select`. EF_UL_EPOLL`=3` cannot support epoll sets which exist across `fork()`. Spinning can be enabled and interrupts are avoided until an application blocks. `onload_ordered_epoll_wait()` is supported.

The relative performance of epoll options 1 and 2 depends on the details of application behavior as well as the number of accelerated file descriptors in the epoll set. Behavior can also differ between earlier and later kernels and between Linux realtime and non-realtime kernels. Generally the OS will allocate short time slices to a user-level CPU intensive application which might result in performance (latency spikes). A kernel-level CPU intensive process is less likely to be de-scheduled resulting in better performance. You are recommended to evaluate options 1 and 2 for applications that manages many file descriptors, or to try option 3 (onload-201502 and later) when using very large sets and all sockets are in the same stack.

Additional environment variables can be employed to control the `epoll_ctl()`, `epoll_wait()` and `epoll_pwait()` functions and to give priority to accelerated sockets over non-accelerated sockets and other file descriptors. Refer to EF_EPOLL_CTL_FAST, EF_EPOLL_SPIN and EF_EPOLL_MT_SAFE.

Refer also to Known Issues with Epoll.

Wire Order Delivery

When a TCP or UDP application is working with multiple network sockets simultaneously it is difficult to ensure data is delivered to the application in the strict order it was received from the wire across these sockets.

The `onload_ordered_epoll_wait()` API is an Onload alternative implementation of `epoll_wait()` providing additional data allowing a receiving application to recover in-order timestamped data from multiple sockets. To maintain wire order delivery, only a specific number of bytes, as identified by the `onload_ordered_epoll_event`, should be recovered from a ready socket.

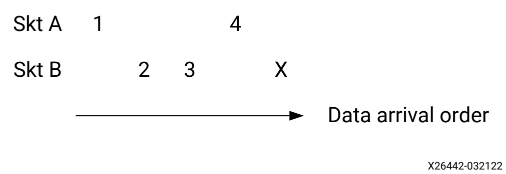
- All sockets must be in the same Onload stack. This can be confirmed using the `onload_fd_stat()` function.
- Ordering is done on a per-stack basis - for TCP and UDP sockets.
- Only data received from an Onload stack with a hardware timestamp will be ordered.
- The environment variable EF_RX_TIMESTAMPING must be enabled:

```
EF_RX_TIMESTAMPING=1
```

- File descriptors where timestamping information is not available can be included in the epoll set, but received data will be returned from these unordered.
- The application must use the epoll API and the `onload_ordered_epoll_wait()` function.
- The application must set the per-process environment variable EF_UL_EPOLL`=1 or EF_UL_EPOLL=3`.
- EPOLLET and ONESHOT flags should NOT be used.
- Concurrent use of the ordering data is not safe, and so `onload_ordered_epoll_wait()` must not be called from multiple threads.
- See onload_ordered_epoll_wait for further details.

To prevent packet coalescing in the receive queue, resulting in multiple packets received with the same hardware timestamp, the EF_TCP_RCVBUF_STRICT variable should be disabled (default setting). The following figure demonstrates the Wire Order Delivery feature.

Figure 14: **Wire Order Delivery**



`onload_ordered_epoll_wait()` returning at point X would allow the following data to be recovered:

- Socket A: timestamp of packet 1, bytes in packet 1.
- Socket B: timestamp of packet 2, bytes in packets 2 and 3.
- `onload_ordered_epoll_wait()` returning again would recover timestamp of packet 4 and bytes in packet 4.

The Wire Order Delivery feature is only available on Solarflare Flareon or XtremeScale™ adapters having a PTP/HW timestamping activation key. When receiving across multiple adapters, Solarflare sfptpd (PTP) can ensure that adapters are closely synchronized with each other and, if required, with an external PTP clock source.

Example API for Wire Order Delivery

The Onload distribution includes example client/server applications to demonstrate the wire order feature:

`wire_order_server` - uses onload_ordered_epoll_wait to receive ordered data over a set of sockets. Received data is echoed back to the client on a single reply socket.

`wire_order_client` - Sends sequenced data across the socket set, reads the reply data from the server and ensures data is received in sequence.

Building the example

Source code for the wire order API is available in:

```
onload-<version>/src/tests/onload/wire_order
```

Although not compiled as part of the Onload install process, to build the example API do the following:

Ensure mmaketool is in the current path (can be found in the `onload-<version>/scripts` directory):

```
# export PATH=$PATH:/onload-<version>/scripts
# cd /onload-<version>/build/gnu_x86_64/tests/onload/wire_order
# USEONLOADEXT=1 make
```

Running the example server

```
# EF_RX_TIMESTAMPING=3 onload ./wire_order_server
```

Running the example client

```
# onload --profile=latency ./wire_order_client <ip server>
```

By default the client will send data over 100 TCP sockets controlled with the -s option. UDP can be selected using the -U option.

***Note*:** To prevent sends being re-ordered between streams, the latency profile should be used on the client side. The environment variable EF_RX_TIMESTAMPING must be set on the server side.

Stack Sharing

By default each process using Onload has its own 'stack'. Refer to Onload Stacks for definition. Several processes can be made to share a single stack, using the EF_NAME environment variable. Processes with the same value for EF_NAME in their environment will share a stack.

Stack sharing is one supported method to enable multiple processes using Onload to be accelerated when receiving the same multicast stream or to allow one application to receive a multicast stream generated locally by a second application. Other methods to achieve this are Multicast Replication and Hardware Multicast Loopback.

Stacks can also be shared by multiple processes to preserve and control resources within the system. Stack sharing can be employed by processes handling TCP as well as UDP sockets.

***Note*:** Onload stacks cannot be shared by different network namespaces.

Stack sharing should only be requested if there is a trust relationship between the processes. If two processes share a stack then they are not completely isolated: a bug in one process can impact the other, or one process can gain access to the privileged information of the other and so breach security. Once the EF_NAME variable is set, any process on the local host can set the same value and gain access to the stack.

By default Onload stacks can only be shared with processes having the same UID. The EF_SHARE_WITH environment variable provides additional security while allowing a different UID to share a stack. Refer to Appendix A: Parameter Reference for a description of the EF_NAME and EF_SHARE_WITH variables.

Processes with different UIDs, sharing an Onload stack cannot use huge pages. Onload will issue a warning at startup and prevent the allocation of huge pages if EF_SHARE_WITH identifies a UID of another process or is set to -1. If a process P1 creates an Onload stack, but is not using huge pages and another process P2 attempts to share the Onload stack by setting EF_NAME, the stack options set by P1 will apply, allocation of huge pages in P2 will be prevented.

To suppress these startup warnings about turning huge pages off, set `EF_USE_HUGE_PAGES` to `0` if EF_SHARE_WITH is non-zero.

An alternative method of implementing stack sharing is to use the Onload Extensions API and the `onload_set_stackname()` function which, through its scope parameter, can limit stack access to the processes created by a particular user. Refer to Appendix D: Onload Extensions API for details.

Application Clustering

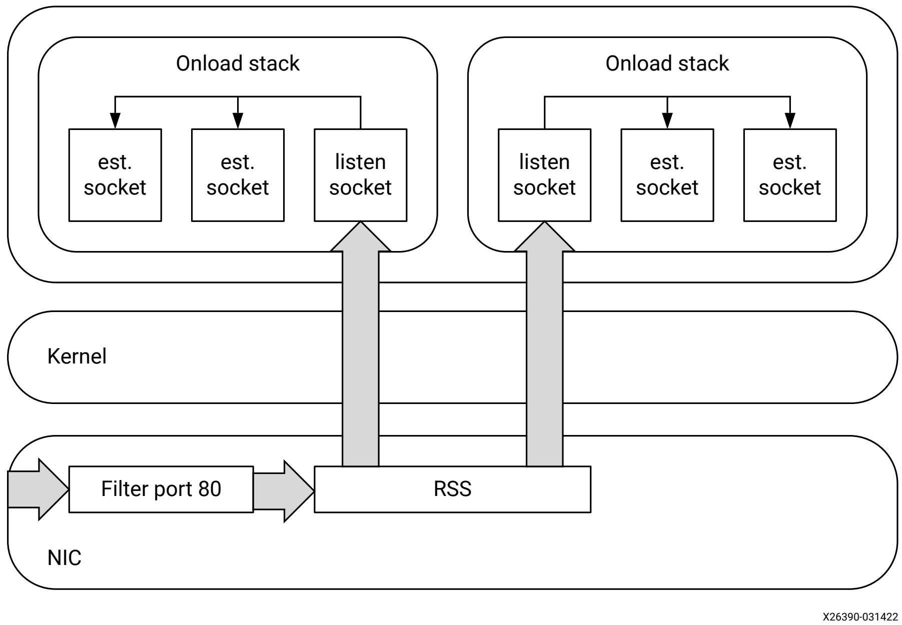
An application cluster is the set of Onload TCP or UDP stack sockets bound to the same port. This feature dramatically improves the scaling of some applications across multiple CPUs (especially those establishing many sockets from a TCP listening socket).

Onload from version 201405 automatically creates a cluster using the `SO_REUSEPORT` socket option. TCP or UDP processes running on RHEL 6.5 (and later) setting this option can bind multiple sockets to the same TCP or UDP port.

Note: Some older Linux kernel/distributions do not have kernel support for `SO_REUSEPORT` (introduced in the Linux 3.9 kernel). Onload contains experimental support for `SO_REUSEPORT` on older kernel versions but this has yet to be fully tested and verified. Users are free to try the Onload application clustering feature on these kernels and report their findings via email to support-nic@amd.com.

For TCP, clustering allows the established connections resulting from a listening socket to be spread over a number of Onload stacks. Each thread/process creates its own listening socket (using `SO_REUSEPORT`) on the same port, with each listening socket residing in its own Onload stack. Handling of incoming new TCP connections are spread via the adapter (using RSS) over the application cluster and therefore over each of the listening sockets resulting in each Onload stack and therefore each thread/process, handling a subset of the total traffic as illustrated in the following figure.

Figure 15: **Application Clustering - TCP**



For UDP, clustering allows UDP unicast traffic to be spread over multiple applications with each application receiving a subset of the total traffic load.

Existing applications that do not use `SO_RESUSEPORT` can use the application clustering feature without the need for re-compilation by using the Onload EF_TCP_FORCE_REUSEPORT or EF_UDP_FORCE_REUSEPORT environment variables identifying the list of ports to which `SO_RESUSEPORT` will be applied.

The size or number of socket members of a cluster in Onload is controlled with EF_CLUSTER_SIZE. To create a cluster the application sets the cluster name with EF_CLUSTER_NAME. A cluster of EF_CLUSTER_SIZE is then created.

Note: The number of socket members must equal the EF_CLUSTER_SIZE value otherwise a portion of the received traffic will be lost.

The spread of received traffic between cluster sockets employs Receive Side Scaling (RSS). The RSS hash is a function of the src_ip:src_port and dst_ip:dst_port 4-tuple.

The reception of traffic within a cluster is dependent on port numbers only.

Restarting an application that includes cluster socket members can fail when orphan stacks are still present. Use EF_CLUSTER_RESTART to force termination of orphaned stacks allowing the creation of the new cluster.

Refer to Chapter 13: Limitations for details of application clustering limitations.

eXpress Data Path (XDP)

From Onload-7.1.0 onwards, the implementation of XDP features is delegated to the currently-running kernel. This expands the set of supported XDP features to those provided by the kernel, but means that XDP is not supported on older OSs that do not provide the corresponding features natively.

***Note*:** On Onload-7.0.0.176, XDP was implemented using a snapshot of the Linux 4.20 implementation. Onload support for XDP allows the user to insert extended Berkeley Packet Filter code (eBPF) very early into the Onload packet receive path. This can be used, for example, to implement low overhead IPtables type filters, for packet statistics collection or for more complex packet manipulation using eBPF programming.

OS Requirements

XDP is not supported on RHEL6 or RHEL7. XDP requires (minimum) RHEL8 or the 4.20 Linux kernel.

Advantages of XDP

The advantages of XDP are as follows:

- An eBPF program is executed before received packets reach the network stack.
- An eBPF program is executed before expensive socket memory allocations for received packets.
- Dropped packets incur low CPU overhead and do not consume packet socket buffers.

Including eBPF with Onload

To use eBPF with Onload:

1. Build Onload with BPF support when using the `onload_install` script:

   ```
   # onload_install --build profile cloud
   ```

2. Enable native kernel BPF program functionality (is enabled by default):

   ```
   #define CI_CFG_WANT_BPF_NATIVE  1
   ```

   This is in: `/src/include/ci/internal/transport_config_opt.h`.

Onload Tools

If you are using the legacy implementation of XDP (instead of the kernel-provided one), see onload_bpftools dependencies in Appendix C: Build Dependencies.

```
# onload_bpftool help
Usage:
    onload_bpftool help
    onload_bpftool prog { show | list }
    onload_bpftool prog dump { xlated | jited } ATTACHMENT
    onload_bpftool prog load object FILE [ section NAME ] [ verbose ]
attach ATTACHMENT [attach ATTACHMENT]...
    onload_bpftool prog detach ATTACHMENT

    ATTACH_POINT := { xdp_ingress }
    ATTACHMENT := ATTACH_POINT [dev IFNAME] [stack STACKNAME]
```

An example eBPF program and build command lines are documented in the Onload Release Notes.

eBPF Return Codes

The following are supported by Onload:

- XDP_DROP

  Reject and silently drop the packet. Drops are not seen by any application including tcpdump and onload_tcpdump.

- XDP_PASS

  Accept and forward the packet to the network stack.

Programmatic Access

Onload includes the `libbpfintf` library exposing the feature set of the onload_bpftool. Users should refer to documentation comments in the header file for further details:

```
/src/include/onload/oobpf.h
```

BPF Statistics

Statistics identify the number of packets passed through the eBPF program, categorized by return code from the program.

```
# onload_stackdump lots | grep xdp
```

Table 20: **Onload_stackdump XDP Counters**

Counter	Description
`unlock_slow_xdp_change`	Number of attempts to release the lock and needed to handle change in XDP program
`rx_xdp_pass`	Number of packets accepted by the eBPF program
`rx_xdp_drop`	Number of packets rejected by the eBPF program
`rx_xdp_tx`	Number of RX packets rejected due to eBPF program returning TX code
`rx_xdp_redirect`	Number of RX packets rejected due to eBPF program returning REDIRECT code
`rx_xdp_aborted`	Number of RX packets rejected due to eBPF program returning ABORTED code
`rx_xdp_unknown`	Number of RX packets rejected due to eBPF program returning unknown code

***Note*:** Onload has no current support for AF_XDP sockets. This might be included in future releases.

Zero-Copy API

The Onload Extensions API includes support for zero-copy of TCP transmit packets and UDP receive packets. Refer to Zero-Copy API for detailed descriptions and example source code of the API.

Debug and Logging

Onload supports various debug and logging options. Logging and debug information will be displayed on an attached console or will be sent to the syslog. To force all debug to the syslog set the Onload environment variable EF_LOG_VIA_IOCTL`=1`.

For more information about debug/logging environment variables refer to Appendix A: Parameter Reference.

To enable debug and logging using the options below, Onload must be installed with debug enabled:

- When Onload was installed from a source tarball:

```
# onload_install --debug
```

- When Onload was installed from a source RPM:

```
# rpmbuild --define "debug true" rebuild enterpriseonload-$VERSION.rpm
```

If Onload is already installed, uninstall as described in Removing an Existing Installation. Then re-install with the --debug option as shown above.

Log options are as follows:

- EF_UNIX_LOG - A bitmask of the types of diagnostic messages to be logged.
- EF_LOG - A comma separated list options which can be logged, enabled, disabled.
- EF_LOG_FILE - When EF_LOG_VIA_IOCTL is unset, the user is able to redirect Onload output to a specified directory and file using the EF_LOG_FILE option. Timestamps can also be added to the logfile when EF_LOG_TIMESTAMPS is also enabled.

  ```
  EF_LOG_FILE=<path/file>
  ```

  Note: Kernel logging is still directed to the syslog.
- TP_LOG (bitmask) - useful for stack debugging. See Onload source code `/src/include/ci/internal/ip_log.h` for bit values.
- Control plane module option:
  - `cplane_debug_bits=[bitmask]` - useful for kernel logging and events involving the control plane. See `src/include/cplane/debug.h` for bit values.
- Onload module options:
  - `ci_tp_log=[bitmask]` - useful for kernel logging and events involving an onload stack. See Onload source code `/src/include/ci/internal/ip_log.h` for details.
  - `oo_debug_bits=[bitmask]` - useful for kernel logging and events not involving an onload stack or the control plane. See `src/include/onload/debug.h` for bit values.

Chapter 7

Timestamps

This chapter describes how to use timestamps with Onload.

Introduction

This section identifies options for using software and hardware timestamps.

Software Timestamps

Setting the `SO_TIMESTAMP` or `SO_TIMESTAMPNS` options using `setsockopt()` enables software timestamping on TCP or UDP sockets. Functions such as `cmesg()`, `recvmsg()` and `recvmmsg()` can then recover timestamps for packets received at the socket.

Onload implements a microsecond resolution software timestamping mechanism, which avoids the need for a per-packet system call thereby reducing the normal timestamp overheads.

Supported adapters will always deliver received packets to the receive ring buffer in the order that these arrive from the network. Onload will append a software timestamp to the packet meta data when it retrieves a packet from the ring buffer - before the packet is transferred to a waiting socket receive buffer.

TCP Streams

From a TCP stream the timestamp returned is that for the first available byte. Due to retransmissions and any reordering, timestamps might not be monotonically increasing as these are delivered to the application.

Interrupt Driven Applications

When a packet is received it is delivered from the adapter to the receive queue and a notification event placed on the event queue. When the Onload application is interrupt driven, a received packet is timestamped when Onload receives the corresponding event.

Spinning Applications

If the Onload application is spinning, a received packet is timestamped when the stack is polled at which point the packet is placed on the socket receive queue. Spinning will generally produce more accurate timestamps so long as the receiving application is able to keep pace with the packet arrival rate.

Software Timestamp Values

The value of the software timestamp is the start time for the poll that fetches the packet from the hardware.

Rarely, a packet might arrive during a poll, and can then be given the same timestamp as an earlier packet fetched by the same poll. If you are also using hardware timestamping for such a packet, its software timestamp might be earlier than its hardware timestamp.

Software Timestamp Formats

The format of timestamps is defined by `struct_timeval`.

Applications preferring timestamps with nanosecond resolution can use `SO_TIMESTAMPNS` in place of the normal (microsecond resolution) `SO_TIMESTAMP` value.

Hardware Timestamps

Setting the `SO_TIMESTAMPING` option using `setsockopt()` enables hardware timestamping on TCP or UDP sockets. Timestamps are generated by the adapter for each received packet. A timestamp is generated when the first byte enters the adapter. Timestamp resolution is <8ns on an X2 series adapter.

Functions such as `cmesg()`, `recvmsg()` and `recvmmsg()` can then recover hardware timestamps for packets recovered from a socket.

Requirements

- Supported only on Solarflare Flareon XtremeScale™ SFN8000 and XtremeScale™ X2 series adapters.
- An activation Key for hardware timestamps must be installed on the adapter:
  - The PTP/timestamping activation key is installed during manufacture on the PLUS variants of SFN8000 and X2 series adapters.

- An appropriate activation key can be installed on other SFN8000 and X2 series adapters by the user.

Hardware Timestamp Format

The format of timestamps is defined by `struct_timespec`. Interested users should read the kernel `SO_TIMESTAMPING` documentation for more details of how to use this socket API – kernel documentation can be found, for example, at:

https://www.kernel.org/doc/Documentation/networking/timestamping/

Received Packets

- The Onload stack for the socket must have the environment variable EF_RX_TIMESTAMPING set - see Appendix A: Parameter Reference for details.
- Received packets are timestamped when they enter the MAC on the SFN8000 or X2 series adapter.

Transmitted Packets

Onload supports hardware timestamping of UDP and TCP packets transmitted over a supported interface. A timestamp is generated when the first byte enters the adapter.

Recent Linux kernels support hardware timestamps for TCP, and Onload 8.1 adds similar capability. To recover hardware timestamps for transmitted TCP packets that are similar to the Linux kernel, set the following socket options:

```
SOF_TIMESTAMPING_TX_HARDWARE | SOF_TIMESTAMPING_SYS_HARDWARE |
SOF_TIMESTAMPING_RAW_HARDWARE
```

Because older Linux kernels do not support hardware timestamps for TCP, Onload provides an extension to the standard SO_TIMESTAMPING API with the `ONLOAD_SOF_TIMESTAMPING_STREAM` socket option to support this. To recover hardware timestamps for transmitted TCP packets that use an Onload proprietary format, set the following socket options:

```
SOF_TIMESTAMPING_TX_HARDWARE | SOF_TIMESTAMPING_SYS_HARDWARE |
SOF_TIMESTAMPING_RAW_HARDWARE | ONLOAD_SOF_TIMESTAMPING_STREAM
```

To recover hardware timestamps for transmitted UDP packets, set the following socket options:

```
SOF_TIMESTAMPING_TX_HARDWARE | SOF_TIMESTAMPING_SYS_HARDWARE |
SOF_TIMESTAMPING_RAW_HARDWARE
```

From Onload 8.1 onwards,`SOF_TIMESTAMPING_OPT_ID` and `SOF_TIMESTAMPING_OPT_TSONLY` are also supported with Linux-style timestamps.

Other socket flag combinations, not listed above, will be silently ignored.

To receive hardware transmit timestamps:

- The adapter must support hardware transmit timestamps. The following adapters currently do so:
  - XtremeScale™ SFN8000 series
  - XtremeScale™ X2 series
  - AMD Alveo™ X3 series.
- The adapter must have a PTP/HW timestamping activation key.

  Note: Alveo X3 series are an exception to this requirement, because they do not use activation keys.
- The adapter must have a SolarCapture Pro activation key or Performance Monitoring activation key.

  Note: Alveo X3 series are an exception to this requirement, because they do not use activation keys.
- You must set EF_TX_TIMESTAMPING on stacks where transmit timestamping is required.
- You must set EF_TIMESTAMPING_REPORTING to control the type of timestamp returned to the application. This is optional, by default Onload will report translated timestamps if the adapter clock has been fully synchronized to correct time by the Solarflare PTP daemon. In all cases Onload will always report raw timestamps. Refer to Appendix A: Parameter Reference for full details of the EF_TIMESTAMPING_REPORTING variable.
- Solarflare PTP (sfptpd) must be running if timestamps are to be synchronized with an external PTP master clock.

For details of the SO_TIMESTAMPING API refer to the Linux documentation:

https://www.kernel.org/doc/Documentation/networking/timestamping/

Zeroed Timestamps

If timestamps returned from the adapter are zeroed, refer to Setting the Adapter Clock Time.

Synchronizing Time

Solarflare Enhanced PTP can be enabled to synchronize the time across all clocks within a server or between multiple servers.

The sfptpd daemon supports clock synchronization with external NTP and PTP time sources and includes an optional PTP/NTP fallback configuration.

For details of Solarflare PTP refer to the *Enhanced PTP User Guide* (UG1602).

Example Timestamping Applications

The onload distribution includes example applications to demonstrate receive and transmit hardware timestamping. With Onload installed, source code is located in the following subdirectory:

```
/onload-<version>/src/tests/onload/hwtimestamping
```

Building the Examples

Following the onload_install, the example applications: rx_timestamping and tx_timestamping are located in the following directory:

```
onload-<version>/build/gnu_x86_64/tests/onload/hwtimestamping
```

Using earlier versions of Onload the user should run the make command in the following directory to build example timestamping applications:

```
openonload-<version>/src/tests/onload/hwtimestamping
```

Running the Examples

The following conditions are required to run the example applications:

- The server must have a Solarflare SFN8000 or X2 series adapter.
- The adapter must have a PTP/HW timestamping activation key.
- The connection from which packets are to be timestamped must be routed over the timestamping adapter.
- To receive TX timestamps, the adapter must have a SolarCapture Pro activation key or Performance Monitoring activation key
- The Onload environment variable EF_RX_TIMESTAMPING or EF_TX_TIMESTAMPING must be enabled in the Onload environment.

***Note*:** User should also read the specific requirements from the RX/TX timestamping sections above.

Setting the Adapter Clock Time

It might be necessary to 'seed' the adapter clock time - otherwise timestamps might be zeroed or reported as 01 Jan 1970. This can be done by briefly running Solarflare PTP (sfptpd) as a slave - the adapter clock is seeded from the system clock.

Running sfptpd in freerun mode will achieve the same result. It is not required to actually receive any PTP packets to seed the adapter clock and sfptpd can be terminated after a few seconds as it is only required to 'seed' the adapter clock.

Users who wish to synchronize the adapter clock with an external time source should refer to the *Enhanced PTP User Guide* (UG1602).

Order of Timestamps in the Example Applications

Timestamps in the example applications are displayed in the following order:

- System: software timestamp from the system clock.
- Transformed: hardware timestamp converted to software timestamp. This can be ignored because the adapter is using UTC time and transformation is not required. Transformed timestamps are identical to Raw timestamps.
- Raw: hardware timestamp generated by the adapter clock.

rx_timestamping Example

This demonstrates the `rx_timestamping` example application.

Server1

Server1 sets the EF_RX_TIMESTAMPING environment variable and starts the `rx_timestamping` application:

```
# EF_RX_TIMESTAMPING=2 onload ./rx_timestamping --proto tcp
oo:rx_timestamping[31250]: Using OpenOnload 201509 Copyright 2006-2015
Solarflare Communications, 2002-2005 Level 5 Networks [0]
Socket created, listening on port 9000
Socket accepted
Selecting hardware timestamping mode.
Packet 1 - 27 bytes     timestamps 1460374944.990960465
1460374944.993421129 1460374944.993421129
Packet 2 - 27 bytes     timestamps 1460374966.478980336
1460374966.481623531 1460374966.481623531
Packet 3 - 0 bytes      no timestamp
recvmsg returned 0 - end of stream
```

Server2

Server2 uses the Linux `netcat` utility to send packets to server1:

```
# nc <server1 ip> 9000
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
```

tx_timestamping Example

This demonstrates the `tx_timestamping` example application.

From Onload 8.1 onwards this example defaults to using Linux format timestamps for TCP. Use the `--stream` option to instead use the Onload proprietary format.

Server1

Server1 sets the EF_TX_TIMESTAMPING environment variable and starts the `tx_timestamping` application:

```
# EF_TX_TIMESTAMPING=3 onload ./tx_timestamping --proto tcp --ioctl eth4
oo:tx_timestamping[16139]: Using OpenOnload 201509 Copyright 2006-2015
Solarflare Communications, 2002-2005 Level 5 Networks [4]
TCP listening on port 9000
 TCP connection accepted
Accepted SIOCHWTSTAMP ioctl.
Selecting hardware timestamping mode.
Packet 1 - 27 bytes
Timestamp for 27 bytes:
First sent timestamp 1453436034.615029223
Last sent timestamp 0.000000000
```

Server2

Server2 uses the Linux `netcat` utility to send a packet to server1 which is then echoed back to the sender:

```
# nc <server1 ip> 9000
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz  (echoed back from server 1)
```

Example UDP Commands

This section gives an example of how the preceding commands can be modified to use UDP.:

Server1

```
# EF_RX_TIMESTAMPING=2 onload ./rx_timestamping --proto udp --port 9000
```

Server2

```
# nc -u <server1_ipaddr> 9000
```

Zeroed Timestamps

If timestamps returned from the example applications are zeroed, refer to Setting the Adapter Clock Time.

Chapter 8

Onload and TCP

This chapter gives information about using TCP with Onload.

TCP Operation

The table below identifies the Onload TCP implementation RFC compliance.

Table 21: **Onload TCP Implementation RFC Compliance**

RFC	Title	Compliance
793	Transmission Control Protocol	Yes
813	Window and Acknowledgement Strategy in TCP	Yes
896	Congestion Control in IP/TCP	Yes
1122	Requirements for Hosts	Yes
1191	Path MTU Discovery	Yes
1323	TCP Extensions for High Performance	Yes
2018	TCP Selective Acknowledgment Options	Yes
2581	TCP Congestion Control	Yes
2582	The NewReno Modification to TCP's Fast Recovery Algorithm	Yes
2883	An Extension to the Selective Acknowledgment (SACK) Option for TCP	Yes
2988	Computing TCP's Retransmission Timer	Yes
3128	Protection Against a Variant of the Tiny Fragment Attack	Yes
3168	The Addition of Explicit Congestion Notification (ECN) to IP	Yes
3465	TCP Congestion Control with Appropriate Byte Counting (ABC)	Yes

TCP Handshake, SYN and SYNACK

During the TCP connection establishment three-way handshake, Onload negotiates the MSS, Window Scale, SACK permitted, ECN, PAWS and RTTM timestamps.

For TCP connections Onload will negotiate an appropriate MSS for the MTU configured on the interface. However, when using jumbo frames, Onload will currently negotiate an MSS value up to a maximum of 2048 bytes minus the number of bytes required for packet headers. This is due to the fact that the size of buffers passed to the Solarflare network interface card is 2048 bytes and the Onload stack cannot currently handle fragmented packets on its TCP receive path.

TCP options advertised during the handshake can be selected using the EF_TCP_SYN_OPTS environment variable. Refer to Appendix A: Parameter Reference for details of environment variables.

TCP SYN Cookies

The Onload environment variable EF_TCP_SYNCOOKIES can be enabled on a per stack basis to force the use of SYNCOOKIES thereby providing a degree of protection against the Denial of Service (DOS) SYN flood attack. EF_TCP_SYNCOOKIES is disabled by default. Refer to Appendix A: Parameter Reference for details of environment variables.

TCP Socket Options

Onload TCP supports the following socket options which can be used in the `setsockopt()` and `getsockopt()` function calls.

Table 22: **Socket Options for setsockopt() and getsockopt()**

Option	Description
`SO_PROTOCOL`	Retrieve the socket protocol as an integer.
`SO_ACCEPTCONN`	Determines whether the socket can accept incoming connections - true for listening sockets. (Only valid as a `getsockopt()`).
`SO_BINDTODEVICE`	Bind this socket to a particular network interface. See SO_BINDTODEVICE.
`SO_CONNECT_TIME`	Number of seconds a connection has been established. (Only valid as a `getsockopt()`).
`SO_DEBUG`	Enable protocol debugging.

Table 22: **Socket Options for setsockopt() and getsockopt()** *(cont'd)*

Option	Description
`SO_ERROR`	The errno value of the last error occurring on the socket. (Only valid as a `getsockopt()`).
`SO_EXCLUSIVEADDRUSE`	Prevents other sockets using the SO_REUSEADDR option to bind to the same address and port.
`SO_KEEPALIVE`	Enable sending of keep-alive messages on connection oriented sockets.
`SO_LINGER`	When enabled, a `close()` or `shutdown()` will not return until all queued messages for the socket have been successfully sent or the linger timeout has been reached. Otherwise the `close()` or `shutdown()` returns immediately and sockets are closed in the background.
`SO_OOBINLINE`	Indicates that out-of-band data should be returned in-line with regular data. This option is only valid for connection-oriented protocols that support out-of-band data.
`SO_PRIORITY`	Set the priority for all packets sent on this socket. Packets with a higher priority might be processed first depending on the selected device queuing discipline.
`SO_RCVBUF`	Sets or gets the maximum socket receive buffer in bytes. EF_TCP_RCVBUF overrides this value, and EF_TCP_RCVBUF_ESTABLISHED_DEFAULT can also override this value. Setting SO_RCVBUF to a value < MTU can result in poorer performance and is not recommended.
`SO_RCVLOWAT`	Sets the minimum number of bytes to process for socket input operations.
`SO_RCVTIMEO`	Sets the timeout for input function to complete.
`SO_RECVTIMEO`	Sets the timeout in milliseconds for blocking receive calls.
`SO_REUSEADDR`	Can reuse local port numbers (another socket can bind to the same port) except when there is an active listening socket bound to the port.
`SO_RESUSEPORT`	Allows multiple sockets to bind to the same port.
`SO_SNDBUF`	Sets or gets the maximum socket send buffer in bytes. The value set is doubled by the kernel and by Onload to allow for bookkeeping overhead when it is set by the `setsockopt()` function call. This value can be overridden by EF_TCP_SNDBUF, EF_TCP_SNDBUF_MODE and EF_TCP_SNDBUF_ESTABLISHED_DEFAULT. When the EF_TCP_SNDBUF_MODE is set to 2, the SNDBUF size is automatically adjusted for each TCP socket to match the window advertised by the peer.
`SO_SNDLOWAT`	Sets the minimum number of bytes to process for socket output operations. Always set to 1 byte.
`SO_SNDTIMEO`	Set the timeout for sending function to send before reporting an error.
`SO_TIMESTAMP`	Report timestamps from system clock in struct timeval.
`SO_TIMESTAMPNS`	Report timestamps from system clock in struct timespec.
`SO_TIMESTAMPING`	Enable/disable hardware timestamps for received packets.
`SOF_TIMESTAMPING_TX_HARDWARE`	Obtain a hardware generated transmit timestamp.

Table 22: **Socket Options for setsockopt() and getsockopt()** *(cont'd)*

Option	Description
`SOF_TIMESTAMPING_SYS_HARDWARE`	Obtain a hardware transmit timestamp adjusted to the system time base.
`SOF_TIMESTAMPING_OPT_CMSG`	Deliver timestamps using the `cmsg` API.
`ONLOAD_SOF_TIMESTAMPING_STREAM`	Onload extension to the standard `SO_TIMESTAMPING` API to support hardware timestamps on TCP sockets.
`SO_TYPE`	Returns the socket type (`SOCK_STREAM` or `SOCK_DGRAM`). (Only valid as a `getsockopt()`).
`IP_TRANSPARENT`	This socket option allows the calling application to bind the socket to a nonlocal IP address.

TCP Level Options

Onload TCP supports the following TCP options which can be used in the `setsockopt()` and `getsockopt()` function calls

Table 23: **TCP Options for setsockopt() and getsockopt()**

Option	Description
`TCP_CORK`	Stops sends on segments less than MSS size until the connection is uncorked.
`TCP_DEFER_ACCEPT`	A connection is ESTABLISHED after handshake is complete instead of leaving it in SYN-RECV until the first real data packet arrives. The connection is placed in the accept queue when the first data packet arrives.
`TCP_INFO`	Populates an internal data structure with tcp statistic values.
`TCP_KEEPALIVE_ABORT_THRESHOLD`	How long to try to produce a successful keepalive before giving up.
`TCP_KEEPALIVE_THRESHOLD`	Specifies the idle time for keepalive timers.
`TCP_KEEPCNT`	Number of keepalives before giving up.
`TCP_KEEPIDLE`	Idle time for keepalives.
`TCP_KEEPINTVL`	Time between keepalives.
`TCP_MAXSEG`	Gets the MSS size for this connection.
`TCP_NODELAY`	Disables Nagle's Algorithm and small segments are sent without delay and without waiting for previous segments to be acknowledged.
`TCP_QUICKACK`	When enabled ACK messages are sent immediately following reception of the next data packet. This flag will be reset to zero following every use. It is a one time option. New connections start in a mode where all packets are acknowledged, and so this value initially defaults to 1.

TCP File Descriptor Control

Onload supports the following options in `socket()` and `accept()` calls.

Table 24: **Options for socket() and accept()**

Option	Description
`SOCK_NONBLOCK`	Supported in `socket()` and `accept()`. Sets the `O_NONBLOCK` file status flag on the new open file descriptor saving extra calls to `fcntl(2)` to achieve the same result.
`SOCK_CLOEXEC`	Supported in `accept()`. Sets the close-on-exec (`FD_CLOEXEC`) flag on the new file descriptor.

TCP Congestion Control

Onload TCP implements congestion control in accordance with RFC3465 and employs the NewReno algorithm with extensions for Appropriate Byte Counting (ABC).

On new or idle connections and those experiencing loss, Onload employs a Fast Start algorithm in which delayed acknowledgments are disabled, thereby creating more ACKs and subsequently 'growing' the congestion window rapidly. Two environment variables; EF_TCP_FASTSTART_INIT and EF_TCP_FASTSTART_LOSS are associated with the fast start - Refer to Appendix A: Parameter Reference for details.

During Slow Start, the congestion window is initially set to 2 × maximum segment size (MSS) value. As each ACK is received the congestion window size is increased by the number of bytes acknowledged up to a maximum 2 × MSS bytes. This allows Onload to transmit the minimum of the congestion window and advertised window size:

```
transmission window (bytes) = min(CWND, receiver advertised window size)
```

If loss is detected - either by retransmission timeout (RTO), or the reception of duplicate ACKs, Onload will adopt a congestion avoidance algorithm to slow the transmission rate. In congestion avoidance the transmission window is halved from its current size - but will not be less than 2 × MSS. If congestion avoidance was triggered by an RTO timeout the Slow Start algorithm is again used to restore the transmit rate. If triggered by duplicate ACKs Onload employs a Fast Retransmit and Fast Recovery algorithm.

If Onload TCP receives three duplicate ACKs this indicates that a segment has been lost - rather than just received out of order and causes the immediate retransmission of the lost segment (Fast Retransmit). The continued reception of duplicate ACKs is an indication that traffic still flows within the network and Onload will follow Fast Retransmit with Fast Recovery.

During Fast Recovery Onload again resorts to the congestion avoidance (without Slow Start) algorithm with the congestion window size being halved from its present value.

Onload supports a number of environment variables that influence the behavior of the congestion window and recovery algorithms identified below. Refer to Appendix A: Parameter Reference:

- EF_TCP_INITIAL_CWND - sets the initial size (bytes) of congestion window
- EF_TCP_LOSS_MIN_CWND - sets the minimum size of the congestion window following loss.
- EF_CONG_AVOID_SCALE_BACK - slows down the rate at which the TCP congestion window is opened to help reduce loss in environments already suffering congestion and loss.

CAUTION! *The congestion variables should be used with caution to avoid violating TCP protocol requirements and degrading TCP performance.*

Small Receive Window Size

Onload, by default, does not behave like the kernel if the remote receiving end of a connection advertises a receive window too small for the data Onload needs to send. The kernel stack will split waiting data to fill the available window space. Onload would probe the remote end with up to three ACKs to prompt the remote end to increase the advertised receive window size before sending the entire packet.

The following compile time option can be enabled to force Onload to behave as the kernel does:

```
#define CI_CFG_SPLIT_SEND_PACKETS_FOR_SMALL_RECEIVE_WINDOWS
```

The flag, in the `include/ci/internal/transport_config_opt.h` file, is disabled by default, Set to 1 to enable.

TCP SACK

Onload will employ TCP Selective Acknowledgment (SACK) if the option has been negotiated and agreed by both ends of a connection during the connection establishment three-way handshake. Refer to RFC 2018 for further information.

TCP QUICKACK

TCP will generally aim to defer the sending of ACKs to minimize the number of packets on the network. Onload supports the standard `TCP_QUICKACK` socket option which allows some control over this behavior. Enabling `TCP_QUICKACK` causes an ACK to be sent immediately in response to the reception of the following data packet. This is a one-shot operation and `TCP_QUICKACK` self clears to zero immediately after the ACK is sent.

TCP Delayed ACK

By default TCP stacks delay sending acknowledgments (ACKs) to improve efficiency and utilization of a network link. Delayed ACKs also improve receive latency by ensuring that ACKs are not sent on the critical path. However, if the sender of TCP packets is using Nagle's algorithm, receive latency will be impaired by using delayed ACKs.

Using the EF_DELACK_THRESH environment variable the user can specify how many TCP segments can be received before Onload will respond with a TCP ACK. Refer to the Parameter List for details of the Onload environment delayed TCP ACK variables.

TCP Dynamic ACK

The sending of excessive TCP ACKs can impair performance and increase receive side latency. Although TCP generally aims to defer the sending of ACKs, Onload also supports a further mechanism. The EF_DYNAMIC_ACK_THRESH environment variable allows Onload to dynamically determine when it is non-detrimental to throughput and efficiency to send a TCP ACK. Onload will force an TCP ACK to be sent if the number of TCP ACKs pending reaches the threshold value.

Refer to the Parameter List for details of the Onload environment delayed TCP ACK variables.

***Note*:** When used together with EF_DELACK_THRESH or EF_DYNAMIC_ACK_THRESH, the socket option `TCP_QUICKACK` will behave exactly as stated above. Both onload environment variables identify the maximum number of segments that can be received before an ACK is returned. Sending an ACK before the specified maximum is reached is allowed.

***Note*:** TCP ACKS should be transmitted at a sufficient rate to ensure the remote end does not drop the TCP connection.

Limit Duplicate ACK Rate

The environment variable, EF_INVALID_ACK_RATELIMIT enables Onload support for the Linux `tcp_invalid_ratelimit` where the aim is to reduce the number of duplicate ACKs in response packets on an existing connection, but invalid for any of the following reasons:

- out-of-window ACK number
- out-of-window sequence number
- ACK for PAWS check failure.

The rate limit, applied per-socket, is the minimal time gap between sending dupacks.

The default rate is that from `/proc/sys/net/ipv4/tcp_invalid_ratelimit`.

Limit Challenge ACK Rate

The per-stack environment variable, EF_CHALLENGE_ACK_RATELIMIT enables Onload support for the Linux `tcp_challenge_ack_limit` where the aim is to limit the number of Challenge ACKs sent per second when mitigating against a TCP blind window attack.

The default rate is that from `/proc/sys/net/ipv4/tcp_challenge_ack_limit`.

TCP Loopback Acceleration

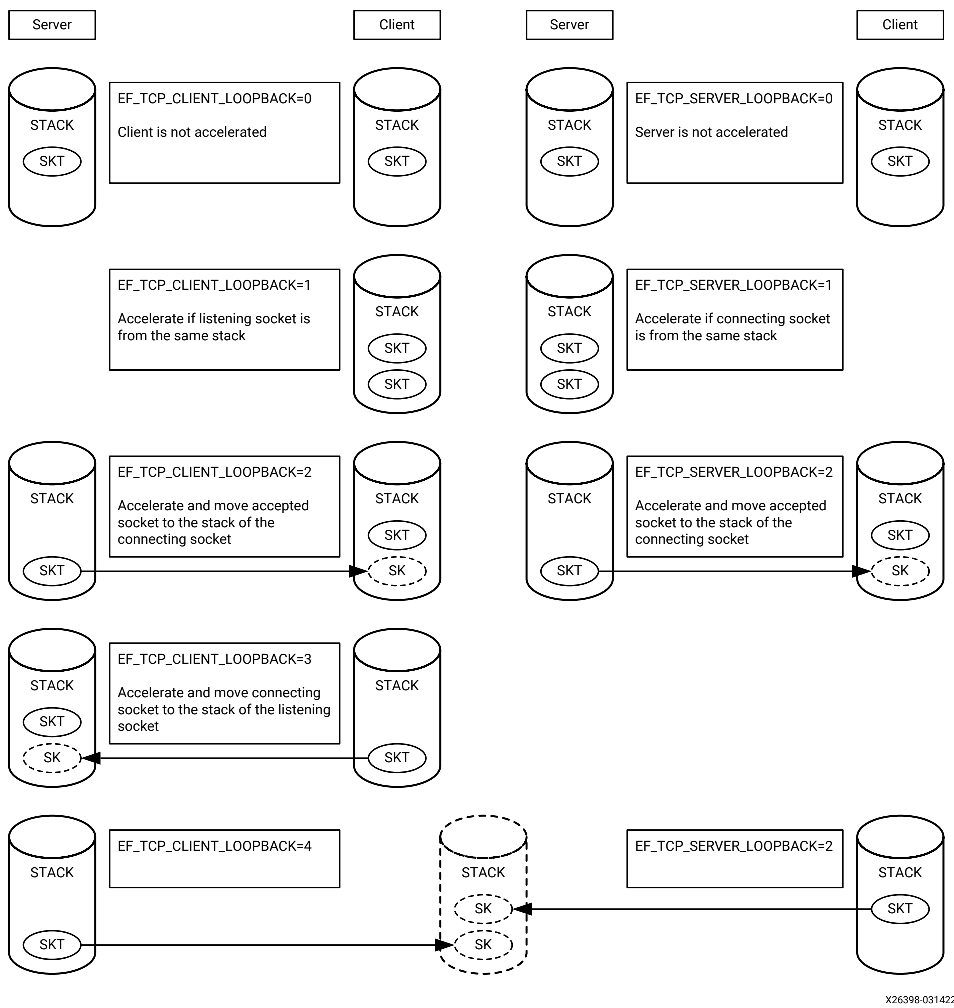
Onload supports the acceleration of TCP loopback connections, providing an accelerated mechanism through which two processes on the same host can communicate. Accelerated TCP loopback connections do not invoke system calls, reduce the overheads for read/write operations and offer improved latency over the kernel implementation.

The server and client processes who want to communicate using an accelerated TCP loopback connection do not need to be configured to share an Onload stack. However, the server and client TCP loopback sockets can only be accelerated if they are in the same Onload stack. Onload has the ability to move a TCP loopback socket between Onload stacks to achieve this.

TCP loopback acceleration is configured via the environment variables EF_TCP_CLIENT_LOOPBACK and EF_TCP_SERVER_LOOPBACK. As well as enabling TCP loopback acceleration these environment variables control Onload's behavior when the server and client sockets do not originate in the same Onload stack. This gives the user greater flexibility and control when establishing loopback on TCP sockets either from the listening (server) socket or from the connecting (client) socket. The connecting socket can use any local address or specify the loopback address.

The following diagram illustrates the client and server loopback options. Refer to Appendix A: Parameter Reference for a description of the loopback variables.

Figure 16: **EF_TCP_CLIENT/SERVER_LOOPBACK**



The client loopback option EF_TCP_CLIENT_LOOPBACK=4, when used with the server loopback option EF_TCP_SERVER_LOOPBACK=2, differs from other loopback options such that rather than move sockets between existing stacks they will create an additional stack and move sockets from both ends of the TCP connection into this new stack. This avoids the possibility of having many loopback sockets sharing and contending for the resources of a single stack.

When client and server are not the same UID, set the environment variable EF_SHARE_WITH to allow both processes to share the created shared stack.

TCP Striping

Onload supports a proprietary TCP striping mechanism that allows a single TCP connection to use both physical ports of a network adapter. Using the combined bandwidth of both ports means increased throughput for TCP streaming applications. TCP striping can be particularly beneficial for Message Passing Interface (MPI) applications.

If the TCP connection's source IP address and destination IP address are on the same subnet as defined by EF_STRIPE_NETMASK then Onload will attempt to negotiate TCP striping for the connection. Onload TCP striping must be configured at both ends of the link.

TCP striping allows a single TCP connection to use the full bandwidth of both physical ports on the same adapter. This should not be confused with link aggregation/port bonding in which any one TCP connection within the bond can only use a single physical port and therefore more than one TCP connection would be required to realize the full bandwidth of two physical ports.

Note: TCP striping is disabled by default. To enable this feature set the parameter CI_CFG_PORT_STRIPING=1 in the onload distribution source directory `src/include/internal/tranport_config_opt.h` file.

TCP Connection Reset on RTO

Under certain circumstances it can be preferable to avoid re-sending TCP data to a peer service when data delivery has been delayed. Once data has been sent, and for which no acknowledgment has been received, the TCP retransmission timeout period represents a considerable delay. When the retransmission timeout (RTO) eventually expires it can be preferable not to retransmit the original data.

Onload can be configured to reset a TCP connection rather than attempt to retransmit data for which no acknowledgment has be received.

This feature is enabled with the EF_TCP_RST_DELAYED_CONN per stack environment variable and applies to all TCP connections in the onload stack. On any TCP connection in the onload stack, if the RTO timer expires before an ACK is received the TCP connection will be reset.

ONLOAD_MSG_WARM

Applications that send data infrequently may see increased send latency compared to an application that is making frequent sends. This is due to the send path and associated data structures not being cache and TLB resident (which can occur even if the CPU has been otherwise idle since the previous send call).

Onload allows applications to repeatedly call send() to keep the TCP fast send path 'warm' in the cache without actually sending data. This is particularly useful for applications that only send infrequently and helps maintain low latency performance for those TCP connections that do not send often. These "fake" sends are performed by setting the `ONLOAD_MSG_WARM` flag when calling the TCP send calls. The message warm feature does not transmit any packets.

```
char buf[10];
send(fd, buf, 10, ONLOAD_MSG_WARM);
```

Onload stackdump supports counters to indicate the level of message warm use:

- `warm_aborted` is a count of the number of times a message warm send function was called, but the sendpath was not exercised due to Onload locking constraints.
- `warm` is a count of the number of times a message warm send function was called when the send path was exercised.
- A `send()` ONLOAD_MSG_WARM can return 0 (length of data sent) if the `send()` was unsuccessful. This will be due to normal stack and TCP networking conditions such as cannot get stack lock, insufficient send window available, other packets in the send queue or retransmit queue etc.

CAUTION! *Onload applications should not invoke send(MSG_WARM) and send(normal) from different threads on the same socket. This is not a supported feature.*

CAUTION! *The ONLOAD_MSG_WARM flag is an Onload feature. It can be applied to sockets created by Onload. However if sockets are subsequently handed off to the kernel - so they are not accelerated by Onload, it can cause the message warm packets to be actually sent. This is due to a limitation in some Linux distributions which appear to ignore this flag. The Onload extensions API can be used to check whether a socket supports the MSG_WARM feature via the `onload_fd_check_feature()` API (onload_fd_check_feature).*

***Note*:** When using the MSG_WARM feature, Onload does not attempt to split large packets into multiple segments and for this reason, the size of data passed to Onload when using the MSG_WARM feature must not exceed the MSS value.

***Note*:** Onload versions earlier than 201310 do not support the `ONLOAD_MSG_WARM` socket flag, therefore setting the flag will cause message warm packets to be sent.

Listen/Accept Sockets

TCP sockets accepted from a listening socket will share a wildcard filter with the parent socket. The following Onload module options can be used to control behavior when the parent socket is closed.

`oof_shared_keep_thresh` - default 100, is the number of accepted sockets sharing a wildcard filter that will cause the filter to persist after the listening socket has closed.

`oof_shared_steal_thresh` - default 200, is the number of sockets sharing a wildcard filter that will cause the filter to persist even when a new listening socket needs the filter.

If the listening socket is closed the behavior depends on the number of remaining accepted sockets as follows:

Table 25: **Behavior If the Listening Socket is Closed**

Number of Accepted Sockets	Onload Action
> `oof_shared_keep_thresh` but < `oof_shared_steal_thresh`	Retain the wildcard filter shared by all accepted sockets. If a new listening socket requires the filter, Onload will install a full-match filter for each accepted socket allowing the listening socket to use the wildcard filter.
> `oof_shared_steal_thresh`	Retain the wildcard filter shared by all accepted sockets. A new listening socket can be created but a filter cannot be installed meaning the socket will receive no traffic until the number of accepted connections is reduced.

Socket Caching

Socket caching means Onload can further reduce the overhead of setting up new TCP connections by reusing existing sockets instead of creating from new.

A cached socket retains a file descriptor and socket buffer when it is returned to the cache of the Onload stack from which it originated.

Socket caching is enabled when EF_SOCKET_CACHE_MAX is set to a value greater than zero. Onload will apply passive or active caching as appropriate for the type of sockets created by the user application.

EF_SOCKET_CACHE_MAX applies to both active and passive sockets, so if set to 100 the cache limit is 100 of each socket type.

TCP Passive Socket Caching

Passive socket caching, supported from the Onload 201502 release, means Onload will re-use socket buffers and file descriptors from passive-open (listening sockets).

This can improve the accept rate of active-open TCP connections and will benefit processes which need to accept lots of connections from these listening sockets.

TCP Active Socket Caching

Active socket caching, supported from the Onload 201509 release, means Onload will re-use socket buffers and file descriptors from active-open sockets when an established TCP connection has terminated.

Active-open sockets setting the IP_TRANSPARENT socket option can be cached.

From Onload 201805, socket caching can be enabled for active-open sockets but disabled for passive-open sockets. To do so, set EF_PER_SOCKET_CACHE_MAX to `0`.

Caching for Web Proxies

Applications such as web proxies can create and close large numbers of sockets. In Onload 201805, socket caching has been extended to improve the support for such applications:

- Applications using many listening sockets with scalable filters can now use a common cache of sockets accepted from them, improving utilization of the cache.
- When a listening socket is used simultaneously by multiple processes, file descriptors can now be cached per-process. In earlier versions of Onload, accepted sockets were cachable only in the process that originated them.

  This is of particular benefit to server applications such as NGINX that support dynamic reconfiguration by spawning a new process reusing existing listening sockets.

  This feature is not compatible with sockets that require O_CLOEXEC.

Caching Stackdump

Onload stackdump can be used to monitor caching activity on Onload stacks.

```
# onload_stackdump lots [| grep cache]
```

Table 26: **Onload_stackdump Cache Counters**

Counter	Description
`active cache: hit=0 avail=0 cache=EMPTY pending=EMPTY`	TCP socket caching: hit = number of cache hits (were cached) avail = number of sockets available for caching current cache state
`sockcache_cached`	Number of sockets cached over the lifetime of the stack
`sockcache_contention`	Number of sockets not cached due to lock contention
`passive_sockcache_stacklim`	Number of passive sockets not cached due to stack limit
`active_sockcache_stacklim`	Number of active sockets not cached due to stack limit
`sockcache_socklim`	Number of sockets not cached due to socket limit
`sockcache_hit`	Number of socket cache hits (were cached)
`sockcache_hit_reap`	Number of socket cache hits (were cached) after reaping
`sockcache_miss_intmismatch`	Number of socket cache misses due to mismatched interfaces
`activecache_cached`	Number of active sockets cached over the lifetime of the stack.
`activecache_stacklim`	Number of active sockets not cached due to stack limit
`activecache_hit`	Number of active socket cache hits (were cached)
`activecache_hit_reap`	Number of active socket cache hits (were cached) after reaping

Caching Requirements

There are some necessary pre-requisites when using socket caching:

- Set EF_UL_EPOLL=3 and set EF_FDS_MT_SAFE=1
- Socket caching is not supported after `fork()`
- Sockets that have been `dup()`ed will not be cached
- Sockets that use the `O_ASYNC` or `O_APPEND` modes will not be cached
- Caching offers no benefit if a single socket accepts connections on multiple local addresses (applicable to passive caching only).
- Set `O_NONBLOCK` or `O_CLOEXEC` if required on the socket, when creating the socket.

  From Onload 201805 onwards, `O_CLOEXEC` cannot be used when a listening socket is used simultaneously by multiple processes.

When socket caching cannot be enabled, sockets will be processed as normal Onload sockets.

Users should refer to details of the following environment variables:

- EF_SOCKET_CACHE_MAX
- EF_PER_SOCKET_CACHE_MAX

- EF_SOCKET_CACHE_PORTS

***Note*:** Allowing more sockets to be cached than there are file descriptors available can result in drastically reduced performance and users should consider that the socket cache limit, EF_SOCKET_CACHE_MAX, applies per stack, unlike the per-process EF_SOCKET_CACHE_PORTS limits.

Refer to Appendix A: Parameter Reference for details of Onload environment variables.

Shared Local Ports

The shared local ports feature improves the performance of TCP active-opens. It:

- reduces the cost of both blocking and non-blocking connect() calls
- reduces the latency to establish new connections
- enables scaling to large numbers of active-open connections
- reduces the cost of closing these connections.

These improvements are achieved by sharing a set of local port numbers amongst active-open sockets, which saves the cost and scaling limits associated with installing packet steering filters for each active-open socket. Shared local ports are only used when the local port is not explicitly assigned by the application.

To enable shared local ports, set the EF_TCP_SHARED_LOCAL_PORTS option to ≥1. The value set gives the initial number of local ports to allocate when the Onload stack is created. More shared local ports are allocated on demand as needed up to the maximum given by EF_TCP_SHARED_LOCAL_PORTS_MAX.

Additional configuration options were added in Onload 201805:

- When EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK is set, connecting TCP sockets will use ports only from the TCP shared local port pool (unless explicitly bound).

  If all shared local ports are in use, the `connect()` call will fail.
- When EF_TCP_SHARED_LOCAL_PORTS_PER_IP is set, ports reserved for the pool of shared local ports will be reserved per local IP address on demand.

  This helps avoid exhaustion of the ephemeral port range.
- When EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST is set, shared local ports can be reused immediately when the previous socket using that port has reached the CLOSED state, even if it did so via LAST-ACK.

  This allows the pool of shared local ports to be recycled more rapidly.

Further configuration options were added in Onload 201811:

- EF_TCP_SHARED_LOCAL_PORTS_PER_IP_MAX sets the maximum size of the pool of local shared ports for a given local IP address.

  When used with scalable RSS mode this setting limits the total number within the cluster.

- EF_TCP_SHARED_LOCAL_PORTS_STEP controls the number of ports allocated when expanding the pool of shared local ports.

  This can be used to fine tune the responsiveness of the pool.

Scalable Filters

Using scalable filters, an Onload stack can install a MAC filter to receive all traffic from a specified interface.

Note: Once the MAC filter is inserted on an interface, ARP, ICMP and IGMP traffic is directed to the kernel, but all other traffic is directed to a single Onload stack.

Using scalable filters removes limitations on:

- the number of listening sockets in scalable filters passive mode
- the number of active-open connections in scalable filters transparent-active mode. This works only for sockets having the IP_TRANSPARENT option set. See Transparent Reverse Proxy Modes below.

On Onload 201805 and later, scalable filters can be combined for both passive and active open connections and with RSS, enabling very high transaction rates for use cases such as a reverse web proxy.

Note: This feature requires modern CPUs that support the `CLMUL` instruction.

The most effective way to use scalable filters is with a dedicated VI created with a MACVLAN. This allows the kernel stack or another application using scalable filters to use the same physical port. The kernel option `inject_kernel_gid` (introduced in Onload 201805) controls the injection of packets not handled by Onload back to the kernel when the VI is instead shared with other functions

Solarflare adapters can be partitioned to expose up to 16 PCIe physical functions (PF). Each PF is presented to the OS as a standard network interface. The adapter is partitioned with the sfboot utility - see example below.

Once a MAC filter has been installed on a PF, other Onload stacks can still receive other traffic on the same PF, but sockets will have to insert IP filters for the required traffic. Apart from ARP, ICMP and IGMP packets, OS kernel sockets, using the same PF, will not receive any traffic.

Per interface, the MAC filter can only be installed by a single Onload stack. If a process creates multiple stacks, the EF_SCALABLE_FILTERS_ENABLE per-stack variable can be used to enable/disable this feature for individual stacks using the existing Onload extensions API. For example:

```
onload_stack_opt_set_int("EF_SCALABLE_FILTERS_ENABLE", 1);
```

The MAC filter is inserted when the stack is created. This is before sockets are created, and sockets need to be created to receive any traffic destined for this stack.

Scalable Filter Restrictions

Scalable filters have the following restrictions:

- Scalable filters are only used for TCP traffic.
- UDP traffic can be received and accelerated by Onload on interfaces where scalable filters are enabled, but kernel UDP sockets will not receive traffic.
- UDP fragmented frames cannot be received on interfaces where scalable filters are enabled. Users should avoid having fragmented frames on these interfaces.
- The adapter must use the full-feature or ultra-low-latency firmware variants.
- Minimum firmware version: 4.6.5.1000.
- Stack per thread options (EF_STACK_PER_THREAD) cannot be used with this feature.
- By default the scalable filters feature requires CAP_NET_RAW. Onload can be configured to avoid capability checks for this using the Onload module option `scalable_filter_gid`. See Module Options for details.
- When using any `rss` mode with scalable filters, the stack cannot be named by either EF_NAME or `onload_set_stackname()`.

Configuring Scalable Filters

To enable scalable filters on a specific interface:

```
EF_SCALABLE_FILTERS=enps0f0
```

Various modes can be specified that can combine both passive and active open connections, optionally with RSS. For example:

```
EF_SCALABLE_FILTERS=enps0f0=rss:transparent_active
```

For full details, see EF_SCALABLE_FILTERS.

Per interface, the MAC filter can only be installed by a single Onload stack. A cluster (see Application Clustering) might have multiple stacks and each stack could install a MAC filter on a different interface.

Sockets must be bound to the IP address of the interface.

This feature is targeted at TCP listening sockets only and connections accepted from a listening socket will share the MAC filter.

See also the following configuration parameters:

- `EF_SCALABLE_FILTERS_ENABLE` turns the scalable filter feature on or off on a stack. It is not normally required, because it defaults to 1 when it is unset but `EF_SCALABLE_FILTERS` is set.

  The value 2, available from Onload 201805, Indicates a special mode to address a master-worker hierarchy of some event driven applications, such as NGINX.

  For full details, see EF_SCALABLE_FILTERS_ENABLE.

- `EF_SCALABLE_LISTEN_MODE`, available from Onload 201805, sets the behavior of scalable listening sockets.

  - The default mode 0 accelerates connections to a local address configured on the scalable interface. Passive connections that come via other interfaces are not accelerated.
  - The non-default mode 1 rejects connections that are not to a local address configured on the scalable interface. This avoids kernel scalability issues with large numbers of listen sockets.

  For full details, see EF_SCALABLE_LISTEN_MODE.

Partitioning the NIC

The following example demonstrates how to partition the adapter to expose more than one PF:

```
# sfboot pf-count=2 vf-count=0 switch-mode=partitioning
```

A cold reboot of the server is needed after changes using sfboot.

The sfboot utility is available in the Solarflare Linux Utilities package (SF-107601-LS).

Scalable Filters and Bonding

Bonded interfaces created with the standard Linux bonding or teaming driver can be used for scalable filters.

Every interface that is part of the bond must be present in the system when the scalable filters stack is created. Removing the bond will cause the scalable filter to stop receiving traffic. After a new bond interface is created, the application must be restarted to use the bond.

Transparent Reverse Proxy Modes

Enhancements such as Scalable Filters, Socket Caching and support for the IP_TRANSPARENT socket option support Onload with greater efficiency and increased scalability in transparent reverse proxy mode server deployments.

These features reduce to a minimum the overheads associated with creating and connecting transparent sockets. Onload can use of up to 2 million transparent active-open sockets per Onload stack.

A transparent socket is created when a socket sets the IP_TRANSPARENT socket option and explicitly binds to IP addresses and port. The IP address can be on a foreign host. IP_TRANSPARENT must be set before the bind.

The EF_SCALABLE_FILTERS variable is used to enable scalable filters and to configure the transparent proxy mode.

Restrictions

The following restrictions apply:

- The IP_TRANSPARENT option must be set before the socket is bound.
- The IP_TRANSPARENT option cannot be cleared after bind on accelerated sockets.
- IP_TRANSPARENT sockets cannot be accelerated if they are bound to port 0 or to INADDR_ANY.
- IP_TRANSPARENT sockets cannot be passed to the kernel stack when bound to a port that is in the list specified by EF_TCP_FORCE_REUSEPORT.
- Reverse path filters must be disabled on all interfaces. The user should check the value returned from the following files:

  ```
  # cat /proc/sys/net/ipv4/conf/all/rp_filter# cat /proc/sys/net/ipv4/
  conf/lo/rp_filter
  ```

- When using the `rss:transparent_active` mode (see below), EF_CLUSTER_NAME must be explicitly set by the process sharing the cluster *and* the stack cannot be named by either EF_NAME or `onload_set_stackname()`.

Example Configuration Settings

Below are examples of configurations using the EF_SCALABLE_FILTERS environment option to set transparent proxy modes.

- Enable scalable filters on interface enps0f0 - this inserts a MAC address filter on the adapter. The filter is shared by all active open connections on the interface. Socket caching will be applied to the passive side of the TCP connection.

```
EF_SCALABLE_FILTERS=enps0f0=passive
```

- Enable scalable filters on enps0f0, then all sockets using this interface that have the IP_TRANSPARENT flag set will use the MAC filter, other sockets will continue to use normal IP filters on this interface. Socket caching will be applied to the active side of a TCP connection:

```
EF_SCALABLE_FILTERS=enps0f0=transparent_active
```

- As for the example above, but uses symmetrical RSS to ensure that requests/responses between clients and backend servers are processed by the same thread.

```
EF_SCALABLE_FILTERS=enps0f0=rss:transparent_active
```

- Enable scalable filters on enps0f0, then all sockets using this interface that have the IP_TRANSPARENT flag set will use the MAC filter, other sockets will continue to use normal IP filters on this interface. Socket buffers are cached from active and passive sides of the TCP connection.

```
EF_SCALABLE_FILTERS=enps0f0=transparent_active:passive
```

Transparent Reverse Proxy on Multiple CPUs

Used together with Application Clustering, transparent scalable modes can deliver linear scalability using multiple CPU cores.

This uses RSS to distribute traffic, both upstream and downstream of the proxy application, mapping streams to the correct Onload stack. When each CPU core is associated exclusively with a single clustered stack there can be no contention between stacks.

For this use-case to function correctly, the proxy application will use the downstream client address:port on the upstream (to server) side of the TCP connection. In this way RSS and hardware filters ensure that client side and server side are handled by the same worker thread and traffic is directed to the correct stack.

In this scenario the client thinks it communicates directly with the server, and the server thinks it communicates directly with the client - the transparent proxy server is ‘transparent’.

Performance in Lossy Network Environments

This release makes several improvements to Onload's TCP core in the presence of loss and reordering, as can be the case, for example, where the route to the peer traverses the Internet.

Tail-drop Probe

Classical TCP implementations recover poorly from the case where the last segment(s) in flight are dropped. This results in no visible gap in sequence space, and so there is nothing to trigger fast retransmissions. The segments are instead retransmitted by the RTO mechanism. In order to attempt to trigger a fast retransmission in the case where such tail-loss is suspected, a “tail-drop probe” segment can be sent after a short timeout. This segment would either be the next segment due to be transmitted, or an opportunistic retransmission of the most recent in-flight segment.

Onload 201805-u1 and earlier had a tail-drop probe implementation, but it was not compiled in by default.

In Onload 201811 the tail-drop probe mechanism has been rewritten, and is now built by default. Its use is controlled at runtime by the following environment variable:

- EF_TAIL_DROP_PROBE

  This now defaults to the value read from the kernel configuration at `/proc/sys/net/ipv4/tcp_early_retrans`, which defaults to on. It previously defaulted to 0 (off).

  For more information, see EF_TAIL_DROP_PROBE.

Early Retransmit (RFC 5827) Algorithm

Onload 201811 implements the Early Retransmit (RFC 5827) algorithm for TCP, and also the Limited Transmit (RFC 3042) algorithm, on which Early Retransmit depends. As for tail-drop probes, the purpose of these algorithms is to allow fast retransmissions to happen more readily. The use of these algorithms is controlled by the following environment variable:

- EF_TCP_EARLY_RETRANSMIT

  The default value is read from the kernel configuration at `/proc/sys/net/ipv4/tcp_early_retrans`.

  For more information, see EF_TCP_EARLY_RETRANSMIT.

SACK Improvements

From Onload 201811 onwards, selective acknowledgments received from the peer are used to grow the congestion window more aggressively when recovering from loss. See also TCP SACK.

Initial Sequence Number Caching

Applications which rapidly open and close a large number of connections to other machines might experience occasional connection failures due to the rapid reuse of TCP sequence numbers being detected as retransmits in the TIME-WAIT state. This is most commonly a problem with Windows and FreeBSD TCP stacks.

The standard RFC-derived algorithm for avoiding this problem relies on a clock ticking at a rate which is faster than bytes are transmitted. A link running at 100 Mb can theoretically transmit faster than the clock can tick, however, and 10 Gb+ links can practically do this.

Onload has added the EF_TCP_ISN_MODE option to provide a solution. The default “clocked” setting uses the standard best-effort algorithm. The “clocked+cache” setting will store the last sequence number used for every remote endpoint to guarantee that the problem is avoided. This mode is recommended for applications such as proxies which rapidly open and close connections to a variety of unknown, third-party servers.

The following settings can be used to fine-tune the clocked+cache mode:

- EF_TCP_ISN_CACHE_SIZE

  Number of entries to allocate in the cache of remote endpoints. The default value of 0 selects a size automatically.

  For more information, see EF_TCP_ISN_CACHE_SIZE.
- EF_TCP_ISN_INCLUDE_PASSIVE

  Store data for closed passively-opened connections in the cache. This data would only be needed by an application which closed its listening socket and continued to run, so the option is disabled by default

  For more information, see EF_TCP_ISN_INCLUDE_PASSIVE.
- EF_TCP_ISN_OFFSET

  Distance by which to step the initial sequence number of new connections relative to the previous connection. Only extremely specialized applications would consider changing the default.

  For more information, see EF_TCP_ISN_OFFSET.
- EF_TCP_ISN_2MSL

Maximum amount of time that any remote TCP stack's implementation will leave a socket in the TIME-WAIT state. This is configurable in many systems, however the default value of 240 seconds is a maximum common value across a variety of operating systems.

For more information, see EF_TCP_ISN_2MSL.

Urgent Data Processing

TCP urgent data processing is a rarely-used feature that is inconsistently implemented on various operating systems. In Onload 201811 a new EF_TCP_URG_MODE environment variable has been added, that can be used to ignore this feature.

IMPORTANT! *If urgent data is received when Onload is configured to ignore urgent data processing, then applications that are written to the Linux convention will experience corrupt data.*

TIMEWAIT Assassination

In Onload 201811 the EF_TCP_TIME_WAIT_ASSASSINATION environment variable has been added, to implement the RFC 1337 behavior of replacing old `TIMEWAIT` sockets with newly-received incoming connections.

The default value is read from `/proc/sys/net/ipv4/tcp_rfc1337`.

AMD

Chapter 9

Onload and UDP

This chapter gives information about using UDP with Onload.

UDP Operation

The table below identifies the Onload UDP implementation RFC compliance.

Table 27: **UDP Implementation RFC Compliance**

RFC	Title	Compliance
768	User Datagram Protocol	Yes
1122	Requirements for Hosts	Yes
3678	Socket Interface Extensions for Multicast Source Filters	Partial See Source Specific Socket Options

Socket Options

Onload UDP supports the following socket options which can be used in the `setsockopt()` and `getsockopt()` function calls.

Table 28: **Socket Options for setsockopt() and getsockopt()**

Option	Description
`SO_PROTOCOL`	Retrieve the socket protocol as an integer.
`SO_BINDTODEVICE`	bind this socket to a particular network interface. See SO_BINDTODEVICE.
`SO_BROADCAST`	When enabled datagram sockets can send and receive packets to/from a broadcast address.
`SO_DEBUG`	Enable protocol debugging.
`SO_ERROR`	The errno value of the last error occurring on the socket. (Only valid as a `getsockopt()`).
`SO_EXCLUSIVEADDRUSE`	Prevents other sockets using the SO_REUSEADDR option to bind to the same address and port.

Table 28: **Socket Options for setsockopt() and getsockopt()** *(cont'd)*

Option	Description
SO_LINGER	When enabled a `close()` or `shutdown()` will not return until all queued messages for the socket have been successfully sent or the linger timeout has been reached. Otherwise the call returns immediately and sockets are closed in the background.
SO_PRIORITY	Set the priority for all packets sent on this socket. Packets with a higher priority might be processed first depending on the selected device queuing discipline.
SO_RCVBUF	Sets or gets the maximum socket receive buffer in bytes. EF_UDP_RCVBUF overrides this value. Setting SO_RCVBUF to a value < MTU can result in poorer performance and is not recommended.
SO_RCVLOWAT	Sets the minimum number of bytes to process for socket input operations.
SO_RECVTIMEO	Sets the timeout for input function to complete.
SO_REUSEADDR	Can reuse local ports (another socket can bind to the same port number) except when there is an active listening socket bound to the port.
SO_RESUSEPORT	Allow multiple sockets to bind to the same port.
SO_SNDBUF	Sets or gets the maximum socket send buffer in bytes. The value set is doubled by the kernel and by Onload to allow for bookkeeping overhead when it is set by the `setsockopt()` function call. EF_UDP_SNDBUF overrides this value.
SO_SNDLOWAT	Sets the minimum number of bytes to process for socket output operations. Always set to 1 byte.
SO_SNDTIMEO	Set the timeout for sending function to send before reporting an error.
SO_TIMESTAMP	Enable or disable receiving the `SO_TIMESTAMP` control message (microsecond resolution). See below.
SO_TIMESTAMPNS	Enable or disable receiving the `SO_TIMESTAMP` control message (nanosecond resolution).
SO_TIMESTAMPING	Enable/disable hardware timestamps for received packets.
SOF_TIMESTAMPING_TX_HARDWARE	obtain a hardware generated transmit timestamp.
SOF_TIMESTAMPING_SYS_HARDWARE	Obtain a hardware transmit timestamp adjusted to the system time base.
SO_TYPE	Returns the socket type (`SOCK_STREAM` or `SOCK_DGRAM`). (Only valid as a `getsockopt()`).

Source Specific Socket Options

The following table identifies source specific socket options supported from onload-201210-u1 onwards. Refer to the `ReleaseNotes` file in your Onload distribution for Onload specific behavior regarding these options.

Table 29: **Source Specific Socket Options**

Option	Description
IP_ADD_SOURCE_MEMBERSHIP	Join the supplied multicast group on the given interface and accept data from the supplied source address.
IP_DROP_SOURCE_MEMBERSHIP	Drops membership to the given multicast group, interface and source address.
MCAST_JOIN_SOURCE_GROUP	Join a source specific group.
MCAST_LEAVE_SOURCE_GROUP	Leave a source specific group.

Onload Sockets vs. Kernel Sockets

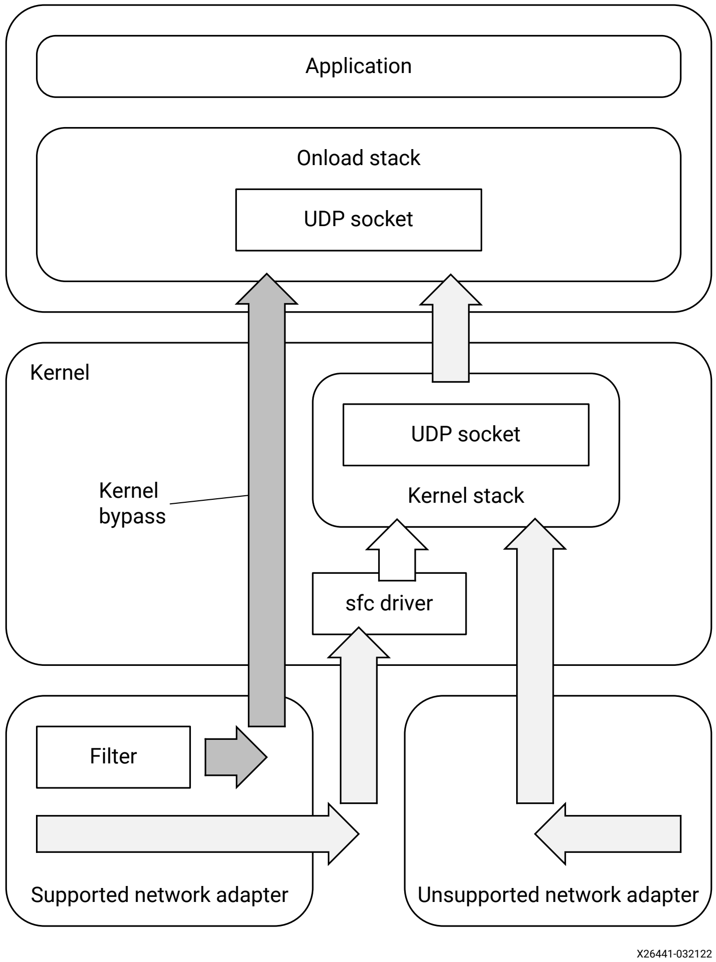
For each UDP socket, Onload creates both an accelerated socket and a kernel socket. Onload will always give priority to the Onload sockets over any kernel sockets.

This is important because if there is always traffic arriving at the Onload receive queue, Onload will might never get to process any packets delivered via the kernel socket (for example if traffic arrives from an unsupported interface).

Send and Receive Paths for UDP Sockets

For each UDP socket, Onload creates both an accelerated socket and a kernel socket. There is usually no file descriptor for the kernel socket visible in the user's file descriptor table. When a UDP process is ready to transmit data, Onload will check a cached ARP table which maps IP addresses to MAC addresses. A cache 'hit' results in sending via the Onload accelerated socket. A cache 'miss' results in a syscall to populate the user mode cached ARP table. If no MAC address can be identified via this process the packet is sent via the kernel stack to provoke ARP resolution. Therefore, it is possible that some UDP traffic will be sent occasionally via the kernel stack.

Figure 17: **UDP Send and Receive Paths**



The preceding figure illustrates the UDP send and receive paths. Dark gray arrows indicate the accelerated 'kernel bypass' path. Light gray arrows identify fragmented UDP packets received by the supported adapter and UDP packets received from an unsupported adapter. UDP packets arriving at the supported adapter are filtered on source and destination address and port number to identify a VNIC the packet will be delivered to. Fragmented UDP packets are received by the application via the kernel UDP socket. UDP packets received by an unsupported adapter are always received via the kernel UDP socket.

Fragmented UDP

When sending datagrams which exceed the MTU, the Onload stack will send multiple Ethernet packets. On hosts running Onload, fragmented datagrams are always received via the kernel stack.

User Level recvmmsg for UDP

The `recvmmsg()` function is intercepted for UDP sockets which are accelerated by Onload.

The Onload user-level `recvmmsg()` is available to systems that do not have kernel/libc support for this function. The `recvmmsg()` is not supported for TCP sockets.

User-Level sendmmsg for UDP

The `sendmmsg()` function is intercepted for UDP sockets which are accelerated by Onload.

The Onload user-level `sendmmsg()` is available to systems that do not have kernel/libc support for this function. The `sendmmsg()` is not supported for TCP sockets.

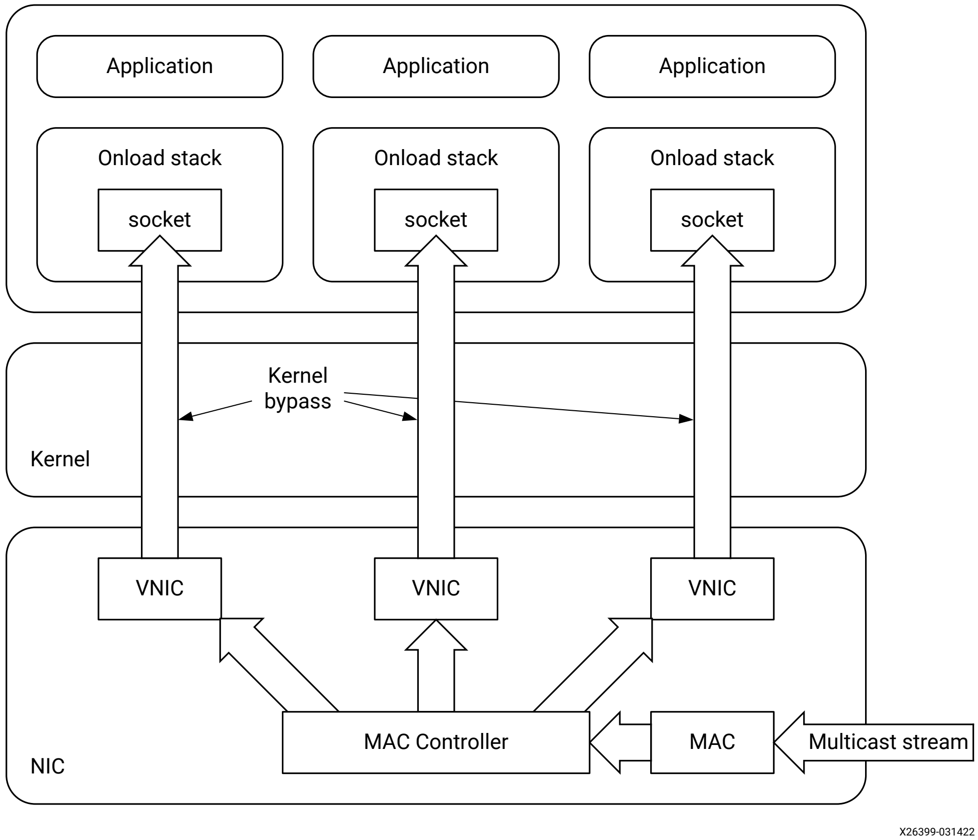
UDP sendfile

The UDP `sendfile()` method is not currently accelerated by Onload. When an Onload accelerated application calls `sendfile()` this will be handled seamlessly by the kernel.

Multicast Replication

The Solarflare SFN8000 and X2 series adapters support multicast replication where received packets are replicated in hardware and delivered to multiple receive queues. This feature allows any number of Onload clients, listening to the same multicast data stream, to receive their own copy of the packets, without an additional software copy and without the need to share Onload stacks. As illustrated below, the packets are delivered multiple times by the controller to each receive queue that has installed a hardware filter to receive the specified multicast stream.

Figure 18: **Hardware Multicast Replication**



Multicast replication is performed in the adapter transparently and does not need to be explicitly enabled.

This feature removes the need to share Onload stacks using the EF_NAME environment variable. Users using EF_NAME exclusively for sharing multicast traffic can now remove EF_NAME from the configurations.

Multicast Operation and Stack Sharing

To illustrate shared stacks, the following examples describe Onload behavior when two processes, on the same host, subscribe to the same multicast stream:

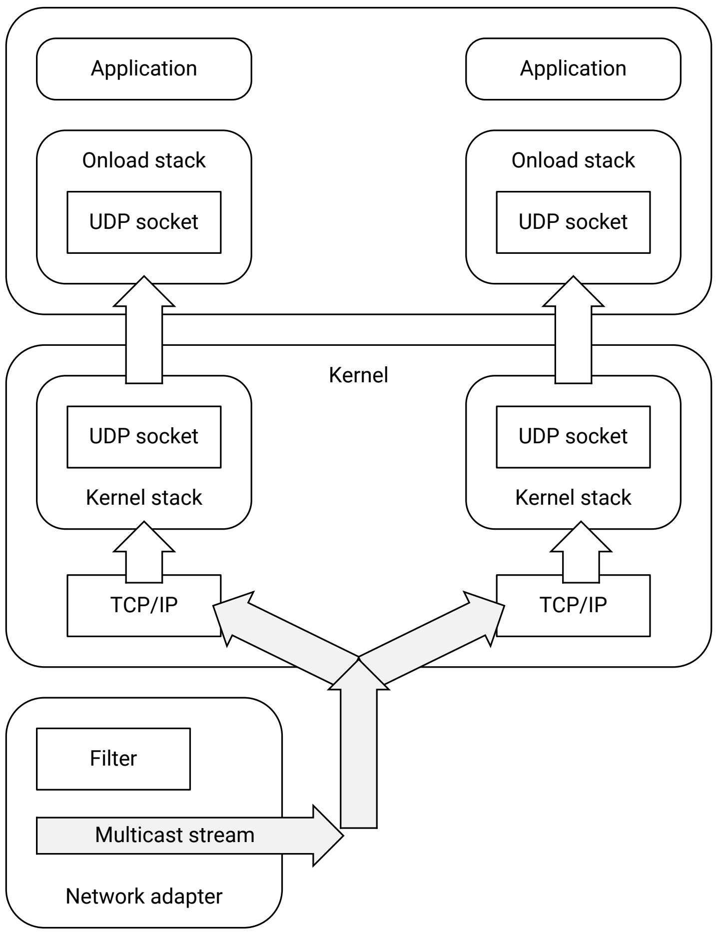
- Multicast Transmit Using Different Onload Stacks
- Multicast Transmit Sharing an Onload Stack
- Multicast Receive to Onload or Kernel Stack
- Multicast Receive and Multiple Sockets.

***Note*:** The following subsections use two processes to demonstrate Onload behavior. In practice multiple processes can share the same Onload stack. Stack sharing is not limited to multicast subscribers and can be employed by any TCP and UDP applications.

Multicast Transmit Using Different Onload Stacks

The following figure illustrates the use of different Onload stacks. Arrows indicate the receive path and fragmented UDP path.

Figure 19: **Using Different Onload Stacks**

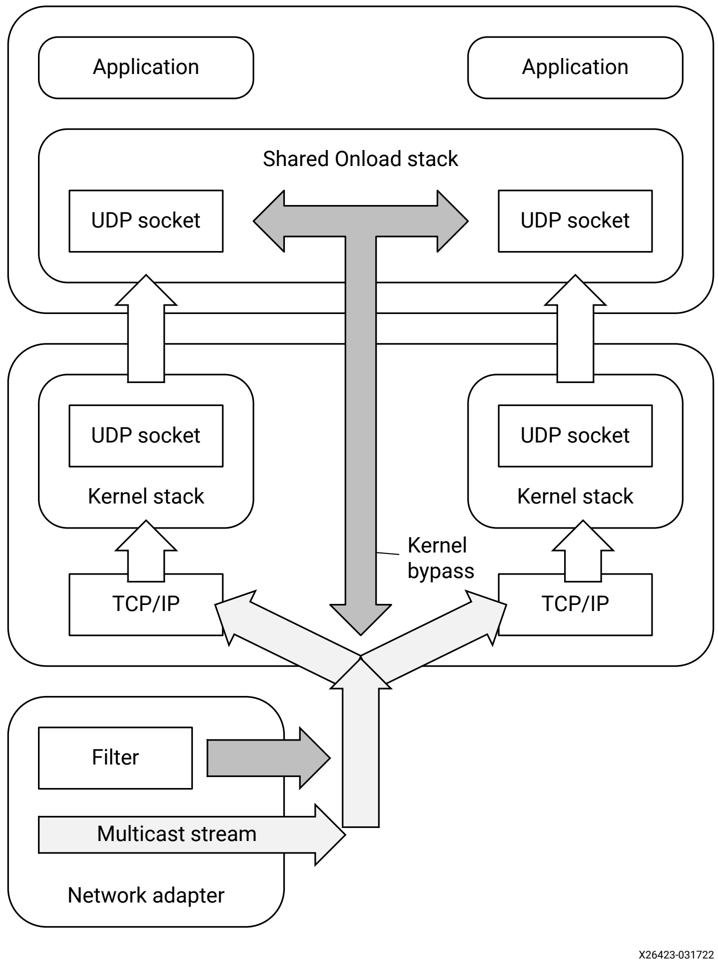


Referring to the preceding figure, if one process were to transmit multicast datagrams, these would not be received by the second process. Onload is only able to accelerate transmitted multicast datagrams when they do not need to be delivered to other applications in the same host. Or more accurately, the multicast stream can only be delivered within the same Onload stack.

Multicast Transmit Sharing an Onload Stack

The following figure illustrates sharing an Onload stack. Dark gray arrows indicate the accelerated (kernel bypass) path. Light gray arrows indicate the fragmented UDP path.

Figure 20: **Sharing an Onload Stack**



Referring to the preceding figure, datagrams transmitted by one process would be received by the second process because both processes share the Onload stack.

Multicast Receive to Onload or Kernel Stack

If a multicast stream is being accelerated by Onload, and another application that is not using Onload subscribes to the same stream, then the second application will not receive the associated datagrams. Therefore if multiple applications subscribe to a particular multicast stream, either all or none should be run with Onload.

Multicast Receive and Multiple Sockets

When multiple sockets join the same multicast group, received packets are delivered to these sockets in the order that they joined the group.

When multiple sockets are created by different threads and all threads are spinning on `recv()`, the thread which is able to receive first will also deliver the packets to the other sockets.

If a thread 'A' is spinning on `poll()`, and another thread 'B', listening to the same group, calls `recv()` but does not spin, 'A' will notice a received packet first and deliver the packet to 'B' without an interrupt occurring.

Multicast Loopback

The socket option IP_MULTICAST_LOOP controls whether multicast traffic sent on a socket can be received locally on the machine. Receiving multicast traffic locally requires both the sender and receiver to be using the same Onload stack. Therefore, when a receiver is in the same application as the sender it will receive multicast traffic. If sender and receiver are in different applications then both must be running Onload and must be configured to share the same Onload stack.

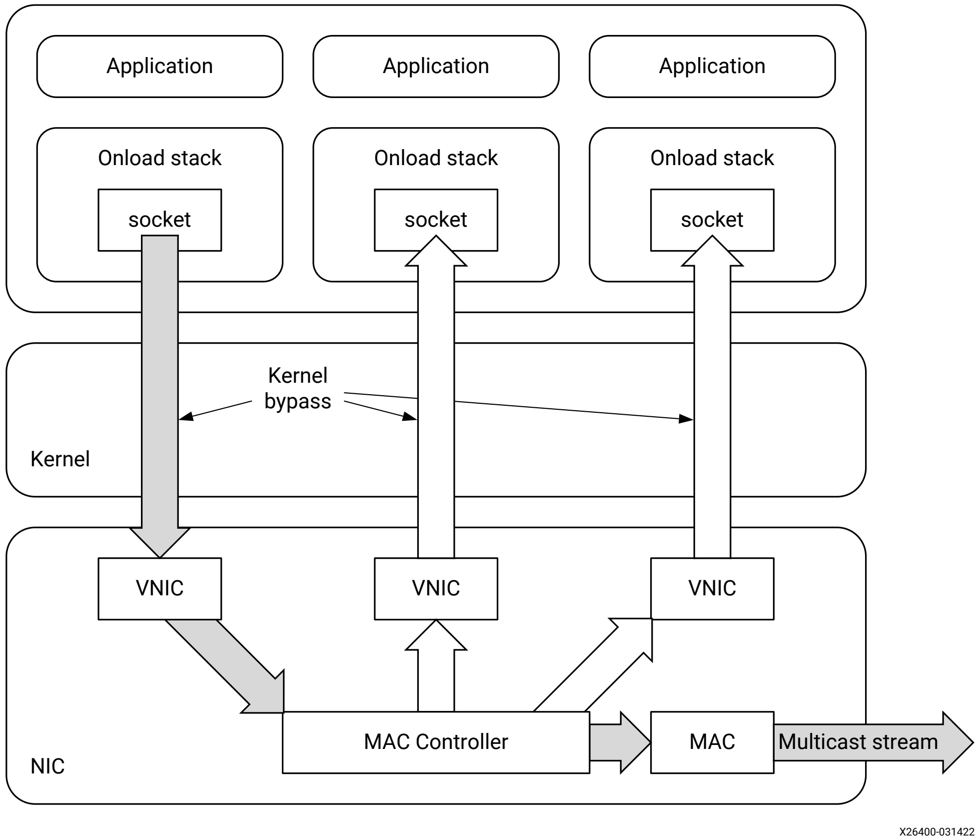
For two processes to share an Onload stack both must set the same value for the EF_NAME parameter (max eight chars). If one local process is to receive the data sent by a sending local process, EF_MCAST_SEND must be set to 1 or 3 on the thread creator of the stack.

User of earlier Onload versions and users of EF_MULTICAST_LOOP_OFF should refer to the Parameter Reference table Appendix A: Parameter Reference for details of deprecated features.

Hardware Multicast Loopback

An alternative to the Onload stack sharing scheme described in Multicast Loopback, Hardware Multicast Loopback, available from openonload-201405, enables the passing of multicast traffic between Onload stacks allowing applications running on the same server to benefit from Onload acceleration without the need to share an Onload stack thereby reducing the risk of stack lock and resource contention.

Figure 21: **Hardware Multicast Loopback**



- Only available on the XtremeScale™ SFN8000 and XtremeScale™ X2 series adapters.
- Adapters must have a minimum firmware version v4.0.7.6710 and "full featured" firmware must be selected using the `firmware-variant` option via the "sfboot" utility. For further details refer to the *Sfboot Parameters* section of the *Solarflare Server User Guide*.

Hardware Multicast Loopback allows data generated by one process to be received by another process on the same host - Multicast Replication does not support local loopback.

Reception of looped back traffic is enabled by default on a per Onload stack basis. A stack can choose not to receive looped back traffic by setting the environment variable EF_MCAST_RECV_HW_LOOP`=0`.

***Note*:** Hardware Multicast Loopback is enabled through a single hardware filter. For this reason, if any single process chooses to receive multicast loopback traffic by EF_MCAST_RECV_HW_LOOP`=1`, then all other processes joined to the same multicast group will also receive the loopback traffic regardless of their setting for EF_MCAST_RECV_HW_LOOP.

Sending of looped back traffic is disabled by default. On a per-stack basis this feature can be enabled by setting the environment variable EF_MCAST_SEND to either 2 or 3.

Setting the socket option `MULTICAST_TTL=0` will disable the sending of traffic on the normal network path and prevent traffic being looped back. The value of the socket option `IP_MULTICAST_LOOP` has no effect on Hardware Multicast Loopback. Refer to Onload and IP_MULTICAST_TTL for differences in Linux kernel and Onload behavior.

IP_MULTICAST_ALL

For an accelerated socket, Onload will usually behave as if IP_MULTICAST_ALL=0. However:

- If two multicast sockets are bound to INADDR_ANY:<same_port> *in different stacks*, then Onload does behave as if IP_MULTICAST_ALL=0. But if the sockets are *in one stack*, then they both receive all packets, as if IP_MULTICAST_ALL=1.

  The behavior is even more complex if the sockets join the same address on different VLANs.

- There is always the potential for messages to arrive at the host - perhaps from an unsupportedsupportedinterface or via the loopback interface - which will also be delivered to the socket under normal UDP port matching rules so the socket could receive traffic for groups not explicitly joined on this socket.

Packet Buffers

This chapter describes packet buffers on SFN8000 series and X2 series adapters. For a description of packet buffers and CTPIO on X3 series adapters refer to the *Alveo X3522 User Guide* (UG1523).

Packet buffers describe the memory used by the Onload stack (and network adapter) to receive, transmit and queue network data. Packet buffers provide a method for user-mode accessible memory to be directly accessed by the network adapter without compromising system integrity.

Onload will request huge pages if these are available when allocating memory for packet buffers. Using huge pages can lead to improved performance for some applications by reducing the number of Translation Lookaside Buffer (TLB) entries needed to describe packet buffers and therefore minimize TLB 'thrashing'.

***Note*:** Onload huge page support should not be enabled if the application uses IPC namespaces and the CLONE_NEWIPC flag.

Onload offers two configuration modes for network packet buffers:

Network Adapter Buffer Table Mode

Solarflare network adapters employ a proprietary hardware-based buffer address translation mechanism to provide memory protection and translation to Onload stacks accessing a VNIC on the adapter. This is the default packet buffer mode and is suitable for the majority of applications using Onload.

This scheme employs a buffer table residing on the network adapter to control the memory an Onload stack can use to send and receive packets.

If the total packet buffer requirements of all applications using Onload require more than the number of packet buffers supported by the adapter's buffer table, the user should consider changing to the Scalable Packet Buffers configuration.

Large Buffer Table Support

The Solarflare SFN8000 and X2 series adapters support many more than the 120,000 packet buffer without the need to switch to Scalable Packet Buffer Mode.

Each buffer table entry in the SFN8000 and X2 series adapter can describe a 4 KB, 64 KB, 1 MB or 4 MB block of memory where each table entry is the page size as directed by the operating system.

Huge Pages

Huge pages are available in 2 MB or 1 GB sizes and can be a benefit to hardware as they create fewer entries in the adapter page table.

Onload applications should not use 1 GB huge pages otherwise every Onload stack would allocate 1 GB huge pages which might not even be required.

The OS huge page size should be set to 2 MB only. A mix of 2 MB and 1 GB sizes is not recommended.

Allocating Huge Pages

Using huge pages can lead to improved performance for some applications by reducing the number of Translation Lookaside Buffer (TLB) entries needed to describe packet buffers and therefore minimize TLB 'thrashing'. Huge pages also deliver many packets buffers, but consume only a single entry in the buffer table. Explicit huge pages are recommended.

Onload is able to use a total of 4096 huge pages.

The current huge page allocation can be checked by inspection of `/proc/meminfo`:

```
cat /proc/meminfo | grep Huge
```

This should return something similar to:

```
AnonHugePages: 2048 kB
HugePages_Total: 2050
HugePages_Free: 2050
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
```

The total number of huge pages available on the system is the value `HugePages_Total`. The following command can be used to dynamically set and/or change the number of huge pages allocated on a system to <N> (where <N> is a non-negative integer):

```
echo <N> > /proc/sys/vm/nr_hugepages
```

On a NUMA platform, the kernel will attempt to distribute the huge page pool over the set of all allowed nodes specified by the NUMA memory policy of the task that modifies `nr_hugepages`. The following command can be used to check the per node distribution of huge pages in a NUMA system:

```
cat /sys/devices/system/node/node*/meminfo | grep Huge
```

Huge pages can also be allocated on a per-NUMA node basis (rather than have the huge pages allocated across multiple NUMA nodes). The following command can be used to allocate <N> huge pages on NUMA node <M>:

```
echo <N> > /sys/devices/system/node/node<M>/hugepages/hugepages-2048kB/
nr_hugepages
```

How Onload Uses Packet Buffers

Each packet buffer is allocated to exactly one Onload stack and is used to receive, transmit or queue network data. Packet buffers are used by Onload in the following ways:

1. Receive descriptor rings. By default the RX descriptor ring will hold 512 packet buffers at all times. This value is configurable using the EF_RXQ_SIZE (per stack) variable.
2. Transmit descriptor rings. By default the TX descriptor ring will hold up to 512 packet buffers. This value is configurable using the EF_TXQ_SIZE (per stack) variable.
3. To queue data held in socket receive and send buffers.
4. TCP sockets can also hold packet buffers in the socket's retransmit queue and in the reorder queue.
5. User-level pipes also consume packet buffer resources.

Identifying Packet Buffer Requirements

When deciding the number of packet buffers required by an Onload stack consideration should be given to the resource needs of the stack to ensure that the available packet buffers can be shared efficiently between all Onload stacks.

- *Example 1:*

  If we consider a hypothetical case of a single host:

  - which employs multiple Onload stacks, for example 10
  - each stack has multiple sockets, for example 6
  - and each socket uses many packet buffers, for example 2000

This would require a total of 120,000 packet buffers

- *Example 2:*

  If on a stack the TCP receive queue is 1 MB and the MSS value is 1472 bytes, this would require at least 700 packet buffers - (and a greater number if segments smaller that the MSS were received).

- *Example 3:*

  A UDP receive queue of 500 received datagrams, each 200 bytes long, has a total payload of approximately 100 KB. However, it will consume approximately 1 MB. This is because each packet buffer is 2048 bytes, even though the packet data is less than 2048 bytes.

The examples above use only approximate calculated values. The onload_stackdump command provides accurate measurements of packet buffer allocation and usage.

Consideration should be given to packet buffer allocation to ensure that each stack is allocated the buffers it will require rather than a 'one size fits all' approach.

When using the Buffer Table Mode the system is limited to 120,000 packet buffers - these are allocated symmetrically across all Solarflare interfaces.

***Note*:** Packet buffers are accessible to all network interfaces and each packet buffer requires an entry in every network adapters' buffer table. Adding more network adapters - and therefore more interfaces does not increase the number of packet buffers available.

Running Out of Packet Buffers

When Onload detects that a stack is close to allocating all available packet buffers it will take action to try and avoid packet buffer exhaustion. Onload will automatically start dropping packets on receive and, where possible, will reduce the receive descriptor ring fill level in an attempt to alleviate the situation. A 'memory pressure' condition can be identified using the `onload_stackdump lots` command where the `pkt_bufs` field will display the `CRITICAL` indicator. See Identifying Memory Pressure below.

Complete packet buffer exhaustion can result in deadlock. In an Onload stack, if all available packet buffers are allocated (for example currently queued in socket receive and send buffers) the stack is prevented from transmitting further data as there are no packet buffers available for the task.

If all available packet buffers are allocated then Onload will also fail to keep its adapters receive queues replenished. If the queues fall empty further data received by the adapter is instantly dropped. On a TCP connection packet buffers are used to hold unacknowledged data in the retransmit queue, and dropping received packets containing ACKs delays the freeing of these packet buffers back to Onload. Setting the value of EF_MIN_FREE_PACKETS`=0` can result in a stack having no free packet buffers and this, in turn, can prevent the stack from shutting down cleanly.

Identifying Memory Pressure

The following extracts from the `onload_stackdump` command identify an Onload stack under memory pressure.

The EF_MAX_PACKETS value identifies the maximum number of packet buffers that can be used by the stack. EF_MAX_RX_PACKETS is the maximum number of packet buffers that can be used to hold packets received. EF_MAX_TX_PACKETS is the maximum number of packet buffers that can be used to hold packets to send. These two values are always less than EF_MAX_PACKETS to ensure that neither the transmit or receive paths can starve the other of packet buffers. Refer to Appendix A: Parameter Reference for detailed descriptions of these per stack variables.

The example Onload stack has the following default environment variable values:

```
EF_MAX_PACKETS:    32768
EF_MAX_RX_PACKETS: 24576
EF_MAX_TX_PACKETS: 24576
```

The `onload_stackdump lots` command identifies packet buffer allocation and the onset of a memory pressure state:

```
pkt_bufs: size=2048 max=32768 alloc=24576 free=32 async=0 CRITICAL
pkt_bufs: rx=24544 rx_ring=9 rx_queued=24535
```

There are potentially 32768 packet buffers available and the stack has allocated (used) 24576 packet buffers.

In the socket receive buffers there are 24544 packets buffers waiting to be processed by the application. This is approaching the EF_MAX_RX_PACKETS limit and is the reason the `CRITICAL` flag is present. The Onload stack is under memory pressure. Only nine packet buffers are available to the receive descriptor ring.

Onload will aim to keep the RX descriptor ring full at all times. If there are not enough available packet buffers to refill the RX descriptor ring this is indicated by the `LOW` memory pressure flag.

The `onload_stackdump lots` command will also identify the number of memory pressure events and number of packets dropped when Onload fails to allocate a packet buffer on the receive path.

```
memory_pressure_enter: 1
memory_pressure_drops: 22096
```

The memory_pressure enter/exit counters count the number of times Onload enter/exits a state when it is trying to refill the receive queue (rxq) when the adapter runs out of packet buffers.

The Onload module option, `max_packets_per_stack`, places an upper limit on EF_MAX_PACKETS. When `max_packets_per_stack` is not set in the `/etc/modprobe.d/<file>`, its default value is 512K.

Controlling Onload Packet Buffer Use

A number of environment variables control the packet buffer allocation on a per stack basis. Refer to Appendix A: Parameter Reference for a description of EF_MAX_PACKETS.

Unless explicitly configured by the user, EF_MAX_RX_PACKETS and EF_MAX_TX_PACKETS will be automatically set to 75% of the EF_MAX_PACKETS value. This ensures that sufficient buffers are available to both receive and transmit. The EF_MAX_RX_PACKETS and EF_MAX_TX_PACKETS are not typically configured by the user.

If an application requires more packet buffers than the maximum configured, then EF_MAX_PACKETS can be increased to meet demand, however it should be recognized that larger packet buffer queues increase cache footprint which can lead to reduced throughput and increased latency.

EF_MAX_PACKETS is the maximum number of packet buffers that could be used by the stack. Setting EF_MAX_RX_PACKETS to a value greater than EF_MAX_PACKETS effectively means that all packet buffers (EF_MAX_PACKETS) allocated to the stack will be used for RX - with nothing left for TX. The safest method is to only increase EF_MAX_PACKETS which keeps the RX and TX packet buffers values at 75% of this value.

Physical Addressing Mode

Physical addressing mode is a Scalable Packet Buffer Mode that also allows Onload stacks to use large amounts of packet buffer memory (avoiding the limitations of the address translation table on the adapter), but without the requirement to configure and use SR-IOV virtual functions.

Physical addressing mode, does however, remove memory protection from the network adapter's access of packet buffers. Unprivileged user-level code is provided and directly handles the raw physical memory addresses of packets buffers. User-level code provides physical memory addresses directly to the adapter and therefore has the ability to direct the adapter to read or write arbitrary memory locations. A result of this is that a malicious or buggy application can compromise system integrity and security. Onload versions earlier than OpenOnload 201210 and EnterpriseOnload 2.1.0.0 are limited to 1 million packet buffers. This limit was raised to 2 million packet buffers in OpenOnload 201210-u1 and EnterpriseOnload 2.1.0.1.

To enable physical addressing mode:

1. Ignore configuration steps 1-4 above.
2. Put the following option into a user-created `.conf` file in the `/etc/modprobe.d` directory:

   ```
   options onload phys_mode_gid=<n>
   ```

Where setting <n> to be -1 allows all users to use physical addressing mode and setting to an integer x restricts use of physical addressing mode to the specific user group x.

3. Reload the Onload kernel drivers:

```
# onload_tool reload
```

4. Enable the Onload environment using EF_PACKET_BUFFER_MODE 2 or 3.

   EF_PACKET_BUFFER_MODE=2 is equivalent to mode 0, but uses physical addresses. Mode 3 uses SR-IOV VFs with physical addresses, but does not use the IOMMU for memory translation and protection. Refer to Appendix A: Parameter Reference for a complete description of all EF_PACKET_BUFFER_MODE options.

Programmed I/O

PIO (programmed input/output) describes the process whereby data is directly transferred by the CPU to or from an I/O device. It is an alternative to bus master DMA techniques where data are transferred without CPU involvement.

Solarflare SFN8000 and X2 series adapters support TX PIO, where packets on the transmit path can be “pushed” to the adapter directly by the CPU. This improves the latency of transmitted packets but can cause a very small increase in CPU utilization. TX PIO is therefore especially useful for smaller packets.

The Onload TX PIO feature is enabled by default but can be disabled via the environment variable EF_PIO. An additional environment variable, EF_PIO_THRESHOLD specifies the size of the largest packet size that can use TX PIO.

The number of PIO buffers available depend on the adapter type being used and the number of PCIe Physical Functions (PF) exposed per port.

Table 30: **Number of PIO Buffers Available**

Solarflare Adapter	Total PIO Buffers	Maximum per PF	PIO Buffer Size
SFN7x02	16	16	2 KB
SFN7x22	16	16	2 KB
SFN7x24	32	16	2 KB
SFN7X42	32	16	2 KB
SFN8522	16	16	4 KB
X2522	16	16	4 KB
X254x	16	16	4 KB
X2552	16	16	4 KB
X2562	16	16	4 KB

For optimum performance, PIO buffers should be reserved for critical processes and other processes should set EF_PIO to 0 (zero).

The Onload stackdump utility provides additional counters to indicate the level of PIO use - see TX PIO Counters for details.

The Solarflare net driver will also use PIO buffers for non-accelerated sockets and this will reduce the number of PIO buffers available to Onload stacks. To prevent this set the driver module option `piobuf_size=0`. Driver module options can be set in a user-created file (`sfc.conf`) in the `/etc/modprobe.d` directory:

```
options sfc piobuf_size=0
```

An Onload stack requires one PIO buffer for each VI it creates. An Onload stack will create one VI for each physical interface that it uses.

When both accelerated and non-accelerated sockets are using PIO, the number of PIO buffers available to Onload stacks can be calculated from the available PIO regions:

Table 31: **Number of PIO Buffers Available**

Input	Description	Example Value
piobuf_size	driver module parameter	256
rss_cpus	driver module parameter	4
region	a chunk of memory 2048 bytes	2048 bytes
PF	PCIe physical function. The adapter can be partitioned to expose up to eight PFs per physical port. Refer to Onload and NIC Partitioning for details	Default one PF

Using the above example values applied to a SFN7x22 adapter, each PF on the adapter requires:

piobuf_size * rss_cpus * num_PFs/ region size = 0.5 regions - (round up - so each port needs 1 region).

This leaves 16-2 = 14 regions for Onload stacks which also require one region per port, per stack. Therefore from our example we can have 7 onload stacks using PIO buffers.

PIO buffers are allocated on a first-come, first-served basis. The following warning might be observed when stacks cannot be allocated any more PIO buffers:

```
WARNING: all PIO bufs allocated to other stacks. Continuing without PIO.
 Use EF_PIO to control this
```

To ensure more buffers are available for Onload, it is possible to prevent the net driver from using PIO buffers. This can be done by setting the sfc driver module option in a user-created file in the `/etc/modprobe.d` directory:

```
options sfc piobuf_size=0
```

Reload the Onload kernel drivers for the changes to be effective:

```
# onload_tool reload
```

The per-stack EF_PIO variable can also be unset for stacks where PIO buffers are not required. If there is contention for PIO buffers, consider disabling PIO for any stacks that primarily receive, so the buffers are available for stacks that perform latency-critical sends.

CTPIO

Onload-201805 introduces the Cut Through Programmed Input Output (CTPIO) feature to deliver the lowest send-path latency enabled by the architecture of the X2 series adapters. Packets are streamed directly over the PCIe bus to the network port, bypassing the main adapter transmit datapath.

CTPIO coexists alongside the standard host-buffered (DMA) transmit mechanism (Network Adapter Buffer Table Mode) and legacy PIO buffering (Programmed I/O). From an Onload VI, traffic using all three methods can be mixed on a frame-by-frame basis and per frame ordering is maintained. CTPIO delivers the lowest transmit latency.

Capabilities

The X2 series adapter supports up to 2048 VIs, and all VIs can transmit using CTPIO. However, only one CTPIO packet, per adapter physical port, can be pushed at a time. When multiple VIs are mapped to the same physical port, a CTPIO push in progress will continue undisturbed, whilst another push attempt, overlapping the first, will fail with the send falling back to use the normal DMA transmit datapath.

The adapter can tolerate back-to-back bursts of CTPIO frames from a VI. An Onload stack creates a VI for each physical port that it uses.

***Note*:** X2 series adapters support CTPIO on a maximum four physical ports.

Requirements for CTPIO

CTPIO has the following requirements:

- CTPIO is a feature of the X2 series of adapters. On the X2 models, CTPIO is enabled by default.
- The minimum adapter driver and firmware versions are identified below:

```
# ethtool -i <interface>driver: sfcversion: 4.13.1.1034firmware-version:
7.1.1.1007 rx1 tx1
```

- CTPIO can be used with all adapter firmware variant settings apart from capture-packed-stream.

  Because CTPIO bypasses the main adapter datapath, packets sent by CTPIO cannot be looped back in hardware. As a result, CTPIO is not enabled by default on interfaces running full-featured firmware. This behavior can be overridden using a configuration variable. See EF_CTPIO_SWITCH_BYPASS.
- CTPIO can be used with Onload 201805 and later versions.

CTPIO Modes

The CTPIO feature can be used in three modes:

1. cut-through (ct)

   Lowest latency. A packet is transmitted onto the network as it is being streamed across the PCIe bus. The adapter starts transmitting the packet even before the entire packet has been delivered over the PCIe bus.

   ***Note*:** This mode is supported at 10 Gb, but not at 25 Gb.
2. store-and-forward (sf)

   The packet is buffered on the adapter before transmitting onto the network. The adapter only transmits when the whole packet has been delivered over the PCIe bus.
3. store-and-forward-with-poison-disabled (sf-np)

   As for [2], but guaranteed that packets are never poisoned. Invalid packets are not transmitted on the wire. This is the default mode for Onload.

CAUTION! *When [3] (sf-np) is enabled on any VI, all VIs are placed into this mode.*

CTPIO Frame Length

The table below shows frame lengths when using CTPIO.

Table 32: **Frame Lengths When Using CTPIO**

Frame	Length
Maximum CTPIO frame	4092 bytes

Table 32: **Frame Lengths When Using CTPIO** *(cont'd)*

Frame	Length
Minimum CTPIO frame	29 bytes
Frame sizes 29-59 bytes	Are padded to 60 bytes by CTPIO logic

Cost of CTPIO

The very low-latency CTPIO transmit path comes at a sacrifice of some of the adapter acceleration features:

The following adapter features are not available to packets sent via CTPIO.

Table 33: **Feature Restrictions with CTPIO**

Feature	Restriction
No checksum offloads	Checksums must be done in the host.
No pacing	Packet is sent as soon as last descriptor is posted.
No switch loopback	No HW loopback of packets to local receivers.
No filter drop	No loopback packets to local receivers and CTPIO packets are not subject to any TX drop filters.
No flow control No Qbb flow control	CTPIO traffic cannot be subject to pause frames or priority flow control measures. Pause frames received at the CTPIO sender will have no effect.

Using CTPIO with Onload

On the X2 series adapter, CTPIO is enabled by default in sf-np mode. The following Onload environment variables are used to configure CTPIO:

Table 34: **Environment Variables for Configuring CTPIO**

EF Variable	Settings
`EF_CTPIO`	0 disable 1 enable (default) 2 enable, fail if not available
`EF_CTPIO_MODE`	ct cut-through mode (not supported for 25 Gb) sf store-and-forward mode sf-np store-and-forward-no-poison (default)
`EF_CTPIO_MAX_FRAME_LEN`	Integer value Packets with length exceeding this value are not sent via CTPIO, but are sent using the legacy DMA datapath.

Latency-best Profile

Onload-201805 includes a new low latency profile that enables CTPIO in cut-through mode to deliver lowest CTPIO latency:

```
# onload --profile=latency-best <application, args>
```

Before using this profile, please read The latency-best Profile, especially the warning on usage.

Using CTPIO with TCPDirect

On the X2 series adapter, CTPIO is enabled by default in sf-np mode. The following TCPDirect attributes are used to configure CTPIO:

Table 35: **TCPDirect attributes for Configuring CTPIO**

TCPDirect attribute	Settings
`ctpio`	0 disable 1 enable (default) 2 enable, warn if not available 3 enable, fail if not available
`ctpio_mode`	ct cut-through mode (not supported for 25 Gb) sf store-and-forward mode sf-np store-and-forward-no-poison (default)

For further details, refer to the *TCPDirect User Guide* (SF-116303-CD).

Using CTPIO with ef_vi

The OpenOnload 201805 distribution includes an example application using CTPIO with ef_vi:

```
onload/src/tests/rtt/rtt_efvi.c.
```

When OpenOnload is installed the test application is found at the following location:

```
/onload-201805/build/gnu_x86_64/tests/rtt
```

The following sequence is required to send via CTPIO:

1. When allocating a VI, `ef_vi_alloc_from_pd()`, set the `EF_VI_TX_CTPIO` flag. Also set the TX timestamping flag if required.
2. To initiate a send, form a complete Ethernet frame (excluding FCS) in host memory. Initiate the send with `ef_vi_transmit_ctpio()` or `ef_vi_transmitv_ctpio()`.

3. Post a fall-back descriptor using `ef_vi_transmit_ctpio_fallback()` or `ef_vi_transmitv_ctpio_fallback()`. These calls are used just like the standard DMA send calls (`ef_vi_transmit()` etc.), and so must be provided with a copy of the frame in registered memory.
4. A TX completion event is returned to the host application regardless of whether the frame is sent via CTPIO or the legacy DMA send method.

The posting of a fall-back descriptor is not on the latency critical path, provided the CTPIO operation succeeds, however it should be posted before posting any further sends on the same VI.

For further details, refer to the *ef_vi User Guide* (SF-114063-CD).

Latency Tests

For benchmark latency tests using CTPIO refer to the Appendix L: X2 Low Latency Quickstart.

Onload 201805 and later include the rtt benchmark test application. The application source code can be found in:

```
<onload_dir>/src/tests/rtt
```

The application binary can be found in:

```
<onload_dir>/build/gnu_x86_64/tests/rtt
```

Example command lines:

- On server1:

  ```
  ./rtt pong efvi:tx=ctpio,intf=<interface>
  ```

- On server2:

  ```
  ./rtt ping efvi:tx=ctpio,intf=<interface> | ef_vi/stats
  ```

rtt produces raw output which can be piped to file.

CTPIO Timestamps

TX timestamps are returned with TX completion events to the VI event queue.

CTPIO Statistics

The adapter driver exposes CTPIO counters via ethtool.

```
# ethtool -S <interface> | grep ctpio
```

Counts in the following table are aggregated for all interfaces on the adapter.

Table 36: **Per-adapter Statistics for CTPIO**

Stats	Description
`ctpio_vi_busy_fallback`	When a CTPIO push occurs from a VI, but the VI DMA datapath is still busy with packets in flight or waiting to be sent. The packet is sent over the DMA datapath.
`ctpio_long_write_success`	Host wrote excess data beyond 32-byte boundary after frame end, but the CTPIO send was successful.
`ctpio_missing_dbell_fail`	When CTPIO push is not accompanied by a TX doorbell.
`ctpio_overflow_fail`	When the host pushes packet bytes too fast and overflows the CTPIO buffer.
`ctpio_underflow_fail`	When the host fails to push packet bytes fast enough to match the adapter port speed. The packet is truncated and data transmitted as a poisoned packet.
`ctpio_timeout_fail`	When host fails to send all bytes to complete the packet to be sent by CTPIO before the VI inactivity timer expires. The packet is truncated and data transmitted as a poisoned packet.
`ctpio_noncontig_wr_fail`	A non-sequential address (for packet data) is encountered during CTPIO, caused when packet data is sent over PCIe interface as out-of-order or with gaps. Packet is truncated and transmitted as a poisoned packet.
`ctpio_frm_clobber_fail`	When a CTPIO push from one VI would have 'clobbered' a push already in progress by the same VI or another VI. One or both packets are sent over the DMA datapath - no packets are dropped.
`ctpio_invalid_wr_fail`	If packet length is less than length advertised in the CTPIO header the CTPIO fails. Or packet write is not aligned to (or multiple of) 32-bytes, Packet maybe transmitted as a poisoned packet if sending has already started. Or erased if send has not already started.
`ctpio_vi_clobber_fallback`	When a CTPIO collided with another already in progress. In-progress packets succeeds, other is sent over the DMA datapath.
`ctpio_unqualified_fallback`	VI is not enabled to send using CTPIO or first write is not the packet header. Packet is discarded and sent over the DMA datapath.
`ctpio_runt_fallback`	Length in header < 29 bytes. Packet is discarded and sent over the DMA datapath.

Counts in the following table are per interface on the adapter.

Table 37: **Per-interface Statistics for CTPIO**

Stats	Description
`ctpio_success`	Number of successful CTPIO transmit events.

Table 37: **Per-interface Statistics for CTPIO** *(cont'd)*

Stats	Description
`ctpio_fallback`	Number of instances when CTPIO push was rejected. This can occur because: • the VI legacy datapath is still busy • another CTPIO is in progress • VI is not enabled to use CTPIO • push request for illegal sized frame Fallback events do not result in poison packets. Rejected packets will fallback to use the legacy DMA datapath path.
`ctpio_poison`	When the packet send has started, if CTPIO has to abort this packet, a corrupt CRC is attached to the packet. A poisoned packet might be sent over the wire - depending on the mode. The packet will fallback to use the legacy DMA datapath.
`ctpio_erase`	Before a packet send has started. Corrupt, undersized or poisoned packets are erased from the CTPIO datapath. Packet send will fallback to use the legacy DMA datapath.

Chapter 11

Interfaces

This chapter identifies Onload support for virtual interfaces types.

Bonding, Link Aggregation and Failover

Bonding (also known as teaming) allows for improved reliability and increased bandwidth by combining physical ports from one or more network adapters into a bond. A bond has a single IP address, single MAC address and functions as a single port or single adapter to provide redundancy.

Onload monitors the OS configuration of the standard kernel bonding module and accelerates traffic over bonds that are detected as suitable (see limitations). As a result no special configuration is required to accelerate traffic over bonded interfaces.

For example, to configure an 802.3ad bond of two SFC interfaces (eth2 and eth3):

```
modprobe bonding miimon=100 mode=4 xmit_hash_policy=layer3+4
ifconfig bond0 up
```

Interfaces must be down before adding to the bond.

```
echo +eth2 > /sys/class/net/bond0/bonding/slaves
echo +eth3 > /sys/class/net/bond0/bonding/slaves
ifconfig bond0 192.168.1.1/24
```

The file `/var/log/messages` should then contain a line similar to:

```
[onload] Accelerating bond0 using Onload
```

Traffic over this interface will then be accelerated by Onload.

Polling the Bonding Configuration

Onload will monitor the underlying bonding configuration and state using netlink where this is supported by the OS or revert to the previous method when netlink is not supported. Polling parameters are now set through the cplane_server_params option via the onload_module parameters:

```
options cplane_server --bond_base_period=800options cplane_server --
bond_peak_period=1600
```

The cplane_server parameters are in milliseconds.

Refer to the Limitations section, Bonding, Link Aggregation for further information.

Teaming

In addition to traditional Linux bonding, Onload also supports link aggregation using the Linux teaming driver that is introduced in RHEL 7, SLES 12, and other recent distributions. There are various methods to configure teaming. The example below demonstrates the use of the NetworkManager CLI which creates the `ifcfg` files in the `/etc/sysconfig/network-scripts` directory. Using `nmcli`, teams persist across server reboots.

1. Create the team:

```
# nmcli connection add type team ifname teamA
Connection 'team-teamA' (b7c39a10-84ac-4840-85f2-66adb5e71183)
successfully added.
```

2. List the created team:

```
# nmcli con show
NAME         UUID                                  TYPE            DEVICE
eno2         4efeb125-d489-4a06-9d8a-407bf03fcc77  802-3-ethernet  --
eno1         f270807d-9904-452e-bbd2-0d6b48840c80  802-3-ethernet  eno1
enp1s0f1     16192f4d-7a97-4154-924b-02ca905c8cd7  802-3-ethernet  --
team-teamA   b7c39a10-84ac-4840-85f2-66adb5e71183  team            teamA
virbr0       ceb1a683-7db9-4721-8ca9-38a577ff5d77  bridge          virbr0
enp1s0f0     cbc92b25-9855-451c-8c6b-186b6d5db9f6  802-3-ethernet  --
```

3. View default settings for the newly created team:

```
# cat /etc/sysconfig/network-scripts/ifcfg-team-teamA
DEVICE=teamA
DEVICETYPE=Team
BOOTPROTO=dhcp
DEFROUTE=yes
PEERDNS=yes
PEERROUTES=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
```

```
IPV6_DEFROUTE=yes
IPV6_PEERDNS=yes
IPV6_PEERROUTES=yes
IPV6_FAILURE_FATAL=no
NAME=team-teamA
UUID=b7c39a10-84ac-4840-85f2-66adb5e71183
ONBOOT=yes
```

4. Add primary interface to the team:

```
# nmcli con add type team-slave con-name teamA-port1 ifname enp1s0f0
master teamA
Connection 'teamA-port1' (015f09d7-3f2a-4578-aaea-7ff89a2769f7)
successfully added.
```

5. Add a second interface to the team:

```
# nmcli con add type team-slave con-name teamA-port2 ifname enp1s0f1
master teamA
Connection 'teamA-port2' (92dfe561-860a-4906-842d-b7ebdf263dbe)
successfully added.
```

6. Bring up the team ports:

```
# nmcli connection up teamA-port1
Connection successfully activated (D-Bus active path: /org/freedesktop/
NetworkManager/ActiveConnection/6)
```

   Repeat command for other team ports.

7. Assign team IP addresses via `ifcfg` files or command line as required.

***Note*:** Teams created with the teamd daemon are non-persistent. Teams created with `nmcli` are persistent across server reboots.

To disable Onload acceleration of teaming, please contact support-nic@amd.com.

VLANS

The division of a physical network into multiple broadcast domains or VLANs offers improved scalability, security and network management.

Onload will accelerate traffic over suitable VLAN interfaces by default with no additional configuration required.

For example to add an interface for VLAN 5 over an SFC interface (eth2):

```
modprobe onload
modprobe 8021q
vconfig add eth2 5
ifconfig eth2.5 192.168.1.1/24
```

Traffic over this interface will then be transparently accelerated by Onload.

Refer to the Limitations section, VLANs for further information.

MACVLAN

The MACVLAN driver is supported from onload-201710 in container configurations and in standard host configurations.

- MACVLAN sub-interfaces will be accelerated by Onload when these are created over a supported interface.
- Nested MACVLAN (MACVLAN on top of MACVLAN on top of base adapter) interfaces are also accelerated by Onload.
- MACVLAN sub-interfaces that are not in the main network namespace will be accelerated only if the base adapter is present in the current or main namespace.

If there are more than one PCI physical interfaces (PF) or virtual interfaces (VF) configured, the adapter firmware-variant must be one of the following:

- the ultra-low-latency firmware variant.
- when using the full-feature firmware variant, the insecure-filters=1 sfboot option must be set.

Restrictions do not apply when there is only a single PF and no VFs configured. Check the adapter configuration with the sfboot utility.

IPVLAN

The IPVLAN driver is supported from Onload-7.0.0 in container and standard host configurations.

IPVLAN requires RHEL8 or Ubuntu 18.10. Other OS variants might have minimum version requirements.

IPVLAN sub-interfaces will be accelerated by Onload when there are created over a supported interface.

Use standard Linux commands to create the IPVLAN interface:

```
# ip link add link <master> name <slave> type ipvlan [mode][flags]
```

example:

```
# ip link add link p2p2 name ipv1 type ipvlan mode l2 bridge
```

Only l2 mode is accelerated, l3 or l3s modes are not accelerated. MAC filter-based scalable filter modes are not supported with IPVLAN interface.

Accelerated pipe()

Onload supports the acceleration of pipes, providing an accelerated IPC mechanism through which two processes on the same host can communicate using shared memory at user-level. Accelerated pipes do not invoke system calls. Accelerated pipes therefore, reduce the overheads for read/write operations and offer improved latency over the kernel implementation.

To create a user-level pipe, and before the `pipe()` or `pipe2()` function is called, a process must be accelerated by Onload and must have created an Onload stack. By default, an accelerated process that has not created an Onload stack is granted only a non-accelerated pipe. See EF_PIPE for other options.

The accelerated pipe is created from the pool of available packet buffers.

The following function calls, related to pipes, will be accelerated by Onload and will not enter the kernel unless they block:

- `pipe()`
- `read()`
- `write()`
- `readv()`
- `writev()`
- `send()`
- `recv()`
- `recvmsg()`
- `sendmsg()`
- `poll()`
- `select()`
- `epoll_ctl()`
- `epoll_wait()`

As with TCP/UDP sockets, the Onload tuning options such as EF_POLL_USEC and EF_SPIN_USEC will also influence performance of the user-level pipe.

Refer also to EF_PIPE, EF_PIPE_RECV_SPIN, EF_PIPE_SEND_SPIN in Appendix A: Parameter Reference.

Note: Only anonymous pipes created with the `pipe()` or `pipe2()` function calls will be accelerated.

Chapter 12

Onload and Virtualization

Using Onload from release 201502, accelerated applications can benefit from the inherent security through isolation, ease of deployment through migration and increased resource management supported by Linux virtualized environments.

This chapter identifies the following:

- Onload and Linux KVM
- Onload and NIC Partitioning
- Onload in a Docker Container

Overview

- Running Onload in a Virtual Machine (VM) or Docker Container means the Onload accelerated application benefits from the inherent isolation policy of the virtualized environment.
- There is minimal degradation of latency and throughput performance. Near native network I/O performance is possible because there is direct hardware access (no hardware emulation) with the guest kernel (and virtualization platform hypervisor) being bypassed.
- Multiple containers/virtual machines can co-exist on the same host and all are isolated from each other.

Onload and Linux KVM

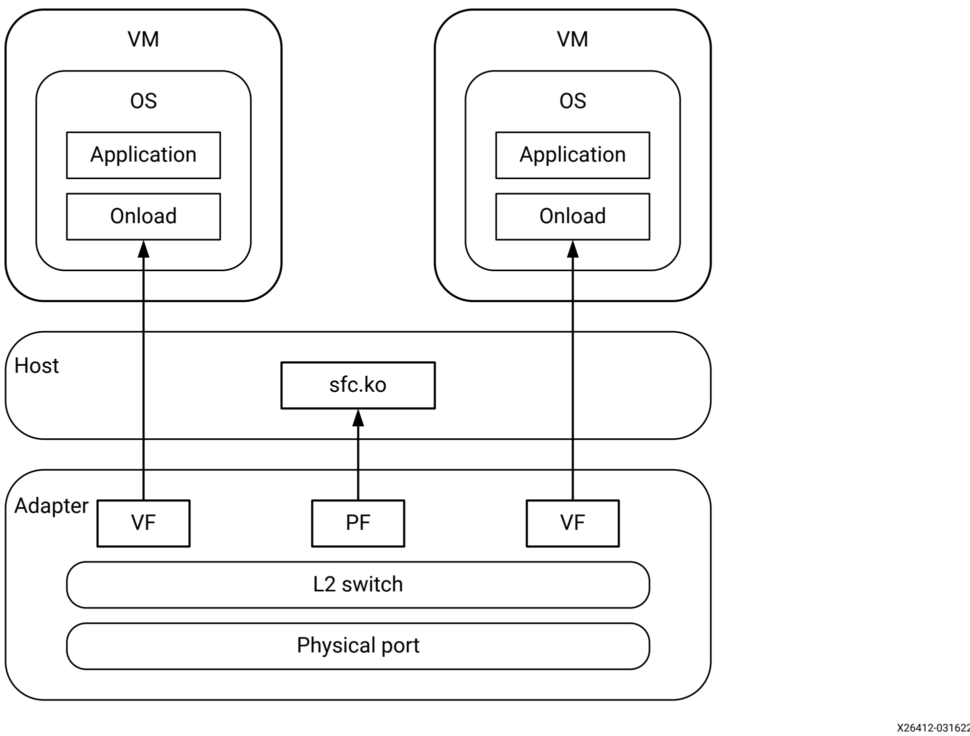
This feature is supported on Solarflare SFN8000 and X2 series adapters.

Onload includes support to accelerate applications running within Linux VMs on a KVM host. Each physical interface on the adapter can be exposed to the host as up to 16 PCIe physical functions (PF) and up to 240 virtual functions (VF). The adapter also supports up to 2048 MSI-X interrupts.

This support requires a VF (or PF) to be exposed directly into the Linux VM – KVM call this network configuration *Network hostdev*. Onload provides user-level access to the adapter via the VF in exactly the same way as is achieved on a non-virtualized Linux install. Firmware on the Solarflare SFN8000 and X2 series adapter configures layer 2 switching capability that supports the transport of network packets between PCI physical functions and virtual functions. This feature supports the transport of network traffic between Onload applications running in different virtual machines. This allows traffic to be replicated across multiple functions and traffic transmitted from one VM can be received on another VM.

The following figure illustrates Onload deployed into the Linux KVM Network Hostdev architecture which exposes Virtual Functions (VF) directly to the VM guest. This configuration allows the Onload data path to fully bypass the host operating system and provides maximum acceleration for network traffic.

Figure 22: **Onload and Network Hostdev Configuration**



To deploy Onload in a Linux KVM:

- As detailed in the *SRIOV* chapter of the *Solarflare Server Adapter User Guide* (SF-103837-CD):
    - Install the Solarflare NET driver version 4.4.1.1017 (or later)
    - Ensure the adapter is using firmware version 4.4.2.1011 (or later)

- Run sfboot to select the full-feature firmware variant, set the switch-mode and identify the required number of VFs:

  ```
  # sfboot firmware-variant=full-feature switch-mode=sriov vf-count=4
  ```

- Reboot the server, so the Linux KVM host can enumerate the VFs.

- Follow the instructions in *Solarflare Server Adapter User Guide* (SF-103837-CD) section *KVM Libvirt network hostdev - Configuration* to:
  - Create a VM
  - Configure the VFs
  - Unbind VFs from the host
  - Pass VFs to the VM.

  Example virsh command line and XML file configuration instructions are provided.

- Install Onload in the VM as in a non-virtualized host - see Building and Installing from a Tarball.

- Set the sfc driver module option `num_vis` to create the number of virtual interfaces. A VI is needed for each Onload stack created on a VF. Driver module options should be set in a user created file (for example `sfc.conf`) in the `/etc/modprobe.d` directory.

  ```
  options sfc num_vis=<NUM>
  ```

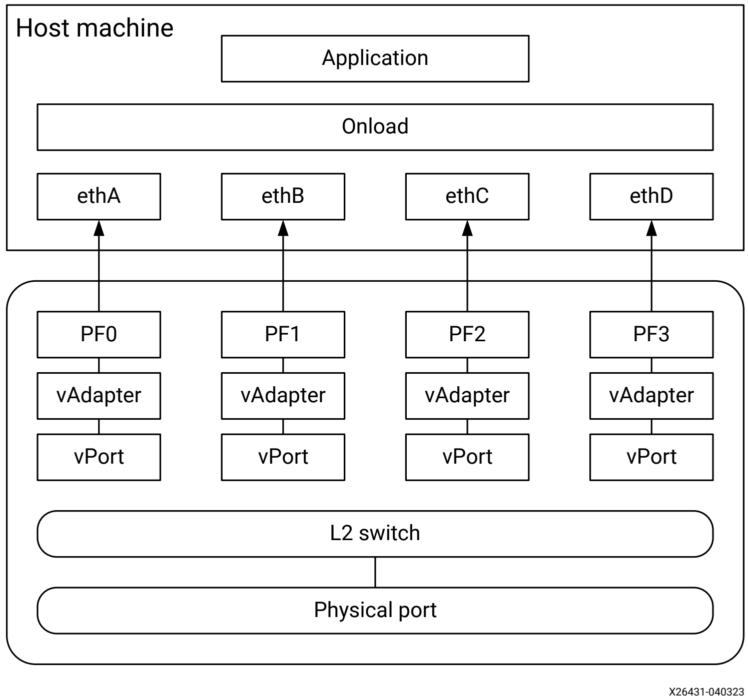
  Note: When using Onload with multiple virtual functions (VF) it is necessary to set the Onload module option `oof_all_ports_required` to zero. See Module Options for details.

The *Solarflare Server Adapter User Guide* is available from the NIC Software and Drivers web page.

Onload and NIC Partitioning

Each physical interface on a Solarflare SFN8000 and X2 series adapter can be exposed to the host as multiple PCIe physical functions (PF). Up to 16 PFs, each having a unique MAC address, are supported per adapter. To Onload, each PF represents a virtual adapter.

Figure 23: **Onload and NIC Partitioning**



On the adapter each PF is backed by a virtual adapter and virtual port - these components are created by the Solarflare NET driver when it finds a partitioned adapter. The PFs can be configured to transparently place traffic on separate VLANS (so each partition is on a separate broadcast domain).

To configure Onload to use the partitioned NIC:

- Ensure the adapter is using firmware version 4.4.2.1011 (minimum)
- Use sfboot to select the full-feature firmware variant
- Use sfboot to partition the NIC into multiple PFs
- Rebooting the host allows the firmware to partition the NIC into multiple PFs.
- To identify which physical port a network interface is using:

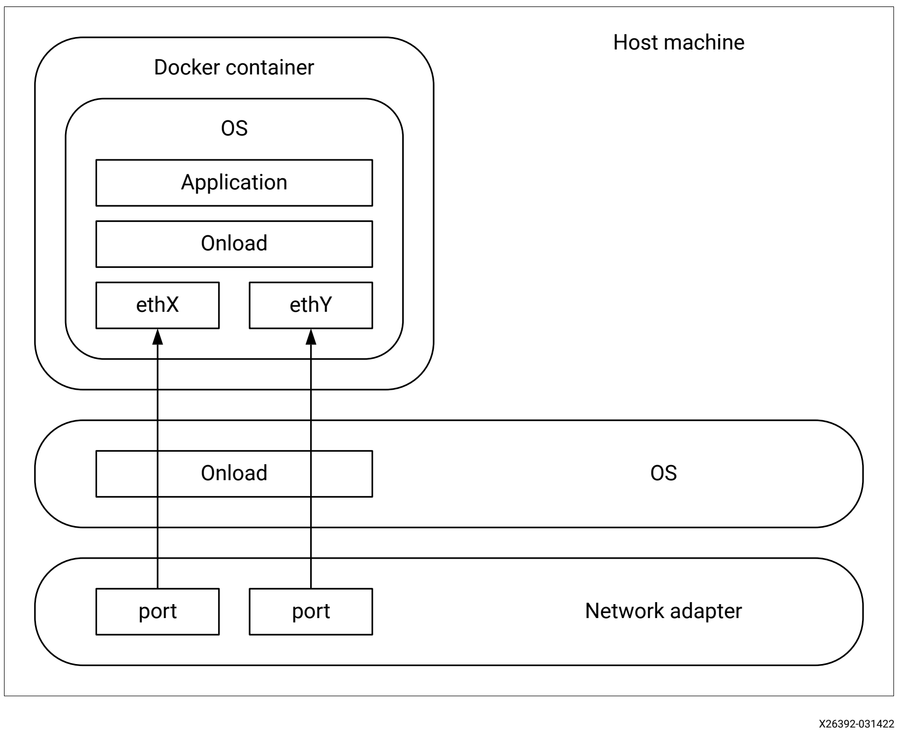
```
# cat /sys/class/net/eth<N>/device/physical_port
```

For complete details of configuring NIC Partitioning refer to the*SRIOV* chapter of the *Solarflare Server Adapter User Guide* (SF-103837-CD), available from the NIC Software and Drivers web page.

Onload in a Docker Container

The following figure illustrates the Onload deployment in a Docker container environment. Only the user-level components are created in the container. Onload in the container uses the Onload drivers installed on the host for network I/O. Network interfaces configured on the host are also visible and usable directly from the container.

Figure 24: **Onload in a Docker Container**



In keeping with the containerization theory, it is envisaged that only a single Onload instance will be running in each container, however, there are no restrictions preventing multiple instances running in the same container.

Pre-Installation

CAUTION! *This install procedure makes the following assumptions - ensure these components are created/installed before continuing:*

- Docker is installed on the host server.
- Identical Onload versions must be installed on the host and in the container.
- The Onload installation in a container must match the Onload installation on the host. Configuration options such as any CI_CFG_* options set in one environment must match those set in the other.

In addition to requiring that char devices nodes for `onload`, `onload_epoll` and `sfc_char` be present in `/dev`, Onload requires that the `procfs` and `sysfs` filesystems be mounted respectively at `/proc` and `/sys`, and that those mounts be in the same network namespace as the stack. These properties are all arranged correctly when containers are created according to the instructions in Installation below.

Installation

1. The docker run command will create a container named onload. The container is created from the centos:latest base image and a bash shell terminal will be started in the container.

   If MACVLAN driver support and namespace support are required, refer to MACVLAN Support below before creating the container.

   ```
   # docker run \
           --net=host \
           --device=/dev/onload \
           --device=/dev/onload_epoll \
           --name=onload \
           -it \
           -v /src/onload-201502.tgz:/tmp/onload-201502.tgz \
           centos:latest \
           /bin/bash
   ```

   The example above copies the `onload-201502.tgz` file from the `/src` directory on the host and placed this file into `/tmp` in the container root file system. *All subsequent commands are run inside the container unless host is specified.*

   Note: The directive `--device=/dev/sfc_char` is required when used with ef_vi.

2. Install required OS tools/packages in the container:

   ```
   # yum install perl autoconf automake libtool tar gcc make net-tools
   ethtool
   ```

   Different docker base images might require additional OS packages installed.

3. Unpack the tarball to build the `onload-<version>` sub-directory:

   ```
   # /usr/bin/tar -zxvf /tmp/onload-201502.tgz
   ```

   Note: It is not possible to use tools/utilities (such as tar) from the host file system on files in the container file system.

4. Change directory to the `onload-<version>/scripts` directory:

   ```
   # cd /tmp/onload-201502/scripts
   ```

5. Build the Onload *user-level* components in the container:

   ```
   # ./onload_build --user
   ```

   If the build process identifies any missing dependencies, return to step 2 to install missing components.

6. Install the Onload *user-level* components in the container:

```
# ./onload_install --userfiles --nobuild
```

   The following warning might appear at the end of the install process, but it is *not necessary* to reload the drivers:

```
onload_install: To load the newly installed drivers run:  onload_tool
reload
```

7. Check Onload installation:

```
# onload
OpenOnload 201502
Copyright 2006-2012 Solarflare Communications, 2002-2005 Level 5 Networks
Built: Feb  5 2015 12:41:04 (release)
Kernel module: 201502

usage:
  onload [options] <command> <command-args>

options:
  --profile=<profile>     -- comma sep list of config profile(s)
  --force-profiles        -- profile settings override environment
  --no-app-handler        -- do not use app-specific settings
  --app=<app-name>        -- identify application to run under onload
  --version               -- print version information
  -v                      -- verbose
  -h --help               -- this help message
```

8. On the host, check that the container has been created and is running:

```
# docker ps -a
```

```

CONTAINER ID  IMAGE          COMMAND      CREATED         STATUS
PORTS   NAMES
e2a12a635359  centos:latest "/bin/bash"  15 seconds ago  Up 14
seconds         onload
```

9. Configure network interfaces.

   Configure network adapter interfaces in the host. Interfaces will also be visible and usable from the container:

```
# ifconfig -a
```

10. Onload is now installed and ready to use in the container.

MACVLAN Support

Onload from 201710 adds supports for network namespaces within Docker containers. Support is also included for the MACVLAN driver and MACVLAN sub-interfaces in container and standard host configurations.

The MACVLAN driver allows a single physical interface to be assigned multiple MAC addresses, creating sub-interfaces, each having a unique MAC address. The hardware address can be randomly generated by the driver, or supplied by the user.

An application running in a Docker container will bind to a specific sub-interface to gain direct access to the network adapter. Onload will accelerate network traffic between the container and the network.

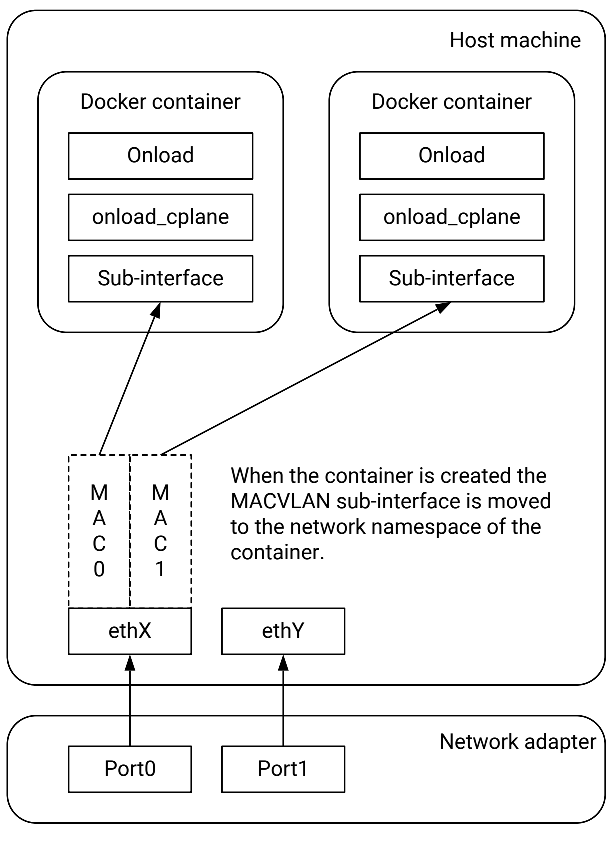
Onload is not able to send packets directly between containers having sub-interfaces from the same parent. Such packets will be delivered between containers only via an underlying switch.

MACVLAN Interface Configurations

Onload will support:

- MACVLAN on top of a supported adapter.
- Nested MACVLAN on top of MACVLAN on top of supported adapter.

Figure 25: **Onload using MACVLAN Sub-interfaces**



Traffic between containers might be delivered depending on the routing configuration/policy of the connected external switch and the configuration settings of EF_MCAST_SEND and EF_MCAST_RECV_HW_LOOP.

Configure MACVLAN for a Docker Network

1. Before creating the Docker container, setup the Docker network:

```
docker network create \
        -d MACVLAN \
        -o parent=<interface> \
        --subnet=<address> \
        --ip-range=<address> \
        my-network
```

   where <interface> is the physical network interface that becomes the parent interface to the created sub-interfaces.

   When the network is created, only the parent interface needs to be present. When the container is created a sub-interface is created and moved to the network namespace of the container.

2. Include the network when creating the Docker container:

```
docker run \
        -it \
        --net=my-network \
        --device=/dev/onload \
        --device=/dev/onload_epoll \
        --device=/dev/sfc_char \
        my-onload-image \
        /bin/bash
```

See the Installation section and examples above.

Create Onload Docker Image

To create a new docker image that includes the Onload installation prior to migration. *All commands are run on the host.*

1. Identify the container (note CONTAINER ID or NAME)

```
# docker ps -a
CONTAINER ID  IMAGE         COMMAND      CREATED       STATUS   PORTS
NAMES
35bfeceb7022  centos:latest  "/bin/bash" 24 hours ago  Exited
onload
```

2. Create new image (this example uses the NAME value)

```
# docker commit -m "installed onload 201502" onload onload:v1
89e95645d5ff1fa02880dee44b433ab577f5a2715daf944fd0b393620d8253f1
```

3. List images

```
# docker images
REPOSITORY   TAG      IMAGE ID        CREATED             VIRTUAL SIZE
onload       v1       89e95645d5ff    28 seconds ago      486 MB
centos       latest   dade6cb4530a    3 days ago          224 MB
```

Migration

The docker `save` command can be used to archive a docker image which includes the Onload installation. This image can then be migrated to other servers having the following configuration:

- Docker is installed and docker service is running
- Host operating system RHEL 7
- The Onload version running on the host must be the same as the migrated image Onload version
- The target server does not need to have the same network adapter types installed.

1. Create a tar file of the container image:

```
# docker save -o <dir path to store image>/<name of image>.tar <current
name of image>
```

Example (store image tar file in host `/tmp` directory):

```
# docker save -o /tmp/dk-onload-201502.tar onload
```

2. The image tar file can then be copied to the target server where it can be loaded with the docker load command:

```
# docker load -i /<path to transferred file>/dk-onload-201502.tar
```

```
# docker images
REPOSITORY   TAG   IMAGE ID       CREATED              VIRTUAL SIZE
onload       v1    303ec2d3e2b5   About an hour ago    486 MB
```

3. Create/run a container from the transferred image.

```
# docker run --net=host --device=/dev/onload --device=/dev/onload_epoll
--name=onload -it onload:v1 /bin/bash
```

When the container has been created, Onload will be running within it.

Note: The directive `--device=/dev/onload_cplane` is required when used with onload-201606 and later releases.

Onload Docker Images

Onload images are not currently available from the default docker registry hub. Images might be made available if there is sufficient customer interest and requirement for this feature.

Copying Files Between Host and Container

The following example demonstrates how to copy files from the host to a container. All commands are run on the host.

1. Get the container Short Name (output truncated):

```
[root@hostname]# docker ps -a
CONTAINER ID
bd1ea8d5526c
```

2. Discover the container Long Name:

```
[root@hostname]# docker inspect -f '{{.Id}}' bd1ea8d5526c
bd1ea8d5526c55df4740de9ba5afe14ed28ac3d127901ccb1653e187962c5156
```

   The container long name can also be discovered using the container name in place of the container identifier.

3. Copy a file to root file system (`/tmp`) on the container:

```
[root@hostname]# cp myfile.txt /var/lib/docker/devicemapper/mnt/
bd1ea8d5526c55df4740de9ba5afe14ed28ac3d127901ccb1653e187962c5156/
rootfs/tmp/myfile.txt
```

Chapter 13

Limitations

IMPORTANT! *Users are advised to read the latest Release Notes distributed with the Onload release for a comprehensive list of known issues.*

Introduction

This chapter outlines configurations that Onload does not accelerate and ways in which Onload can change behavior of the system and applications. It is a key goal of Onload to be fully compatible with the behavior of the regular kernel stack, but there are some cases where behavior deviates.

Resources

Onload uses certain physical resources on the network adapter. If these resources are exhausted, it is not possible to create new Onload stacks and not possible to accelerate new sockets or applications. The `onload_stackdump` utility should be used to monitor hardware resources. Physical resources include:

Virtual NICs

Virtual NICs provide the interface by which a user level application sends and receives network traffic. When these are exhausted it is not possible to create new Onload stacks, meaning new applications cannot be accelerated. However, Solarflare network adapters support large numbers of Virtual NICs, and this resource is not typically the first to become unavailable.

Endpoints

Onload represents sockets and pipes as structures called endpoints. The maximum number of accelerated endpoints permitted by each Onload stack is set with the EF_MAX_ENDPOINTS variable. The stack limit can be reached sooner than expected when syn-receive states (the number of half-open connections) also consume endpoint buffers. Four syn-receive states consume one endpoint. The maximum number of syn-receive states can be limited using the EF_TCP_SYNRECV_MAX variable.

Filters

Filters are used to deliver packets received from the wire to the appropriate application. When filters are exhausted it is not possible to create new accelerated sockets. The general recommendation is that:

- On X3-series adapters, applications do not allocate more than 256 filters or create more than 256 outgoing connections.
- On X2-series adapters, applications do not allocate more than 16384 filters or create more than 16384 outgoing connections.
- On earlier adapters, applications do not allocate more than 4096 filters or create more than 4096 outgoing connections.

The limit does not apply to inbound connections to a listening socket.

Buffer Table

The buffer table provides address protection and translation for DMA buffers. When all buffer resources are exhausted it is not possible to create new Onload stacks, and existing stacks are not able to allocate more DMA buffers.

When hardware resources are exhausted, normal operation of the system should continue, but it will not be possible to accelerate new sockets or applications.

TX, RX Ring Buffer Size

Onload does not obey RX, TX ring sizes set in the kernel, but instead uses the values specified by EF_RXQ_SIZE and EF_TXQ_SIZE. Both default to 512.

Devices

The `efrm` driver used by Onload supports a maximum of 64 devices.

Changes to Behavior

Onload exhibits the following changes to behavior compared with the kernel stack.

Multithreaded Applications Termination

As Onload handles networking in the context of the calling application's thread it is recommended that applications ensure all threads exit cleanly when the process terminates. In particular the `exit()` function causes all threads to exit immediately - even those in critical sections. This can cause threads currently within the Onload stack holding the per stack lock to terminate without releasing this shared lock - this is particularly important for shared stacks where a process sharing the stack could 'hang' when Onload locks are not released.

An unclean exit can prevent the Onload kernel components from cleanly closing the application's TCP connections, a message similar to the following will be observed:

```
[onload] Stack [0] released with lock stuck
```

and any pending TCP connections will be reset. To prevent this, applications should always ensure that all threads exit cleanly.

Thread Cancellation

Unexpected behavior can result when an accelerated application uses a pthread_cancel function. There is increased risk from multi-threaded applications or a PTHREAD_CANCEL_ASYNCHRONOUS thread calling a non-async safe function. Onload users are strongly advised that applications should not use pthread_cancel functions.

Packet Capture

Packets delivered to an application via the accelerated path are not visible to the OS kernel. As a result, diagnostic tools such as tcpdump and wireshark do not capture accelerated packets. The supplied `onload_tcpdump` application does support capture of UDP and TCP packets from Onload stacks - Refer to Appendix G: onload_tcpdump for details.

Firewalls

Packets delivered to an application via the accelerated path are not visible to the OS kernel. As a result, these packets are not visible to the kernel firewall (iptables) and therefore firewall rules will not be applied to accelerated traffic. The onload_iptables feature can be used to enforce Linux iptables rules as hardware filters on the Solarflare adapter, refer to Appendix I: onload_iptables.

Note: Hardware filtering on the network adapter will ensure that accelerated applications receive traffic only on ports to which they are bound.

Socket Visibility to System Tools

With the exception of 'listening' sockets, TCP sockets accelerated by Onload are not visible to the netstat tool. UDP sockets are visible to netstat.

Accelerated sockets appear in the `/proc` directory as symbolic links to `/dev/onload`. Tools that rely on `/proc` will probably not identify the associated file descriptors as being sockets. Refer to File Descriptors, Stacks, and Sockets for more details.

Accelerated sockets can be inspected in detail with the Onload `onload_stackdump` tool, which exposes considerably more information than the regular system tools. For details of `onload_stackdump` refer to Appendix E: onload_stackdump.

Signals

If an application receives a `SIGSTOP` signal, it is possible for the processing of network events to be stalled in an Onload stack used by the application. This happens if the application is holding a lock inside the stack when the application is stopped, and if the application remains stopped for a long time, this can cause TCP connections to time-out.

A signal which terminates an application can prevent threads from exiting cleanly. Refer to Multithreaded Applications Termination for more information.

Undefined content can result when a signal handler uses the third argument (ucontext) and if the signal is postponed by Onload. To avoid this, use the Onload module option `safe_signals_and_exit=0` or use EF_SIGNALS_NOPOSTPONE to prevent specific signals being postponed by Onload.

Onload and IP_MULTICAST_TTL

Onload will act in accordance with RFC 791 when it comes to the IP_MULTICAST_TTL setting. Using Onload, if IP_MULTICAST_TTL=0, packets will never be transmitted on the wire.

This differs from the Linux kernel where the following behavior has been observed:

Kernel - IP_MULTICAST_TTL 0 - if there is a local listener, packets will not be transmitted on the wire.

Kernel - IP_MULTICAST_TTL 0 - if there is NO local listener, packets will always be transmitted on the wire.

Source/Policy Based Routing

OpenOnload 201710 / EnterpriseOnload 6.0

The Onload 201710 and EnterpriseOnload 6.0 releases include support for source based policy routing for unicast and multicast packets. The following are supported:

- source ip address
- destination ip address
- outgoing interface (SO_BINDTODEVICE)
- TOS (Type of Service)

Policy rules based on other criteria are not supported and will be ignored by Onload.

Earlier Onload Versions

Earlier Onload versions do not support source based or policy based routing. Whereas the Linux kernel will select a route and interface based on routing metrics, Onload will select any of the valid routes and Onload interfaces to a destination that are available.

The EF_TCP_LISTEN_REPLIES_BACK environment variable provides a pseudo source-based routing solution. This option forces a reply to an incoming SYN to ignore routes and reply to the originating network interface.

Enabling this option will allow new TCP connections to be setup, but does not guarantee that all replies from an Onloaded application will go via the receiving interface - and some re-ordering of the routing table might be needed to guarantee this OR an explicit route (to go via the receiving interface) should be added to the routing table.

Routing Table Metrics

Onload, from version 201606, introduced support for routing table metrics, therefore, if two entries in the routing table will route traffic to the destination address, the entry with the best metric will be selected even if that means routing over an unaccelerated interface.

Multipath Routes

Onload does not support a multipath route simultaneously via Onload-accelerated and non-Onload-accelerated interfaces. The paths in a multipath route should either all be acceleratable, or all be non-acceleratable.

Reverse Path Filtering

Onload does not support Reverse Path Filtering. When Onload cannot route traffic to a remote endpoint over an accelerated interface (no suitable route table entry), the traffic will be handled via the kernel.

SO_REUSEPORT

Onload vs. kernel behavior is described in Application Clustering.

Thread Safe

Onload assumes that file descriptor modifications are thread-safe and that file descriptors are not concurrently modified by different threads. Concurrent access should not cause problems. This is different from kernel behaviour and users should set `EF_FDS_MT_SAFE=0` if the application is not considered thread-safe.

Similar consideration should be given when using `epoll()` where default concurrency control are disabled in Onload. Users should set `EF_EPOLL_MT_SAFE=0`.

Control of Duplicated Sockets

When a socket has been duplicated, for example, using fork(), and where the parent fd is controlled by the kernel, the child fd controlled by Onload. Changes by the kernel using fcntl() to modify flags such as O_NONBLOCK will not be reflected in the Onload socket.

UDP Sockets shutdown()

When a kernel UDP socket is unconnected, a shutdown() call will prompt a blocking recv() operation on the socket to successfully complete. When an Onload UDP socket is unconnected, a shutdown() call does not successfully complete a blocking recv() call and thereafter the socket fd cannot be reused.

When a UDP socket is connected, kernel and Onload behavior is the same, a shutdown() call will prompt a blocking recv() operation to complete successfully.

***Note*:** Kernel behavior might differ between different kernel versions.

SOF_TIMESTAMPING_OPT_ID

Onload does not support the `SOF_TIMESTAMPING_OPT_ID` socket option.

Limits to Acceleration

Onload has the following limits to acceleration.

IP Fragmentation

Fragmented IP traffic is not accelerated by Onload on the receive side, and is instead received transparently via the kernel stack. IP fragmentation is rarely seen with TCP, because the TCP/IP stacks segment messages into MTU-sized IP datagrams. With UDP, datagrams are fragmented by IP if they are too large for the configured MTU. Refer to Fragmented UDP for a description of Onload behavior.

Broadcast Traffic

Broadcast sends and receives function as normal but will not be accelerated. Multicast traffic can be accelerated.

IPv6 Traffic

The following are not supported when Onload is accelerating IPv6 traffic:

- Multicast traffic is not accelerated
- Fragmented packets are not accelerated
- IPv6 specific socket options are not supported except IPV6_V6ONLY
- SYN cookies are not supported
- Flow information is neither set not considered
- Opportunistic packet processing

IPv6 is not supported when using the Onload extensions API: delegated send or zero-copy send.
TCP NOP Options

Onload will silently discard packets that include IP header No Operation (NOP) options. Discards will not increment drop packet counters.

Onload will process packets that include NOP options in the TCP header, but the options themselves will be ignored.

IPv6 Kernel Support

If the kernel does not support IPv6, the following error message is output:

```
sock_create(10, <1 or 2>, 0) failed (-97)
```

where

- -97 is the error code EAFNOSUPPORT (Address family not supported by protocol)
- the other numbers indicate an IPv6 TCP or UDP socket

One possible cause of this error is when using Java which often creates IPv6 sockets alongside IPv4 sockets.

Raw Sockets

Raw Socket sends and receives function as normal but will not be accelerated.

Socketpair and UNIX Domain Sockets

Onload will intercept, but does not accelerate the `socketpair()` system call. Sockets created with `socketpair()` will be handled by the kernel. Onload also does not accelerate UNIX domain sockets.

UDP sendfile()

The UDP `sendfile()`method is not currently accelerated by Onload. When an Onload accelerated application calls `sendfile()` this will be handled seamlessly by the kernel.

Statically Linked Applications

Onload will not accelerate statically linked applications. This is due to the method in which Onload intercepts libc function calls (using `LD_PRELOAD`).

Local Port Address

Onload is limited to `OOF_LOCAL_ADDR_MAX` number of local interface addresses. A local address can identify a physical port or a VLAN, and multiple addresses can be assigned to a single interface where each address contributes to the maximum value. Users can allocate additional local interface addresses by increasing the compile time constant `OOF_LOCAL_ADDR_MAX` in the `/src/lib/efthrm/oof_impl.h` file and rebuilding Onload. In onload-201205 `OOF_LOCAL_ADDR_MAX` was replaced by the onload module option `max_layer2_interfaces`.

Bonding, Link Aggregation

- Onload will only accelerate traffic over 802.3ad and active-backup bonds.
- Onload will not accelerate traffic if a bond contains any slave interfaces that are not supported network devices.
- Adding an unsupported network device to a bond that is currently accelerated by Onload can result in unexpected results such as connections being reset.
- Acceleration of bonded interfaces in Onload requires a kernel configured with `sysfs` support and a bonding module version of 3.0.0 or later.
- Traffic on the backup link is not filtered.

In cases where Onload will not accelerate the traffic it will continue to work via the OS network stack.

VLANs

The following limitations apply to VLANs:

- Onload will only accelerate traffic over VLANs where the master device is either a Solarflare network device, or over a bonded interface that is accelerated. If the VLAN's master is accelerated, then so is the VLAN interface itself.
- Nested VLAN tags are not accelerated, but will function as normal.
- The ifconfig command will return inconsistent statistics on VLAN interfaces (not master interface).
- When a Solarflare VLAN tagged interface is subsequently placed in a bond, the interface will continue to be accelerated, but the bond is not accelerated.
- Using SFN8000 and X2 series adapters with the low-latency firmware variant, the following limitation applies:

  Hardware filters installed by Onload on the adapter will only act on the IP address and port, but not the VLAN identifier. Therefore if the same IP address:port combination exists on different VLAN interfaces, only the first interface to install the filter will receive the traffic.

  This limitation does not apply to SFN8000 and X2 series adapters using the full-feature firmware variant.

In cases where Onload will not accelerate the traffic it will continue to work via the OS network stack.

For more information and details and configuration options refer to the *Setting Up VLANs* section of the *Solarflare Server Adapter User Guide*.

Ethernet Bridge Configuration

Onload does not currently support acceleration of interfaces added to an Ethernet bridge configured/added with the Linux `brctl` command.

TCP RTO During Overload Conditions

Using Onload, under very high load conditions an increased frequency of TCP retransmission timeouts (RTOs) might be observed. This has the potential to occur when a thread servicing the stack is descheduled by the CPU whilst still holding the stack lock thus preventing another thread from accessing/polling the stack. A stack not being serviced means that ACKs are not received in a timely manner for packets sent, resulting in RTOs for the unacknowledged packets and increased jitter on the Onload stack.

Enabling the per stack environment variable EF_INT_DRIVEN can reduce the likelihood of this behavior and reduce jitter by ensuring the stack is serviced promptly. TCP with Jumbo Frames

When using jumbo frames with TCP, Onload will limit the MSS to 2048 bytes to ensure that segments do not exceed the size of internal packet buffers.

This should present no problems unless the remote end of a connection is unable to negotiate this lower MSS value.

Packet Loss on the Transmission Path

Occasionally Onload needs to send a packet, which would normally be accelerated, via the kernel. This occurs when there is no destination address entry in the ARP table or to prevent an ARP table entry from becoming stale.

By default, the Linux sysctl, `unres_qlen`, will enqueue three packets per unresolved address when waiting for an ARP reply, and on a server subject to a very high UDP or TCP traffic load this can result in packet loss on the transmit path and packets being discarded.

The `unres_qlen` value can be identified using the following command:

```
sysctl -a | grep unres_qlen
net.ipv4.neigh.eth2.unres_qlen = 3
net.ipv4.neigh.eth0.unres_qlen = 3
net.ipv4.neigh.lo.unres_qlen = 3
net.ipv4.neigh.default.unres_qlen = 3
```

Changes to the queue lengths can be made permanent in the `/etc/sysctl.conf` file. It is recommended to set the `unres_qlen` value to at least 50.

If packet discards are suspected, this extremely rare condition can be indicated by the `cp_defer` counter produced by the `onload_stackdump lots` command on UDP sockets or from the `unresolved_discards` counter in the Linux `/proc/net/stat arp_cache` file.

TCP Packets with Unsupported Routing

If TCP packets are received over an Onload accelerated interface, but Onload cannot find a suitable Onload accelerated return route, no response will be sent resulting in the connection timing out.

Application Clustering

For details of Application Clustering, refer to Application Clustering.

- Onload matches the Linux kernel implementation such that clustering is not supported for multicast traffic and where setting of `SO_REUSEPORT` has the same effect as `SO_REUSEADDR`.
- Calling `connect()` on a TCP socket which was previously subject to a `bind()` call is not currently supported. This will be supported in a future release.
- An application cluster will not persist over adapter/server/driver reset. Before restarting the server or resetting the adapter the Onload applications should be terminated.
- The environment variable EF_CLUSTER_RESTART determines the behavior of the cluster when the application process is restarted - refer to EF_CLUSTER_RESTART in Appendix A: Parameter Reference.
- If the number of sockets in a cluster is less than EF_CLUSTER_SIZE, a portion of the received traffic will be lost.
- There is little benefit when clustering involves a TCP loopback listening socket as connections will not be distributed amongst all threads. A non-loopback listening socket - which might occasionally get some loopback connections can benefit from Application Clustering.

Duplicate IP or MAC Addresses

Onload does not support multiple interfaces with the same IP address or MAC address.

Known Issues with Epoll

Onload supports different implementations of epoll controlled by the EF_UL_EPOLL environment variable - see Multiplexed I/O for configuration details.

There are various limitations and differences in Onload vs. kernel behavior - refer to Multiplexed I/O for details.

- When using EF_UL_EPOLL=1 or 3, it has been identified that the behavior of `epoll_wait()` differs from the kernel when the `EPOLLONESHOT` event is requested, resulting in two 'wakeups' being observed, one from the kernel and one from Onload. This behavior is apparent on `SOCK_DGRAM` and `SOCK_STREAM` sockets for all combinations of `EPOLLONESHOT`, `EPOLLIN` and `EPOLLOUT` events. This applies for all types of accelerated sockets. EF_EPOLL_CTL_FAST is enabled by default and this modifies the semantics of epoll. In particular, it buffers up calls to `epoll_ctl()` and only applies them when `epoll_wait()` is called. This can break applications that do `epoll_wait()` in one thread and `epoll_ctl()` in another thread. The issue only affects EF_UL_EPOLL=2 and the solution is to set EF_EPOLL_CTL_FAST=0 if this is a problem. The described condition does not occur if EF_UL_EPOLL=1 or EF_UL_EPOLL=3.
- When EF_EPOLL_CTL_FAST is enabled and an application is testing the readiness of an epoll file descriptor without actually calling `epoll_wait()`, for example by doing epoll within epoll() or epoll within `select()`, if one thread is calling `select()` or `epoll_wait()` and another thread is doing `epoll_ctl()`, then EF_EPOLL_CTL_FAST should be disabled. This applies when using EF_UL_EPOLL 1, 2 or 3.

  If the application is monitoring the state of the epoll file descriptor indirectly, for example by monitoring the epoll fd with poll, then EF_EPOLL_CTL_FAST can cause issues and should be set to zero.

  To force Onload to follow the kernel behavior when using the `epoll_wait()` call, the following variables should be set:

  `EF_UL_EPOLL=2`

  `EF_EPOLL_CTL_FAST=0`

  `EF_EPOLL_CTL_HANDOFF=0` (when using `EF_UL_EPOLL=1`)
- A socket should be removed from an epoll set only when all references to the socket are closed.

  With EF_UL_EPOLL=1 (default) or EF_UL_EPOLL=3, a socket is removed from the epoll set if the file descriptor is closed, even if other references to the socket exist. This can cause problems if file descriptors are duplicated using `dup(), dup2() or fork()`. For example:

  ```
  s = socket();
  s2 = dup(s);
  epoll_ctl(epoll_fd, EPOLL_CTL_ADD, s, ...);
  close(s);  /* socket referenced by s is removed from epoll set when using
  onload */
  ```

  Workaround is set EF_UL_EPOLL=2.
- When Onload is unable to accelerate a connected socket, for example because no route to the destination exists which uses a Solarflare interface, the socket will be handed off to the kernel and is removed from the epoll set. Because the socket is no longer in the epoll set, attempts to modify the socket with `epoll_ctl()` will fail with the `ENOENT` (descriptor not present) error. The described condition does not occur if EF_UL_EPOLL=1 or 3.

- If an epoll file descriptor is passed to the `read()` or `write()` functions these will return a different errorcode than that reported by the kernel stack. This issue exists for all implementations of epoll.
- When `EPOLLET` is used and the event is ready, `epoll_wait()` is triggered by ANY event on the socket instead of the requested event. This issue should not affect application correctness.
- Users should be aware that if a server is overclocked the `epoll_wait()` timeout value will increase as CPU MHz increases resulting in unexpected timeout values. This has been observed on Intel based systems and when the Onload epoll implementation is EF_UL_EPOLL=1 or 3. Using EF_UL_EPOLL=2 this behavior is not observed.
- On a spinning thread, if epoll acceleration is disabled by setting EF_UL_EPOLL=0, sockets on this thread will be handed off to the kernel, but latency will be worse than expected kernel socket latency.
- To ensure that non-accelerated file descriptors are checked in poll and select functions, the following options should be disabled (set to zero):
- EF_SELECT_FAST and EF_POLL_FAST
- When using `poll()` and `select()` calls, to ensure that non-accelerated file descriptors are checked when there are no events on any accelerated descriptors, set the following options:
- EF_POLL_FAST_USEC and EF_SELECT_FAST_USEC, setting both to zero.

Nested Epoll Sets

When an epoll set includes accelerated sockets and is nested inside another epoll set, the following can occur with the outer set:

- it might not always get notified about socket readiness
- after a socket becomes ready, its state cannot be cleared.

This limitation is known to affect EF_UL_EPOLL=3.

Timing Issues and Spinning

Onload users should consider that as different software is being run, timings will be affected which can result in unexpected scheduling behavior and memory use. Spinning applications, in particular, require a dedicated core per spinning Onload thread.

Configuration Issues

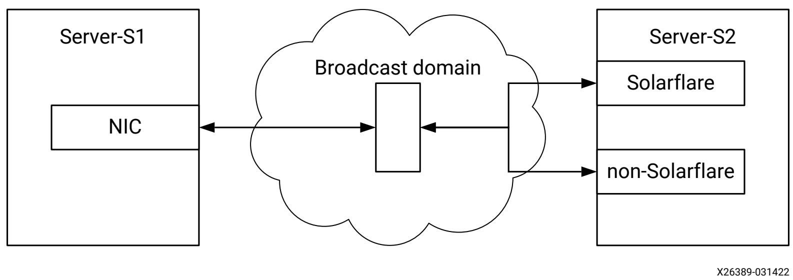
Onload has the following configuration issues.

Mixed Adapters Sharing a Broadcast Domain

Onload should not be used when Solarflare and non-Solarflare interfaces in the same network server are configured in the same broadcast domain, as depicted by the following figure.

Note: A Broadcast domain can be a local network segment or VLAN.

Figure 26: **Mixed Adapters Sharing a Broadcast Domain**



When an originating server (S1) sends an ARP request to a remote server (S2) having more than one interface within the same broadcast domain, ARP responses from S2 will be generated from all interfaces and it is non-deterministic which response the originator uses. When Onload detects this situation, it prompts a message identifying '`duplicate claim of ip address`' to appear in the (S1) host `syslog` as a warning of potential problems.

Problem 1

Traffic from S1 to S2 can be delivered through either of the interfaces on S2, irrespective of the IP address used. This means that if one interface is accelerated by Onload and the other is not, you might or might not get acceleration.

To resolve the situation (for the current session) issue the following command:

```
echo 1 >/proc/sys/net/ipv4/conf/all/arp_ignore
```

or to resolve it permanently add the following line to the `/etc/sysctl.conf` file:

```
net.ipv4.conf.all.arp_ignore = 1
```

and run the `sysctl` command for this be effective.

```
sysctl -p
```

These commands ensure that an interface will only respond to an ARP request when the IP address matches its own. Refer to the Linux documentation `Linux/Documentation/networking/ip-sysctl.txt` for further details.

Problem 2

A more serious problem arises if one interface on S2 carries Onload accelerated TCP connections and another interface on the same host and same broadcast domain is non-Solarflare:

A TCP packet received on the non-Solarflare interface can result in accelerated TCP connections being reset by the kernel stack and therefore appear to the application as if TCP connections are being dropped/terminated at random.

To prevent this situation the Solarflare and non-Solarflare interfaces should not be configured in the same broadcast domain. The solution described for Problem 1 above can reduce the frequency of Problem 2, but does not eliminate it.

TCP packets can be directed to the wrong interface because:

- the originator S1 needs to refresh its ARP table for the destination IP address - so sends an ARP request and subsequently directs TCP packets to the non-Solarflare interface
- a switch within the broadcast domain broadcasts the TCP packets to all interfaces.

IGMP Operation and Multicast Process Priority

It is important that the priority of processes using UDP multicast do not have a higher priority than the kernel thread handling the management of multicast group membership.

Failure to observe this could lead to the following situations:

1. Incorrect kernel IGMP operation.
2. The higher priority user process is able to effectively block the kernel thread and prevent it from identifying the multicast group to Onload which will react by dropping packets received for the multicast group.

A combination of indicators can identify this:

- ethtool reports good packets being received while multicast mismatch does not increase.
- ifconfig identifies data is being received.
- onload_stackdump will show the `rx_discard_mcast_mismatch` counter increasing.

Lowering the priority of the user process will remedy the situation and allow the multicast packets through Onload to the user process.

Dynamic Loading

If the onload library `libonload` is opened with `dlopen()` and closed with `dlclose()` it can leave the application in an unpredictable state. Users are advised to use the `RTLD_NODELETE` flag to prevent the library from being unloaded when `dlclose()` is called.

Huge Pages with IPC Namespace

Huge page support should not be enabled if the application uses IPC namespaces and the `CLONE_NEWIPC` flag. Failure to observe this might result in a segfault.

Huge Pages with Shared Stacks

Processes having the same UID, which share an Onload stack, should not attempt to use huge pages. Refer to Stack Sharing for limitation details.

Huge Page Size

When using huge pages, it is recommended to avoid setting the page size greater than 2 MB. A failure to observe this could lead to Onload unable to allocate further buffer table space for packet buffers.

Huge Pages and shmmni

Users should ensure that the number of system wide shared memory segments (shmmni) exceeds the number of huge pages required.

- To identify current shmmni setting:

```
# cat /proc/sys/kernel/shmmni
```

- To set (no reboot required - but not permanent):

```
# echo 8000 > /proc/sys/kernel/shmmni
```

- To set (permanent - reboot required):

```
# echo "kernel.shmmni=8000" >> /etc/sysctl.conf
```

For example, if 4000 huge pages are required, increase the current shmmni value by 4000.

Use of vfork() in Java 7 Applications

Onload accelerated Java 7 applications that call `vfork()` should set the environment variable EF_VFORK_MODE=2 and thereafter the application should not create sockets or accelerated pipes in `vfork()` child before exec.

PIO Not Supported in KVM/ESXi

Due to limitations with write-combine mapping in a virtual guest environment, PIO is not currently supported for Onload applications running in a virtual machine in KVM or ESXi.

Users should ensure that EF_PIO is set to 0 for all Onload stacks running in VMs.

IP_MTU_DISCOVER Socket Option

Onload does not support the `IP_PMTUDISC_INTERFACE` and `IP_PMTUDISC_OMIT` values for the `IP_MTU_DISCOVER` socket option.

AMD

Chapter 14

Onload Change History

This chapter provides a brief history of changes, additions and removals to Onload releases affecting Onload behavior and Onload environment variables.

The *OOL* column identifies the OpenOnload release supporting the feature. The *EOL* column identifies the EnterpriseOnload release supporting the feature (*NS* = not supported).

Mapping Onload Versions

The following table maps major EnterpriseOnload releases to the closest functionally equivalent OpenOnload release. Users should always also refer to the *Release Notes* and *Change Logs* to identify feature support in the Enterprise releases.

Table 38: **Onload Version Mapping**

OpenOnload	EnterpriseOnload
201011-u1	1.0
201109-u2	2.0
201310-u2	3.0
201502-u2	4.0
201606-u1	5.0
201811	6.0
201811-u1	6.0

Notes:

1. Version numbers are common from 7.0 onwards.

Features

The following table shows new feature availability in Onload releases from OpenOnload 7.0 onwards.

Table 39: **Feature Availability**

Feature	OOL	EOL	Description/Notes
Controlling access to RX queues on X3522	8.1	8.1	—
Hardware filter IDs on X3522	8.1	8.1	—
Transmit path warming on X3522	8.1	8.1	—
X3 series adapters supported	8.0	8.0	—
TCPDirect multi-send	7.0	7.1	See *TCPDirect User Guide*.
IPv6 acceleration	7.0	7.1	See IPv6 Traffic
Equal Cost Multipath routing	7.0	7.1	—
External timestamps received in packet trailers	7.0	7.1	Refer to onload_timestamping_request
eXpress Data Path filtering	7.0	7.1	See eXpress Data Path (XDP)
IPVLANs	7.0	7.1	IPVLAN
Installation of Onload control plane as non-root user	7.0	6.0	Default behavior

Environment Variables

The following table shows changes to environment variables in Onload releases from OpenOnload 7.0 onwards.

Table 40: **Environment Variables**

Variable	OOL	EOL	Changed	Notes
EF_CHALLENGE_ACK_LIMIT	—	—	8.1	Default and maximum increased to INT_MAX to match current Linux kernels.
EF_AF_XDP_ZEROCOPY	8.0	8.0	—	Enables zerocopy on AF_XDP NICs.
EF_CLUSTER_SIZE	—	—	8.0	Defaults to `0`, which disables clustering.
EF_COMPOUND_PAGES_MODE	—	—	8.0	Value `1` is obsolete.
EF_EVS_PER_POLL	—	—	8.0	When EF_POLL_IN_KERNEL is set the default value is `192`.
EF_ICMP_PKTS	8.0	8.0	—	Maximum number of ICMP messages which can be queued to one Onload stack.
EF_IRQ_MODERATION	—	—	8.0	Removed.
EF_PACKET_BUFFER_MODE	—	—	8.0	Modes `1` and `3` removed.
EF_RXQ_MIN	—	—	8.0	Minimum is `0`.
EF_RXQ_SIZE	—	—	8.0	Minimum is `0`.
EF_CTPIO_MAX_FRAME_LEN	—	—	7.1.3	Maximum value is `4092`.

Table 40: **Environment Variables** *(cont'd)*

Variable	OOL	EOL	Changed	Notes
EF_TCP_URG_MODE	—	—	7.1.3	Default is `ignore`.
EF_TXQ_LIMIT	—	—	7.1.2	Removed.
EF_TXQ_RESTART	—	—	7.1.2	Removed.
EF_TCP_COMBINE_SENDS_MODE	7.1.1	7.1	—	Controls how Onload fills packets in the TCP send buffer.
EF_UDP_CONNECT_HANDOVER	—	—	7.1	This option now also accepts a value of `2`, which will cause all UDP sockets to be handed over when calling `connect()`, regardless of whether the socket could have been accelerated.
EF_CHALLENGE_ACK_LIMIT	7.0	7.1	—	Limit the number of "challenge ACK packets" sent. See Limit Challenge ACK Rate.
EF_DEFER_ARP_MAX	7.0	7.1	—	Maximum packets to keep while resolving MAC address.
EF_DEFER_ARP_TIMEOUT	7.0	7.1	—	Time to keep packets and try to resolve MAC address.
EF_INVALID_ACK_RATELIMIT	7.0	7.1	—	Limit the ACKs sent because of invalid incoming TCP packet. See Limit Duplicate ACK Rate.
EF_POLL_IN_KERNEL	7.0	7.1	—	Do polling of eventq in kernel when using mode 3.
EF_RX_TIMESTAMPING_ORDERING	7.0	7.1	—	Select the timestamps to use to order received packets.
EF_AUTO_FLOWLABELS	7.0	7.1	—	Defines whether to generate non-zero IPv6 flow labels. Defaults to the sysctl net.ipv6.auto_flowlabels setting.
EF_USE_DSACK	7.0	7.1	—	Controls use of Duplicate Selective Acknowledgment. Defaults to 1 (on).

Module Options

Onload has module options:

- To list all onload module options:

```
# modinfo onload
# modinfo onload_cplane
```

- Onload module options can be set in a user-created file (for example `onload.conf`) in the `/etc/modprobe.d` directory. For example:

```
options onload max_layer2_interfaces=16
options onload_cplane max_layer2_interfaces=16
```

The following table shows changes to module options in Onload releases from OpenOnload 7.0 onwards.

Table 41: **Module Options**

Option	OOL	EOL	Changed	Notes
`cplane_track_xdp`	7.1.2	7.1	—	Track XDP programs linked to network interfaces. It is needed for EF_XDP_MODE=compatible mode to function properly
`cplane_use_prefsrc_as_local`	7.1	7.1	—	Use a preferred source of any accelerated route in the same way as an address assigned to accelerated interface.
`oo_accelerate_veth`	7.0	7.1	—	Controls whether Onload will accelerate traffic over veth interfaces.

***Note*:** The user should always refer to the Onload distribution *Release Notes* and *Change Log*. These are available from the NIC Software and Drivers web page.

Adapter Net Drivers

The following table identifies the Solarflare adapter net driver included in Onload releases from OpenOnload 7.0 onwards.

Table 42: **Net Drivers**

OOL	EOL	Net Driver	Notes
8.1.0.15	8.1.0.3	5.3.14.1019	Linux kernels 4.15 - 6.3
8.0.2.51	NS	5.3.13.1006	Linux kernels 4.15 - 5.18
8.0.1.39	NS	5.3.13.1006	Linux kernels 4.15 - 5.17
8.0.0.34	NS	5.3.12.1023	Linux kernels 4.15 - 5.15
7.1.3.202	7.1.3.4	4.15.14.1002	Linux kernels 4.4 - 5.12
7.1.2.141	7.1.2.3	4.15.12.1008	Linux kernels 4.4 - 5.12
7.1.1.75	7.1.1.2	4.15.12.1008	Linux kernels 4.4 - 5.9
7.1.0.218	NS	4.15.6.1003	Linux kernels 3.10 - 5.4
NS	6.0.6	4.15.14.1001	—
NS	6.0.5	4.15.12.1008	—
NS	6.0.4	4.15.6.1004	—
NS	6.0.3	4.15.6.1000	—
NS	6.0.2	4.15.5.1003	—
7.0.0.173	NS	4.15.4.1003	Linux kernels 3.10 - 5.2

AMD

Appendix A

Parameter Reference

This appendix gives reference descriptions of parameters that can be used with Onload.

Parameter List

The parameter list details the following:

- The environment variable used to set the parameter.
- Parameter name: the name used by `onload_stackdump`.
- The default, minimum and maximum values.
- Whether the variable scope applies per-stack or per-process.
- Description.

EF_ACCEPTQ_MIN_BACKLOG

- **Name:** `acceptq_min_backlog`
- **Default:** `1`
- **Scope:** per-stack

Sets a minimum value to use for the 'backlog' argument to the `listen()` call. If the application requests a smaller value, use this value instead.

EF_ACCEPT_INHERIT_NONBLOCK

- **Name:** `accept_force_inherit_nonblock`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

If set to `1`, TCP sockets accepted from a listening socket inherit the `O_NONBLOCK` flag from the listening socket.

EF_AF_XDP_ZEROCOPY

- **Name:** `af_xdp_zerocopy`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enables zerocopy on AF_XDP NICs. Support for zerocopy is required.

EF_AUTO_FLOWLABELS

- **Name:** `auto_flowlabels`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `3`
- **Scope:** per-process

If set to `1`, will generate non-zero IPv6 flow labels. Defaults to sysctl net.ipv6.auto_flowlabels setting.

EF_BINDTODEVICE_HANDOVER

- **Name:** `bindtodevice_handover`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Hand sockets over to the kernel stack that have the `SO_BINDTODEVICE` socket option enabled.

EF_BURST_CONTROL_LIMIT

- **Name:** `burst_control_limit`
- **Default:** `0`
- **Scope:** per-stack

If non-zero, limits how many bytes of data are transmitted in a single burst. This can be useful to avoid drops on low-end switches which contain limited buffering or limited internal bandwidth. This is not usually needed for use with most modern, high-performance switches.

EF_BUZZ_USEC

- **Name:** `buzz_usec`
- **Default:** `0`
- **Scope:** per-stack

Sets the timeout in microseconds for lock buzzing options. Set to zero to disable lock buzzing (spinning). Will buzz forever if set to `-1`. Also set by the EF_POLL_USEC option.

EF_CHALLENGE_ACK_LIMIT

- **Name:** `challenge_ack_limit`
- **Default:** `INT_MAX`
- **Minimum:** `0`
- **Maximum:** `INT_MAX`
- **Scope:** per-stack

Limit the number of “challenge ACK packets” sent as part of TCP blind window attack mitigation, RFC 5961; in packets per second. The limitation applies for each Onload stack separately.

The value from `/proc/sys/net/ipv4/tcp_challenge_ack_limit` is used by default.

EF_CLUSTER_HOT_RESTART

- **Name:** `cluster_hot_restart_opt`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option controls whether or not clusters support the hot/seamless restart of applications. Enabling this reuses existing stacks in the cluster to allow up to two processes per stack to bind to the same port simultaneously.

***Note*:** It is required there will be as many new sockets on the port as old ones; traffic will be lost otherwise when the old sockets close.

- `0` - disable per-port stack sharing (default)
- `1` - enable per-port stack sharing for hot restarts.

EF_CLUSTER_IGNORE

- **Name:** `cluster_ignore`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

EF_CLUSTER_IGNORE is deprecated, use EF_CLUSTER_SIZE=0 to disable clustering.

When set, this option instructs Onload to ignore attempts to use clusters and effectively ignore attempts to set `SO_REUSEPORT`.

EF_CLUSTER_NAME

- **Name:** `cluster_name`
- **Default:** none
- **Minimum:** none
- **Maximum:** none
- **Scope:** per-process

This option sets the name for an Onload stack that is created when using clusters. The name should have the following maximum length:

- 5 characters in scalable mode
- 7 characters in normal mode with EF_CLUSTER_SIZE ≥ 10
- 8 characters in normal mode with EF_CLUSTER_SIZE < 10.

EF_CLUSTER_RESTART

- **Name:** `cluster_restart_opt`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option controls the behavior when recreating a stack (e.g. due to restarting a process) in an `SO_REUSEPORT` cluster and it encounters a resource limitation such as an orphan stack from the previous process:

- `0` - return an error
- `1` - terminate the orphan to allow the new process to continue.

EF_CLUSTER_SIZE

- **Name:** `cluster_size`
- **Default:** `0`
- **Minimum:** `0`
- **Scope:** per-process

If use of `SO_REUSEPORT` creates a cluster, this option specifies size of the cluster to be created. This option has no impact if use of `SO_REUSEPORT` joins a cluster that already exists.

***Note*:** If fewer sockets than specified here join the cluster, then some traffic will be lost. Refer to Application Clustering for more detail.

EF_COMPOUND_PAGES_MODE

- **Name:** `compound_pages`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Debug option, not suitable for normal use.

For packet buffers, allocate system pages in the following way:

- `0` - try to use compound pages if possible (default)
- `1` - obsolete, same behavior as 0
- `2` - do not use compound pages at all.

EF_CONG_AVOID_SCALE_BACK

- **Name:** `cong_avoid_scale_back`
- **Default:** `0`
- **Scope:** per-stack

When `>0`, this option slows down the rate at which the TCP congestion window is opened. This can help to reduce loss in environments where there is lots of congestion and loss.

EF_CTPIO

- **Name:** `ctpio`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Controls whether the CTPIO low-latency transmit mechanism is enabled:

- `0` – no (use DMA and/or PIO)
- `1` – enable CTPIO if available (default)
- `2` – enable CTPIO and fail stack creation if not available.

Mode `1` will fall back to DMA or PIO if CTPIO is not currently available. Mode `2` will fail to create the stack if the hardware supports CTPIO but CTPIO is not currently available. On hardware that does not support CTPIO there is no difference between mode `1` and mode `2`.

In all cases, CTPIO is only be used for packets if length ≤ EF_CTPIO_MAX_FRAME_LEN and when the VI's transmit queue is empty. If these conditions are not met DMA or PIO is used, even in mode `2`.

Note: CTPIO is currently only available on x86_64 systems.

Note: Mode `2` will not prevent a stack from operating without CTPIO in the event that CTPIO allocation is originally successful but then fails after an adapter is rebooted or hotplugged while that stack exists.

EF_CTPIO_CT_THRESH

- **Name:** `ctpio_ct_thresh`
- **Default:** `64`
- **Minimum:** `0`
- **Scope:** per-stack

Experimental: Sets the cut-through threshold for CTPIO transmits, when EF_CTPIO_MODE=`ct`. This option is for test purposes only and is likely to be changed or removed in a future release.

EF_CTPIO_MAX_FRAME_LEN

- **Name:** `ctpio_max_frame_len`
- **Default:** `500` if EF_CTPIO_MODE=`ct`, else `1518`
- **Minimum:** `0`
- **Maximum:** `4092`
- **Scope:** per-stack

Sets the maximum frame length for the CTPIO low-latency transmit mechanism. Packets up to this length will use CTPIO, if CTPIO is supported by the adapter and if CTPIO is enabled (see EF_CTPIO). Longer packets will use PIO and/or DMA. The cost per byte of packet payload varies between host architectures, as does the effect of packet size on the probability of poisoning, and so on some hosts it might be beneficial to reduce this value.

EF_CTPIO_MODE

- **Name:** `ctpio_mode`
- **Default:** `sf-np`
- **Scope:** per-stack

CTPIO transmission mode:

- `sf` - store and forward

  The NIC will buffer the entire packet before starting to send it on the wire.

- `sf-np` - store and forward, no poison

  Similar to `sf` mode but the NIC will guarantee never to emit a poisoned frame under any circumstances. This will force store-and-forward semantics for all users of CTPIO on the same port.

- `ct` - cut-through

  The NIC will start to send the outgoing packet onto the wire before it has been fully received, improving latency at the cost of occasionally transmitting a poisoned frame under some circumstances (such as the process being descheduled before it has finished writing the packet to the NIC).

EF_CTPIO_SWITCH_BYPASS

- **Name:** `ctpio_switch_bypass`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Allows CTPIO to be enabled on interfaces using the adapter's internal switch (i.e. on interfaces running full-feature firmware). This switching functionality is used to implement hardware multicast loopback and hardware loopback between interfaces, as used by virtual machines. CTPIO bypasses the switch, and hence is not compatible with those features.

EF_DEFER_ARP_MAX

- **Name:** `defer_arp_pkts`
- **Default:** `128`
- **Minimum:** `0`
- **Maximum:** `4096`
- **Scope:** per-stack

Limits the total number of packets to queue in a stack for all next hops while waiting for the OS to complete neighbor resolution (via ARP protocol for IPv4 or Neighbor Discovery for IPv6).

EF_DEFER_ARP_TIMEOUT

- **Name:** `defer_arp_timeout`
- **Default:** `60`
- **Minimum:** `1`
- **Maximum:** `600`
- **Scope:** per-stack

Onload now handles the deferral of packet sends during neighbor resolution (via ARP protocol for IPv4 or Neighbor Discovery for IPv6) rather than delegating the sends to the OS This option specifies a timeout (seconds) to wait for the OS to complete neighbor resolution before Onload sends deferred packets.

EF_DEFER_WORK_LIMIT

- **Name:** `defer_work_limit`
- **Default:** `32`
- **Scope:** per-stack

The maximum number of times that work can be deferred to the lock holder before we force the unlocked thread to block and wait for the lock

EF_DELACK_THRESH

- **Name:** `delack_thresh`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `65535`
- **Scope:** per-stack

This option controls the delayed acknowledgment algorithm. A socket can receive up to the specified number of TCP segments without generating an ACK. Setting this option to `0` disables delayed acknowledgments.

Note: This option is overridden by EF_DYNAMIC_ACK_THRESH, so both options need to be set to `0` to disable delayed acknowledgments.

EF_DONT_ACCELERATE

- **Name:** `dont_accelerate`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Do not accelerate by default. This option is usually used with `onload_set_stackname()` to allow individual sockets to be accelerated selectively.

EF_DYNAMIC_ACK_THRESH

- **Name:** `dynack_thresh`
- **Default:** `16`
- **Minimum:** `0`
- **Maximum:** `65535`
- **Scope:** per-stack

If set to >`0` this will turn on dynamic adaptation of the ACK rate to increase efficiency by avoiding ACKs when they would reduce throughput. The value is used as the threshold for number of pending ACKs before an ACK is forced. If set to zero then the standard delayed-ack algorithm is used.

EF_ENDPOINT_PACKET_RESERVE

- **Name:** `endpoint_packet_reserve`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1024`
- **Scope:** per-stack

This option enables reservation of packets per endpoint. No other endpoints would be able to use that reserved quota. Furthermore, new endpoints will only be created if there are enough free packets to reserve. Currently, this option is limited to TCP sockets and enforced on incoming TCP connections.

EF_EPOLL_CTL_FAST

- **Name:** `ul_epoll_ctl_fast`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Avoid system calls in `epoll_ctl()` when using an accelerated epoll implementation. System calls are deferred until `epoll_wait()` blocks, and in some cases removed completely. This option improves performance for applications that call `epoll_ctl()` frequently.

Caveats:

- This option has no effect when EF_UL_EPOLL=`0`.
- Do not turn this option on if your application uses `dup()`, `fork()` or `exec()` with epoll file descriptors or with the sockets monitored by epoll.
- If you monitor the epoll fd in another poll, select or epoll set, and have this option enabled, it might not give correct results.
- If you monitor the epoll fd in another poll, select or epoll set, and the effects of `epoll_ctl()` are latency critical, then this option can cause latency spikes or even deadlock.
- With EF_UL_EPOLL=`2`, this option is harmful if you are calling `epoll_wait()` and `epoll_ctl()` simultaneously from different threads or processes.

EF_EPOLL_CTL_HANDOFF

- **Name:** `ul_epoll_ctl_handoff`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Allow `epoll_ctl()` calls to be passed from one thread to another to avoid lock contention, in EF_UL_EPOLL=`1` or `3` case. This optimization is particularly important when `epoll_ctl()` calls are made concurrently with `epoll_wait()` and spinning is enabled.

This option is enabled by default.

Caveat:

- This option can cause an error code returned by `epoll_ctl()` to be hidden from the application when a call is deferred. In such cases an error message is emitted to `stderr` or the system log.

EF_EPOLL_MT_SAFE

- **Name:** `ul_epoll_mt_safe`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option disables concurrency control inside the accelerated epoll implementations, reducing CPU overhead. It is safe to enable this option if, for each epoll set, all calls on the epoll set and all calls that can modify a member of the epoll set are concurrency safe. Calls that can modify a member are `bind()`, `connect()`, `listen()` and `close()`.

This option improves performance with EF_UL_EPOLL=`1` or `3` and also with EF_UL_EPOLL=`2` and EF_EPOLL_CTL_FAST=`1`.

EF_EPOLL_SPIN

- **Name:** `ul_epoll_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in `epoll_wait()` calls until an event is satisfied or the spin timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_EVS_PER_POLL

- **Name:** `evs_per_poll`
- **Default:** `64`
- **Minimum:** `0`
- **Maximum:** `0x7fffffff`
- **Scope:** per-stack

Sets the number of hardware network events to handle before performing other work. This is a hint for internal tuning, and the actual number handled might differ. The value chosen represents a trade-off: Larger values increase batching (which typically improves efficiency) but can also increase the working set size (which harms cache efficiency). When EF_POLL_IN_KERNEL is set (either explicitly or implicitly) then the default value is `192`, to increase batching efficiency.

EF_FDS_MT_SAFE

- **Name:** `fds_mt_safe`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option allows less strict concurrency control when accessing the user-level file descriptor table, resulting in increased performance, particularly for multi-threaded applications. Single-threaded applications get a small latency benefit, but multi-threaded applications benefit most due to decreased cache-line bouncing between CPU cores.

This option is unsafe for applications that make changes to file descriptors in one thread while accessing the same file descriptors in other threads. For example, closing a file descriptor in one thread while invoking another system call on that file descriptor in a second thread. Concurrent calls that do not change the object underlying the file descriptor remain safe.

Calls to `bind()`, `connect()`, `listen()` can change the underlying object. If you call such functions in one thread while accessing the same file descriptor from the other thread, this option is also unsafe. In some special cases, any functions may change the underlying object.

Also concurrent calls might happen from signal handlers, so set this to `0` if your signal handlers call `bind()`, `connect()`, `listen()` or `close()`

EF_FDTABLE_SIZE

- **Name:** `fdtable_size`
- **Default:** `0`
- **Scope:** per-process

Limit the number of opened file descriptors by this value. If zero, the initial hard limit of open files (`` `ulimit -n -H` ``) is used. Hard and soft resource limits for opened file descriptors (`help ulimit`, `man 2 setrlimit`) are bound by this value.

EF_FDTABLE_STRICT

- **Name:** `fdtable_strict`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Enables more strict concurrency control for the user-level file descriptor table. Enabling this option can reduce performance for applications that create and destroy many connections per second.

EF_FORCE_SEND_MULTICAST

- **Name:** `force_send_multicast`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option causes all multicast sends to be accelerated. When disabled, multicast sends are only accelerated for sockets that have cleared the `IP_MULTICAST_LOOP` flag.

This option disables loopback of multicast traffic to receivers on the same host, unless (a) those receivers are sharing an Onload stack with the sender (see EF_NAME) and EF_MCAST_SEND is set to `1` or `3`, or(b) prerequisites to support loopback to other Onload stacks are met (see EF_MCAST_SEND).

EF_FORCE_TCP_NODELAY

- **Name:** `tcp_force_nodelay`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

This option allows the user to override the use of `TCP_NODELAY`. This can be useful in cases where third-party software is (not) setting this value and the user would like to control its behavior:

- `0` - do not override
- `1` - always set `TCP_NODELAY`
- `2` - never set `TCP_NODELAY`

EF_FORK_NETIF

- **Name:** `fork_netif`
- **Default:** `3`
- **Minimum:** `CI_UNIX_FORK_NETIF_NONE`
- **Maximum:** `CI_UNIX_FORK_NETIF_BOTH`
- **Scope:** per-process

This option controls behavior after an application calls `fork()`:

- `0` - Neither fork parent nor child creates a new Onload stack
- `1` - Child creates a new stack for new sockets
- `2` - Parent creates a new stack for new sockets
- `3` - Parent and child each create a new stack for new sockets.

EF_FREE_PACKETS_LOW_WATERMARK

- **Name:** `free_packets_low`
- **Default:** `0`
- **Scope:** per-stack

Keep free packets number to be at least this value. EF_MIN_FREE_PACKETS defines initialization behavior, and this value is about normal application runtime. In some combinations of hardware and software, Onload is not able allocate packets at any context, so it makes sense to keep some spare packets. Default value `0` is interpreted as EF_RXQ_SIZE`/2`.

EF_HELPER_PRIME_USEC

- **Name:** `timer_prime_usec`
- **Default:** `250`
- **Scope:** per-stack

Sets the frequency with which software should reset the count-down timer. Usually set to a value that is significantly smaller than EF_HELPER_USEC to prevent the count-down timer from firing unless needed. Defaults to (EF_HELPER_USEC`/2`).

EF_HELPER_USEC

- **Name:** `timer_usec`
- **Default:** `500`
- **Scope:** per-stack

Timeout in microseconds for the count-down interrupt timer. This timer generates an interrupt if network events are not handled by the application within the given time. It ensures that network events are handled promptly when the application is not invoking the network, or is descheduled.

Set this to `0` to disable the count-down interrupt timer. It is disabled by default for stacks that are interrupt driven.

EF_HIGH_THROUGHPUT_MODE

- **Name:** `rx_merge_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option causes onload to optimize for throughput at the cost of latency.

EF_ICMP_PKTS

- **Name:** `icmp_msg_max`
- **Default:** `64`
- **Minimum:** `2`
- **Maximum:** `1024`
- **Scope:** per-stack

Maximum number of ICMP messages which can be queued to one Onload stack.

EF_INTERFACE_BLACKLIST

- **Name:** `iface_blacklist`
- **Default:** none
- **Minimum:** none
- **Maximum:** none
- **Scope:** per-stack

List of names of interfaces for which use by the stack is denied. Space separated.See EF_INTERFACE_WHITELIST for notes as the same caveats apply.

***Note*:** The denylist takes priority over the allowlist so an interface present in both lists will not be accelerated.

EF_INTERFACE_WHITELIST

- **Name:** `iface_whitelist`
- **Default:** none
- **Minimum:** none
- **Maximum:** none
- **Scope:** per-stack

Space separated list of names of interfaces for which use by the stack is allowed. Beside passing the network interface of Solarflare NIC itself, it is allowed to provide name of higher order interface such as VLAN, MACVLAN, team or bond. At stack creation time these names will be used to identify underlying Solarflare NICs on which the allowlisting operates.

***Note*:** The granularity of allowlisting is limited: all interfaces based on allowlisted Solarflare NICs are accelerated.

EF_INT_DRIVEN

- **Name:** `int_driven`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Put the stack into an 'interrupt driven' mode of operation. When this option is not enabled Onload uses heuristics to decide when to enable interrupts, and this can cause latency jitter in some applications. So enabling this option can help avoid latency outliers.

This option is enabled by default except when spinning is enabled.

This option can be used with spinning to prevent outliers caused when the spin timeout is exceeded and the application blocks, or when the application is descheduled. In this case we recommend that interrupt moderation be set to a reasonably high value (e.g. 100 µs) to prevent too high a rate of interrupts.

EF_INT_REPRIME

- **Name:** `int_reprime`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enable interrupts more aggressively than the default.

EF_INVALID_ACK_RATELIMIT

- **Name:** `oow_ack_ratelimit`
- **Default:** `500`
- **Minimum:** `0`
- **Maximum:** `65535`
- **Scope:** per-stack

Limit the rate of ACKs sent because of invalid incoming TCP packet, in milliseconds. The limitation is applied per-socket. The value from `/proc/sys/net/ipv4/tcp_invalid_ratelimit` is used by default.

EF_IRQ_CHANNEL

- **Name:** `irq_channel`
- **Default:** `-1`
- **Minimum:** `-1`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Set the net-driver receive channel that will be used to handle interrupts for this stack. The core that receives interrupts for this stack will be whichever core is configured to handle interrupts for the specified net driver receive channel.

EF_IRQ_CORE

- **Name:** `irq_core`
- **Default:** `-1`
- **Minimum:** `-1`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Specify which CPU core interrupts for this stack should be handled on.

Onload interrupts are handled via net driver receive channel interrupts. The sfc_affinity driver is normally used to choose which net-driver receive channel is used, however this value can be used to override that mechanism.. It is only possible for interrupts to be handled on the requested core if a net driver interrupt is assigned to the selected core. Otherwise a nearby core will be selected.

***Note*:** If the IRQ balancer service is enabled it can redirect interrupts to other cores.

EF_KEEPALIVE_INTVL

- **Name:** `keepalive_intvl`
- **Default:** `75000`
- **Scope:** per-stack

Default interval between keepalives, in milliseconds.

The value from `/proc/sys/net/ipv4/tcp_keepalive_intvl` (which is in seconds) is used to find the default.

EF_KEEPALIVE_PROBES

- **Name:** `keepalive_probes`
- **Default:** `9`
- **Scope:** per-stack

Default number of keepalive probes to try before aborting the connection.

The value from `/proc/sys/net/ipv4/tcp_keepalive_probes` is used by default.

EF_KEEPALIVE_TIME

- **Name:** `keepalive_time`
- **Default:** `7200000`
- **Scope:** per-stack

Default idle time before keepalive probes are sent, in milliseconds.

The value from `/proc/sys/net/ipv4/tcp_keepalive_time` (which is in seconds) is used to find the default.

EF_KERNEL_PACKETS_BATCH_SIZE

- **Name:** `kernel_packets_batch_size`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `64`
- **Scope:** per-stack

In some cases (for example, when using scalable filters), packets that should be delivered to the kernel stack are instead delivered to Onload. Onload will forward these packets to the kernel, and can do so in batches of size up to the value of this option.

EF_KERNEL_PACKETS_TIMER_USEC

- **Name:** `kernel_packets_timer_usec`
- **Default:** `500`
- **Scope:** per-stack

Controls the maximum time for which Onload will queue up a packet that was received by Onload but should be forwarded to the kernel.

EF_LOAD_ENV

- **Name:** `load_env`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Onload will only consult other environment variables if this option is set. i.e. Clearing this option will cause all other EF_ environment variables to be ignored.

EF_LOG

- **Name:** `log_category`
- **Default:** `27`
- **Minimum:** `0`
- **Scope:** per-stack

Designed to control how chatty Onload's informative/warning messages are. Specified as a comma separated list of options to enable and disable (with a minus sign). Valid options are:

- 'banner' (on by default)
- 'resource_warnings' (on by default)
- 'config_warnings' (on by default)
- 'conn_drop' (off by default)
- 'usage_warnings' (on by default).

For example:

- To enable conn_drop:

  ```
  EF_LOG=conn_drop
  ```

- To enable conn_drop and turn off resource warnings:

```
EF_LOG=conn_drop,-resource_warnings
```

EF_LOG_FILE

- **Scope:** per-process

When EF_LOG_VIA_IOCTL is unset, the user can direct Onload debug and output data to a directory/file instead of `stdout` and instead of the `syslog`.

EF_LOG_TIMESTAMPS

- **Name:** `log_timestamps`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** global

If enabled this will add a timestamp to every Onload output log entry. Timestamps are originated from the FRC counter.

EF_LOG_VIA_IOCTL

- **Name:** `log_via_ioctl`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Causes error and log messages emitted by Onload to be written to the system log rather than written to standard error. This includes the copyright banner emitted when an application creates a new Onload stack.

By default, Onload logs are written to the application standard error if and only if it is a TTY.

Enable this option when it is important not to change what the application writes to standard error.

Disable it to guarantee that log goes to standard error even if it is not a TTY.

EF_MAX_ENDPOINTS

- **Name:** `max_ep_bufs`
- **Default:** `8192`
- **Minimum:** `4`
- **Maximum:** `CI_CFG_NETIF_MAX_ENDPOINTS_MAX` (default `1<<21`)
- **Scope:** per-stack

This option places an upper limit on the number of accelerated endpoints (sockets, pipes etc.) in an Onload stack. This option should be set to a power of two between 4 and 2^21. When this limit is reached listening sockets are not able to accept new connections over accelerated interfaces. New sockets and pipes created via `socket()` and `pipe()` etc. are handed over to the kernel stack and so are not accelerated.

Note: ~4 syn-receive states consume one endpoint, see also EF_TCP_SYNRECV_MAX.

EF_MAX_PACKETS

- **Name:** `max_packets`
- **Default:** `32768`
- **Minimum:** `1024`
- **Scope:** per-stack

Upper limit on number of packet buffers in each Onload stack. Packet buffers require hardware resources which may become a limiting factor if many stacks are each using many packet buffers. This option can be used to limit how much hardware resource and memory a stack uses. This option has an upper limit determined by the `max_packets_per_stack` onload module option.

Note: When 'scalable packet buffer mode' is not enabled (see EF_PACKET_BUFFER_MODE) the total number of packet buffers possible in aggregate is limited by a hardware resource.

EF_MAX_RX_PACKETS

- **Name:** `max_rx_packets`
- **Default:** `24576`
- **Minimum:** `0`
- **Maximum:** `1000000000`
- **Scope:** per-stack

The maximum number of packet buffers in a stack that can be used by the receive data path. This should be set to a value smaller than EF_MAX_PACKETS to ensure that some packet buffers are reserved for the transmit path.

EF_MAX_TX_PACKETS

- **Name:** `max_tx_packets`
- **Default:** `24576`
- **Minimum:** `0`
- **Maximum:** `1000000000`
- **Scope:** per-stack

The maximum number of packet buffers in a stack that can be used by the transmit data path. This should be set to a value smaller than EF_MAX_PACKETS to ensure that some packet buffers are reserved for the receive path.

EF_MCAST_JOIN_BINDTODEVICE

- **Name:** `mcast_join_bindtodevice`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When a UDP socket joins a multicast group (using `IP_ADD_MEMBERSHIP` or similar), this option causes the socket to be bound to the interface that the join was on. The benefit of this is that it ensures the socket will not accidentally receive packets from other interfaces that happen to match the same group and port. This can sometimes happen if another socket joins the same multicast group on a different interface, or if the switch is not filtering multicast traffic effectively.

If the socket joins multicast groups on more than one interface, then the binding is automatically removed.

EF_MCAST_JOIN_HANDOVER

- **Name:** `mcast_join_handover`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

When this option is set to `1`, and a UDP socket joins a multicast group on an interface that is not accelerated, the UDP socket is handed-over to the kernel stack. This can be a good idea because it prevents that socket from consuming Onload resources, and may also help avoid spinning when it is not wanted.

When set to `2`, UDP sockets that join multicast groups are always handed-over to the kernel stack.

EF_MCAST_RECV

- **Name:** `mcast_recv`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Controls whether or not to accelerate multicast receives. When set to zero, multicast receives are not accelerated, but the socket continues to be managed by Onload.

See also EF_MCAST_JOIN_HANDOVER.

EF_MCAST_RECV_HW_LOOP

- **Name:** `mcast_recv_hw_loop`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When enabled allows udp sockets to receive multicast traffic that originates from other Onload stacks.

EF_MCAST_SEND

- **Name:** `mcast_send`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `3`
- **Scope:** per-stack

Controls loopback of multicast traffic to receivers in the same and other Onload stacks.

- When set to `0` (default) disables loopback within the same stack as well as to other Onload stacks.
- When set to `1` enables loopback to the same stack.
- When set to `2` enables loopback to other Onload stacks.
- When set to `3` enables loopback to the same as well as other Onload stacks.

In respect to loopback to other Onload stacks the options is just a hint and the feature requires all the following:

- 8000-series or newer device
- selecting firmware variant with loopback support.

EF_MIN_FREE_PACKETS

- **Name:** `min_free_packets`
- **Default:** `100`
- **Minimum:** `0`
- **Maximum:** `1000000000`
- **Scope:** per-stack

Minimum number of free packets to reserve for each stack at initialization. If Onload is not able to allocate sufficient packet buffers to fill the RX rings and fill the free pool with the given number of buffers, then creation of the stack will fail.

EF_MULTICAST_LOOP_OFF

- **Name:** `multicast_loop_off`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

EF_MULTICAST_LOOP_OFF is deprecated in favor of EF_MCAST_SEND.

When set, disables loopback of multicast traffic to receivers in the same Onload stack.

This option only takes effect when EF_MCAST_SEND is not set and is equivalent to EF_MCAST_SEND=`1` or EF_MCAST_SEND=`0` for values of `0` and `1` respectively.

EF_NAME

Default: none Maximum: 8 chars Scope: per-stack

The environment variable EF_NAME will be honored to control Onload stack sharing. However, a call to `onload_set_stackname()` overrides this variable, and EF_DONT_ACCELERATE and EF_STACK_PER_THREAD both take precedence over EF_NAME.

EF_NETIF_DTOR

- **Name:** `netif_dtor`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-process

This option controls the lifetime of Onload stacks when the last socket in a stack is closed.

EF_NONAGLE_INFLIGHT_MAX

- **Name:** `nonagle_inflight_max`
- **Default:** `50`
- **Minimum:** `1`
- **Scope:** per-stack

This option affects the behavior of TCP sockets with the `TCP_NODELAY` socket option. Nagle's algorithm is enabled when the number of packets in-flight (sent but not acknowledged) exceeds the value of this option. This improves efficiency when sending many small messages, while preserving low latency.

Set this option to `-1` to ensure that Nagle's algorithm never delays sending of TCP messages on sockets with `TCP_NODELAY` enabled.

EF_NO_FAIL

- **Name:** `no_fail`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option controls whether failure to create an accelerated socket (due to resource limitations) is hidden by creating a conventional unaccelerated socket. Set this option to `0` to cause out-of-resources errors to be propagated as errors to the application, or to `1` to have Onload use the kernel stack instead when out of resources.

Disabling this option can be useful to ensure that sockets are being accelerated as expected (i.e. to find out when they are not).

EF_ONLOAD_FD_BASE

- **Name:** `fd_base`
- **Default:** `4`
- **Scope:** per-process

Onload uses fds internally that are not visible to the application. This can cause problems for applications that make assumptions about their use of the fd space, for example by doing dup2/3 onto a specific file descriptor. If this is done on an fd that is internally used by Onload than an error of the form 'citp_ep_dup3(29, 3): target is reserved, see EF_ONLOAD_FD_BASE' will be generated.

This option specifies a base file descriptor value, that Onload should try to make its internal file descriptors greater than or equal to. This allows the application to direct Onload to a part of the fd space that it is not expecting to explicitly use.

EF_PACKET_BUFFER_MODE

- **Name:** `packet_buffer_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

This option affects how DMA buffers are managed. The default packet buffer mode uses a limited hardware resource, and so restricts the total amount of memory that can be used by Onload for DMA.

Setting EF_PACKET_BUFFER_MODE`!=0` enables 'scalable packet buffer mode' which removes that limit. See details for each mode below:

- `2` - Physical address mode.

  Inherently unsafe, with no address space separation between different stacks or net driver packets.

Mode 1 was relevant only to adapters which are no longer supported.

For unsafe physical address mode (`2`), you should tune phys_mode_gid module parameter of the onload module.

EF_PERIODIC_TIMER_CPU

- **Name:** `periodic_timer_cpu`
- **Default:** `-1`
- **Minimum:** `-1`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Affinitizes Onload's periodic tasks to the specified CPU core. To ensure that Onload internal tasks such as polling timers are correctly serviced, the user should select a CPU that is receiving periodic timer ticks.

EF_PER_SOCKET_CACHE_MAX

- **Name:** `per_sock_cache_max`
- **Default:** `-1`
- **Minimum:** `-1`
- **Maximum:** `SMAX`
- **Scope:** per-stack

When socket caching is enabled, (i.e. when EF_SOCKET_CACHE_MAX `> 0`), this sets a further limit on the size of the cache for each socket.

- If set to `-1` in Onload 201805 onwards, or to `0` in earlier versions, no limit is set beyond the global limit specified by EF_SOCKET_CACHE_MAX.

  This behavior is the default.

- If set to `0` in Onload 201805 onwards, no accepted sockets will be cached for any listening sockets. This allows active-open socket caching to be enabled without also enabling passive-open socket caching.

EF_PIO

- **Name:** `pio`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Control of whether Programmed I/O is used instead of DMA for small packets:

- `0` - no (use DMA)
- `1` - use PIO for small packets if available (default)

Mode 1 will fall back to DMA if PIO is not currently available.

- 2 - use PIO for small packets and fail if PIO is not available.

  Mode 2 will fail to create the stack if the hardware supports PIO but PIO is not currently available.

On hardware that does not support PIO there is no difference between mode 1 and mode 2.

In all cases, PIO will only be used for small packets (see EF_PIO_THRESHOLD) and if the VI's transmit queue is currently empty. If these conditions are not met DMA will be used, even in mode 2.

Note: PIO is currently only available on x86_64 systems.

Note: Mode 2 will not prevent a stack from operating without PIO in the event that PIO allocation is originally successful but then fails after an adapter is rebooted or hotplugged while that stack exists.

EF_PIO_THRESHOLD

- **Name:** pio_thresh
- **Default:** 1514
- **Minimum:** 0
- **Scope:** per-stack

Sets a threshold for the size of packet that will use PIO (if turned on using EF_PIO) or CTPIO (if turned on using EF_CTPIO). Packets up to the threshold will use PIO or CTPIO. Larger packets will not.

EF_PIPE

- **Name:** ul_pipe
- **Default:** 2
- **Minimum:** 0
- **Maximum:** 2
- **Scope:** per-process

- 0 - disable pipe acceleration
- 1 - enable pipe acceleration
- 2 - accelerate pipes only if an Onload stack already exists in the process.

EF_PIPE_RECV_SPIN

- **Name:** `pipe_recv_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in pipe receive calls until data arrives or the spin timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_PIPE_SEND_SPIN

- **Name:** `pipe_send_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in pipe send calls until space becomes available in the socket buffer or the spin timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_PIPE_SIZE

- **Name:** `pipe_size`
- **Default:** `237568`
- **Minimum:** `OO_PIPE_MIN_SIZE` (default `4096`)
- **Maximum:** `CI_CFG_MAX_PIPE_SIZE` (default `1<<20`)
- **Scope:** per-process

Default size of the pipe in bytes. Actual pipe size will be rounded up to the size of packet buffer and subject to modifications by fcntl `F_SETPIPE_SZ` where supported.

EF_PKT_WAIT_SPIN

- **Name:** `pkt_wait_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin while waiting for DMA buffers. If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_POLL_FAST

- **Name:** `ul_poll_fast`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Allow a `poll()` call to return without inspecting the state of all polled file descriptors when at least one event is satisfied. This allows the accelerated `poll()` call to avoid a system call when accelerated sockets are 'ready', and can increase performance substantially.

This option changes the semantics of `poll()`, and as such could cause applications to misbehave. It effectively gives priority to accelerated sockets over non-accelerated sockets and other file descriptors. In practice a vast majority of applications work fine with this option.

EF_POLL_FAST_USEC

- **Name:** `ul_poll_fast_usec`
- **Default:** `32`
- **Scope:** per-process

When spinning in a `poll()` call, causes accelerated sockets to be polled for N usecs before unaccelerated sockets are polled. This reduces latency for accelerated sockets, possibly at the expense of latency on unaccelerated sockets. Since accelerated sockets are typically the parts of the application which are most performance-sensitive this is typically a good tradeoff.

EF_POLL_IN_KERNEL

- **Name:** `poll_in_kernel`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Ensures that mode 3 polling always occurs in kernel context. Defaults to 0 (off).

EF_POLL_NONBLOCK_FAST_USEC

- **Name:** `ul_poll_nonblock_fast_usec`
- **Default:** `200`
- **Scope:** per-process

When invoking `poll()` with timeout==`0` (non-blocking), this option causes non-accelerated sockets to be polled only every N usecs.

This reduces latency for accelerated sockets, possibly at the expense of latency on unaccelerated sockets. Since accelerated sockets are typically the parts of the application which are most performance-sensitive this is often a good tradeoff.

Set this option to zero to disable, or to a higher value to further improve latency for accelerated sockets.

This option changes the behavior of `poll()` calls, so could potentially cause an application to misbehave.

EF_POLL_ON_DEMAND

- **Name:** `poll_on_demand`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Poll for network events in the context of the application calls into the network stack. This option is enabled by default.

This option can improve performance in multi-threaded applications where the Onload stack is interrupt-driven (EF_INT_DRIVEN=`1`), because it can reduce lock contention. Setting EF_POLL_ON_DEMAND=`0` ensures that network events are (mostly) processed in response to interrupts.

EF_POLL_SPIN

- **Name:** `ul_poll_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in `poll()` calls until an event is satisfied or the spin timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_POLL_USEC

- **Name:** `ef_poll_usec_meta_option`
- **Default:** `0`
- **Scope:** per-process

This option enables spinning and sets the spin timeout in microseconds.

Setting this option is equivalent to: Setting EF_SPIN_USEC and EF_BUZZ_USEC, enabling spinning for UDP sends and receives, TCP sends and receives, select, poll and `epoll_wait()`, and enabling lock buzzing.

Spinning typically reduces latency and jitter substantially, and can also improve throughput. However, in some applications spinning can harm performance, particularly application that have many threads. When spinning is enabled you should normally dedicate a CPU core to each thread that spins.

You can use the `EF_*_SPIN` options to selectively enable or disable spinning for each API and transport. You can also use the `onload_thread_set_spin()` extension API to control spinning on a per-thread and per-API basis.

See also EF_POLL_USEC.

EF_PREALLOC_PACKETS

- **Name:** `prealloc_packets`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

If set ensures all packet buffers (EF_MAX_PACKETS) get allocated during stack creation or the stack creation fails. Also when set EF_MIN_FREE_PACKETS option is not taken into account.

EF_PREFAULT_PACKETS

- **Name:** `prefault_packets`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1000000000`
- **Scope:** per-stack

When set, this option causes the process to 'touch' the specified number of packet buffers when the Onload stack is created. This causes memory for the packet buffers to be pre-allocated, and also causes them to be memory-mapped into the process address space. This can prevent latency jitter caused by allocation and memory-mapping overheads.

The number of packets requested is in addition to the packet buffers that are allocated to fill the RX rings. There is no guarantee that it will be possible to allocate the number of packet buffers requested.

The default setting causes all packet buffers to be mapped into the user-level address space, but does not cause any extra buffers to be reserved. Set to `0` to prevent prefaulting.

EF_PROBE

- **Name:** `probe`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

When set, file descriptors accessed following `exec()` will be 'probed' and Onload sockets will be mapped to user-land so that they can be accelerated. Otherwise Onload sockets are not accelerated following `exec()`.

EF_RETRANSMIT_THRESHOLD

- **Name:** `retransmit_threshold`
- **Default:** `15`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Number of retransmit timeouts before a TCP connection is aborted.

The value from `/proc/sys/net/ipv4/tcp_retries2` is used by default.

EF_RETRANSMIT_THRESHOLD_ORPHAN

- **Name:** `retransmit_threshold_orphan`
- **Default:** `8`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Number of retransmit timeouts before a TCP connection is aborted in case of orphaned connection.

The value from `/proc/sys/net/ipv4/tcp_orphan_retries` is used by default.

EF_RETRANSMIT_THRESHOLD_SYN

- **Name:** `retransmit_threshold_syn`
- **Default:** `4`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Number of times a SYN will be retransmitted before a `connect()` attempt will be aborted.

The value from `/proc/sys/net/ipv4/tcp_syn_retries` is used by default.

EF_RETRANSMIT_THRESHOLD_SYNACK

- **Name:** `retransmit_threshold_synack`
- **Default:** `5`
- **Minimum:** `0`
- **Maximum:** `CI_CFG_TCP_SYNACK_RETRANS_MAX` (default `10`)
- **Scope:** per-stack

Number of times a SYN-ACK will be retransmitted before an embryonic connection will be aborted.

The value from `/proc/sys/net/ipv4/tcp_synack_retries` is used by default.

EF_RFC_RTO_INITIAL

- **Name:** `rto_initial`
- **Default:** `1000`
- **Scope:** per-stack

Initial retransmit timeout in milliseconds. i.e. The number of milliseconds to wait for an ACK before retransmitting packets.

EF_RFC_RTO_MAX

- **Name:** `rto_max`
- **Default:** `120000`
- **Scope:** per-stack

Maximum retransmit timeout in milliseconds.

EF_RFC_RTO_MIN

- **Name:** `rto_min`
- **Default:** `200`
- **Scope:** per-stack

Minimum retransmit timeout in milliseconds.

EF_RXQ_LIMIT

- **Name:** `rxq_limit`
- **Default:** `65535`
- **Minimum:** `CI_CFG_RX_DESC_BATCH` (default `16`)
- **Maximum:** `65535`
- **Scope:** per-stack

Maximum fill level for the receive descriptor ring. This has no effect when it has a value larger than the ring size (EF_RXQ_SIZE).

EF_RXQ_MIN

- **Name:** `rxq_min`
- **Default:** `256`
- **Minimum:** `0` (default `33`)
- **Scope:** per-stack

Minimum initial fill level for each RX ring. If Onload is not able to allocate sufficient packet buffers to fill each RX ring to this level, then creation of the stack will fail.

EF_RXQ_SIZE

- **Name:** `rxq_size`
- **Default:** `512`
- **Minimum:** `0`
- **Maximum:** `4096`
- **Scope:** per-stack

Set the size of the receive descriptor ring. Valid values: `512`, `1024`, `2048` or `4096`.

A larger ring size can absorb larger packet bursts without drops, but might reduce efficiency because the working set size is increased.

EF_RX_TIMESTAMPING

- **Name:** `rx_timestamping`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `3`
- **Scope:** per-stack

Control of hardware timestamping of received packets, possible values:

- `0` - do not do timestamping (default)
- `1` - request timestamping but continue if hardware is not capable or it does not succeed
- `2` - request timestamping and fail if hardware is capable and it does not succeed
- `3` - request timestamping and fail if hardware is not capable or it does not succeed.

EF_RX_TIMESTAMPING_ORDERING

- **Name:** `rx_timestamping_ordering`
- **Default:** `nic`
- **Scope:** per-stack

Select the source of timestamps to use to order received packets:

- `nic` - use hardware timestamps generated by the NIC
- `cpacket` - use cPacket timestamps on received packets.

EF_SA_ONSTACK_INTERCEPT

- **Name:** `sa_onstack_intercept`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Intercept signals when signal handler is installed with `SA_ONSTACK` flag.

- `0` - Do not intercept.

  If you call socket-related functions such as send, file-related functions such as close or dup from your signal handler, then your application might deadlock. (default)

- `1` - Intercept.

There is no guarantee that `SA_ONSTACK` flag will really work, but Onload library will do its best.

EF_SCALABLE_ACTIVE_WILDS_NEED_FILTER

- **Name:** `scalable_active_wilds_need_filter`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When set to `1`, IP filter is installed for every cached active-opened socket (see EF_TCP_SHARED_LOCAL_PORTS). Otherwise it is assumed that scalable filters do the job.

Default: `1` if EF_SCALABLE_FILTERS_ENABLE=`1` and scalable mode in EF_SCALABLE_FILTERS_MODE is "active"; `0` otherwise.

EF_SCALABLE_FILTERS

- **Name:** `scalable_filter_string`
- **Default:** `0`
- **Minimum:** none
- **Maximum:** none
- **Scope:** per-stack

Specifies the interface on which to enable support for scalable filters, and configures the scalable filter mode(s) to use. Scalable filters allow Onload to use a single hardware MAC-address filter to avoid hardware limitations and overheads. This removes restrictions on the number of simultaneous connections and increases performance of active connect calls, but kernel support on the selected interface is limited to ARP/DHCP/ICMP protocols and some Onload features that rely on unaccelerated traffic (such as receiving fragmented UDP datagrams) will not work. Please see the Onload user guide for full details.

Depending on the mode selected this option will enable support for:

- scalable listening sockets
- `IP_TRANSPARENT` socket option
- scalable active open.

Format of EF_SCALABLE_FILTERS variable is as follows:

```
EF_SCALABLE_FILTERS=[<interface-name>[=mode[:mode]],]<interface-
name>[=mode[:mode]]
```

The following modes and their combinations can be specified:

- `transparent_active`
- `rss:transparent_active`
- `passive`
- `rss:passive`
- `transparent_active:passive`
- `active`
- `rss:active`
- `rss:passive:active`

It is possible to specify both an active mode interface and a passive mode interface. If two interfaces are specified then both the active and passive interfaces must have the same rss qualifier. Furthermore, if the interface is the string “any”, scalable filters are installed on all interfaces.

EF_SCALABLE_FILTERS_ENABLE

- **Name:** `scalable_filter_enable`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Turn the scalable filter feature on or off on a stack. Takes one of the following values:

- `0` – Scalable filters are not used for this stack.
- `1` – The configuration selected in EF_SCALABLE_FILTERS will be used.
- `2` – Indicates a special mode to address a master-worker hierarchy of some event driven applications.

  The scalable filter gets created for reuse by port bound sockets in the master process context. However, active mode will become available in worker processes once they add one of the sockets to their epoll set.

  Applies to `rss:*active` scalable modes.

  Note: This mode is not compatible with use of the onload extensions stackname API.

If unset this will default to `1` if EF_SCALABLE_FILTERS is configured.

EF_SCALABLE_FILTERS_IFINDEX_ACTIVE

- **Name:** `scalable_filter_ifindex_active`
- **Default:** `0`
- **Minimum:** `CITP_SCALABLE_FILTERS_MIN`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Stores active scalable filter interface set with EF_SCALABLE_FILTERS. To be set indirectly with EF_SCALABLE_FILTERS variable

EF_SCALABLE_FILTERS_IFINDEX_PASSIVE

- **Name:** `scalable_filter_ifindex_passive`
- **Default:** `0`
- **Minimum:** `CITP_SCALABLE_FILTERS_MIN`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Stores passive scalable filter interface set with EF_SCALABLE_FILTERS. To be set indirectly with EF_SCALABLE_FILTERS variable

EF_SCALABLE_FILTERS_MODE

- **Name:** `scalable_filter_mode`
- **Default:** `-1`
- **Minimum:** `-1`
- **Maximum:** `13`
- **Scope:** per-stack

Stores scalable filter mode set with EF_SCALABLE_FILTERS. To be set indirectly with EF_SCALABLE_FILTERS variable.

EF_SCALABLE_LISTEN_MODE

- **Name:** `scalable_listen`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Choose behavior of scalable listening sockets when using EF_SCALABLE_FILTERS

- `0` – Listening sockets bound to a local address configured on the scalable interface use the scalable filter (default). Connections on other interfaces are not accelerated.
- `1` – Listening sockets bound to a local address configured on the scalable interface use the scalable filter. Connections on other interfaces including loopback are refused.

  This mode avoids kernel scalability issues with large numbers of listen sockets.

EF_SELECT_FAST

- **Name:** `ul_select_fast`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Allow a `select()` call to return without inspecting the state of all selected file descriptors when at least one selected event is satisfied. This allows the accelerated `select()` call to avoid a system call when accelerated sockets are 'ready', and can increase performance substantially.

This option changes the semantics of `select()`, and as such could cause applications to misbehave. It effectively gives priority to accelerated sockets over non-accelerated sockets and other file descriptors. In practice a vast majority of applications work fine with this option.

EF_SELECT_FAST_USEC

- **Name:** `ul_select_fast_usec`
- **Default:** `32`
- **Scope:** per-process

When spinning in a `select()` call, causes accelerated sockets to be polled for N µsecs before unaccelerated sockets are polled. This reduces latency for accelerated sockets, possibly at the expense of latency on unaccelerated sockets. Because accelerated sockets are typically the parts of the application which are most performance-sensitive this is typically a good tradeoff.

EF_SELECT_NONBLOCK_FAST_USEC

- **Name:** `ul_select_nonblock_fast_usec`
- **Default:** `200`
- **Scope:** per-process

When invoking `select()` with timeout==`0` (non-blocking), this option causes non-accelerated sockets to be polled only every N µsecs.

This reduces latency for accelerated sockets, possibly at the expense of latency on unaccelerated sockets. Because accelerated sockets are typically the parts of the application which are most performance-sensitive this is often a good tradeoff.

Set this option to zero to disable, or to a higher value to further improve latency for accelerated sockets.

This option changes the behavior of `select()` calls, so could potentially cause an application to misbehave.

EF_SELECT_SPIN

- **Name:** `ul_select_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking `select()` calls until the select set is satisfied or the spin timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_SEND_POLL_MAX_EVS

- **Name:** `send_poll_max_events`
- **Default:** `96`
- **Minimum:** `1`
- **Maximum:** `65535`
- **Scope:** per-stack

When polling for network events after sending, this places a limit on the number of events handled.

EF_SEND_POLL_THRESH

- **Name:** `send_poll_thresh`
- **Default:** `64`
- **Minimum:** `0`
- **Maximum:** `65535`
- **Scope:** per-stack

Poll for network events after sending this many packets.

Setting this to a larger value might improve transmit throughput for small messages by allowing batching. However, such batching can cause sends to be delayed leading to increased jitter.

EF_SHARE_WITH

- **Name:** `share_with`
- **Default:** `0`
- **Minimum:** `-1`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Set this option to allow a stack to be accessed by processes owned by another user. Set it to the UID of a user that should be permitted to share this stack, or set it to `-1` to allow any user to share the stack. By default stacks are not accessible by users other than root.

Processes invoked by root can access any stack. Setuid processes can only access stacks created by the effective user, not the real user. This restriction can be relaxed by setting the Onload kernel module option `allow_insecure_setuid_sharing=1`.

IMPORTANT! *A user that is permitted to access a stack is able to: snoop on any data transmitted or received via the stack; inject or modify data transmitted or received via the stack; damage the stack and any sockets or connections in it; cause misbehavior and crashes in any application using the stack.*

EF_SIGNALS_NOPOSTPONE

- **Name:** `signals_no_postpone`
- **Default:** `67110088`
- **Minimum:** `0`
- **Maximum:** `(ci_uint64)(-1)`
- **Scope:** per-process

Comma-separated list of signal numbers to avoid postponing of the signal handlers. Your application will deadlock if one of the handlers uses socket function. By default, the list includes `SIGILL`, `SIGBUS`, `SIGFPE`, `SIGSEGV` and `SIGPROF`.

Please specify numbers, not string aliases: `EF_SIGNALS_NOPOSTPONE=7,11,27` instead `of EF_SIGNALS_NOPOSTPONE=SIGBUS,SIGSEGV,SIGPROF`.

You can set EF_SIGNALS_NOPOSTPONE to empty value to postpone all signal handlers in the same way if you suspect these signals to call network functions.

EF_SLEEP_SPIN_USEC

- **Name:** `sleep_spin_usec`
- **Default:** `0`
- **Scope:** per-process

Sets the duration in microseconds of sleep after each spin iteration. Currently applies to EPOLL3 `epoll_wait` only. Enabling the option trades some of the benefits of spinning - latency - for reduction in CPU utilization and power consumption.

Spinning typically reduces latency and jitter substantially, and can also improve throughput. However, in some applications spinning can harm performance; particularly application that have many threads. When spinning is enabled you should normally dedicate a CPU core to each thread that spins.

You can use the `EF_*_SPIN` options to selectively enable or disable spinning for each API and transport. You can also use the `onload_thread_set_spin()` extension API to control spinning on a per-thread and per-API basis.

EF_SOCKET_CACHE_MAX

- **Name:** `sock_cache_max`
- **Default:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Sets the maximum number of TCP sockets to cache for this stack. When set > 0, Onload will cache resources associated with sockets to improve connection set-up and tear-down performance. This improves performance for applications that make new TCP connections at a high rate.

EF_SOCKET_CACHE_PORTS

- **Name:** sock_cache_ports
- **Default:** 0
- **Scope:** per-process

This option specifies a comma-separated list of port numbers. When set (and socket caching is enabled), only sockets bound to the specified ports will be eligible to be cached.

EF_SOCK_LOCK_BUZZ

- **Name:** sock_lock_buzz
- **Default:** 0
- **Minimum:** 0
- **Maximum:** 1
- **Scope:** per-process

Spin while waiting to obtain a per-socket lock. If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_BUZZ_USEC.

The per-socket lock is taken in recv() calls and similar. This option can reduce jitter when multiple threads invoke recv() on the same socket, but can reduce fairness between threads competing for the lock.

EF_SO_BUSY_POLL_SPIN

- **Name:** so_busy_poll_spin
- **Default:** 0
- **Minimum:** 0
- **Maximum:** 1
- **Scope:** per-process

Spin poll, select and epoll in a Linux-like way: enable spinning only if a spinning socket is preset in the poll/select/epoll set. See Linux documentation on SO_BUSY_POLL socket option for details.

You should also enable spinning via EF_{POLL, SELECT, EPOLL}_SPIN variable if you'd like to spin in poll, select or epoll correspondingly. The spin duration is set via EF_SPIN_USEC, which is equivalent to the Linux sysctl.net.busy_poll value. EF_POLL_USEC is all-in-one variable to set for all 4 variables mentioned here.

Most versions of Linux never spin in epoll, but Onload does. This variable does not affect epoll behavior if EF_UL_EPOLL=`2`.

EF_SPIN_USEC

- **Name:** `ul_spin_usec`
- **Default:** `0`
- **Scope:** per-process

Sets the timeout in microseconds for spinning options. Set this to `-1` to spin forever. The spin timeout can also be set by the EF_POLL_USEC option.

Spinning typically reduces latency and jitter substantially, and can also improve throughput. However, in some applications spinning can harm performance, particularly application that have many threads. When spinning is enabled you should normally dedicate a CPU core to each thread that spins.

You can use the `EF_*_SPIN` options to selectively enable or disable spinning for each API and transport. You can also use the `onload_thread_set_spin()` extension API to control spinning on a per-thread and per-API basis.

EF_STACK_LOCK_BUZZ

- **Name:** `stack_lock_buzz`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin while waiting to obtain a per-stack lock. If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_BUZZ_USEC.

This option reduces jitter caused by lock contention, but can reduce fairness between threads competing for the lock.

EF_STACK_PER_THREAD

- **Name:** `stack_per_thread`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Create a separate Onload stack for the sockets created by each thread.

EF_SYNC_CPLANE_AT_CREATE

- **Name:** `sync_cplane`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-process

When this option is set to `2` Onload will force a sync of control plane information from the kernel when a stack is created. This can help to ensure up to date information is used where a stack is created immediately following interface configuration.

If this option is set to `1` then Onload will perform a lightweight sync of control plane information without performing a full dump. It is the default mode.

Setting this option to `0` will disable forced sync. Synchronizing data from the kernel will continue to happen periodically.

Sync operation time is limited by `cplane_init_timeout` onload module option.

EF_TAIL_DROP_PROBE

- **Name:** `tail_drop_probe`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Whether to probe if the tail of a TCP burst isn't ACKed quickly.

The value from `/proc/sys/net/ipv4/tcp_early_retrans` is used to derive the default.

EF_TCP

- **Name:** `ul_tcp`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Clear to disable acceleration of new TCP sockets.

EF_TCP_ACCEPT_SPIN

- **Name:** `tcp_accept_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking TCP `accept()` calls until incoming connection is established, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_TCP_ADV_WIN_SCALE_MAX

- **Name:** `tcp_adv_win_scale_max`
- **Default:** `14`
- **Minimum:** `0`
- **Maximum:** `14`
- **Scope:** per-stack

Maximum value for TCP window scaling that will be advertised. Set it to 0 to turn window scaling off.

The value from `/proc/sys/net/ipv4/tcp_window_scaling` is used by default.

EF_TCP_BACKLOG_MAX

- **Name:** `tcp_backlog_max`
- **Default:** `256`
- **Scope:** per-stack

Places an upper limit on the number of embryonic (half-open) connections for one listening socket. See also EF_TCP_SYNRECV_MAX.

The value from `/proc/sys/net/ipv4/tcp_max_syn_backlog` is used by default.

EF_TCP_CLIENT_LOOPBACK

- **Name:** `tcp_client_loopback`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `4`
- **Scope:** per-stack

Enable acceleration of TCP loopback connections on the connecting (client) side:

- `0` - not accelerated (default)
- `1` - accelerate if the listening socket is in the same stack (you should also set EF_TCP_SERVER_LOOPBACK`!=0`)
- `2` - accelerate and move accepted socket to the stack of the connecting socket (server should allow this via EF_TCP_SERVER_LOOPBACK=`2`)
- `3` - accelerate and move the connecting socket to the stack of the listening socket (server should allow this via EF_TCP_SERVER_LOOPBACK`!=0`)
- `4` - accelerate and move both connecting and accepted sockets to the new stack (server should allow this via EF_TCP_SERVER_LOOPBACK=`2`).

Note: Options `3` and `4` break some applications using `epoll()`, `fork()` and `dup()` calls.

Note: Options `2` and `4` cause `accept()` to misbehave if the client exits too early.

EF_TCP_COMBINE_SENDS_MODE

- **Name:** `tcp_combine_sends_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option controls how Onload fills packets in the TCP send buffer:

- `0` - Onload prefers to use all the space at the end of a previous packet before allocating a new one (default)
- `1` - Onload prefers to allocate a new packet for each new send.

In all cases this is a hint rather than guaranteed behavior, and there are conditions where the preference indicated by this option will not be possible. For example memory pressure might cause packets in the send queue to be combined. `MSG_MORE` and `TCP_CORK` can override this option when set. The zero-copy sends API can also use the segmentation provided by the caller's buffers. For full control of message segmentation the delegated sends API can be used. Setting this option can affect the capacity of send buffers belonging to sockets in this stack and increase packet buffer usage. It can also reduce efficiency as packets must be allocated for each send call rather than reusing one that is already available.

Setting this option is only recommended if you have an explicit need to avoid combined or split sends.

EF_TCP_CONNECT_HANDOVER

- **Name:** `tcp_connect_handover`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When an accelerated TCP socket calls `connect()`, hand it over to the kernel stack. This option disables acceleration of active-open TCP connections.

EF_TCP_CONNECT_SPIN

- **Name:** `tcp_connect_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking TCP `connect()` calls until connection is established, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_TCP_EARLY_RETRANSMIT

- **Name:** `tcp_early_retransmit`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enables the Early Retransmit (RFC 5827) algorithm for TCP, and also the Limited Transmit (RFC 3042) algorithm, on which Early Retransmit depends.

The value from `/proc/sys/net/ipv4/tcp_early_retrans` is used to derive the default.

EF_TCP_FASTSTART_IDLE

- **Name:** `tcp_faststart_idle`
- **Default:** `65536`
- **Minimum:** `0`
- **Scope:** per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so might reduce performance. FASTSTART is enabled when a connection is new, following loss and after the connection has been idle for a while.

This option sets the number of bytes that must be ACKed by the receiver before the connection exits FASTSTART. Set to zero to prevent a connection entering FASTSTART after an idle period.

EF_TCP_FASTSTART_INIT

- **Name:** `tcp_faststart_init`
- **Default:** `65536`
- **Minimum:** `0`
- **Scope:** per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so might reduce performance. FASTSTART is enabled when a connection is new, following loss and after the connection has been idle for a while.

This option sets the number of bytes that must be ACKed by the receiver before the connection exits FASTSTART. Set to zero to disable FASTSTART on new connections.

EF_TCP_FASTSTART_LOSS

- **Name:** `tcp_faststart_loss`
- **Default:** `65536`
- **Minimum:** `0`
- **Scope:** per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so might reduce performance. FASTSTART is enabled when a connection is new, following loss and after the connection has been idle for a while.

This option sets the number of bytes that must be ACKed by the receiver before the connection exits FASTSTART following loss. Set to zero to disable FASTSTART after loss.

EF_TCP_FIN_TIMEOUT

- **Name:** `fin_timeout`
- **Default:** `60`
- **Scope:** per-stack

Time in seconds to wait for an orphaned connection to be closed properly by the network partner (e.g. FIN in the TCP FIN_WAIT2 state, zero window opening to send our FIN, etc).

The value from `/proc/sys/net/ipv4/tcp_fin_timeout` is used by default.

EF_TCP_FORCE_REUSEPORT

- **Name:** `tcp_reuseports`
- **Default:** `0`
- **Scope:** per-process

This option specifies a comma-separated list of port numbers. TCP sockets that bind to those port numbers will have `SO_REUSEPORT` automatically applied to them.

EF_TCP_INITIAL_CWND

- **Name:** `initial_cwnd`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Sets the initial size of the congestion window (in bytes) for TCP connections. Some care is needed as, for example, setting smaller than the segment size might result in Onload being unable to send traffic.

IMPORTANT! *Modifying this option can violate the TCP protocol.*

EF_TCP_ISN_2MSL

- **Name:** `tcp_isn_2msl`
- **Default:** `240`
- **Maximum:** `CITP_TCP_ISN_2MSL_MAX`
- **Scope:** per-stack

Maximum time that peers are assumed to stay in `TIMEWAIT` state. In seconds. Relevant when EF_TCP_ISN_MODE is set to clocked+cache.

EF_TCP_ISN_CACHE_SIZE

- **Name:** `tcp_isn_cache_sizel`
- **Default:** `0`
- **Scope:** per-stack

Cache size for recently used four-tuples and their last sequence number. `0` - automatically chosen. Relevant when EF_TCP_ISN_MODE is set to clocked+cache.

EF_TCP_ISN_INCLUDE_PASSIVE

- **Name:** `tcp_isn_include_passive`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enables populating isn cache with passively opened connections. Relevant when EF_TCP_ISN_MODE is set to clocked+cache.

EF_TCP_ISN_MODE

- **Name:** `tcp_isn_mode`
- **Default:** `clocked+cache`
- **Scope:** per-stack

Selects behavior with which Onload interacts with peers when reusing four-tuples:

- `clocked` – Linux compatible behavior (default)
- `clocked+cache` – additional cache to avoid failed connection attempts.

  Note: The behavior is relevant to high connection rate use cases with high outgoing data rates.

When in clocked+cache mode, sequence numbers used by closed TCP connections are remembered so that initial sequence numbers for subsequent uses of the same four-tuple can be selected so as not to overlap with the previous connection's sequence space.

EF_TCP_ISN_OFFSET

- **Name:** `tcp_isn_offset`
- **Default:** `65537`
- **Scope:** per-stack

Increase in sequence number between subsequent connections reusing the same four-tuple. Lower value allows to reduce use of ISN cache, however potentially being unsafe with some host types or rare use cases.

EF_TCP_LISTEN_HANDOVER

- **Name:** `tcp_listen_handover`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When an accelerated TCP socket calls `listen()`, hand it over to the kernel stack. This option disables acceleration of TCP listening sockets and passively opened TCP connections.

EF_TCP_LOSS_MIN_CWND

- **Name:** `loss_min_cwnd`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Sets the minimum size of the congestion window for TCP connections following loss.

IMPORTANT! *Modifying this option can violate the TCP protocol.*

Deprecated. Please use EF_TCP_MIN_CWND instead.

EF_TCP_MIN_CWND

- **Name:** `min_cwnd`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX`
- **Scope:** per-stack

Sets the minimum size of the congestion window for TCP connections. This value is used for any congestion window changes: connection start, packet loss, connection being idle, etc.

IMPORTANT! *Modifying this option can violate the TCP protocol.*

EF_TCP_RCVBUF

- **Name:** `tcp_rcvbuf_user`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX/2`
- **Scope:** per-stack

Override `SO_RCVBUF` for TCP sockets. (Note: the actual size of the buffer is double the amount requested, mimicking the behavior of the Linux kernel.)

EF_TCP_RCVBUF_ESTABLISHED_DEFAULT

- **Name:** `tcp_rcvbuf_est_def`
- **Default:** `131072`
- **Minimum:** `0`
- **Maximum:** `SMAX/4`
- **Scope:** per-stack

Overrides the OS default `SO_RCVBUF` value for TCP sockets in the `ESTABLISHED` state if the OS default `SO_RCVBUF` value falls outside bounds set with this option. This value is used when the TCP connection transitions to `ESTABLISHED` state, to avoid confusion of some applications like netperf.

The lower bound is set to this value and the upper bound is set to 4 * this value. If the OS default `SO_RCVBUF` value is less than the lower bound, then the lower bound is used. If the OS default `SO_RCVBUF` value is more than the upper bound, then the upper bound is used.

This variable overrides OS default `SO_RCVBUF` value only, it does not change `SO_RCVBUF` if the application explicitly sets it (see EF_TCP_RCVBUF variable which overrides application-supplied value).

EF_TCP_RCVBUF_MODE

- **Name:** `tcp_rcvbuf_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option controls how the RCVBUF is set for TCP Mode `0` (default) gives fixed size RCVBUF.

Mode `1` will enable automatic tuning of RCVBUF using Dynamic Right Sizing. If `SO_RCVBUF` is explicitly set by the application this value will be used. EF_TCP_SOCKBUF_MAX_FRACTION can be used to control the maximum size of the buffer for an individual socket.

The effect of EF_TCP_RCVBUF_STRICT is independent of this setting.

EF_TCP_RCVBUF_STRICT

- **Name:** `tcp_rcvbuf_strict`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option prevents TCP small segment attack. With this option set, Onload limits the number of packets inside TCP receive queue and TCP reorder buffer. In some cases, this option causes performance penalty. You probably want this option if your application is connecting to untrusted partner or over untrusted network.

Off by default.

EF_TCP_RECV_SPIN

- **Name:** `tcp_recv_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking TCP receive calls until data arrives, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_TCP_RST_DELAYED_CONN

- **Name:** `rst_delayed_conn`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option tells Onload to reset TCP connections rather than allow data to be transmitted late. Specifically, TCP connections are reset if the retransmit timeout fires. (This usually happens when data is lost, and normally triggers a retransmit which results in data being delivered hundreds of milliseconds late).

IMPORTANT! *This option is likely to cause connections to be reset spuriously if ACK packets are dropped in the network.*

EF_TCP_RX_CHECKS

- **Name:** `tcp_rx_checks`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Internal/debugging use only: perform extra debugging/consistency checks on received packets.

EF_TCP_RX_LOG_FLAGS

- **Name:** `tcp_rx_log_flags`
- **Default:** `0`
- **Scope:** per-stack

Log received packets that have any of these flags set in the TCP header. Only active when EF_TCP_RX_CHECKS is set.

EF_TCP_SEND_NONBLOCK_NO_PACKETS_MODE

- **Name:** `tcp_nonblock_no_pkts_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option controls how a non-blocking TCP `send()` call should behave if it is unable to allocate sufficient packet buffers. By default Onload will mimic Linux kernel stack behavior and block for packet buffers to be available. If set to `1`, this option will cause Onload to return error `ENOBUFS`. Note this option can cause some applications (that assume that a socket that is writable is able to send without error) to malfunction.

EF_TCP_SEND_SPIN

- **Name:** `tcp_send_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking TCP send calls until window is updated by peer, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_TCP_SERVER_LOOPBACK

- **Name:** `tcp_server_loopback`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Enable acceleration of TCP loopback connections on the listening (server) side:

- `0` - not accelerated (default)
- `1` - accelerate if the connecting socket is in the same stack (you should also set EF_TCP_CLIENT_LOOPBACK`!=0`)
- `2` - accelerate and allow accepted socket to be in another stack (this is necessary for clients with EF_TCP_CLIENT_LOOPBACK=`2,4`).

EF_TCP_SHARED_LOCAL_PORTS

- **Name:** `tcp_shared_local_ports`
- **Default:** `0`
- **Minimum:** `0`
- **Scope:** per-stack

This feature improves the performance of TCP active-opens. It reduces the cost of both blocking and non-blocking connect() calls, reduces the latency to establish new connections, and enables scaling to large numbers of active-open connections. It also reduces the cost of closing these connections.

These improvements are achieved by sharing a set of local port numbers amongst active-open sockets, which saves the cost and scaling limits associated with installing packet steering filters for each active-open socket. Shared local ports are only used when the local port is not explicitly assigned by the application. Set this option to `>=1` to enable local port sharing.

The value set gives the initial number of local ports to allocate when the Onload stack is created. More shared local ports are allocated on demand as needed up to the maximum given by EF_TCP_SHARED_LOCAL_PORTS_MAX.

Note: Typically only one local shared port is needed, as different local ports are only needed when multiple connections are made to the same remote IP:port.

EF_TCP_SHARED_LOCAL_PORTS_MAX

- **Name:** `tcp_shared_local_ports_max`
- **Default:** `100`
- **Minimum:** `0`
- **Scope:** per-stack

This setting sets the maximum size of the pool of local shared ports in the stack. See EF_TCP_SHARED_LOCAL_PORTS for details.

EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK

- **Name:** `tcp_shared_local_no_fallback`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When set, connecting TCP sockets will use ports only from the TCP shared local port pool (unless explicitly bound). If all shared local ports are in use, the `connect()` call will fail.

EF_TCP_SHARED_LOCAL_PORTS_PER_IP

- **Name:** `tcp_shared_local_ports_per_ip`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When set, ports reserved for the pool of shared local ports will be reserved per local IP address on demand.

EF_TCP_SHARED_LOCAL_PORTS_PER_IP_MAX

- **Name:** `tcp_shared_local_ports_per_ip_max`
- **Default:** `0`
- **Minimum:** `0`
- **Scope:** per-stack

Sets the maximum size of the pool of local shared ports for given local IP address. When used with scalable RSS mode this setting limits the total number within the cluster. `0` – no limit. See EF_TCP_SHARED_LOCAL_PORTS for details.

EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST

- **Name:** `tcp_shared_local_ports_reuse_fast`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When enabled, this option allows shared local ports (as controlled by the EF_TCP_SHARED_LOCAL_PORTS option) to be reused immediately when the previous socket using that port has reached the CLOSED state, even if it did so via LAST-ACK.

EF_TCP_SHARED_LOCAL_PORTS_STEP

- **Name:** `tcp_shared_local_ports_step`
- **Default:** `1`
- **Minimum:** `1`
- **Scope:** per-stack

Controls the number of ports allocated when expanding the pool of shared local ports.

EF_TCP_SNDBUF

- **Name:** `tcp_sndbuf_user`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX/2`
- **Scope:** per-stack

Override `SO_SNDBUF` for TCP sockets (Note: the actual size of the buffer is double the amount requested, mimicking the behavior of the Linux kernel.)

EF_TCP_SNDBUF_ESTABLISHED_DEFAULT

- **Name:** `tcp_sndbuf_est_def`
- **Default:** `131072`
- **Minimum:** `0`
- **Maximum:** `SMAX/4`
- **Scope:** per-stack

Overrides the OS default `SO_SNDBUF` value for TCP sockets in the `ESTABLISHED` state if the OS default `SO_SNDBUF` value falls outside bounds set with this option. This value is used when the TCP connection transitions to `ESTABLISHED` state, to avoid confusion of some applications like netperf.

The lower bound is set to this value and the upper bound is set to 4 * this value. If the OS default `SO_SNDBUF` value is less than the lower bound, then the lower bound is used. If the OS default `SO_SNDBUF` value is more than the upper bound, then the upper bound is used.

This variable overrides OS default `SO_SNDBUF` value only, it does not change `SO_SNDBUF` if the application explicitly sets it (see EF_TCP_SNDBUF variable which overrides application-supplied value).

EF_TCP_SNDBUF_MODE

- **Name:** `tcp_sndbuf_mode`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

This option controls how the `SO_SNDBUF` limit is applied to TCP sockets. In the default mode the limit applies to the size of the send queue and retransmit queue combined. When this option is set to `0` the limit applies to the send queue only.

When this option is set to 2, the SNDBUF size is automatically adjusted for each TCP socket to match the window advertised by the peer (limited by EF_TCP_SOCKBUF_MAX_FRACTION). If the application sets SO_SNDBUF explicitly then automatic adjustment is not used for that socket. The limit is applied to the size of the send queue and retransmit queue combined. You might also want to set EF_TCP_RCVBUF_MODE to give automatic adjustment of RCVBUF.

EF_TCP_SOCKBUF_MAX_FRACTION

- **Name:** tcp_sockbuf_max_fraction
- **Default:** 1
- **Minimum:** 1
- **Maximum:** 10
- **Scope:** per-stack

This option controls the maximum fraction of the TX buffers that can be allocated to a single socket with EF_TCP_SNDBUF_MODE=2.

It also controls the maximum fraction of the RX buffers that can be allocated to a single socket with EF_TCP_RCVBUF_MODE=1.

The maximum allocation for a socket is EF_MAX_TX_PACKETS/(2^N) for TX and EF_MAX_RX_PACKETS/(2^N) for RX, where N is specified here.

EF_TCP_SYNCOOKIES

- **Name:** tcp_syncookies
- **Default:** 0
- **Minimum:** 0
- **Maximum:** 1
- **Scope:** per-stack

Use TCP syncookies to protect from SYN flood attack.

EF_TCP_SYNRECV_MAX

- **Name:** tcp_synrecv_max
- **Default:** 1024
- **Maximum:** CI_CFG_NETIF_MAX_ENDPOINTS_MAX (default 1<<21)
- **Scope:** per-stack

Places an upper limit on the number of embryonic (half-open) connections in an Onload stack. See also EF_TCP_BACKLOG_MAX.

By default, EF_TCP_SYNRECV_MAX `= 4 *` EF_TCP_BACKLOG_MAX.

EF_TCP_SYN_OPTS

- **Name:** `syn_opts`
- **Default:** `7`
- **Scope:** per-stack

A bitmask specifying the TCP options to advertise in SYN segments:

- bit `0` (`0x1`) is set to `1` to enable PAWS and RTTM timestamps (RFC1323)
- bit `1` (`0x2`) is set to `1` to enable window scaling (RFC1323)
- bit `2` (`0x4`) is set to `1` to enable SACK (RFC2018)
- bit `3` (`0x8`) is set to `1` to enable ECN (RFC3128).

The values from `/proc/sys/net/ipv4/tcp_sack`, `/proc/sys/net/ipv4/tcp_timestamp` and `/proc/sys/net/ipv4/tcp_window_scaling` are used to find the default.

EF_TCP_TCONST_MSL

- **Name:** `msl_seconds`
- **Default:** `25`
- **Scope:** per-stack

The Maximum Segment Lifetime (as defined by the TCP RFC). A smaller value causes connections to spend less time in the `TIME_WAIT` state.

EF_TCP_TIME_WAIT_ASSASSINATION

- **Name:** `time_wait_assassinate`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Allow TCP `TIMEWAIT` state assassination, as with `/proc/sys/net/ipv4/tcp_rfc1337` set to `0`.

EF_TCP_TSOPT_MODE

- **Name:** `tcp_tsopt_mode`
- **Default:** `2`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Enable or disable per-stack TCP header timestamps (as defined in RFC 1323). Overrides system setting ipv4.tcp_timestamps and EF_TCP_SYN_OPTS. Possible values are:

- `0` - Disable TCP header timestamps
- `1` - Enable TCP header timestamps
- `2` - Use system settings (default).

EF_TCP_URG_MODE

- **Name:** `urg_mode`
- **Default:** `ignore`
- **Scope:** per-stack

- `allow` – process the urgent flag and pointer.
- `ignore` – ignore the urgent flag and pointer in received packets.

IMPORTANT! *Applications actually using urgent data will see corrupt streams.*

EF_TIMESTAMPING_REPORTING

- **Name:** `timestamping_reporting`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Controls timestamp reporting, possible values:

- `0`: report translated timestamps only when the NIC clock has been set
- `1`: report translated timestamps only when the system clock and the NIC clock are in sync (e.g. using ptpd)

If the above conditions are not met Onload will only report raw (not translated) timestamps.

EF_TXQ_SIZE

- **Name:** `txq_size`
- **Default:** `512`
- **Minimum:** `512`
- **Maximum:** `4096`
- **Scope:** per-stack

Set the size of the transmit descriptor ring. Valid values: `512`, `1024`, `2048` or `4096`.

EF_TX_MIN_IPG_CNTL

- **Name:** `tx_min_ipg_cntl`
- **Default:** `0`
- **Minimum:** `-1`
- **Maximum:** `20`
- **Scope:** per-stack

Rate pacing value.

EF_TX_PUSH

- **Name:** `tx_push`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enable low-latency transmit.

EF_TX_PUSH_THRESHOLD

- **Name:** `tx_push_thresh`
- **Default:** `100`
- **Minimum:** `1`
- **Scope:** per-stack

Sets a threshold for the number of outstanding sends before we stop using TX descriptor push. This has no effect if EF_TX_PUSH=`0`. It makes sense to set this value similar to EF_SEND_POLL_THRESH.

EF_TX_QOS_CLASS

- **Name:** `tx_qos_class`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Set the QOS class for transmitted packets on this Onload stack. Two QOS classes are supported: `0` and `1`. By default both Onload accelerated traffic and kernel traffic are in class `0`. You can minimize latency by placing latency sensitive traffic into a separate QOS class from bulk traffic.

EF_TX_TIMESTAMPING

- **Name:** `tx_timestamping`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `3`
- **Scope:** per-stack

Control of hardware timestamping of transmitted packets, possible values:

- `0` - do not do timestamping (default)
- `1` - request timestamping but continue if hardware is not capable or it does not succeed
- `2` - request timestamping and fail if hardware is capable and it does not succeed
- `3` - request timestamping and fail if hardware is not capable or it does not succeed.

EF_UDP

- **Name:** `ul_udp`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Clear to disable acceleration of new UDP sockets.

EF_UDP_CONNECT_HANDOVER

- **Name:** `udp_connect_handover`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

When set to 1, if a UDP socket is connected to an IP address that cannot be accelerated by Onload, or resource restrictions prevent RX acceleration, hand the socket over to the kernel stack.

When this option is disabled the socket remains under the control of Onload. This might be worthwhile because the socket might subsequently be re-connected to an IP address that can be accelerated, or the socket might be intended for TX use only.

When set to 2, hand the socket over on `connect()` even if the address could have been accelerated.

EF_UDP_FORCE_REUSEPORT

- **Name:** `udp_reuseports`
- **Default:** `0`
- **Scope:** per-process

This option specifies a comma-separated list of port numbers. UDP sockets that bind to those port numbers will have `SO_REUSEPORT` automatically applied to them.

EF_UDP_PORT_HANDOVER2_MAX

- **Name:** `udp_port_handover2_max`
- **Default:** `1`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER2_MIN), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_PORT_HANDOVER2_MIN

- **Name:** `udp_port_handover2_min`
- **Default:** `2`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER2_MAX), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_PORT_HANDOVER3_MAX

- **Name:** `udp_port_handover3_max`
- **Default:** `1`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER3_MIN), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_PORT_HANDOVER3_MIN

- **Name:** `udp_port_handover3_min`
- **Default:** `2`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER3_MAX), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_PORT_HANDOVER_MAX

- **Name:** `udp_port_handover_max`
- **Default:** `1`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER_MIN), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_PORT_HANDOVER_MIN

- **Name:** `udp_port_handover_min`
- **Default:** `2`
- **Scope:** per-stack

When set (together with EF_UDP_PORT_HANDOVER_MAX), this causes UDP sockets explicitly bound to a port in the given range to be handed over to the kernel stack. The range is inclusive.

EF_UDP_RCVBUF

- **Name:** `udp_rcvbuf_user`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX/2`
- **Scope:** per-stack

Override `SO_RCVBUF` for UDP sockets. (Note: the actual size of the buffer is double the amount requested, mimicking the behavior of the Linux kernel.)

EF_UDP_RECV_SPIN

- **Name:** `udp_recv_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking UDP receive calls until data arrives, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

EF_UDP_SEND_NONBLOCK_NO_PACKETS_MODE

- **Name:** `udp_nonblock_no_pkts_mode`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

This option controls how a non-blocking UDP `send()` call should behave if it is unable to allocate sufficient packet buffers. By default Onload will mimic Linux kernel stack behavior and block for packet buffers to be available. If set to `1`, this option will cause Onload to return error `ENOBUFS`. Note this option can cause some applications (that assume that a socket that is writable is able to send without error) to malfunction.

EF_UDP_SEND_SPIN

- **Name:** `udp_send_spin`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Spin in blocking UDP send calls until space becomes available in the socket buffer, the spin timeout expires or the socket timeout expires (whichever is the sooner). If the spin timeout expires, enter the kernel and block. The spin timeout is set by EF_SPIN_USEC or EF_POLL_USEC.

***Note*:** UDP sends usually complete very quickly, but can block if the application does a large burst of sends at a high rate. This option reduces jitter when such blocking is needed.

EF_UDP_SEND_UNLOCKED

- **Name:** `udp_send_unlocked`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Enables the 'unlocked' UDP send path. When enabled this option improves concurrency when multiple threads are performing UDP sends.

EF_UDP_SEND_UNLOCK_THRESH

- **Name:** `udp_send_unlock_thresh`
- **Default:** `1500`
- **Scope:** per-stack

UDP message size below which we attempt to take the stack lock early. Taking the lock early reduces overhead and latency slightly, but might increase lock contention in multi-threaded applications.

EF_UDP_SNDBUF

- **Name:** `udp_sndbuf_user`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `SMAX/2`
- **Scope:** per-stack

Override `SO_SNDBUF` for UDP sockets. (Note: the actual size of the buffer is double the amount requested, mimicking the behavior of the Linux kernel.)

EF_UL_EPOLL

- **Name:** `ul_epoll`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `3`
- **Scope:** per-process

Choose epoll implementation. The choices are:

- `0` - kernel (unaccelerated)
- `1` - user-level (accelerated, lowest latency)
- `2` - kernel-accelerated (best when there are lots of sockets in the set and mode `3` is not suitable)
- `3` - user-level (accelerated, lowest latency, scalable, supports socket caching).

The default is the user-level implementation (`1`).

Mode `3` can offer benefits over mode `1`, particularly with larger sets. However, this mode has some restrictions:

- It does not support epoll sets that exist across `fork()`.
- It does not support monitoring the readiness of the set's epoll fd via a another epoll/poll/select.

EF_UL_POLL

- **Name:** `ul_poll`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Clear to disable acceleration of `poll()` calls at user-level.

EF_UL_SELECT

- **Name:** `ul_select`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

Clear to disable acceleration of `select()` calls at user-level.

EF_UNCONFINE_SYN

- **Name:** `unconfine_syn`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Accept TCP connections that cross into or out-of a private network.

EF_UNIX_LOG

- **Name:** `log_level`
- **Default:** `3`
- **Scope:** per-process

A bitmask determining which kinds of diagnostics messages will be logged:

- `0x1` errors
- `0x2` unexpected
- `0x4` setup
- `0x8` verbose
- `0x10select()`
- `0x20poll()`
- `0x100` socket set-up
- `0x200` socket control
- `0x400` socket caching
- `0x1000` signal interception
- `0x2000` library enter/exit

- `0x4000` log call arguments
- `0x8000` context lookup
- `0x10000` pass-through
- `0x20000` very verbose
- `0x40000` verbose returned error
- `0x80000` very verbose errors: show 'ok' too
- `0x20000000` verbose transport control
- `0x40000000` very verbose transport control
- `0x80000000` verbose pass-through

EF_URG_RFC

- **Name:** `urg_rfc`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Choose between compliance with RFC1122 (`1`) or BSD behavior (`0`) regarding the location of the urgent point in TCP packet headers.

EF_USE_DSACK

- **Name:** `use_dsack`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

Whether or not to use DSACK (duplicate SACK).

The value from `/proc/sys/net/ipv4/tcp_dsack` is used by default.

EF_USE_HUGE_PAGES

- **Name:** `huge_pages`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-stack

Control of whether huge pages are used for packet buffers:

- `0` - no
- `1` - use huge pages if available (default)
- `2` - always use huge pages and fail if huge pages are not available.

Mode `1` prints syslog message if there is not enough huge pages in the system.

Mode `2` guarantees only initially-allocated packets to be in huge pages. It is recommended to use this mode together with EF_MIN_FREE_PACKETS, to control the number of such guaranteed huge pages. All non-initial packets are allocated in huge pages when possible. A syslog message is printed if the system is out of huge pages.

Non-initial packets can be allocated in non-huge pages without any warning in syslog for both mode `1` and `2` even if the system has free huge pages.

EF_VALIDATE_ENV

- **Name:** `validate_env`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-stack

When set this option validates Onload related environment variables (starting with EF_).

EF_VFORK_MODE

- **Name:** `vfork_mode`
- **Default:** `1`
- **Minimum:** `0`
- **Maximum:** `2`
- **Scope:** per-process

This option dictates how `vfork()` intercept should work. After a `vfork()`, parent and child still share address space but not file descriptors. We have to be careful about making changes in the child that can be seen in the parent. We offer three options here. Different apps might require different options depending on their use of `vfork()`. If using `EF_VFORK_MODE=2`, it is not safe to create sockets or pipes in the child before calling `exec()`.

- `0` - Old behavior. Replace `vfork()` with `fork()`
- `1` - Replace `vfork()` with `fork()` and block parent till child exits/execs
- `2` - Replace `vfork()` with `vfork()`.

EF_WODA_SINGLE_INTERFACE

- **Name:** `woda_single_if`
- **Default:** `0`
- **Minimum:** `0`
- **Maximum:** `1`
- **Scope:** per-process

This option alters the behavior of `onload_ordered_epoll_wait()`. This function would normally ensure correct ordering across multiple interfaces. However, this impacts latency, as only events arriving before the first interface polled can be returned and still guarantee ordering.

If the traffic being ordered is only arriving on a single interface then this additional constraint is not necessary. When this option is enabled, traffic will only be ordered relative to other traffic arriving on the same interface.

AMD

Appendix B

Meta Options

This appendix gives reference descriptions of meta options that can be used with Onload.

Environment Variables

There are several environment variables which act as meta options and set several of the options detailed in Appendix A: Parameter Reference. These are:

EF_POLL_USEC

Setting EF_POLL_USEC causes the following options to be set:

- EF_SPIN_USEC=EF_POLL_USEC
- EF_SELECT_SPIN=1
- EF_EPOLL_SPIN=1
- EF_POLL_SPIN=1
- EF_PKT_WAIT_SPIN=1
- EF_TCP_SEND_SPIN=1
- EF_TCP_RECV_SPIN=1
- EF_UDP_SEND_SPIN=1
- EF_UDP_RECV_SPIN=1
- EF_BUZZ_USEC=EF_POLL_USEC
- EF_SOCK_LOCK_BUZZ=1
- EF_STACK_LOCK_BUZZ=1

It does *not* set the following options, which must be set individually if required:

- EF_TCP_ACCEPT_SPIN
- EF_TCP_CONNECT_SPIN
- EF_PIPE_RECV_SPIN

- EF_PIPE_SEND_SPIN

Note: If neither of the spinning options; EF_POLL_USEC and EF_SPIN_USEC are set, Onload will resort to default interrupt driven behavior because the EF_INT_DRIVEN environment variable is enabled by default.

Note: When EF_POLL_USEC or EF_SPIN_USEC are greater than zero, EF_INT_DRIVEN will be zero.

See also EF_BUZZ_USEC.

EF_BUZZ_USEC

Setting EF_BUZZ_USEC sets the following options:

- EF_SOCK_LOCK_BUZZ=`1`
- EF_STACK_LOCK_BUZZ=`1`

Note: If EF_POLL_USEC is set to value N, then EF_BUZZ_USEC is also set to N only if N ≤ 100, If N > 100 then EF_BUZZ_USEC will be set to 100. This is deliberate as spinning for too long on internal locks may adversely affect performance. However the user can explicitly set EF_BUZZ_USEC value. For example `export EF_POLL_USEC=10000 export EF_BUZZ_USEC=1000`.

Appendix C

Build Dependencies

This appendix describes the build dependencies for Onload.

General

Before Onload network and kernel drivers can be built and installed, the target platform must support the following capabilities:

- Support a general C build environment that has `gcc`, `make`, `libc` and `libc-devel`.
- From version 201502 the following are required: `perl`, `autoconf`, `automake` and `libtool`.
- Can compile kernel modules, and so has the correct kernel-devel package for the installed kernel version.

Note: Onload builds have been tested against libtool versions 1.5.26 to 2.4.2. Users experiencing build issues with other libtool versions should contact support-nic@amd.com.

Building Kernel Modules

The kernel must be built with the following options enabled, where supported:

- `CONFIG_NETFILTER`
- `CONFIG_KALLSYMS`
- `EFRM_KALLSYMS_ALL`
- `CONFIG_FIB_RULES`
- `CONFIG_IP_MULTIPLE_TABLES`.

Standard distributions will already have these enabled, but they must also be enabled when building a custom kernel. These options do not affect performance.

The following commands can be used to install kernel development headers.

- Debian based Distributions - including Ubuntu (any kernel):

```
apt-get install linux-headers-$(uname -r)
```

- For RedHat/Fedora (not for 32 bit kernel):
    - If the system supports a 32 bit kernel and the kernel is PAE, then:

      ```
      yum -y install kernel-PAE-devel
      ```

    - otherwise:

      ```
      yum -y install kernel-devel
      ```

- For SuSE:

  ```
  yast -i kernel-source
  ```

onload

Onload requires the following to be installed:

- `autoconf`
- `automake`
- `bash`
- `binutils`
- `gawk`
- `gcc`
- `gettext`
- `glibc-common`
- `libcap-devel`.
- `libtool`
- `make`
- `sed`

onload_tcpdump

onload_tcpdump requires the following additional packages to be installed:

- `libpcap`
- `libpcap-devel`.

***Note*:** If these additional packages are not installed then onload_tcpdump will not be built, but the Onload build will succeed.

solar_clusterd

solar_clusterd requires the following additional packages to be installed:

- `python-devel`.

Note: If these additional packages are not installed then solar_clusterd will not be built, but the Onload build will succeed.

onload_bpftools

onload_bpftools requires the following additional packages to be installed:

- `libelf` development package
- the `elfutil-devel` or `libelf-dev` packages are required to use the 'load' option.

Note: If these additional packages are not installed then onload_bpftools will not be built, but the Onload build will succeed.

IPv6 Support

Onload IPv6 support requires Linux kernel 4.4 or later, or RHEL7.4 or later.

Red Hat Enterprise Linux 8.x

Users of RHEL 8.0 and later releases must ensure the CodeReady Builder repository is included in the system repository list. This is a requirement to provide the libcap build dependency.

Appendix D

Onload Extensions API

The Onload Extensions API allows the user to customize an application using advanced features to improve performance.

The Extensions API does not create any runtime dependency on Onload and an application using the API can run without Onload. The license for the API and associated libraries is a BSD 2-Clause License.

API groups

This appendix covers the following API groups within the Extensions API:

- Common Components
- Stacks API
- Zero-Copy API
- Templated Sends API
- Delegated Sends API

Source code

The Extensions API source code is provided with the Onload distribution. Entry points for the source code are:

- `src/lib/transport/unix/onload_ext_intercept.c`
- `src/lib/transport/unix/zc_intercept.c`

Java Native Interface Wrapper

The Onload distribution includes a JNI wrapper for use with the Extensions API. Java users should also refer to the files:

- `/onload-<version>/src/tools/jni`

Common Components

For all applications employing the Extensions API the following components are provided:

- `#include <onload/extensions.h>`

  An application should include the header file containing function prototypes and constant values required when using the API.

- `libonload_ext.a`, `libonload_ext.so`

  This library provides stub implementations of the extended API. An application that wishes to use the extensions API should link against this library.

  When Onload is not present, the application will continue to function, but calls to the extensions API will have no effect (unless documented otherwise).

  - To link dynamically to this library include the '-l' linker option on the compiler command line:

    ```
    -lonload_ext
    ```

  - You can instead link against the `libonload_ext.a` static library. This is required to run the application on servers that do not have the dynamic libraries installed. When doing so, it is necessary to also link with the dynamic library by adding the 'ldl' option to the compiler command line.

    ```
    -ldl -l:libonload_ext.a
    ```

onload_is_present

Description

If the application is linked with `libonload_ext`, but not running with Onload this will return 0. If the application is running with Onload this will return 1.

Definition

```
int onload_is_present (void)
```

Formal Parameters

None.

Return Value

1 from `libonload.so` library, or 0 from `libonload_ext.a` library

onload_fd_stat

```
struct onload_stat
{
  int32_t    stack_id;
  char*      stack_name;
  int32_t    endpoint_id;
  int32_t    endpoint_state;
};

extern int onload_fd_stat(int fd, struct onload_stat* stat);
```

Description

Retrieves internal details about an accelerated socket.

Definition

See above.

Formal Parameters

See above.

Return Value

0 socket is not accelerated

1 socket is accelerated

-ENOMEM when memory cannot be allocated

Notes

When calling `free()` on `stack_name` use the (char *) because memory is allocated using malloc.

This function will call `malloc()` and so should never be called from any other function requiring a malloc lock.

***Note*:** Can be used to check if a fd is accelerated without allocating memory if stat is declared as NULL.

onload_fd_check_feature

```
enum onload_fd_feature {
  /* Check whether this fd supports ONLOAD_MSG_WARM or not */
  ONLOAD_FD_FEAT_MSG_WARM,
  /* see Notes for details */
  ONLOAD_FD_FEAT_UDP_TX_TS_HDR
};

int onload_fd_check_feature (int fd, enum onload_fd_feature feature);
```

Description

Used to check whether the Onload file descriptor supports a feature or not.

Definition

See above.

Formal Parameters

See above.

Return Value

0 if the feature is supported but not on this fd

>0 if the feature is supported both by onload and this fd

<0 if the feature is not supported:

-ENOSYS if `onload_fd_check_feature()` is not supported.

- ENOTSUPP if the feature is not supported by onload.

Notes

Onload-201509 and later versions support the ONLOAD_FD_FEAT_UDP_TX_TS_HDR option. `onload_fd_check_feature()` will return 1 to indicate that a recvmsg used to retrieve TX timestamps for UDP packets will return the entire Ethernet header.

***Note*:** When run on older versions of onload this will return -EOPNOTSUPP.

onload_thread_set_spin

Description

For a thread calling this function, `onload_thread_set_spin()` sets the per-thread spinning actions, it is not per-stack and not per-socket.

Definition

```
int onload_thread_set_spin(enum onload_spin_type type, unsigned spin)
```

Formal Parameters

- `type`: Which operation to change the spin status of. The `type` must be one of the following:

```
enum onload_spin_type {
  ONLOAD_SPIN_ALL,         /* enable or disable all spin options */
  ONLOAD_SPIN_UDP_RECV,
  ONLOAD_SPIN_UDP_SEND,
  ONLOAD_SPIN_TCP_RECV,
  ONLOAD_SPIN_TCP_SEND,
  ONLOAD_SPIN_TCP_ACCEPT,
  ONLOAD_SPIN_PIPE_RECV,
  ONLOAD_SPIN_PIPE_SEND,
  ONLOAD_SPIN_SELECT,
  ONLOAD_SPIN_POLL,
  ONLOAD_SPIN_PKT_WAIT,
  ONLOAD_SPIN_EPOLL_WAIT,
  ONLOAD_SPIN_STACK_LOCK,
  ONLOAD_SPIN_SOCK_LOCK,
  ONLOAD_SPIN_SO_BUSY_POLL,
  ONLOAD_SPIN_TCP_CONNECT,
  ONLOAD_SPIN_MIMIC_EF_POLL, /* thread spin configuration which mimics
                              * spin settings in EF_POLL_USEC. Note
                              * that this has no effect on the
                              * usec-setting part of EF_POLL_USEC.
                              * This needs to be set separately
                              */
  ONLOAD_SPIN_MAX /* special value to mark largest valid input */
};
```

- `spin`: A boolean which indicates whether the operation should spin or not.

Return Value

0 on success

-EINVAL if unsupported type is specified.

Notes

Spin time (for all threads) is set using the per-process EF_SPIN_USEC parameter.

Examples

The `onload_thread_set_spin` API can be used to control spinning on a per-thread or per-API basis. The existing spin-related configuration options set the default behavior for threads, and the `onload_thread_set_spin` API overrides the default for the thread calling this function.

- **Disable All Sorts of Spinning:**

```
onload_thread_set_spin(ONLOAD_SPIN_ALL, 0);
```

- **Enable All Sorts of Spinning:**

```
onload_thread_set_spin(ONLOAD_SPIN_ALL, 1);
```

- **Enable Spinning only for Certain Threads:**

  1. Set the spin timeout by setting EF_SPIN_USEC, and disable spinning by default by setting EF_POLL_USEC=0.
  2. In each thread that should spin, invoke `onload_thread_set_spin()`.

- **Disable Spinning only in Certain Threads:**

  1. Enable spinning by setting EF_POLL_USEC=<timeout>.
  2. In each thread that should not spin, invoke `onload_thread_set_spin()`.

> **IMPORTANT!** *If a thread is set to NOT spin and then blocks this might invoke an interrupt for the whole stack. Interrupts occurring on moderately busy threads can cause unintended and undesirable consequences.*

- **Enable Spinning for UDP Traffic, but not TCP Traffic:**

  1. Set the spin timeout by setting EF_SPIN_USEC, and disable spinning by default by setting EF_POLL_USEC=0.
  2. In each thread that should spin (UDP only), do:

```
onload_thread_set_spin(ONLOAD_SPIN_UDP_RECV, 1)
onload_thread_set_spin(ONLOAD_SPIN_UDP_SEND, 1)
```

- **Enable Spinning for TCP Traffic, but not UDP Traffic:**

  1. Set the spin timeout by setting EF_SPIN_USEC, and disable spinning by default by setting EF_POLL_USEC=0.
  2. In each thread that should spin (TCP only), do:

```
onload_thread_set_spin(ONLOAD_SPIN_TCP_RECV, 1)
onload_thread_set_spin(ONLOAD_SPIN_TCP_SEND, 1)
onload_thread_set_spin(ONLOAD_SPIN_TCP_ACCEPT, 1)
```

Spinning and Sockets

When a thread calls `onload_thread_set_spin()` it sets the spinning actions applied when the thread accesses any socket - irrespective of whether the socket is created by this thread.

If a socket is created by thread-A and is accessed by thread-B, calling `onload_thread_set_spin(`ONLOAD_SPIN_ALL, 1`)` only from thread-B will enable spinning for thread-B, but not for thread-A. In the same scenario, if `onload_thread_set_spin(`ONLOAD_SPIN_ALL, 1`)` is called only from thread-A, then spinning is enabled only for thread-A, but not for thread-B.

The `onload_thread_set_spin()` function sets the per-thread spinning action.

onload_thread_get_spin

Description

For the current thread, identify which operations should spin.

Definition

```
int onload_thread_get_spin(unsigned *state)
```

Formal Parameters

- `state`: Location at which to write the spin status as a bitmask. Bit *n* of the mask is set if spinning has been enabled for spin type *n* (see onload_thread_set_spin).

Return Value

0 on success.

Notes

Spin time (for all threads) is set using the EF_SPIN_USEC parameter.

Examples

Determine if spinning is enabled for UDP receive:

```
unsigned state;
onload_thread_get_spin(&state);
if (state & (1 << ONLOAD_SPIN_UDP_RECV)) {
  // spinning is enabled for UDP receive
}
```

onload_socket_nonaccel

Description

Create a socket which is not accelerated by Onload. This function is useful when attempting to reserve a port for an ephemeral ef_vi instance without installing Onload filters. It is also possible to use the stackname API to disable acceleration for specific socket(s).

Definition

```
int onload_socket_nonaccel(int domain int type, int protocol)
```

Formal Parameters

This function takes arguments and returns values that correspond exactly to the standard `socket()` function call.

Return Value

Return the file descriptor that refers to the created endpoint.

-1 with errno ENOSYS if the Onload extensions library is not in use.

onload_socket_unicast_nonaccel

Description

Create a socket that will only accelerate multicast traffic. If this socket is not able to receive multicast, for example, because it is bound to a unicast local address, or it is a TCP socket, then it will be handed over to the kernel.

This function is useful for cases where a socket will be used solely for multicast traffic to avoid consuming limited filter table resource. This does not prevent unicast traffic from arriving at the socket, and if appropriate traffic is received, it will still be delivered via the unaccelerated path. It is most useful for sockets that are bound to INADDR_ANY, because for these Onload must install a filter per IP address that is configured on an accelerated interface, on each accelerated hardware port.

If a socket is bound to a multicast local address, then no unicast filters will be installed, so there is no need for this function.

Definition

```
int onload_socket_unicast_nonaccel(int domain, int type, int protocol)
```

Formal Parameters

This function takes arguments and returns values that correspond exactly to the standard `socket()` function call.

Return Value

Return the file descriptor that refers to the created endpoint.

-1 with errno ENOSYS if the Onload extensions library is not in use.

Stacks API

Using the Onload Extensions API an application can bind selected sockets to specific Onload stacks and in this way ensure that time-critical sockets are not starved of resources by other non-critical sockets. The API allows an application to select sockets which are to be accelerated thus reserving Onload resources for performance critical paths. This also prevents non-critical paths from creating jitter for critical paths.

onload_set_stackname

Description

Select the Onload stack that new sockets are placed in. A socket can exist only in a single stack. A socket can be moved to a different stack - see onload_move_fd() below.

Moving a socket to a different stack does not create a copy of the socket in originator and target stacks.

Definition

```
int onload_set_stackname(int who, int scope, const char *name)
```

Formal Parameters

- `who`: Must be one of the following:
  - ONLOAD_THIS_THREAD - to modify the stack name in which all subsequent sockets are created by this thread.
  - ONLOAD_ALL_THREADS - to modify the stack name in which all subsequent sockets are created by all threads in the current process. ONLOAD_THIS_THREAD takes precedence over ONLOAD_ALL_THREADS.
- `scope`: Must be one of the following:

- ONLOAD_SCOPE_THREAD - name is scoped with current thread
- ONLOAD_SCOPE_PROCESS - name is scoped with current process
- ONLOAD_SCOPE_USER - name is scoped with current user
- ONLOAD_SCOPE_GLOBAL - name is global across all threads, users and processes.
- ONLOAD_SCOPE_NOCHANGE - undo effect of a previous call to onload_set_stackname(ONLOAD_THIS_THREAD, ...), see Note 4.

- `name`: One of the following:
  - the stack name up to eight characters.
  - an empty string to set no stackname
  - the special value ONLOAD_DONT_ACCELERATE to prevent sockets created in this thread, user, process from being accelerated.

Sockets identified by the options above will belong to the Onload stack until a subsequent call using onload_set_stackname identifies a different stack or the ONLOAD_SCOPE_NOCHANGE option is used.

Return Value

0 on success

-1 with errno set to ENAMETOOLONG if the name exceeds permitted length

-1 with errno set to EINVAL if other parameters are invalid.

Note 1

This applies for stacks selected for sockets created by `socket()` and for `pipe()`, it has no effect on `accept()`. Passively opened sockets created via `accept()` will always be in the same stack as the listening socket that they are linked to. This means that the following are functionally identical:

```
onload_set_stackname(foo)
socket
listen
onload_set_stackname(bar)
accept
```

and:

```
onload_set_stackname(foo)
socket
listen
accept
onload_set_stackname(bar)
```

In both cases the listening socket and the accepted socket will be in stack foo.

Note 2

Scope defines the namespace in which a stack belongs. A stackname of foo in scope user is not the same as a stackname of foo in scope thread. Scope restricts the visibility of a stack to either the current thread, current process, current user or is unrestricted (global). This has the property that with, for example, process based scoping, two processes can have the same stackname without sharing a stack - as the stack for each process has a different namespace.

Note 3

Scoping can be thought of as adding a suffix to the supplied name, for example:

ONLOAD_SCOPE_THREAD: <stackname>-t<thread_id>

ONLOAD_SCOPE_PROCESS: <stackname>-p<process_id>

ONLOAD_SCOPE_USER: <stackname>-u<user_id>

ONLOAD_SCOPE_GLOBAL: <stackname>

This is an example only and the implementation is free to do something different such as maintaining different lists for different scopes.

Note 4

ONLOAD_SCOPE_NOCHANGE will undo the effect of a previous call to onload_set_stackname(ONLOAD_THIS_THREAD, ...).

If you have previously used onload_set_stackname(ONLOAD_THIS_THREAD, ...) and want to revert to the behavior of threads that are using the ONLOAD_ALL_THREADS configuration, without changing that configuration, you can do the following:

```
onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_NOCHANGE, "");
```

Related Environment Variables

Related environment variables are:

- **EF_DONT_ACCELERATE:**

  Default: `0`

  Minimum: `0`

  Maximum: `1`

  Scope: Per-process

If this environment variable is set then acceleration for ALL sockets is disabled and handed off to the kernel stack until the application overrides this state with a call to `onload_set_stackname()`.

- **EF_STACK_PER_THREAD:**

  Default: `0`

  Minimum: `0`

  Maximum: `1`

  Scope: Per-process

  If this environment variable is set each socket created by the application will be placed in a stack depending on the thread in which it is created. Stacks could, for example, be named using the thread ID of the thread that creates the stack, but this should not be relied upon.

  A call to `onload_set_stackname` overrides this variable. EF_DONT_ACCELERATE takes precedence over this variable.

- **EF_NAME:**

  Default: none

  Minimum: `0`

  Maximum: `8`

  Scope: Per-stack

  The environment variable EF_NAME will be honored to control Onload stack sharing. However, a call to `onload_set_stackname` overrides this variable and, EF_DONT_ACCELERATE and EF_STACK_PER_THREAD both take precedence over EF_NAME.

onload_move_fd

Description

Move the file descriptor to the current stack. The target stack can be specified with `onload_set_stackname()`, then use `onload_move_fd()` to put the socket into the target stack.

A socket can exist only in a single stack. Moving a socket to a different stack does not create a copy of the socket in originator and target stacks. Limited to TCP closed or accepted sockets only.

Definition

```
int onload_move_fd (int fd)
```

Formal Parameters

fd - the file descriptor to be moved to the current stack.

Return Value

0 on success

non-zero otherwise.

Notes

- Useful to move fds obtained by `accept()` to a different Onload stack from the listening socket.
- Cannot be used on actively opened connections, although it is possible to use `onload_set_stackname()` before calling `connect()` to achieve the same result.
- The socket must have empty send and retransmit queues (send not called on this socket)
- The socket must have a simple receive queue (no loss, reordering, etc)
- The fd is not yet in an epoll set.
- The onload_move_fd function should not be used if SO_TIMESTAMPING is set to a non-zero value for the originating socket.
- Should not be used simultaneously with other I/O multiplex actions such as `poll()`, `select()`, `recv()` etc on the file descriptor.
- This function is not async-safe and should never be called from any process function handling signals.
- This function cannot be used to hand sockets over to the kernel. It is not possible to use onload_set_stackname (ONLOAD_DONT_ACCELERATE) and then onload_move_fd().

***Note*:** The onload_move_fd function does not check whether a destination stack has either RX or TX timestamping enabled.

onload_stackname_save

Description

Save the state of the current onload stack identified by the previous call to `onload_set_stackname()`.

Definition

```
int onload_stackname_save (void)
```

Formal Parameters

none

Return Value

0 on success

-ENOMEM when memory cannot be allocated.

onload_stackname_restore

Description

Restore stack state saved with a previous call to `onload_stackname_save()`. All updates/ changes to state of the current stack will be deleted and all state previously saved will be restored. To avoid unexpected results, the stack should be restored in the same thread as used to call `onload_stackname_save()`.

Definition

```
int onload_stackname_restore (void)
```

Formal Parameters

none

Return Value

0 on success

non-zero if an error occurs.

Notes

The API stackname save and restore functions provide flexibility when binding sockets to an Onload stack.

Using a combination of `onload_set_stackname()`, `onload_stackname_save()` and `onload_stackname_restore()`, the user is able to create default stack settings which apply to one or more sockets, save this state and then create changed stack settings which are applied to other sockets. The original default settings can then be restored to apply to subsequent sockets.

onload_stack_opt_set_int

Description

Set/modify per-stack options for all subsequently created stacks. These override any global per-stack environment options already set. When using this function - check the scope field on the environment variable, for example:

- **EF_NAME:**

  Default: none

  Minimum: `0`

  Maximum: `8`

  Scope: Per-stack

The `onload_stack_opt_set_int()` function has no effect for per-process, per-thread options which should be set with `onload_thread_set_spin()`.

Definition

```
int onload_stack_opt_set_int(const char* name, int64_t value)
```

Formal Parameters

- `name`: Stack option to modify
- `value`: New value for the stack option.

Example

```
onload_stack_opt_set_int("EF_SCALABLE_FILTERS_ENABLE", 1);
```

Return Value

0 on success

errno set to EINVAL if the requested option is not found or ENOMEM.

Notes

- Cannot be used to modify options on existing stacks - only for new stacks.
- Cannot be used to modify per-process options - only per-stack options.
- Modified options will be used for all newly created stacks until `onload_stack_opt_reset()` is called.

onload_stack_opt_reset

Description

Revert to using global stack options for newly created stacks.

Definition

```
int onload_stack_opt_reset(void)
```

Formal Parameters

None.

Return Value

0 always

Notes

Should be called following a call to `onload_stack_opt_set_int()` to revert to using global stack options for all newly created stacks.

onload_ordered_epoll_wait

For details of the Wire Order Delivery feature refer to Wire Order Delivery.

Description

If the epoll set contains accelerated sockets in only one stack this function can be used instead of `epoll_wait()` to return events in the order these were recovered from the wire. There is no explicit check on sockets, so applications must ensure that the rules are applied to avoid mis-ordering of packets.

Definition

```
int onload_ordered_epoll_wait (
  int epfd,
  struct epoll_event *events,
  struct onload_ordered_epoll_event *oo_events,
  int maxevents,
  int timeout);
```

Formal Parameters

See definition `epoll_wait()`.

Return Value

- A positive value identifies the number of epoll_evs / ordered_evs to process.
- A zero value indicates there are no events which can be processed while maintaining ordering, that is there might be no data or only unordered data.
- A negative return value identifies an error condition.

Notes

Any file descriptors returned as ready without a valid timestamp (tv_sec = 0), should be considered unordered with respect to the rest of the set. This can occur for data received via the kernel or data returned without a hardware timestamp, that is from an interface that does not support hardware timestamping.

The environment variable EF_UL_EPOLL`=1` must be set if hardware timestamps are required. This feature is only available on the SFN8000 and X2 series adapters.

```
struct onload_ordered_epoll_event{
  /* The hardware timestamp of the first readable data */
  struct timespec ts;
  /* Number of bytes that may be read to maintain wire order */
  int bytes
};
```

ONLOAD_MSG_ONEPKT and EF_TCP_RCVBUF_STRICT are incompatible with the wire order delivery feature. Refer to Wire Order Delivery for details.

Use the environment variable EF_RX_TIMESTAMPING_ORDERING to select either the (default) NIC hardware timestamps or external timestamps from cPacket trailers applied by upstream external equipment. When using external timestamps, packets might appear out of order due to external delays unknown to Onload.

onload_timestamping_request

Can be called instead of `setsockopt(SO_TIMESTAMPING)` to enable packet timestamping in multiple formats.

Description

Receive packet timestamps from multiple sources with sub-nanosecond resolution. Supported sources are hardware timestamps generated by the Solarflare adapter and timestamps applied by an external equipment using the cPacket format.

Definition

```
int onload_timestamping_request (int fd, unsigned flags);
```

Formal Parameters

- `fd`: Socket file descriptor
- `flags`: Onload_timestamping_flags:

```
enum onload_timestamping_flags {
  /* Request NIC timestamps for sent packets */
  ONLOAD_TIMESTAMPING_FLAG_TX_NIC = 1 << 0,

  /* Request NIC and/or external timestamps for received packets */
  ONLOAD_TIMESTAMPING_FLAG_RX_NIC = 1 << 1,
  ONLOAD_TIMESTAMPING_FLAG_RX_CPACKET = 1 << 2,
};
```

Return Value

0 on success

EINVAL unknown flag is set

ENOTTY fd does not refer to an Onload-accelerated socket

EOPNOTSUPP this build of Onload does not support timestamping

Stacks API Examples

Using a combination of the EF_DONT_ACCELERATE environment variable and the function `onload_set_stackname()`, the user is able to control/select sockets which are to be accelerated and isolate these performance critical sockets and threads from the rest of the system.

The following examples demonstrate using `onload_set_stackname()`:

- This thread will use stack foo, other threads in the stack will continue as before.

```
onload_set_stackname(ONLOAD_THIS_THREAD, ONLOAD_SCOPE_GLOBAL, "foo")
```

- All threads in this process will get their own stack called foo. This is equivalent to the EF_STACK_PER_THREAD environment variable.

```
onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_THREAD, "foo")
```

- All threads in this process will share a stack called foo. If another process did the same function call it will get its own stack.

```
onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_PROCESS, "foo")
```

- All threads in this process will share a stack called foo. If another process run by the same user did the same, it would share the same stack as the first process. If another process run by a different user did the same it would get is own stack.

  ```
  onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_USER, "foo")
  ```

- Equivalent to EF_NAME. All threads will use a stack called foo which is shared by any other process which does the same.

  ```
  onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_GLOBAL, "foo")
  ```

- Equivalent to EF_DONT_ACCELERATE. New sockets/pipes will not be accelerated until another call to `onload_set_stackname()`.

  ```
  onload_set_stackname(ONLOAD_ALL_THREADS, ONLOAD_SCOPE_GLOBAL,
  ONLOAD_DONT_ACCELERATE)
  ```

Zero-Copy API

Zero-Copy can improve the performance of networking applications by eliminating intermediate buffers when transferring data between application and network adapter.

The Onload Extensions Zero-Copy API supports zero-copy of UDP received packet data and TCP transmit packet data.

The API provides the following components:

- `#include <onload/extensions_zc.h>`

  In addition to the common components, an application should include this header file which contains all function prototypes and constant values required when using the API. The header file also includes comprehensive documentation, required data structures and function definitions.

Zero-Copy Data Buffers

To avoid the copy data is passed to and from the application in special buffers described by a `struct onload_zc_iovec`. A message or datagram can consist of multiple iovecs using a `struct onload_zc_msg`. A single call to send can involve multiple messages using an array of `struct onload_zc_mmsg`.

Figure 27: **Zero-Copy Data Buffers**

```
/* A zc_iovec describes a single buffer */
struct onload_zc_iovec {
  void* iov_base;              /* Address within buffer */
  size_t iov_len;              /* Length of data */
  onload_zc_handle buf;        /* (opaque) buffer handle */
```

```
  unsigned iov_flags;            /* Not currently used */
};
/* A msg describes array of iovecs that make up datagram */
struct onload_zc_msg {
  struct onload_zc_iovec* iov;   /* Array of buffers */
  struct msghdr msghdr;          /* Message metadata */
};
/* An mmsg describes a message, the socket, and its result */
struct onload_zc_mmsg {
  struct onload_zc_msg msg;      /* Message */
  int rc;                        /* Result of send operation */
  int fd;                        /* socket to send on */
};
```

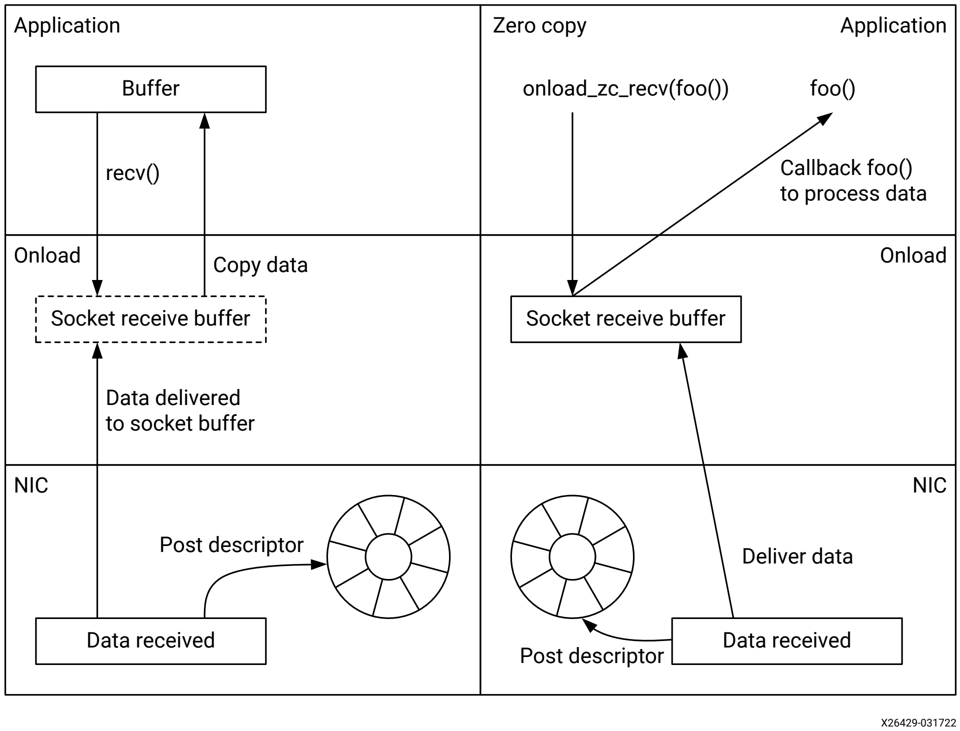
Zero-Copy UDP Receive Overview

The following figure illustrates the difference between the normal UDP receive mode and the zero-copy method.

When using the standard POSIX socket calls, the adapter delivers packets to an Onload packet buffer which is described by a descriptor previously placed in the RX descriptor ring. When the application calls `recv()`, Onload copies the data from the packet buffer to an application-supplied buffer.

Using the zero-copy UDP receive API the application calls the `onload_zc_recv()` function including a callback function which will be called when data is ready. The callback can directly access the data inside the Onload packet buffer avoiding a copy.

Figure 28: **Traditional vs. Zero-Copy UDP Receive**



A single call using `onload_zc_recv()` function can result in multiple datagrams being delivered to the callback function. Each time the callback returns to Onload the next datagram is delivered. Processing stops when the callback instructs Onload to cease delivery or there are no further received datagrams.

If the receiving application is filtering and so does not require to look at all data received this can result in a considerable performance advantage because this data is not pulled into the processor's cache, thereby reducing the application cache footprint.

As a general rule, the callback function should avoid calling other system calls which attempt to modify or close the current socket.

Zero-copy UDP Receive is implemented within the Onload Extensions API.

Zero-Copy UDP Receive

The `onload_zc_recv()` function specifies a callback to invoke for each received UDP datagram. The callback is invoked in the context of the call to `onload_zc_recv()` (it blocks/ spins waiting for data).

Before calling, the application must set the following in the `struct onload_zc_recv_args`:

Table 43: **onload_zc_recv_args Fields**

Field	Value
`cb`	Set to the callback function pointer.
`user_ptr`	Set to point to application state, this is not touched by Onload

`msg.msghdr.msg_control`
`msg_controllen`
`msg_name`
`msg_namelen`
	The user application should set these to appropriate buffers and lengths (if required) as you would for recvmsg (or NULL and 0 if not used).

`flags`	Set to indicate behavior (for example ONLOAD_MSG_DONTWAIT).

Figure 29: **Zero-Copy recv_args**

```
typedef enum onload_zc_callback_rc
(*onload_zc_recv_callback)(struct onload_zc_recv_args *args, int flags);
struct onload_zc_recv_args
{
  struct onload_zc_msg msg;
  onload_zc_recv_callback cb;
  void* user_ptr;
  int flags;
};
int onload_zc_recv(int fd, struct onload_zc_recv_args *args);
```

The callback gets to examine the data, and can control what happens next:

- whether or not the buffer(s) are kept by the callback or are immediately freed by Onload
- whether or not `onload_zc_recv()` will internally loop and invoke the callback with the next datagram, or immediately return to the application.

The next action is determined by setting flags in the return code as follows:

Table 44: **Flags for Setting in the Return Code**

Flag	Description
`ONLOAD_ZC_KEEP`	The callback function can elect to retain ownership of received buffer(s) by returning ONLOAD_ZC_KEEP. Following this, the correct way to release retained buffers is to call `onload_zc_release_buffers()` to explicitly release the first buffer from each received datagram. Subsequent buffers pertaining to the same datagram will then be automatically released.
`ONLOAD_ZC_CONTINUE`	To suggest that Onload should loop and process more datagrams

Table 44: **Flags for Setting in the Return Code** *(cont'd)*

Flag	Description
`ONLOAD_ZC_TERMINATE`	To insist that Onload immediately return from the `onload_zc_recv()`

Flags can also be set by Onload:

Table 45: **Flags That Can Be Set By Onload**

Flag	Description
`ONLOAD_ZC_END_OF_BURST`	Onload sets this flag to indicate that this is the last packet
`ONLOAD_ZC_MSG_SHARED`	Packet buffers are read only

If there is unaccelerated data on the socket from the kernel's receive path this cannot be handled without copying. The application has two choices as follows:

Table 46: **Flags for Handling Unaccelerated Data**

Flag	Description
`ONLOAD_MSG_RECV_OS_INLINE`	Set this flag when calling `onload_zc_recv()`. Onload will deal with the kernel data internally and pass it to the callback
check return code	Check the return code from `onload_zc_recv()`. If it returns `ENOTEMPTY` then the application must call `onload_recvmsg_kernel()` to retrieve the kernel data.

Zero-Copy Receive Example #1

Figure 30: **Zero-Copy Receive Example #1**

```
struct onload_zc_recv_args args;
struct zc_recv_state state;
int rc;
state.bytes = bytes_to_wait_for;
/* Easy way to set msg_control* and msg_name* to zero */
memset(&args.msg, 0, sizeof(args.msg));
args.cb = &zc_recv_callback;
args.user_ptr = &state;
args.flags = ONLOAD_ZC_RECV_OS_INLINE;
rc = onload_zc_recv(fd, &args);
//---
enum onload_zc_callback_rc
zc_recv_callback(struct onload_zc_recv_args *args, int flags)
{
  int i;
  struct zc_recv_state *state = args->user_ptr;
```

```
  for( i = 0; i < args->msg.msghdr.msg_iovlen; ++i ) {
    printf("zc callback iov %d: %p, %d", i,
           args->msg.iov[i].iov_base,
           args->msg.iov[i].iov_len);
    state->bytes -= args->msg.iov[i].iov_len;
  }
  if( state->bytes <= 0 ) return ONLOAD_ZC_TERMINATE;
  else return ONLOAD_ZC_CONTINUE;
}
```

Zero-Copy Receive Example #2

Figure 31: **Zero-Copy Receive Example #2**

```
static enum onload_zc_callback_rc
zc_recv_callback(struct onload_zc_recv_args *args, int flag)
{
  struct user_info *zc_info = args->user_ptr;
  int i, zc_rc = 0;
  for( i = 0; i < args->msg.msghdr.msg_iovlen; ++i ) {
    zc_rc += args->msg.iov[i].iov_len;
    handle_msg(args->msg.iov[i].iov_base,
               args->msg.iov[i].iov_len);
  }
  if( zc_rc == 0 )
    return ONLOAD_ZC_TERMINATE;
  zc_info->zc_rc += zc_rc;
  if( (zc_info->flags & MSG_WAITALL) &&
      (zc_info->zc_rc < zc_info->size) )
    return ONLOAD_ZC_CONTINUE;
  else return ONLOAD_ZC_TERMINATE;
}
struct onload_zc_recv_args zc_args;
ssize_t do_recv_zc(int fd, void* buf, size_t len, int flags)
{
  struct user_info info; int rc;
  init_user_info(&info);
  memset(&zc_args, 0, sizeof(zc_args));
  zc_args.user_ptr = &info;
  zc_args.flags = 0;
  zc_args.cb = &zc_recv_callback;
  if( flags & MSG_DONTWAIT )
    zc_args.flags |= ONLOAD_MSG_DONTWAIT;
  rc = onload_zc_recv(fd, &zc_args);
  if( rc == -ENOTEMPTY) {
    if( ( rc = onload_recvmsg_kernel(fd, &msg, 0) ) < 0 )
      printf("onload_recvmsg_kernel failed\n");
  }
  else if( rc == 0 ) {
    /* zc_rc gets set by callback to bytes received, so we
     * can return that to appear like standard recv call */
    rc = info.zc_rc;
  }
  return rc;
}
```

Note: `onload_zc_recv()` should not be used together with `onload_set_recv_filter()` and only supports accelerated (Onloaded) sockets. For example, when bound to a broadcast address the socket fd is handed off to the kernel and this function will return `ESOCKNOTSUPPORT`.

Zero-Copy TCP Send Overview

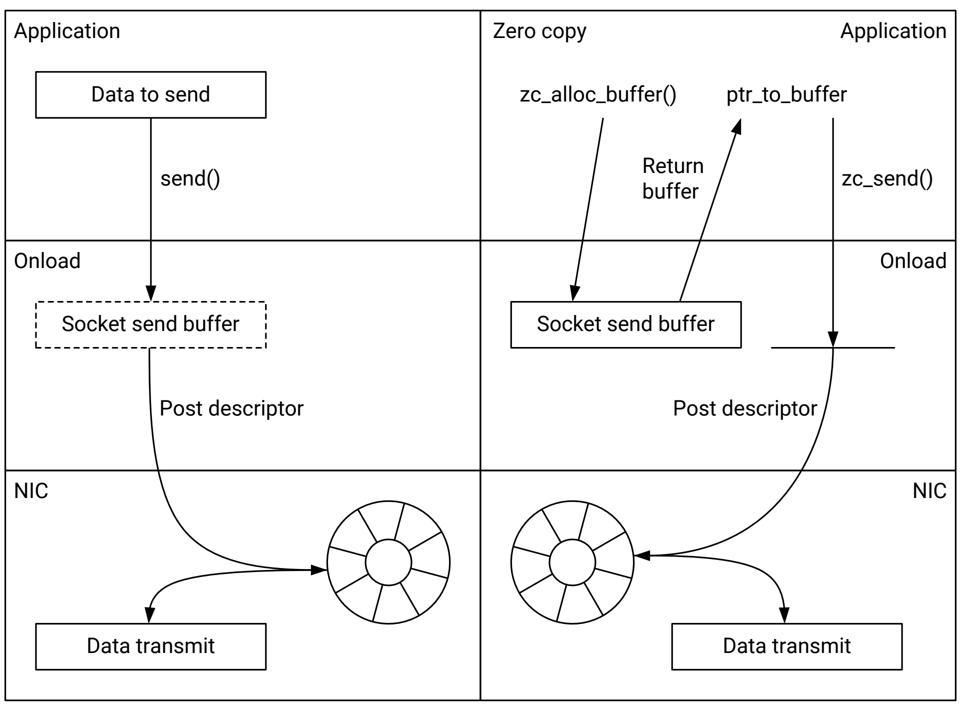
The following figure illustrates the difference between the normal TCP transmit method and the zero- copy method.

When using standard POSIX socket calls, the application first creates the payload data in an application allocated buffer before calling the `send()` function. Onload will copy the data to a Onload packet buffer in memory and post a descriptor to this buffer in the network adapter TX descriptor ring.

Using the zero-copy TCP transmit API the application calls the `onload_zc_alloc_buffers()` function to request buffers from Onload. A pointer to a packet buffer is returned in response. The application places the data to send directly into this buffer and then calls `onload_zc_send()` to indicate to Onload that data is available to send.

Onload will post a descriptor for the packet buffer in the network adapter TX descriptor ring and ring the TX doorbell. The network adapter fetches the data for transmission.

Figure 32: **Traditional vs. Zero-Copy TCP Transmit**



***Note*:** The socket used to allocate zero-copy buffers must be in the same stack as the socket used to send the buffers. When using TCP loopback, Onload can move a socket from one stack to another. Users must ensure that they *always use buffers from the correct stack*.

***Note*:** The `onload_zc_send` function does not currently support the ONLOAD_MSG_MORE or TCP_CORK flags.

Zero-copy TCP transmit is implemented within the Onload Extensions API.

Zero-Copy TCP Send

The zero-copy send API supports the sending of multiple messages to different sockets in a single call. Data buffers must be allocated in advance and for best efficiency these should be allocated in blocks and off the critical path. The user should avoid simply moving the copy from Onload into the application, but where this is unavoidable, it should also be done off the critical path.

Figure 33: **Zero-Copy Send**

```
int onload_zc_send(struct onload_zc_mmsg* msgs, int mlen, int flags);
```

Figure 34: **Zero-Copy Allocate Buffers**

```
int onload_zc_alloc_buffers(int fd,
                            struct onload_zc_iovec* iovecs,
                            int iovecs_len,
                            onload_zc_buffer_type_flags flags);
int onload_zc_release_buffers(int fd,
                              onload_zc_handle* bufs,
                              int bufs_len);
```

The `onload_zc_send()` function return value identifies how many of the `onload_zc_mmsg` array's rc fields are set. Each `onload_zc_mmsg.rc` returns how many bytes (or error) were sent in for that message. Refer to the table below.

`rc = onload_zc_send()`	
`rc < 0`	application error calling `onload_zc_send()`. rc is set to the error code
`rc == 0`	should not happen
`0 < rc <= n_msgs`	`rc` is set to the number of messages whose status has been sent in `mmsgs[i].rc`. `rc == n_msgs` is the normal case
`rc = mmsg[i].rc`	
`rc < 0`	error sending this message. `rc` is set to the error code
`rc >= 0`	`rc` is set to the number of bytes that have been sent in this message. Compare to the message length to establish which buffers sent

Sent buffers are owned by Onload. Unsent buffers are owned by the application and must be freed or reused to avoid leaking.

***Note*:** Buffers sent with the ONLOAD_MSG_WARM feature enabled are not actually sent buffers, ownership remains with the user who is responsible for freeing these buffers.

Zero-Copy Send with Single Message and Buffer

Figure 35: **Zero-Copy with Single Message and Buffer Example**

```
struct onload_zc_iovec iovec;
struct onload_zc_mmsg mmsg;
rc = onload_zc_alloc_buffers(fd, &iovec, 1,
                             ONLOAD_ZC_BUFFER_HDR_TCP);
assert(rc == 0);
assert(my_data_len <= iovec.iov_len);
memcpy(iovec.iov_base, my_data, my_data_len);
iovec.iov_len = my_data_len;
mmsg.fd = fd;
mmsg.msg.iov = &iovec;
mmsg.msg.msghdr.msg_iovlen = 1;
rc = onload_zc_send(&mmsg, 1, 0);
if( rc <= 0) {
  /* Probably application bug */
  return rc;
} else {
  /* Only one message, so rc should be 1 */
  assert(rc == 1);
  /* rc == 1 so we can look at the first (only) mmsg.rc */
  if( mmsg.rc < 0 )
    /* Error sending message */
    onload_zc_release_buffers(fd, &iovec.buf, 1);
  else
    /* Message sent, single msg, single iovec so
     * shouldn't worry about partial sends */
    assert(mmsg.rc == my_data_len);
}
```

The example above demonstrates error code handling. Note it contains an examples of bad practice where buffers are allocated and populated on the critical path.

Zero-Copy Send with Multiple Messages and Buffers

Figure 36: **Zero-Copy with Multiple Messages and Buffers Example**

```
#define N_BUFFERS 2
#define N_MSGS 2
struct onload_zc_iovec iovec[N_MSGS][N_BUFFERS];
struct onload_zc_mmsg mmsg[N_MSGS];
for( i = 0; i < N_MSGS; ++i ) {
  rc = onload_zc_alloc_buffers(fd, iovec[i], N_BUFFERS,
                               ONLOAD_ZC_BUFFER_HDR_TCP);
  assert(rc == 0);
  /* TODO store data in iovec[i][j].iov_base,
   * set iovec[i][j]iov_len */
  mmsg[i]fd = fd; /* Could be different for each message */
  mmsg[i].iov = iovec[i];
  mmsg[i].msg.msghdr.msg_iovlen = N_BUFFERS;
}
rc = onload_zc_send(mmsg, N_MSGS, 0);
if( rc <= 0 ) {
  /* Probably application bug */
  return rc;
} else {
```

```
  for( i = 0; i < N_MSGS; ++i ) {
    if( i < rc ) {
      /* mmsg[i] is set and we can use it */
      if( mmsg[i] < 0) {
        /* error sending this message - release buffers */
        for( j = 0; j < N_BUFFERS; ++j )
          onload_zc_release_buffers(fd, &iovec[i][j].buf, 1);
      } else if( mmsg(i] < sum_over_j(iovec[i][j].iov_len) ) {
        /* partial success */
        /* TODO use mmsg[i] to determine which buffers in
         * iovec[i] array are sent and which are still
         * owned by application */
      } else {
        /* Whole message sent, buffers now owned by Onload */
      }
    } else {
      /* mmsg[i] is not set, this message was not sent */
      for( j = 0; j < N_BUFFERS; ++j )
        onload_zc_release_buffers(fd, &iovec[i][j].buf, 1);
    }
  }
}
```

The example above demonstrates error code handling and contains some examples of bad practice where buffers are allocated and populated on the critical path.

Zero-Copy Send Full Example

Figure 37: **Zero-Copy Send**

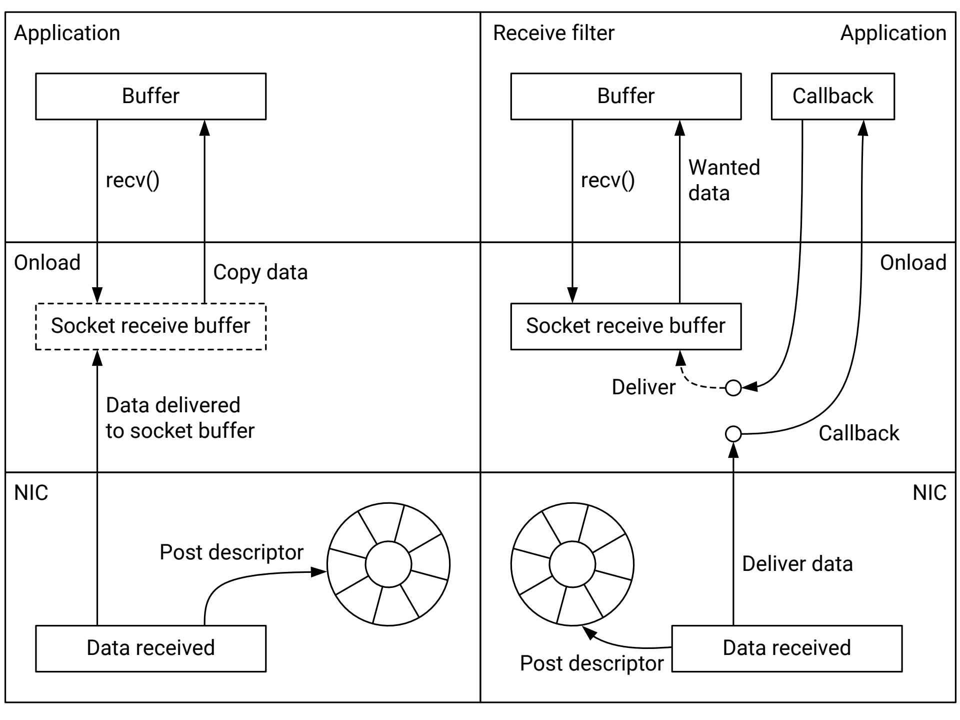
```
static struct onload_zc_iovec iovec[NUM_ZC_BUFFERS];
static ssize_t do_send_zc(int fd, const void* buf, size_t len, int flags)
{
  int bytes_done, rc, i, bufs_needed;
  struct onload_zc_mmsg mmsg;
  mmsg.fd = fd;
  mmsg.msg.iov = iovec;
  bytes_done = 0;
  mmsg.msg.msghdr.msg_iovlen = 0;
  while( bytes_done < len ) {
    if( iovec[mmsg.msg.msghdr.msg_iovlen].iov_len > (len - bytes_done) )
      iovec[mmsg.msg.msghdr.msg_iovlen].iov_len = (len - bytes_done);
    memcpy(iovec[i].iov_base, buf+bytes_done, iov_len);
    bytes_done += iovec[mmsg.msg.msghdr.msg_iovlen].iov_len;
    ++mmsg.msg.msghdr.msg_iovlen;
  }
  rc = onload_zc_send(&mmsg, 1, 0);
  if( rc != 1 /* Number of messages we sent */ ) {
    printf("onload_zc_send failed to process msg, %d\n", rc);
    return -1;
  } else {
    if( mmsg.rc < 0 )
      printf("onload_zc_send message error %d\n", mmsg.rc);
    else {
      /* Iterate over the iovecs; any that were sent we must replenish. */
      i = 0; bufs_needed= 0;
      while( i < mmsg.msg.msghdr.msg_iovlen ) {
        if( bytes_done == mmsg.rc ) {
          printf(onload_zc_send did not send iovec %d\n", i);
          /* In other buffer allocation schemes we would have to release
```

```
             * these buffers, but seems pointless as we guarantee at the
             * end of this function to have iovec array full, so do nothing.
*/
        } else {
          /* Buffer sent, now owned by Onload, so replenish iovec array */
          ++bufs needed;
          bytes_done += iovec[i].iov_len;
        }
        ++i;
      }
      if( bufs_needed ) /* replenish the iovec array */
        rc = onload_zc_alloc_buffers(fd, iovec, bufs_needed,
                                     ONLOAD_ZC_BUFFER_HDR_TCP);
    }
  }
  /* Set a return code that looks similar enough to send(). NB. we're
   * not setting (and neither does onload_zc_send()) errno */
  if( mmsg.rc < 0 ) return -1;
  else return bytes_done;
}
```

Receive Filtering API

The Onload Extensions Receive Filtering API allows a user-defined callback to inspect data received on a UDP socket before it enters the socket receive buffer. It provides an alternative to the `onload_zc_recv()` function described in the previous sections.

Figure 38: **UDP Receive Filtering**



Receive filtering is implemented within the Onload Extensions API.

Note: An application using the Receive Filtering API can continue to use any POSIX function on the socket such as `select()`, `poll()`, `epoll_wait()` or `recv()`, but must not use the `onload_zc_receive()` function.

Receive Filtering API

The Onload Extensions Receive Filtering API provides the following components:

- `#include <onload/extensions_zc.h>`

  In addition to the common components, an application should include this header file which contains all function prototypes and constant values required when using the API.

  This file includes comprehensive documentation, required data structures and function definitions.

The Receive Filtering API is a variation on the zero-copy receive whereby the normal socket methods are used for accessing the data, but the application can specify a callback to inspect each datagram before it is received.

```
typedef enum onload_zc_callback_rc
(*onload_zc_recv_filter_callback)(struct onload_zc_msg *msg,
                                  void* arg,
                                  int flags);
int onload_set_recv_filter(int fd,
                           onload_zc_recv_filter_callback filter,
                           void* cb_arg,
                           int flags);
```

Figure 39: **Receive Filter**

The `onload_set_recv_filter()` function returns immediately.

The callback is invoked once per message in the context of subsequent calls to `recv()`, `recvmsg()` etc. The `cb_arg` value is passed to the callback along with the message. The `flags` argument of the callback is set to `ONLOAD_ZC_MSG_SHARED` if the message is shared with other sockets, and the caller should take care not to modify the contents of the iovec.

The message can be found in `msg->iov[]`, and the `iovec` is of length `msg->msghdr.msg_iovlen`.

The callback *must* return `ONLOAD_ZC_CONTINUE` to allow the message to be delivered to the application. Other return codes such as `ONLOAD_ZC_TERMINATE` and `ONLOAD_ZC_MODIFIED` are deprecated and no longer supported.

This function can only be used with accelerated sockets (those being handled by Onload). If a socket has been handed over to the kernel stack (for example because it has been bound to an address that is not routed over a SFC interface), it will return `-ESOCKTNOSUPPORT`.

Receive Filtering Example

```
static enum onload_zc_callback_rczc_recv_filter(struct onload_zc_msg* msg,
void* arg, int flags)
{
  return ONLOAD_ZC_CONTINUE;
}struct zc_recv_state zc_filter_state;
static int do_zc_filter(controller_t* c)
{
  zc_filter_state.c = c;
  zc_filter_state.bytes = 0;
  return onload_set_recv_filter(the_socket, zc_recv_filter,
                                &zc_filter_state, 0);
}
```

Figure 40: **Receive Filtering Example**

Note: The `onload_set_recv_filter()` function should *not* be used together with the `onload_zc_recv()` function.

Templated Sends API

"Templated sends" is a feature for the SFN8000 and X2 series adapters that builds on top of TX PIO to provide further transmit latency improvements. Refer to Programmed I/O for details of TX PIO.

Description

Templated sends can be used in applications that know the majority of the content of packets in advance of when the packet is to be sent. For example, a market feed handler might publish packets that vary only in the specific value of certain fields, possibly different symbols and price information, but are otherwise identical.

The Onload templated sends feature uses the Onload Extensions API to generate the packet template which is then instantiated on the adapter ready to receive the "missing" data before each transmission.

Templated sends involve allocating a template of a packet on the adapter containing the bulk of the data prior to the time of sending the packet. Then, when the packet is to be sent, the remaining data is pushed to the adapter to complete and send the packet.

When the socket, associated with an allocated template, is shutdown or closed, allocated templates are freed and subsequent calls to access these template will return an error.

The API details are available in the Onload distribution at:

- `/src/include/onload/extensions_zc.h`

MSG Template

```
struct oo_msg_template {
  /* To verify subsequent templated calls are used with the same socket */
  oo_sp    oomt_sock_id;
};
```

MSG Update

```
/* An update_iovec describes a single template update */
struct onload_template_msg_update_iovec {
  void*    otmu_base;          /* Pointer to new data */
  size_t   otmu_len;           /* Length of new data */
  off_t    otmu_offset;        /* Offset within template to update */
  unsigned otmu_flags;         /* For future use.  Must be set to 0. */
};
```

MSG Allocation

Description

Populated from an array of iovecs to specify the initial packet data. This function is called once to allocate the packet template and populate the template with the bulk of the payload data.

Definition

```
extern int onload_msg_template_alloc(
int fd,
struct iovec* initial_msg,
int iovlen,
onload_template_handle* handle,
unsigned flags);
```

Formal Parameters

- `fd`: File descriptor to send on.
- `initial_msg`: Array of iovecs which are the bulk of the payload.
- `iovlen`: Length of initial msg.
- `handle`: Template handle, used to refer to this template.
- `flags`: See notes below. Can also be set to zero.

Return Value

0 on success.

Non-zero otherwise.

Notes

The initial `iovec` array passed to `onload_msg_template_alloc()` must have at least one element having a valid address and non-zero length.

If PIO allocation fails, then `onload_msg_template_alloc()` will fail. Setting the flags to ONLOAD_TEMPLATE_FLAGS_PIO_RETRY will force allocation without PIO while attempting to allocate the PIO in later calls to `onload_msg_template_update()`.

MSG Template Update

Description

Takes an array of `onload_template_msg_update_iovec` to describe changes to the base packet populated by the `onload_msg_template_alloc()` function. Each of the update iovecs should describe a single change. The update function is used to overwrite existing template content or to send the complete template content when the ONLOAD_TEMPLATE_FLAGS_SEND_NOW flag is set.

Definition

```
extern int onload_msg_template_update(
int fd,
onload_template_handle* handle,
struct onload_template_msg_update_iovec* updates,
int ulen,
unsigned flags);
```

Formal Parameters

- `fd`: File descriptor to send on.
- `handle`: Template handle, returned from the alloc function.
- `onload_template_msg_update_iovec`: Array of `onload_template_msg_update_iovec` each of which is a change to the template payload.
- `ulen`: Length of updates array (the number of changes).
- `flags`: See below. Can also be set to zero.

Return Value

0 on success.

Non-zero otherwise.

Notes

If the ONLOAD_TEMPLATE_FLAGS_SEND_NOW flag is set, ownership of the template is passed to Onload.

This function can be called multiple times and changes are cumulative.

Flags

- `ONLOAD_TEMPLATE_FLAGS_SEND_NOW`: Perform the template update, send the template contents and pass ownership of the template to Onload.

  To send without updating template contents – updates=NULL, ulen=0 and set the send now flag.

- `ONLOAD_TEMPLATE_FLAGS_DONTWAIT` **(same as MSG_DONTWAIT):** Do not block.

MSG Template Abort

Abort use of the template without sending the template and free the template resources including the template handle and PIO region.

Definition

```
extern int onload_msg_template_alloc(
int fd,
onload_template_handle* handle);
```

Formal Parameters

- `fd`: File descriptor owning the template.
- `handle`: Template handle, used to refer to this template.

Return Value

0 on success.

Non-zero otherwise.

Delegated Sends API

The delegated send API can lower the latency overhead incurred when calling `send()` on TCP sockets by controlling TCP socket creation and management through Onload, but allowing TCP sends directly through the Onload layer 2 ef_vi API or other similar API.

Description

An application using the delegated sends API will prepare a packet buffer with IP/TCP header data, before adding payload data to the packet. The packet buffer can be prepared in advance and payload added just before the send is required.

After each delegated send, the actual data sent (and length of that data) is returned to Onload. This allows Onload to update the TCP internal state and have the data to hand if retransmissions are required on the socket.

This feature is intended for applications that make sporadic TCP sends as opposed to large amounts of bi-directional TCP traffic. The API should be used with caution to send small amounts of TCP data. Although the packet buffer can be prepared in advance of the send, the idea is to complete the delegated send operation (`onload_delegated_send_complete()`) soon after the initial send to maintain the integrity of the TCP internal state ensuring that sequence/acknowledgment numbers are correct.

The user is responsible for serialization when using the delegated send API. The first call should always be `onload_delegated_send_prepare()`. If a normal send is required following the prepare, the user should use `onload_delegated_send_cancel()`.

Note: For a given file descriptor, while a delegated send is in progress, and until complete has been called, the user should NOT attempt any standard `send()`, `write()` or `sendfile()close()` etc operations.

Performance

For best latency the application should call `onload_delegated_send_complete()` as soon as a delegated send is complete. This allows Onload to continue if retransmissions are required.

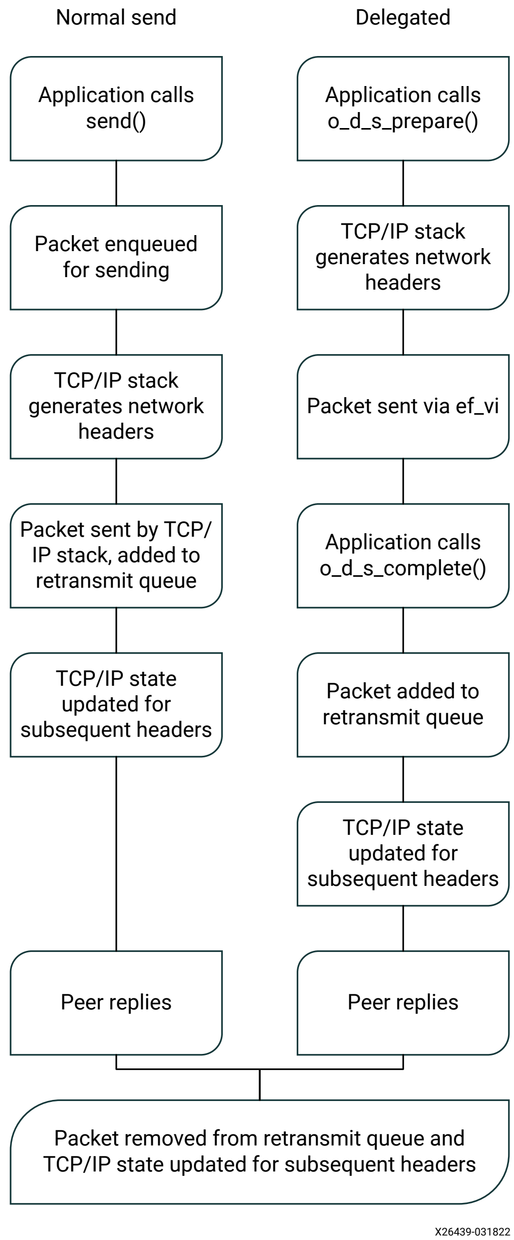
IMPORTANT! *Onload cannot perform any retransmission until complete has been called.*

When a link partner has already acknowledged data before complete has been called, Onload will not have to copy the sent data to the TCP retransmit queue. So delaying the complete call might avoid a data copy, but latency might suffer in the event of packet loss.

Standard Send vs. Delegated Send

The following sequence demonstrates the events sequence of a normal TCP send and the Delegated send.

Figure 41: **Standard vs. Delegated Send**



A packet could be delayed before sending when the receiver or network is not ready. When this occurs using delegated send, the `onload_delegated_send_prepare()` function will return zero values in the cong/send window fields of the delegated send state and the caller can elect to send with the standard method.

Example Code

The Onload distribution includes the `exchange.c` and `trader_onload_ds_efvi.c` example applications to demonstrate the delegated sends API. Variables and constants definitions, including socket flags and function return codes required when using the API can be found in the `extensions.h` header file.

- `onload-<version>/src/tests/trade_sim`
- `onload-<version>/build/gnu_x86_64/tests/trade_sim`

Run Client/Server

```
[server1] # onload -p latency-best ./exchange <interface>
oo:exchange[22157]: Using OpenOnload 201811 Copyright 2006-2018 Solarflare
Communications, 2002-2005 Level 5 Networks [1]
Waiting for client to connect
Accepted client connection
Starting event loop
n_lost_msgs:  0
n_samples:    50000
latency_mean: 1441
latency_min:  1312
latency_max:  38322
```

```
[server2]# onload -p latency-best ./trader_onload_ds_efvi <interface>
<exchange-host-ip>
oo:trader_onload_ds[2430]: Using OpenOnload 201811 Copyright 2006-2018
Solarflare Communications, 2002-2005 Level 5 Networks [2]
n_normal_sends: 55000
n_delegated_sends: 0
```

struct onload_delegated_send

```
struct onload_delegated_send {
  void* headers;
  int   headers_len;   /* buffer len on input, headers len on output */
  int   mss;           /* one packet payload may not exceed this */
  int   send_wnd;      /* send window */
  int   cong_wnd;      /* congestion window */
  int   user_size;     /* the "size" value from send_prepare() call */
  int   tcp_seq_offset;
  int   ip_len_offset;
  int   ip_tcp_hdr_len;
  int   reserved[5];
};
```

onload_delegated_send_rc

The return codes for the onload_delegated_send_prepare() function are defined in `enum onload_delegated_send_rc`, and are as follows

- `enum onload_delegated_send_rc { ONLOAD_DELEGATED_SEND_RC_OK = 0`: Send successful.
- `ONLOAD_DELEGATED_SEND_RC_BAD_SOCKET`: Non-onloaded, non-TCP, non-connected or write-shutdowned.
- `ONLOAD_DELEGATED_SEND_RC_SMALL_HEADER`: Too small header_len value.
- `ONLOAD_DELEGATED_SEND_RC_SENDQ_BUSY`: Send queue is not empty.
- `ONLOAD_DELEGATED_SEND_RC_NOWIN`: Send window is closed, the peer cannot receive more data.
- `ONLOAD_DELEGATED_SEND_RC_NOARP`: Failed to find the destination MAC address. See extensions.h for further information.
- `ONLOAD_DELEGATED_SEND_RC_NOCWIN`: Congestion window is closed. It is a violation of the TCP protocol to send anything. However, all the headers are filled in and the caller can use them for sending.

onload_delegated_send_prepare

Description

Prepare to send up to size bytes. Allocate TCP headers and prepare them with Ethernet IP/TCP header data - including current sequence number and acknowledgment number.

Definition

```
enum onload_delegated_send_prepare (
int fd,
int size,
uint flags,
struct onload_delegated_send* )
```

Formal Parameters

- `fd`: File descriptor to send on.
- `size`: Size of payload data.
- `flags`: See below.
- `struct onload_delegated_send*`: See struct onload_delegated_send.

Return Value

Refer to onload_delegated_send_rc above.

Notes

This function can be called speculatively so that the packet buffer is prepared in advance, headers are added so that the packet payload data can be added immediately before the send is required.

This function assumes the packet length is equal to MSS in which case there is no need to call `onload_delegated_send_tcp_update()`.

Flags are used for ARP resolution:

- default flags = 0
- `ONLOAD_DELEGATED_SEND_FLAG_IGNORE_ARP` - do not do ARP lookup, the caller will provide destination MAC address.
- `ONLOAD_DELEGATED_SEND_FLAG_RESOLVE_ARP` - if ARP information is not available, send a speculative TCP_ACK to provoke kernel into ARP resolution, and wait up to 1 ms for ARP information to appear.

Note: *TCP send window/congestion windows must be respected during delegated sends.*

See `extensions.h` for flags and return code values.

onload_delegated_send_tcp_update

Description

This function does not send TCP data, but is called to update packet headers with the sequence number and flags following successive sends via the onload_delegated_send_tcp_advance() function.

Note: This function does not update the ACK number.

Definition

```
void onload_delegated_send_tcp_update (
struct onload_delegated_send*,
int size,
int flags )
```

Formal Parameters

- `struct onload_delegated_send*`: See struct onload_delegated_send.
- `size`: Size of payload data.
- `flags`: See below.

Return Value

None.

Notes

This function is called when, during a send, the payload length is not equal to the MSS value. See onload_delegated_send_prepare.

Flag TCP_FLAG_PSH is expected to be set on the last packet when sending a large data chunk.

onload_delegated_send_tcp_advance

Description

Advance TCP headers after sending a TCP packet. This function if good for:

- Sending a few small packets in rapid succession
- Sending large data chunk (>MSS) over multiple packets.

The sequence number is updated for each outgoing packet. When a packet has been sent, the application must call `onload_delegated_send_tcp_update()` to update packet headers with the payload length - thereby ensuring that the sequence number is correct for the next send.

This function does not update the ACK number in outgoing packets. The ACK number in successive outgoing packets is the value from the last call to the `onload_delegated_send_prepare()` function.

The advance function is used to send a small number of successive outgoing packets, but the application should then call `onload_delegated_send_complete()` to return control to Onload to maintain sequence/acknowledgment number integrity and allow Onload to remove sent data from the retransmit queue.

Definition

```
void onload_delegated_send_tcp_advance (
struct onload_delegated_send*,
int bytes )
```

Formal Parameters

- `struct onload_delegated_send*`: See struct onload_delegated_send
- `bytes`: Number of bytes sent.

Return Value

None.

Notes

When sending a packet using multiple sends, the function is called to update the header data with the number of bytes after each send.

The actual data sent is not returned to Onload until the function `onload_delegated_send_complete()` is called.

onload_delegated_send_complete

Description

Following a delegated send, this function is used to return the actual data sent (and length of that data) to Onload which will update the internal TCP state (the sequence numbers) and remove packets from the retransmit queue (when appropriate ACKs are received).

Definition

```
int onload_delegated_send_complete (
int fd,
const struct iovec *,
int iovlen,
int flags )
```

Formal Parameters

- `fd`: The file descriptor.
- `struct iovec`: Pointer to the data sent.
- `iovlen`: Size (bytes) of the iovec array.
- `flags`: (`MSG_DONTWAIT` | `MSG_NOSIGNAL`)

Return Value

Number of bytes accepted, or return -1 if an error occurs with `errno` set.

Notes

Onload is unable to do any retransmit until this function has been called.

This function should be called even if some (but not all) bytes specified in the prepare function have been sent. The user must also call `onload_delegated_send_cancel()` if some of the bytes are not going to be sent (they are reserved-but-not-sent) - see `onload_delegated_send_cancel()` notes below.

***Note*:** This function differs from the `send()` function in its handling of a "resource temporarily unavailable" or "operation would block" situation. This function returns `0`, but the `send()` function would return `-1` with `errno` set to `EAGAIN`.

This function can block because of SO_SNDBUF limitation and will ignore the SO_SNDTIMEO value.

onload_delegated_send_cancel

Description

No more delegated send is planned.

Normal `send()`, `shutdown()` or `close()` etc can be called after this call.

Definition

```
int onload_delegated_send_cancel (int fd)
```

Formal Parameters

- `fd`: The file descriptor to be closed.

Return Value

0 on success

-1 on failure with `errno` set.

Notes

When TCP headers have been allocated with `onload_delegated_send_prepare()`, but it is subsequently required to do a normal send, this function can be used to cancel the delegated send operation and do a normal send.

There is no need to call this function before calling `onload_delegated_send_prepare()`.

There is no need to call this function if all the bytes specified in the `onload_delegated_send_prepare()` function have been sent.

If some, but not all bytes have been sent, you must call `onload_delegated_send_complete()` for the sent bytes, *then* call `onload_delegated_send_cancel()` for the remaining bytes (reserved-but-not-sent) bytes. This applies even if the reason for not sending is that the window limits returned from the prepare function have been reached.

Normal `send()`, `shutdown()` or `close()` etc can be called after this call.

onload_stackdump

The `onload_stackdump` diagnostic utility is a component of the Onload distribution which can be used to monitor Onload performance, set tuning options and examine aspects of the system performance.

Note: To view data for all stacks, created by all users, the user must be `root` when running `onload_stackdump`. Non-root users can only view data for stacks created by themselves or accessible to them via the EF_SHARE_WITH environment variable.

The following examples of `onload_stackdump` are demonstrated elsewhere in this user guide:

- Monitoring Using onload_stackdump
- Processing at User-Level
- As Few Interrupts as Possible
- Eliminating Drops
- Minimizing Lock Contention
- Stack Contention - Deferred Work

General Use

The `onload_stackdump` tool can produce an extensive range of data and it can be more useful to limit output to specific stacks or to specific aspects of the system performance for analysis purposes.

- For help, and to list all onload_stackdump commands and options:

  ```
  onload_stackdump --help
  ```

- To display documentation of EF_* environment variables:

  ```
  onload_stackdump doc
  ```

- For descriptions of statistics variables:

  ```
  onload_stackdump describe_stats
  ```

  Describes all statistics listed by the `onload_stackdump lots` command.

- To identify all stacks, by identifier and name, and all processes accelerated by Onload:

```
onload_stackdump
#stack-id stack-name      pids
6         teststack       28570
```

- To limit the command/option to a specific stack, for example stack 4:

```
onload_stackdump 4 lots
```

List Onloaded Processes

The '`onload_stackdump processes`' command will show the PID and name of processes being accelerated by Onload and the Onload stack being used by each process. For example:

```
# onload_stackdump processes
#pid  stack-id cmdline
25587    3      ./sfnt-pingpong
```

Onloaded processes which have not created a socket are not displayed, but can be identified using the `lsof` command.

List Onloaded Threads, Priority and Affinity

The 'onload_stackdump threads' command will identify threads within each Onload-accelerated process, the CPU affinity of the thread and its runtime priority.

```
# onload_stackdump threads | column -t
#pid   thread   affinity  priority  realtime
12606  12606    00000002  0         0
```

List Onload Environment Variables

The '`onload_stackdump env`' command will identify onloaded processes running in the current environment and list all Onload variables set in the current environment. For example:

```
# EF_POLL_USEC=100000 EF_TXQ_SIZE=4096 EF_INT_DRIVE=1 onload <application>
# onload_stackdump env
pid: 25587
cmdline: ./sfnt-pingpong
env: EF_POLL_USEC=100000
env: EF_TXQ_SIZE=4096
env: EF_INT_DRIVEN=1
```

TX PIO Counters

The Onload stackdump utility exposes counters to indicate how often TX PIO is being used - see Debug and Logging. To view PIO counters run the following command:

```
$ onload_stackdump stats | grep pio
pio_pkts: 2485971
no_pio_err: 0
```

The values returned will identify the number of packets sent via PIO and number of times when PIO was not used due to an error condition.

Send RST on a TCP Socket

To send a reset on an Onload accelerated TCP socket, specify the stack and socket using the rst command:

```
# onload_stackdump <stack:socket> rst
```

CAUTION! *This resets the TCP connection, and so is likely to disrupt the application.*

Removing Zombie and Orphan Stacks

Onload stacks and sockets can remain active even after all processes using them have been terminated or have exited, for example to ensure sent data is successfully received by the TCP peer or to honor TCP TIME_WAIT semantics. Such stacks should always eventually self-destruct and disappear with no user intervention. However, these stacks, in some instances, cause problems for restarting applications, for example the application might be unable to use the same port numbers when these are still being used by the persistent stack socket. Persistent stacks also retain resources such as packet buffers which are then denied to other stacks.

Such stacks are termed 'zombie' or 'orphan' stacks and it can be either undesirable or desirable that they exist.

- To list all persistent stacks:

  ```
  # onload_stackdump -z all
  ```

  No output to the console or syslog means that no such stacks exist.

- To list a specific persistent stack:

```
# onload_stackdump -z <stack ID>
```

- To display the state of persistent stacks:

```
# onload_stackdump -z [dump | lots]
```

- To terminate persistent stacks

```
# onload_stackdump -z kill
```

- To display all options available for zombie/orphan stacks:

```
# onload_stackdump --help
```

Snapshot vs. Dynamic Views

The `onload_stackdump` tool presents a snapshot view of the system when invoked. To monitor state and variable changes whilst an application is running use `onload_stackdump` with the Linux `watch` command. For example:

- snapshot: `onload_stackdump netif`
- dynamic: `watch -d -n1 onload_stackdump netif`

Some `onload_stackdump` commands also update periodically whilst monitoring a process. These commands usually have the `watch_` prefix. For example:

`watch_stats`, `watch_more_stats`, `watch_tcp_stats`, `watch_ip_stats` etc.

Use the `onload_stackdump -h` option to list all commands.

Monitoring Receive and Transmit Packet Buffers

```
onload_stackdump packets
```

```
# onload_stackdump packets
ci_netif_pkt_dump_all: id=1
  pkt_sets: pkt_size=2048 set_size=1024 max=32 alloc=2
  pkt_set[0]: free=544
  pkt_set[1]: free=437 current
  pkt_bufs: max=32768 alloc=2048 free=981 async=0
  pkt_bufs: rx=1067 rx_ring=1001 rx_queued=2 pressure_pool=64
```

```
  pkt_bufs: tx=0 tx_ring=0 tx_oflow=0
  pkt_bufs: in_loopback=0 in_sock=0
    1003: 0x200 Rx
   n_zero_refs=1045 n_freepkts=981 estimated_free_nonb=64
   free_nonb=0 nonb_pkt_pool=ffffffffffffff
```

The `onload_stackdump packets` command can be useful to review packet buffer allocation, use and reuse within a monitored process.

The example above identifies that the process has a maximum of 32768 buffers (each of 2048 bytes) available. From this pool 2048 buffers have been allocated and 981 from that allocation are currently free for reuse - that means they can be pushed onto the receive or transmit ring buffers ready to accept new incoming/outgoing data.

On the receive side of the stack, 1067 packet buffers have been allocated, 1001 have been pushed to the receive ring - and are available for incoming packets, and 2 are currently in the receive queue for the application to process.

On the transmit side of the stack, zero buffers are currently allocated or being used. The remaining values are calculations based on the packet buffer values.

Using the EF_PREFAULT_PACKETS environment variable, packets can be pre-allocated to the user-process when an Onload stack is created. This can reduce latency jitter and improve Onload performance - for further details see Reducing Jitter from Page Faults.

Packet Sets

A packet set is a 2 MB chunk of packet buffers being used by an Onload application. An application might use buffers from a single set or from several sets depending on its complexity and packet buffer requirements.

With an aim to further reduce TLB thrashing and eliminate packets drops, Onload will try to reuse buffers from the same set.

The `onload_stackdump lots` command will report on the current use of packets sets. For example:

```
$ onload_stackdump lots | grep pkt_set

  pkt_sets: pkt_size=2048 set_size=1024 max=32 alloc=2
  pkt_set[0]: free=544
  pkt_set[1]: free=442 current
```

In the above output there are two packet sets, the counters identify the number of free packet buffers in each set and identify the set currently being used.

The packet sets feature is not available to user applications using the ef_vi layer directly.

TCP Application Statistics

The following `onload_stackdump` commands can be used to monitor accelerated TCP connections:

```
onload_stackdump tcp_stats
```

Table 48: **Output from onload_stackdump tcp_stats**

Field	Description
`tcp_active_opens`	Number of socket connections initiated by the local end. This is the number of times TCP connections have made a direct transition to the SYN-SENT state from the CLOSED state
`tcp_passive_opens`	Number of sockets connections accepted by the local end. This is the number of times TCP connections have made a direct transition to the SYN-RCVD state from the LISTEN state.
`tcp_l3xudp_active_opens`	Number of l3xudp socket connections initiated by the local end. This is the number of times TCP connections have made a direct transition to the SYN-SENT state from the CLOSED state and that socket is using l3xudp encapsulation.
`tcp_l3xudp_passive_opens`	Number of l3xudp sockets connections accepted by the local end. This is the number of times TCP connections have made a direct transition to the SYN-RCVD state from the LISTEN state and that socket is using l3xudp encapsulation.
`tcp_attempt_fails`	Number of failed connection attempts. This is the number of times TCP connection have made a direct transition to the CLOSED state from the SYN-SENT state or the SYN-RCVD state, plus the number of times TCP connections have made a direct transition to the LISTEN state from the SYN-RCVD state.
`tcp_estab_resets`	Number of established connections which were subsequently reset. This is the number of times TCP connections have made a direct transition to the CLOSED state from either the ESTABLISHED state or the CLOSE-WAIT state.
`tcp_curr_estab`	Number of TCP connections for which the current state is either ESTABLISHED or CLOSE-WAIT.
`tcp_in_segs`	Total number of segments received, including those received in error.
`tcp_out_segs`	Total number of segments sent, including those on current connections but excluding those containing only retransmitted octets.
`tcp_retran_segs`	Total number of segments retransmitted.
`tcp_in_errs`	Number of erroneous segments received.
`tcp_out_rsts`	Number of RST segments sent.

```
onload_stackdump more_stats | grep tcp
```

Table 49: **Output from onload_stackdump more_stats | grep tcp**

Field	Description
`tcp_has_recvq`	The number of TCP sockets that currently have data in a receive queue.

Table 49: **Output from onload_stackdump more_stats | grep tcp** *(cont'd)*

Field	Description
`tcp_recvq_bytes`	The number of bytes currently waiting in TCP receive queues.
`tcp_recvq_pkts`	The number of packets currently waiting in TCP receive queues.
`tcp_has_recv_reorder`	The number of sockets with out of sequence bytes. This is the number of sockets with packets in the re-ordering queue. Re-ordering usually (though not always) indicates loss. We hold on to the future packets until the intervening ones arrive, then push them to the receive queue. So unless Onload is currently waiting for some retransmits; this counter will be zero. It is not a historical log.
`tcp_recv_reorder_pkts:`	Number of out of sequence packets received. This is the number of packets currently in re-ordering queues. Re-ordering usually (though not always) indicates loss. We hold on to the future packets until the intervening ones arrive, then push them to the receive queue. So unless Onload is currently waiting for some retransmits; this counter will be zero. It is not a historical log.
`tcp_has_sendq`	Non zero if send queues have data ready This is the number of TCP sockets with packets in the send queue. This counter will usually be zero; unless something is preventing Onload from sending immediately (for example congestion window). See also `send+pre=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.
`tcp_sendq_bytes`	Number of bytes currently in all send queues for this connection This is the count of bytes in TCP send queues. This counter will usually be zero; unless something is preventing Onload from sending immediately (for example congestion window). See also `send+pre=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.
`tcp_sendq_pkts`	Number of packets currently in all send queues for this connection This is the number of packets in TCP send queues. This counter will usually be zero; unless something is preventing Onload from sending immediately (for example congestion window). See also `send+pre=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.
`tcp_has_inflight`	Non zero if some data remains unacknowledged This is the number of sockets that have packets 'in-flight' (sent but for which Onload has not yet received an ACK). See also `inflight=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.
`tcp_inflight_bytes`	Total number of unacknowledged bytes This is the number of bytes that are 'in-flight' (sent but for which Onload has not yet received an ACK). See also `inflight=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.
`tcp_inflight_pkts`	Total number of unacknowledged packets This is the number of packets 'in-flight' (sent but for which Onload has not yet received an ACK). See also `inflight=` in Table 54: Stackdump Output: TCP Established Connection Socket for per-socket information.

Table 49: **Output from onload_stackdump more_stats | grep tcp** *(cont'd)*

Field	Description
`tcp_n_in_listenq`	Number of sockets in SYN-RECEIVED state. This is the number of sockets (summed across all listening sockets) where the local end has responded to SYN with a SYN_ACK, but this has not yet been acknowledged by the remote end The size of the listen queue is limited by EF_TCP_BACKLOG_MAX.
`tcp_n_in_acceptq`	Number of sockets that have reached ESTABLISHED state, that the application has not yet called `accept()` for.

Use the `onload_stackdump -h` command to list all TCP connection, stack and socket commands.

The onload_stackdump LOTS Command.

The `onload_stackdump lots` command will produce extensive data for all accelerated stacks and sockets. The command can also be restricted to a specific stack and its associated connections when the stack number is entered on the command line. For example:

```
onload_stackdump lots
onload_stackdump 2 lots
```

The following sections each describe a part of the output from the `onload_stackdump lots` command. See:

- TCP Stacks
- TCP ESTABLISHED Connection Sockets
- TCP LISTEN Sockets
- UDP Sockets
- Statistics
- Environment Variables

TCP Stacks

This section describes typical output for a TCP stack.

***Note*:** Depending on the state of the stack, some of the output shown in this section might be omitted. The sample output in this section is from several different stacks, to illustrate these different states, and so might not be self-consistent.

The output starts with basic information about the stack, shown in the following table:

Table 50: **Stackdump Output: TCP Stack**

Sample Output	Description
`ci_netif_dump_to_logger:`	Function dumping the stack.
`stack=7`	Stack id.
`name=`	Stack name as set by EF_NAME.
`cplane_pid=813`	Process id of Onload control plane server.
`namespace=net:[4026531956]`	Namespace id.
`ver=201811`	Onload version.
`uid=0`	User id.
`pid=1930`	Process id of creator process.
`ns_flags=0`	Flags as a hexadecimal value, followed by names of flags including: • ONLOAD_UNSUPPORTED • SOCKCACHE_FORKED.
`creation_time=2019-01-25 15:16:14`	Creation time of stack.
`(delta=19secs)`	Age of stack, as a delta between stack creation and the stackdump.
`lock=20000000 LOCKED`	Internal stack lock status.
`nics=3`	Hexadecimal bitfield identifying adapters used by this stack. For example, 0x3 = 0b11, so the stack is using adapters 1 and 2.
`primed=1`	1 if the event queue will generate an interrupt when the next event arrives, otherwise 0.
`ref=`	Count of references to this stack.
`trusted_lock=`	Kernel side stack lock: • `0`: unlocked • `2`: awaiting free • Otherwise: locked, bitmask gives more information.
`k_ref=`	Count of kernel references to this stack.
`n_ep_closing=`	Count of kernel references to this stack for closing endpoints.
`sock_bufs:`	Sockets buffers which can be allocated:
`max=8192`	Maximum number.
`n_allocated=4`	Number currently allocated.
`aux_bufs:`	Aux buffers, used by partially opened TCP connections (incoming connections) before they are established and promoted to use socket buffers. The number of aux buffers is limited to EF_TCP_SYNRECV_MAX `* 2`
`free=6`	Number of free aux buffers.

Table 50: **Stackdump Output: TCP Stack** *(cont'd)*

Sample Output	Description

`aux_bufs[syn-recv state]:`
`n=0`
`max=2048`
	Aux buffers for the syn-recv state:
	Number currently allocated
	Maximum number.

`aux_bufs[syn-recv bucket]:`
`n=0`
`max=8192`
	Aux buffers for the syn-recv bucket:
	Number currently allocated
	Maximum number.

`aux_bufs[epoll3 state]:`
`n=0`
`max=8192`
	Aux buffers for the epoll3 state:
	Number currently allocated
	Maximum number.

`pkt_sets:`
`pkt_size=2048`
`set_size=1024`
`max=32`
`alloc=2`
	Packet sets:
	Size of a packet buffer, in bytes.
	Number of packet buffers in each packet set.
	Maximum number of packet sets available to this stack.
	Number of packet sets currently allocated.

`pkt_set[0]:` `free=112`	Packet set 0: Number of free packet buffers in the set, each of size `pkt_sets -> pkt_size`.

`pkt_set[1]:`
`free=880`
`current`
	Packet set 1:
	Number of free packet buffers in the set, each of size `pkt_sets -> pkt_size`.
	This is the packet set currently being used.

Table 50: **Stackdump Output: TCP Stack** *(cont'd)*

Sample Output	Description
`pkt_bufs:`	Packet buffers:
`max=32768`	Maximum number of packet buffers this stack can allocate, each of size `pkt_sets -> pkt_size`.
`alloc=576`	Number of packet buffers that have been allocated.
`free=57`	Number of packet buffers that are free, and can be reused by either receive or transmit rings.
`async=0`	Number of packet buffers used by Onload in one of its asynchronous queues.
`pkt_bufs:`	Receive packet buffers:
`rx=1056`	Number of receive packet buffers that are currently in use.
`rx_ring=992`	Number of packet buffers that have been pushed to the receive ring.
`rx_queued=0`	Number of packet buffers that are in the application's receive queue.
`pressure_pool=64`	Number of packet buffers in the pressure pool. This is a pool of packet buffers used when the stack is under memory pressure. Its size is `rx - (rx_ring + rx_queued)`. This might be followed by flags indicating a memory pressure condition: `CRITICAL`: the number of packets in the receive socket buffers is approaching the EF_MAX_RX_PACKETS value. `LOW`: there are not enough packet buffers available to refill the RX descriptor ring.
`pkt_bufs:`	Transmit packet buffers:
`tx=2`	Number of transmit packet buffers that are currently in use.
`tx_ring=1`	Number of packet buffers that remain in the transmit ring.
`tx_oflow=0`	The number of extra packets that are ready to send to the transmit queue, but that the transmit queue does not have space to accept.

Table 50: **Stackdump Output: TCP Stack** *(cont'd)*

Sample Output	Description

`pkt_bufs:`
 	
`in_loopback=0`
 	
`in_sock=991`
	Other packet buffer totals:
	
	Number of packet buffers currently used in TCP loopback connection.
	
	Number of packet buffers currently used by a TCP socket.

`pkt_bufs:`
 	
`rx_reserved=`
	Other packet buffer totals:
	
	Total number of receive packet buffers that are reserved by `ESTABLISHED` sockets.

`signal_q=[%d,%d]` `completion_q=%d`	Asynchronous signal queue head and tail (Windows only). Asynchronous completion queue (Windows only).

`time:`
 	
`netif=5eb5c61`
 	
`poll=5eb5c61`
 	
`now=5eb5c61 (diff=0.000sec)`
	Internal timer values.
	To convert ticks to milliseconds, multiply by `ci_ip_time_tick2ms`:
	
	Current cached time, in ticks.
	
	Scheduler's view of time, in ticks.
	
	Time now from cache of real ticks, and difference between this time and the `netif` time.
	If the difference is more than 5 seconds, it is followed by: `!! STUCK !!`

`ERRORS:`	Errors, if any, including: • `PPL` • `LOOP` • `ASS` • `SYNRECV`.

`active cache:`
 	
`hit=0`
 	
`avail=0`
 	
`cache=EMPTY`
 	
`pending=EMPTY`
	TCP socket caching:
	
	Number of cache hits (were cached).
	
	Number of sockets available for caching.
	
	Current cache state, either “EMPTY” or “yes”.
	
	Current pending state, either “EMPTY” or “yes”.

Table 50: **Stackdump Output: TCP Stack** *(cont'd)*

Sample Output	Description
`passive scalable cache:`	TCP socket caching:
`cache=EMPTY`	Current cache state, either “EMPTY” or “yes”.
`pending=EMPTY`	Current pending state, either “EMPTY” or “yes”.
`readylist:`	Ready list (one line per list):
`id=%d`	Ready list id
`pid=%d`	Process id of process managing ready list
`ready=%s`	Current ready list state, either “EMPTY” or “yes”.
`unready=%s`	Current unready list state, either “EMPTY” or “yes”.
`flags=%x`	Ready list flags, as a hexadecimal value.

There is then a section that is repeated for each virtual interface associated with the stack, describing the virtual interface to the NIC. This is shown in the following table:

Table 51: **Stackdump Output: Virtual Interface for a TCP Stack**

Sample Output	Description
`ci_netif_dump_vi:`	Function dumping the stack’s virtual interface to the NIC
`stack=7`	Stack id.
`intf=0`	Interface (port) number.
`dev=(pci address)`	PCI address of NIC.
`hw=0C0`	Hardware version, given as an architecture / variant / revision tuple.

Table 51: **Stackdump Output: Virtual Interface for a TCP Stack** *(cont'd)*

Sample Output	Description
`vi=240`	Identifies the VI in use by the stack.
`pd_owner=1`	Will be zero when using physical addressing mode.
`channel=0`	Identifies the receive queue being used on this interface.
`tcpdump=off`	One of the following: • `all` • `nomatch` • `off`.
`vi_flags=3800000`	VI flags, as a hexadecimal value.
`oo_vi_flags=3`	Hexadecimal bitfield identifying features requested on this VI. For details, see `src/include/ci/internal/oo_vi_flags.h`.
`evq:`	Event queue data:
`cap=2048`	Maximum number of events the queue can hold, set by EF_RXQ_SIZE, EF_TXQ_SIZE.
`current=16de30`	The current event queue location.
`is_32_evs=0`	Is 1 if there are 32 or more events pending.
`is_ev=0`	Is 1 if there are any events pending.
`evq:`	Further event queue data:
`sync_major=ffffffff`	Major part of the timestamp (seconds).
`sync_minor=0`	Minor part of the timestamp (upper part of ns).
`sync_min=0`	Smallest possible seconds value for timestamp.
`evq:`	Further event queue data:
`sync_synced=0`	Timestamp synchronized with adapter
`sync_flags=0`	Time synchronization flags

Table 51: **Stackdump Output: Virtual Interface for a TCP Stack** *(cont'd)*

Sample Output	Description

`rxq:`
 	
`cap=511`
 	
`lim=511`
 	
`spc=1`
 	
`level=510`
 	
`total_desc=93666`
	Receive queue data:
	
	Total capacity.
	
	Maximum fill level for receive descriptor ring, specified by EF_RXQ_LIMIT.
	
	Amount of empty buffers ready to be used.
	
	How full the receive queue currently is.
	
	Total number of descriptors that have been pushed to the receive queue.

`txq:`
 	
`cap=511`
 	
`lim=511`
 	
`spc=511`
 	
`level=0`
 	
`pkts=0`
 	
`oflow_pkts=0`
	Transmit queue data:
	
	Total capacity.
	
	Maximum fill level for transmit descriptor ring.
	
	Amount of empty buffers ready to be used.
	
	How full the transmit queue currently is.
	
	How many packets are represented by the descriptors in the transmit queue.
	
	How many packets are in the overflow transmit queue (that is, waiting for space in the NIC's transmit queue).

`txq:`
 	
`pio_buf_size=2048`
 	
`tot_pkts=93669`
 	
`bytes=0`
	Further transmit queue data:
	
	PIO buffer size.
	
	Total number of packet buffers used.
	
	Number of packet bytes currently in the queue.

`txq:`
 	
`ts_nsec=40000000`
	Further transmit queue data:
	
	Nanoseconds from timestamp in tx queue state.

`clk:`	Flags from last synchronization: • `SET` • `SYNC`.

Table 51: **Stackdump Output: Virtual Interface for a TCP Stack** *(cont'd)*

Sample Output	Description
`last_rx_stamp:`	Last receive timestamp:
`0:0`	Given as seconds:nanoseconds
`ctpio:`	Cut-through PIO
`max_frame_len=500`	Maximum frame length for the CTPIO low-latency transmit mechanism.
`frame_len_check=500`	Frame length check, CTPIO is disabled if this is zero.
`ct_thresh=65535`	Cut-through threshold for CTPIO transmits.
`ERRORS:`	Errors, if any, including: • `REMAP`.
`vi=240`	Identifies the VI in use by the stack when there is a separate receive queue for UDP.
`evq:`	Event queue data for when there is a separate receive queue for UDP:
`cap=2048`	Maximum number of events the queue can hold, set by EF_RXQ_SIZE, EF_TXQ_SIZE.
`current=16de30`	The current event queue location.
`is_32_evs=0`	Is 1 if there are 32 or more events pending, otherwise 0.
`is_ev=0`	Is 1 if there are any events pending, otherwise 0.
`rxq:`	Receive queue data for when there is a separate receive queue for UDP:
`cap=511`	Total capacity.
`lim=511`	Maximum fill level for receive descriptor ring, specified by EF_RXQ_LIMIT.
`spc=1`	Amount of empty buffers ready to be used.
`level=510`	How full the receive queue currently is.
`total_desc=93666`	Total number of descriptors that have been pushed to the receive queue.

There is then a section giving extra information about the stack, shown in the following table:

Table 52: **Stackdump Output: Extra Information for a TCP Stack**

Sample Output	Description

`ci_netif_dump_extra:`
 	
`stack=7`
	Function dumping the extra information
	
	Stack id.

`in_poll=0`
`post_poll_list_empty=1`
 	
`poll_did_wake=0`
	Is 1 if the process is currently polling, otherwise 0.
	Is 1 if there are tasks to be done once polling is complete, otherwise 0.
	Is 1 if while polling, the process identified a socket which needs to be woken following the poll, otherwise 0.

`rx_defrag_head=-1`
 	
`rx_defrag_tail=-1`
	Reassembly sequence number. -1 means no re-assembly has occurred.
	Reassembly sequence number. -1 means no re-assembly has occurred.

`tx_may_alloc=1`
`can=1`
`nonb_pool=1`
 	
`send_may_poll=0`
`is_spinner=0,0`
	The number of packet buffers TCP could use.
	The number of packet buffers TCP can use now.
	The number of packet buffers available to TCP process without holding the lock.
	Is 1 if using EF_POLL_ON_DEMAND, otherwise 0.
	First value is 1 if a thread is spinning, otherwise 0. Second value is the number of spinning threads.

`hwport_to_intf_i=0,-1,-1,-1,-1,-1` `intf_i_to_hwport=0,0,0,0,0,0`	Internal mapping of hardware ports to internal interfaces. Internal mapping of internal interfaces to hardware ports.
`uk_intf_ver=03e89aa26d20b98fd08793e771f2cdd9`	md5 user/kernel interface checksum computed by both kernel and user application to verify internal data structures.
`deferred count 0/32`	NUMA node parameters - refer to Onload Deployment on NUMA Systems.

`numa nodes:`
 	
`creation=0`
 	
`load=0`
	Further NUMA node parameters - refer to Onload Deployment on NUMA Systems.

`numa node masks:`
 	
`packet alloc=1`
 	
`sock alloc=1`
 	
`interrupt=1`
	Further NUMA node parameters - refer to Onload Deployment on NUMA Systems.

Finally, there is a list of process ids shown in the following table:

Table 53: **Stackdump Output: Process Ids for a TCP Stack**

Sample Output	Description
`pids:14025`	List of processes being accelerated by Onload on this stack.

TCP ESTABLISHED Connection Sockets

The following table shows typical output for a TCP ESTABLISHED connection socket.

Table 54: **Stackdump Output: TCP Established Connection Socket**

Sample Output	Description

`TCP`
`7:1`
`lcl=192.168.1.2:50773`
`rmt=192.168.1.1:34875`
`ESTABLISHED`
	TCP socket.
	Stack:socket id.
	Local ip:port address.
	Remote ip:port address.
	Connection is ESTABLISHED.

`lock: 10000000 UNLOCKED`	Internal socket lock status, as a hexadecimal number, followed by status names including: • `LOCKED` • `CONTENDED`.

`rx_wake=0000b6f4(RQ)`
 	
`tx_wake=00000002`
 	
`flags:`
	Internal sequence value that is incremented each time a receive queue is 'woken'.
	`(RQ)` indicates that a wake has been requested.
	Internal sequence value that is incremented each time a transmit queue is 'woken'.
	`(RQ)` indicates that a wake has been requested.
	Flags (if any), including:
	• `WK_TX`, `WK_RX`
	• `TCP_PP`
	• `ORPH`
	• `ACCEPTQ`
	• `DEFERRED`
	• `AVOID_INT`
	• `O_ASYNC`, `O_NONBLOCK`, `O_NDELAY`, `O_APPEND`, `O_CLOEXEC`
	• `CACHE`, `PASSIVE_CACHE`, `CACHE_NO_FD`
	• `OS_BACKED`
	• `NONB_UNSYNCED`.

`ul_poll:`
`301326900 spin cycles`
`100000 usec`
	User-level `poll()`:
	in spin cycles
	in µs.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
uid=0	User id that owns this socket.
s_flags:	Socket flags, including: • CORK • SHUTRD, SHUTWR • TCP_NODELAY • ACK • REUSE • KALIVE • BCAST • OOBIN • LINGER • DONTROUTE • FILTER • BOUND, ABOUND, PBOUND • SNDBUF, RCVBUF • SW_FILTER_FULL • TRANSPARENT • SCALACTIVE, SCALPASSIVE • MAC_FILTER • REUSEPORT • BOUND_ALIEN • CONNECT_MUST_BIND • PMTU_DO • ALWAYS_DF • IP_TTL • DEFERRED_BIND • V6ONLY • NOMCAST.
rcvbuf=129940	Socket receive buffer size.
sndbuf=131072	Socket send buffer size.
bindtodev=-1(-1,0:0)	Device to which the socket is bound, given as an interface, or -1 if unbound. This is followed by an (interface index, hardware port:vlan) tuple.
ttl=64	Initial TTL value.
rcvtimeo_ms=0	Timeout value (microseconds) before an error is generated for receive functions, as set by SO_RCVTIMEO.
sndtimeo_ms=0	Timeout value (microseconds) before an error is generated for send functions, as set by SO_SNDTIMEO.
sigown=0	The PID receiving signals from this socket.
cmsg=	Current message flags, including: • NO_MCAST_TX.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
`rx_errno=0`	Zero whilst data can still arrive, otherwise contains error code.
`tx_errno=0`	Zero if transmit can still happen, otherwise contains error code.
`so_error=0`	Current socket error, or zero if no error.
`os_sock=0`	0 if the socket is handled by Onload, 1 if the socket is handled by the OS and not by Onload.
`TX`	Socket is being used for transmit.
`epoll3: ready_list_id 0`	List of ready sockets from the epoll3 set.
`tcpflags: TSO WSCL SACK ESTAB PASSIVE`	TCP flags currently set for this socket, including: • `TSO` • `WSCL` • `SACK` • `ECN` • `STRIPE` • `SYNCOOKIE` • ESTAB • `NONBCON` • `PASSIVE` • `ARP_FAIL` • `NO_TX_ADVANCE` • `LOOP_DEFER` • `NO_QUICKACK` • `MEM_DROP` • `FIN_RECV` • `ACTIVE_WILD` • `MSG_WARM` • `LOOP_FAKE` • `TOA` • `TLP_TIMER` • `TLP_SENT` • `FIN_PENDING`.
`local_peer: -1`	The peer socket in a local loopback connection.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
`snd:`	Send data:
`up=b554bb86`	*U*rgent *P*ointer: the sequence number of the byte following the 00B byte.
`una-nxt-max=b554bb86-b554bb87-b556b6a6`	Sequence numbers of: `una`: first unacknowledged byte `nxt`: next byte we expect to be acknowledged `max`: last byte in the current send window.
`enq=b554bb87`	Sequence number of last byte currently queued for transmit.
`snd:`	Further send data:
`send=0(0)`	Number of bytes (packets) held in the send buffer.
`send+pre=0`	Number of packets in the pre-send queue. A process can add data to this queue when it is prevented from sending the data immediately. The data will be sent when the current sending operation is complete.
`inflight=1(1)`	Number of bytes (packets) sent but not yet acknowledged.
`wnd=129824`	Advertised window size of the receiver, in bytes.
`unused=129823`	Number of unused (free) bytes in that window.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description

`snd:`
 	
`cwnd=49733+0`
 	
`used=0`
 	
`ssthresh=65535`
 	
`bytes_acked=0`
 	
`Open`
	Further send data:
	
	Congestion window size, in bytes.
	
	Portion of the congestion window that is currently in use.
	
	Number of bytes that have to be sent before the process can exit slow start.
	
	Number of bytes acknowledged.
	This value is used to calculate the rate at which the congestion window is opened.
	
	Current congestion window status, one of:
	`Open`
	`RTO`
	`RTORecovery`
	`FastRecovery`
	`Cooling`
	`RTOCooling`
	`Notified`.

`snd:`
 	
`timed_seq 0`
 	
`timed_ts 3fa5511c`
	Further send data:
	
	First byte of timed packet.
	
	Timestamp for timed packet.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
`snd:`	Further send data:
`sndbuf_pkts=136`	Size of the send buffer (packets). Send buffer is calculated as bytes.

`Onloaded(Valid)`	Status of cached control plane information, one of:
	`Onloaded` = can reach the destination via an accelerated interface
	`NoMac`
	`NoRoute`
	`ViaOs`
	`Local`
	`MacFail`
	followed by its validity:
	`(Valid)`: information is up-to-date. Can send immediately using this information.
	`(Old)`: information might be out-of-date. On next send Onload will do a control plane lookup - this will add some latency.

`if=6`	Interface being used.
`mtu=1500`	MTU being used.
`intf_i=0`	Intf_i value.
`vlan=0`	VLAN being used.
`encap=4`	Types of encapsulation supported by the NIC, as a hexadecimal mask.
`snd:`	Further send data:
`limited`	Counts of transmission being stopped for the following reasons:
`rwnd=0`	- receive window size
`cwnd=0`	- congestion window size
`nagle=0`	- Nagle's algorithm
`more=0`	- more (CORK, MSG_MORE)
`app=412548`	- transmit queue being empty.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
`rcv:`	Receiver data:
`nxt-max=0e9251fe-0e944d1d`	Next byte we expect to receive and last byte we expect to receive (because of window size).
`wnd adv=129823`	Receiver advertised window size.
`cur=0e944d92`	Byte currently being processed.
`FASTSTART FAST`	Possible flags: `FASTSTART`: is in faststart `FAST`: can use fast path.
`rcv:`	Further receiver data:
`isn=b8f5ec59`	Initial sequence number.
`up=b8f5ec58`	*U*rgent *P*ointer: the sequence number of the byte following the 00B byte.
`urg_data=0000`	Urgent data: byte and associated flags.
`q=recv1`	Queue in use: `rcv1` or `rcv2`.
`rcv:`	Further receiver data:
`bytes=13201600`	Total number of bytes received.
`tot_pkts=`	Total number of packets received.
`rob_pkts=0`	Number of packets in the reorder buffer. Bytes received out of sequence are put into a reorder buffer awaiting further bytes before reordering can occur.
`q_pkts=2+0`	Number of packets queued in (`recv1`+`recv2`).
`usr=0`	Number of bytes of received data available to the user.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description
`eff_mss=1448`	Effective maximum segment size.
`smss=1460`	Sender maximum segment size.
`amss=1460`	Advertised maximum segment size.
`used_bufs=2`	Number of transmit buffers used.
`wscl`	Window scaling:
`s=2`	Send window scaling.
`r=2`	Receive window scaling.
`srtt=01`	Smoothed round trip time (RTT), in milliseconds.
`rttvar=000`	Round trip time (RTT) variation, in milliseconds.
`rto=189`	Current RTO timeout value, in ticks. To convert ticks to milliseconds, multiply by `ci_ip_time_tick2ms`.
`zwins=0,0`	Zero windows for probes,acks: times when advertised window has gone to zero size.
`curr_retrans=0`	Current re-transmissions.
`total_retrans=0`	Total re-transmissions.
`dupacks=0`	Number of duplicate acks received.
`congrecover=0`	Next sequence number to send when loss detected.
`rtos=0`	Number of retrans timeouts.
`frecs=0`	Number of fast recoveries.
`seqerr=0,0`	Number of sequence errors.
`ooo_pkts=0`	Number of out of sequence packets.
`ooo=0`	Number of out of order events.
`tx:`	Transmit data:
`defer=0`	Number of packets where send is deferred to stack lock holder.
`nomac=0`	Number of packets sent via the OS using raw sockets when up to date ARP data is not available.
`warm=0`	Number of packets sent using MSG_WARM.
`warm_aborted=0`	Number of times a message warm send function was called, but not sent due to onload lock constraints.

Table 54: **Stackdump Output: TCP Established Connection Socket** *(cont'd)*

Sample Output	Description

`tmpl:`
`send_fast=0`
`send_slow=0`
`active=0`
	Templated send data:
	Number of fast templated sends.
	Number of slow templated sends.
	Number of active templated sends.

`timers:` `rto(200ms[3fa586ca])`	Currently active timers: Retransmit timeout timer.

TCP LISTEN Sockets

The following table shows typical output for a TCP LISTEN socket.

Table 55: **Stackdump Output: TCP Stack Listen Socket**

Sample Output	Description

`TCP`
`7:3`
`lcl=0.0.0.0:50773`
`rmt=0.0.0.0:0`
`LISTEN`
	TCP socket.
	Stack:socket id.
	Listening on port 50773, local ip address not set (not bound to any IP address).
	Remote ip:port address not set (not bound to any IP address).
	Connection is LISTENing.

`lock: 10000000 UNLOCKED`	Internal socket lock status, as a hexadecimal number, followed by status names including: • `LOCKED` • `CONTENDED`.

Table 55: **Stackdump Output: TCP Stack Listen Socket** *(cont'd)*

Sample Output	Description
`rx_wake=0000b6f4(RQ)`	Internal sequence value that is incremented each time a receive queue is ‘woken’. `(RQ)` indicates that a wake has been requested.
`tx_wake=00000002`	Internal sequence value that is incremented each time a transmit queue is ‘woken’. `(RQ)` indicates that a wake has been requested.
`flags:`	Flags (if any), including: • `WK_TX`, `WK_RX` • `TCP_PP` • `ORPH` • `ACCEPTQ` • `DEFERRED` • `AVOID_INT` • `O_ASYNC`, `O_NONBLOCK`, `O_NDELAY`, `O_APPEND`, `O_CLOEXEC` • `CACHE`, `PASSIVE_CACHE`, `CACHE_NO_FD` • `OS_BACKED` • `NONB_UNSYNCED`.

`ul_poll:`
`369599500 spin cycles`
`100000 usec`
	User-level `poll()`:
	in spin cycles
	in µs.

Table 55: **Stackdump Output: TCP Stack Listen Socket** *(cont'd)*

Sample Output	Description
`uid=0`	User id that owns this socket.
`s_flags: REUSE BOUND PBOUND`	Socket flags, including: • `CORK` • `SHUTRD`, `SHUTWR` • `TCP_NODELAY` • `ACK` • `REUSE` • `KALIVE` • `BCAST` • `OOBIN` • `LINGER` • `DONTROUTE` • `FILTER` • `BOUND`, `ABOUND`, `PBOUND` • `SNDBUF`, `RCVBUF` • `SW_FILTER_FULL` • `TRANSPARENT` • `SCALACTIVE`, `SCALPASSIVE` • `MAC_FILTER` • `REUSEPORT` • `BOUND_ALIEN` • `CONNECT_MUST_BIND` • `PMTU_DO` • `ALWAYS_DF` • `IP_TTL` • `DEFERRED_BIND` • `V6ONLY` • `NOMCAST`. The sample output allows bind to reuse local port.
`rcvbuf=129940`	Socket receive buffer size.
`sndbuf=131072`	Socket send buffer size.
`bindtodev=0(0,0x0:0)`	Device to which the socket is bound, given as an interface, or -1 if unbound. This is followed by an (interface index, hardware port:vlan) tuple.
`ttl=64`	Initial TTL value.
`rcvtimeo_ms=0`	Timeout value (microseconds) before an error is generated for receive functions, as set by SO_RCVTIMEO.
`sndtimeo_ms=0`	Timeout value (microseconds) before an error is generated for send functions, as set by SO_SNDTIMEO.
`sigown=0`	The PID receiving signals from this socket.
`cmsg=`	Current message flags, including: • `NO_MCAST_TX`.

Table 55: **Stackdump Output: TCP Stack Listen Socket** *(cont'd)*

Sample Output	Description
`rx_errno=6b`	Zero whilst data can still arrive, otherwise contains error code.
`tx_errno=20`	Zero if transmit can still happen, otherwise contains error code.
`so_error=0`	Current socket error, or zero if no error.
`os_sock=0`	0 if the socket is handled by Onload, 1 if the socket is handled by the OS and not by Onload.
`TX`	Socket is being used for transmit.
`listenq:`	Listen Queue: This is a queue of half open connects (SYN received and SYNACK sent, waiting for final ACK).
`max=1024`	Maximum number of connections in the queue.
`n=0`	Current number of connections in the queue.
`new=0`	Length of buffer for first connection in the queue.
`buckets=1`	Number of buckets in hash table for queue lookup.
`acceptq:`	Accept Queue: This is a queue of open connections, waiting for the application to call `accept()`.
`max=5`	Maximum number of connections in the queue.
`n=0`	Current number of connections in the queue.
`accepted=0`	Number of connections that have been accepted, and so removed from queue.
`defer_accept=0`	Number of times `TCP_DEFER_ACCEPT` kicked in (see TCP Level Options), or 255 if `TCP_DEFER_ACCEPT` is disabled.
`sockcache:`	Socket endpoint cache:
`n=0`	Number of endpoints currently known to this socket.
`sock_n=0`	Number of available cache entries for this socket.
`cache=EMPTY`	`EMPTY`, or `yes` if endpoints are in the cache (can be used for an accept).
`pending=EMPTY`	`EMPTY`, or `yes` if endpoints are waiting to be cached because they are in close-wait (closed but not dropped).
`connected=EMPTY`	`EMPTY`, or `yes` if endpoints are connected (accepted).

Table 55: **Stackdump Output: TCP Stack Listen Socket** *(cont'd)*

Sample Output	Description
`l_overflow=0`	Number of times listen queue was full and had to reject a SYN request.
`l_no_synrecv=0`	Number of times unable to allocate internal resource for a SYN request.
`aq_overflow=0`	Number of times unable to promote a connection to the accept queue because the queue was full.
`aq_no_sock=0`	Number of times unable to promote a connection to the accept queue because could not create a socket.
`aq_no_pkts=0`	Number of times unable to promote a connection to the accept queue because could not create a packet buffer.
`a_loop2_closed=0`	Number of times the real client for a loopback has gone, and so the connection was closed.
`a_no_fd=0`	Number of times a file descriptor could not be acquired.
`ack_rsts=0`	Number of times received an ACK before SYN, so the connection was reset.
`os=2`	Number of sockets being processed in the kernel.
`rx_pkts=0`	Number of packets received.

UDP Sockets

The following table shows typical output for a UDP socket

Table 56: **Stackdump Output: UDP Socket**

Sample Output	Description
`UDP`	Socket configuration:
`4:1`	Stack:socket id.
`lcl=192.168.1.2:38142`	Local ip:port address.
`rmt=192.168.1.1:42638`	Remote ip:port address.
`UDP`	Connection is UDP.
`lock: 20000000 LOCKED`	Internal socket lock status, as a hexadecimal number, followed by status names including: • `LOCKED` • `CONTENDED`.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`rx_wake=000e69b0`	Internal sequence value that is incremented each time a receive queue is 'woken'. `(RQ)` indicates that a wake has been requested.
`tx_wake=000e69b1`	Internal sequence value that is incremented each time a transmit queue is 'woken'. `(RQ)` indicates that a wake has been requested.
`flags:`	Flags (if any), including: • `WK_TX`, `WK_RX` • `TCP_PP` • `ORPH` • `ACCEPTQ` • `DEFERRED` • `AVOID_INT` • `O_ASYNC`, `O_NONBLOCK`, `O_NDELAY`, `O_APPEND`, `O_CLOEXEC` • `CACHE`, `PASSIVE_CACHE`, `CACHE_NO_FD` • `OS_BACKED` • `NONB_UNSYNCED`.

`ul_poll:`
`0 spin cycles`
`0 (usec)`
	User-level `poll()`:
	in spin cycles
	in µs.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`uid=0`	User id that owns this socket.
`s_flags: FILTER`	Socket flags, including: • `CORK` • `SHUTRD`, `SHUTWR` • `TCP_NODELAY` • `ACK` • `REUSE` • `KALIVE` • `BCAST` • `OOBIN` • `LINGER` • `DONTROUTE` • `FILTER` • `BOUND`, `ABOUND`, `PBOUND` • `SNDBUF`, `RCVBUF` • `SW_FILTER_FULL` • `TRANSPARENT` • `SCALACTIVE`, `SCALPASSIVE` • `MAC_FILTER` • `REUSEPORT` • `BOUND_ALIEN` • `CONNECT_MUST_BIND` • `PMTU_DO` • `ALWAYS_DF` • `IP_TTL` • `DEFERRED_BIND` • `V6ONLY` • `NOMCAST`.
`rcvbuf=129024`	Socket receive buffer size.
`sndbuf=129024`	Socket send buffer size.
`bindtodev=-1(01,0:0)`	Device to which the socket is bound, given as an interface, or -1 if unbound. This is followed by an (interface index, hardware port:vlan) tuple.
`ttl=64`	Initial TTL value.
`rcvtimeo_ms=0`	Timeout value (microseconds) before an error is generated for receive functions, as set by SO_RCVTIMEO.
`sndtimeo_ms=0`	Timeout value (microseconds) before an error is generated for send functions, as set by SO_SNDTIMEO.
`sigown=0`	The PID receiving signals from this socket.
`cmsg=`	Current message flags, including: • `NO_MCAST_TX`.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description

`rx_errno=0`
 	
`tx_errno=0`
 	
`so_error=0`
`os_sock=0`
 	
`TX`
	Zero whilst data can still arrive, otherwise contains error code.
	Zero if transmit can still happen, otherwise contains error code.
	Current socket error, or zero if no error.
	0 if the socket is handled by Onload, 1 if the socket is handled by the OS and not by Onload.
	Socket is being used for transmit.

`epoll3: ready_list_id 0`	List of ready sockets from the epoll3 set.
`udpflags: FILT MCAST_LOOP RXOS`	UDP flags currently set for this socket, including: • `FILT` • `MCAST_LOOP` • `IMP_BIND` • `EFSND` • `LAST_RCV_ON` • `BIND` • `MC_B2D` • `NO_MC_B2D` • `PEEKOS` • `SO_TS` • `MC` • `MC_FILT` • `NO_UC_FILT`.

`rcv:`
`q_pkts=0`
`reap=2`
`tot_pkts=944560`
	Receive data:
	Number of packets currently in receive queue.
	Number of packet buffers in the process of being freed for reuse.
	Total number of packet buffers used.
	Note any packets that are delivered to multiple receive queues get wrapped in an additional 0-length buffer, to reference those queues.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`rcv:`	Further receive data:
`oflow_drop=0(0%)`	Number of packets dropped because the buffer is full.
`mem_drop=0`	Number of packets dropped due to running out of packet buffer memory.
`eagain=0`	Number of times the application tried to read from a socket when there is no data ready. This value can be ignored on the receive side.
`pktinfo=0`	Number of times an `IP_PKTINFO` control message was received.
`q_max_pkts=0`	Maximum depth reached by the receive queue (packets).
`rcv:`	Further receive data:
`os=0(0%)`	Number of packets received via the operating system, both as a number and as a percentage of total packets received.
`os_slow=0`	Number of packets received via the operating system slow path.
`os_error=0`	Number of times a `recv()` function call via the operating system returned an error.
`snd:`	Send data:
`q=0+0`	Number of bytes sent to the interface but not yet transmitted, + number of bytes waiting because the interface lock is contended in `sendmsg()`.
`ul=944561`	Number of packets sent via Onload.
`os=0(0%)`	Number of packets sent via the operating system, both as a number and as a percentage of total packets sent.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description

`snd:`
`LOCK`
`cp=1(0%)`
`pkt=737815(99%)`
`snd=3(0%)`
`poll=0(0%)`
`defer=1(0%)`
	Send data...
	...about locks:
	Count of locks held while updating the control plane.
	Count of locks to get a packet buffer.
	Count of locks held when sending.
	Count of locks held to poll the stack.
	Count of sends deferred to the lock holder.

`snd:`
`MCAST`
`if=9`
`src=172.16.128.28`
`ttl=1`
	Send data...
	...about multicast:
	The interfaces being used by the UDP stack.
	Source IP address for multicast.
	Initial TTL value.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`snd:`	Send data...
`TO`	...about unconnected sends:
`n=737820`	Total number of UDP packets sent on this socket via Onload.
`match=737819(99%)`	Number of packets that matched the cache, both as a number and as a percentage of total packets sent.
`lookup=1+0(0%)`	Number of packets needing lookup, from the control plane + because unlocked, both as numbers and as a percentage of total packets sent.

`Onloaded(Valid)`	Status of cached control plane information, one of:
	`Onloaded` = can reach the destination via an accelerated interface
	`NoMac`
	`NoRoute`
	`ViaOs`
	`Local`
	`MacFail`
	followed by its validity:
	`(Valid)`: information is up-to-date. Can send immediately using this information.
	`(Old)`: information might be out-of-date. On next send Onload will do a control plane lookup - this will add some latency.

`snd:`	Further send data...
`TO`	...about unconnected sends:
`if=9`	Interface being used.
`mtu=1500`	MTU being used.
`intf_i=0`	Intf_i value.
`vlan=0`	VLAN being used.
`encap=4`	Types of encapsulation supported by the NIC, as a hexadecimal mask.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description

`snd:`
`TO`
`172.16.128.28:34645 =>`
`224.1.2.3:8001`
	Further send data...
	...about unconnected sends:
	UDP send multicast source ip:port address.
	UDP send multicast ip:port address.

`snd:`
`CON`
`n=0`
`lookup=0`
`NoRoute(Old)`
	Further send data...
	...about connected sends:
	Total number of UDP packets sent on this socket via Onload.
	Number of packets needing lookup from the control plane.
	Status of cached control plane information, one of:
	`Onloaded` = can reach the destination via an accelerated interface
	`NoMac`
	`NoRoute`
	`ViaOs`
	`Local`
	`MacFail`
	followed by its validity:
	`(Valid)`: information is up-to-date. Can send immediately using this information.
	`(Old)`: information might be out-of-date. On next send Onload will do a control plane lookup - this will add some latency.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`snd:`	Further send data...
`CON`	...about connected sends:
`if=9`	Interface being used.
`mtu=0`	MTU being used.
`intf_i=-1`	Intf_i value.
`vlan=0`	VLAN being used.
`encap=0`	Types of encapsulation supported by the NIC, as a hexadecimal mask.
`snd:`	Further send data:
`eagain=0`	Count of the number of times the application tried to send data, but the transmit queue is already full. A high value on the send side might indicate transmit issues.
`spin=0`	Number of times process had to spin when the send queue was full.
`block=0`	Number of times process had to block when the send queue was full.
`snd:`	Further send data:
`poll_avoids_full=0`	Number of times polling created space in the send queue.
`fragments=0`	Number of (non first) fragments sent.
`confirm=0`	Number of packets sent with `MSG_CONFIRM` flag.

Table 56: **Stackdump Output: UDP Socket** *(cont'd)*

Sample Output	Description
`snd:`	Further send data:
`os_slow=1`	Number of packets sent via the operating system slow path.
`os_late=0`	Number of packets sent via the operating system after copying
`unconnect_late=0`	Number of packets silently dropped when process/ thread becomes disconnected during a send procedure.
`nomac=0(0%)`	Number of times when no MAC address was known, so ARP was required before delivering traffic.

Statistics

Following the stack and socket data `onload_stackdump lots` will display a list of statistical data. For descriptions of the fields refer to the output from the following command:

```
onload_stackdump describe_stats
```

Environment Variables

The final list produced by `onload_stackdump lots` shows the current values of all environment variables in the monitored process environment. For descriptions of the environment variables refer to Appendix A: Parameter Reference or use the following command:

```
onload_stackdump doc
```

Onload Stackdump Filters

Use the onload_stackdump filters commands to identify filters installed by the Onload application.

```
# onload_stackdump filters

1. oof_manager_dump: hwports up=f, down=0 unavailable=0 local_addr_n=1
2.   172.16.130.252 active=1 sockets=0
3. oof_local_port_dump: UDP:8001 n_refs=1
4.   wild sockets:
5.     : 0:3 UDP 0.0.0.0:8001 0.0.0.0:0 ACCELERATED
```

```
6.   FILTER 172.16.130.252:8001 hwports=c stack=0
7.   mcast filters:
8.     maddr=224.1.2.3:8001 stack=0 hwports=1,1,0
9.       0:3 UDP 0.0.0.0:8001 0.0.0.0:0 if=6 hwports=1,1,0 KERNEL
10. oof_manager_dump: scalable interfaces and MAC filters
```

```
# onload_stackdump filter_table

1. ci_netif_filter_dump: 0 size=16384 n_entries=2 n_slots=2 max=1 mean=1
2. 0000000431 id=3          rt_ct=0 UDP 224.1.2.3:8001 0.0.0.0:0
0000000431:1109197297
3. 0000012173 id=3          rt_ct=0 UDP 172.16.130.252:8001 0.0.0.0:0
0000012173:-1113780035
```

Remote Monitoring

The Onload Remote Monitor (ORM) provides similar details to `onload_stackdump` about Onload stacks and sockets. This data is exported in JSON format, which might be easier for an application to consume. The data is typically processed by a remote third-party monitoring application, such as `collectd`.

Two scripts are supplied:

- `orm_webserver` provides the data via a webserver, for remote consumption.

  See orm_webserver.

- `orm_json` provides the data to stdout, for local consumption.

  See orm_json.

These scripts are installed via the `onload_install` script, in the following directory:

```
onload-<version>/src/tools/onload_remote_monitor
```

(from OpenOnload 201606-u1 onwards and EnterpriseOnload 5.0.0 onwards).

From Onload-7.1.0 onwards, most of the ORM functionality has been separated into a new `orm_json_lib` library. The `orm_webserver` and `orm_json` scripts now internally use this library, and customers can also use it to build their own custom monitoring solutions. See orm_json_lib.

orm_webserver

To allow the statistics to be queried from a remote machine, run the `orm_webserver` Python script on the machine that is running Onload, specifying a port through which HTTP clients can connect:

```
# orm_webserver <port>
```

The script starts a webserver process that provides the following URLs, where `<stackname>` is the EF_NAME for a stack:

- http://<serverhost>:<port>/onload/stats
- http://<serverhost>:<port>/onload/stack
- http://<serverhost>:<port>/onload/opts
- http://<serverhost>:<port>/onload/lots
- http://<serverhost>:<port>/onload/all
- http://<serverhost>:<port>/onload/stackname/<stackname>/stats
- http://<serverhost>:<port>/onload/stackname/<stackname>/stack
- http://<serverhost>:<port>/onload/stackname/<stackname>/opts
- http://<serverhost>:<port>/onload/stackname/<stackname>/lots
- http://<serverhost>:<port>/onload/stackname/<stackname>/all

Gathering the Statistics

An example of how to set up `collectd` to gather statistics from Onload Remote Monitor is provided in:

```
onload-<version>/src/tests/onload/onload_remote_monitor/using_collectd/
```

orm_json

Alternatively, `orm_json` can be run directly, in a similar manner to `onload_stackdump`. It sends the JSON output to `stdout` on the local machine.

To see the available options, type `orm_json -h`.

orm_json_lib

A simple example of how to use `orm_json_lib` is provided in `src/tools/onload_remote_monitor/orm_zmq_publisher.c`. This will publish the chosen statistics via a ZeroMQ publisher every *N* seconds. A companion example `src/tools/onload_remote_monitor/zmq_subscriber.c` can be used to receive the JSON statistics via ZeroMQ.

sfnettest

Sfnettest is a set of benchmark tools and test utilities created by Solarflare for benchmark and performance testing of network servers and network adapters. Sfnettest is available in source form from:

https://github.com/Xilinx-CNS/cns-sfnettest

Download the sfnettest source and unpack if necessary, for example:

```
# unzip cns-sfnettest-master.tgz
```

Run the `make` utility from the `src` subdirectory to build the benchmark applications.

Refer to the `README.sfnt-pingpong` or `README.sfnt-stream` files in the distribution directory once sfnettest is installed.

sfnt-pingpong

Description

The `sfnt-pingpong` application measures TCP and UDP latency by creating a single socket between two servers and running a simple message pattern between them. The output identifies latency and statistics for increasing TCP/UDP packet sizes.

Usage

```
sfnt-pingpong [options] [<tcp|udp|pipe|unix_stream|unix_datagram>
[<host[:port]>]]
```

Options

The following table lists the sfnt-pingpong options:

Table 57: **sfnt-pingpong Options**

Option	Description
`--port`	server port

Table 57: **sfnt-pingpong Options** *(cont'd)*

Option	Description
`--sizes`	single message size (bytes)
`--connect`	`connect()` UDP socket
`--spin`	spin on non-blocking `recv()`
`--muxer`	select, poll or epoll
`--serv-muxer`	none, select, poll or epoll (same as client by default)
`--rtt`	report round-trip-time
`--raw`	dump raw results to files
`--percentile`	percentile
`--minmsg`	minimum message size
`--maxmsg`	maximum message size
`--minms`	min time per msg size (ms)
`--maxms`	max time per msg size (ms)
`--miniter`	minimum iterations for result
`--maxiter`	maximum iterations for result
`--mcast`	use multicast addressing
`--mcastintf`	set the multicast interface. The client sends this parameter to the server. `--mcastintf=eth2` both client and server use eth2 `--mcastintf='eth2;eth3'` client uses eth2 and server uses eth3 (quotes are required for this format)
`--mcastloop`	IP_MULTICAST_LOOP
`--bindtodev`	SO_BINDTODEVICE
`--forkboth`	fork client and server
`--n-pipe`	include pipes in file descriptor set
`--n-unix-d`	include Unix datagrams in the file descriptor set
`--n-unix-s`	include Unix streams in the file descriptor set
`--n-udp`	include UDP sockets in file descriptor set
`--n-tcpc`	include TCP sockets in file descriptor set
`--n-tcpl`	include TCP listening sockets in file descriptor set
`--tcp-serv`	host:port for TCP connections
`--timeout`	socket SND/RECV timeout
`--affinity`	`'<client-core>;<server-core>'` Enclose values in quotes. This option should be set on the client side only. The client sends the <server_core> value to the server. The user must ensure that the identified server core is available on the server machine. **IMPORTANT!** *This option will override any value set by taskset on the same command line.*
`--n-pings`	number of ping messages
`--n-pongs`	number of pong messages
`--nodelay`	enable TCP_NODELAY

The following table lists the standard options:

Table 58: **Standard Options**

Option	Description
`-? --help`	this message
`-q --quiet`	quiet
`-v --verbose`	display more information

Example TCP Latency Command Lines

```
[server]# onload --profile=latency taskset -c 1 ./sfnt-pingpong
```

```
[client]# onload --profile=latency taskset -c 1 ./sfnt-pingpong \
                      --maxms=10000 --affinity "1;1" tcp <server-ip>
```

Example UDP Latency Command Lines

```
[server]# onload --profile=latency taskset -c 9 ./sfnt-pingpong
```

```
[client]# onload --profile=latency taskset -c 9 ./sfnt-pingpong \
                      --maxms=10000 --affinity "9;9" udp <server_ip>
```

Example Output

```
# version: 1.5.0
# src: 8dc3b027d85b28bedf9fd731362e4968
# date: Tue  9 Feb 13:15:46 GMT 2016
# uname: Linux dellr210g2q.uk.level5networks.com 3.10.0-327.el7.x86_64 #1
SMP Thu Oct 29 17:29:29 EDT 2015 x86_64 x86_64 x86_64 GNU/Linux
# cpu: model name        : Intel(R) Xeon(R) CPU E3-1280 V2 @ 3.60GHz
# lspci: 05:00.0 Ethernet controller: Intel Corporation I350 Gigabit
Network Connection (rev 01)
# lspci: 05:00.1 Ethernet controller: Intel Corporation I350 Gigabit
Network Connection (rev 01)
# lspci: 83:00.0 Ethernet controller: Solarflare Communications SFC9020
[Solarstorm]
# lspci: 83:00.1 Ethernet controller: Solarflare Communications SFC9020
[Solarstorm]
# lspci: 85:00.0 Ethernet controller: Intel Corporation 82574L Gigabit
Network Connection
# eth0: driver: igb
# eth0: version: 3.0.6-k
# eth0: bus-info: 0000:05:00.0
# eth1: driver: igb
# eth1: version: 3.0.6-k
# eth1: bus-info: 0000:05:00.1
# eth2: driver: sfc
# eth2: version: 3.2.1.6083
# eth2: bus-info: 0000:83:00.0
# eth3: driver: sfc
# eth3: version: 3.2.1.6083
# eth3: bus-info: 0000:83:00.1
# eth4: driver: e1000e
# eth4: version: 1.4.4-k
# eth4: bus-info: 0000:85:00.0
# virbr0: driver: bridge
```

```
# virbr0: version: 2.3
# virbr0: bus-info: N/A
# virbr0-nic: driver: tun
# virbr0-nic: version: 1.6
# virbr0-nic: bus-info: tap
# ram: MemTotal:        32959748 kB
# tsc_hz: 3099966880
# LD_PRELOAD=libonload.so
# server LD_PRELOAD=libonload.so
# onload_version=201205
# EF_TCP_FASTSTART_INIT=0
# EF_POLL_USEC=100000
# EF_TCP_FASTSTART_IDLE=0
#
#       size    mean    min     median  max     %ile    stddev  iter
        1       2453    2380    2434    18288   2669    77      1000000
        2       2453    2379    2435    45109   2616    90      1000000
        4       2467    2380    2436    10502   2730    82      1000000
        8       2465    2383    2446    8798    2642    70      1000000
        16      2460    2380    2441    7494    2632    68      1000000
        32      2474    2399    2454    8758    2677    71      1000000
        64      2495    2419    2474    12174   2716    77      1000000
```

The output identifies mean, minimum, median and maximum (nanosecond) ½ RTT latency for increasing packet sizes including the 99% percentile and standard deviation for these results. In the above example a message size of 32 bytes has a mean latency of 2.4 microseconds with a 99%ile latency less than 2.7 microseconds.

sfnt-stream

The `sfnt-stream` application measures RTT latency (not ½ RTT) for a fixed size message at increasing message rates. Latency is calculated from a sample of all messages sent. Message rates can be set with the `rates` option and the number of messages to sample using the `sample` option.

sfnt-stream only functions on UDP sockets. This limitation will be removed to support other protocols in the future.

Refer to the `README.sfnt-stream` file which is part of the Onload distribution for further information.

Usage

```
sfnt-stream [options] [tcp|udp|pipe|unix_stream|unix_datagram [host[:port]]]
```

Options

The following table lists the sfnt-stream options:

Table 59: **sfnt-stream Options**

Option	Description
--msgsize	message size (bytes)
--rates	msg rates <min>-<max>[+<step>]
--millisec	time per test (milliseconds)
--samples	number of samples per test
--stop	stop when TX rate achieved is below give percentage of target rate
--maxburst	maximum burst length
--port	server port number
--connect	connect() UDP socket
--spin	spin on non-blocking recv()
--muxer	select, poll, epoll or none
--rtt	report round-trip-time
--raw	dump raw results to file
--percentile	percentile
--mcast	set the multicast address
--mcastintf	set multicast interface. The client sends this parameter to the server. --mcastintf=eth2 both client and server use eth2 --mcastintf='eth2;eth3' client uses eth2 and server uses eth3 (quotes are required for this format)
--mcastloop	IP_MULTICAST_LOOP
--ttl	IP_TTL and IP_MULTICAST_TTL
--bindtodevice	SO_BINDTODEVICE
--n-pipe	include pipes in file descriptor set
--n-unix-d	include Unix datagram in file descriptor set
--n-unix-s	include Unix stream in file descriptor set
--n-udp	include UDP sockets in file descriptor set
--n-tcpc	include TCP sockets in file descriptor set
--n-tcpl	include TCP listening sockets in file descriptor set
--tcpc-serv	host:port for TCP connections
--nodelay	enable TCP_NODELAY
--affinity	"<client-tx>,<client-rx>;<server-core>" enclose the values in double quotes, for example "4,5;3". This option should be set on the client side only. The client sends the <server_core> value to the server. The user must ensure that the identified server core is available on the server machine. **IMPORTANT!** *This option will override any value set by taskset on the same command line.*
--rtt-iter	iterations for RTT measurement

The following table lists the standard options:

Table 60: **Standard Options**

Option	Description
`-? --help`	this message
`-q --quiet`	quiet
`-v --verbose`	display more information
`--version`	display version information

Example Command Lines for Client/Server

```
[server]# ./sfnt-stream
```

```
[client]# ./sfnt-stream --affinity 1,1 udp <server-ip>
```

```
[client]# ./taskset -c 1 ./sfnt-stream --affinity="3,5;3" --mcastintf=eth4
udp \
                              <remote-ip>
```

Example of Bonded Interfaces

The following example configures a single bond, having two slaves interfaces, on each machine. Both server and client machines use eth4 and eth5.

```
[root@server src]# ifconfig eth4 0.0.0.0 down
[root@server src]# ifconfig eth5 0.0.0.0 down
[root@server src]# modprobe bonding miimon=100 mode=1
xmit_hash_policy=layer2 primary=eth5
[root@server src]# ifconfig bond0 up
[root@server src]# echo +eth4 > /sys/class/net/bond0/bonding/slaves
[root@server src]# echo +eth5 > /sys/class/net/bond0/bonding/slaves
[root@server src]# ifconfig bond0 172.16.136.28/21
NOTE: server sends to IP address of client bond
[root@server src]# onload --profile=latency taskset -c 1 ./sfnt-stream --
mcastintf=bond0 --affinity "1,1;3" udp 172.16.136.27
```

```
[root@client src]# ifconfig eth4 0.0.0.0 down
[root@client src]# ifconfig eth5 0.0.0.0 down
[root@client src]# modprobe bonding miimon=100 mode=1
xmit_hash_policy=layer2 primary=eth5
[root@client src]# ifconfig bond0 up
[root@client src]# echo +eth4 > /sys/class/net/bond0/bonding/slaves
[root@client src]# echo +eth5 > /sys/class/net/bond0/bonding/slaves
[root@client src]# ifconfig bond0 172.16.136.27/21
[root@client src]# onload --profile=latency taskset -c 3 ./sfnt-stream
sfnt-stream: server: waiting for client to connect...
sfnt-stream: server: client connected
sfnt-stream: server: client 0 at 172.16.136.28:45037
```

Output Fields

All time measurements are nanoseconds unless otherwise stated.

Table 61: **sfnt-stream Output Fields**

Field	Description
`mps target`	Msg per sec target rate
`mps send`	Msg per sec actual rate
`mps recv`	Msg receive rate
`latency mean`	RTT mean latency
`latency min`	RTT minimum latency
`latency median`	RTT median latency
`latency max`	RTT maximum latency
`latency %ile`	RTT 99%ile
`latency stddev`	Standard deviation of sample
`latency samples`	Number of messages used to calculate latency measurement
`sendjit mean`	Mean variance when sending messages
`sendjit min`	Minimum variance when sending messages
`sendjit max`	Maximum variance when sending messages
`sendjit behind`	Number of times the sender falls behind and is unable to keep up with the transmit rate
`gaps n_gaps`	Count the number of gaps appearing in the stream
`gaps n_drops`	Count the number of drops from stream
`gaps n_ooo`	Count the number of sequence numbers received out of order

Running Without Spinning

Both sfnt-pingpong and sfnt-stream use scripts found in the `onload_apps` subdirectory which invoke the onload latency profile thereby causing the application to ‘spin’.

To run these test programs in an interrupt driven mode, replace the `--profile=latency` option on the command line, with the `--no-app-handler` option.

onload_tcpdump

By definition, Onload is a kernel bypass technology and this prevents packets from being captured by packet sniffing applications such as tcpdump, netstat and wireshark.

Onload supports the onload_tcpdump application that supports packet capture from onload stacks to a file or to be displayed on standard out (stdout). Packet capture files produced by onload_tcpdump can then be imported to the regular tcpdump, wireshark or other third party application where users can take advantage of search and analysis features.

Onload_tcpdump allows for the capture of all TCP and UDP unicast and multicast data sent or received via Onload stacks - including shared stacks.

Note: Onload tcpdump is not a replacement for the standard Linux tcpdump utility. Onload tcpdump captures traffic only from Onload stacks.

Building onload_tcpdump

The onload_tcpdump script is supplied with the Onload distribution and is located in the Onload-<version>/scripts sub-directory.

Note: libpcap and libpcap-devel must be built and installed *before* Onload is installed.

Using onload_tcpdump

For help use the ./onload_tcpdump -h command:

```
Usage:
onload_tcpdump [-o stack-(id|name) [-o stack ...]]
tcpdump_options_and_parameters
"man tcpdump" for details on tcpdump parameters.
You may use stack id number or shell-like pattern for the stack name
to specify the Onload stacks to listen on.
If you do not specify stacks, onload_tcpdump will monitor all onload stacks.
If you do not specify interface via -i option, onload_tcpdump
listens on ALL interfaces instead of the first one.
```

For further information refer to the Linux man tcpdump pages.

***Note*:** Onload tcpdump only accepts separate command line options - combined options will be ignored by the application parser:

The following example will work:

```
onload_tcpdump -n -i <interface>
```

The following example will *not* work:

```
onload_tcpdump -ni <interface>
```

Examples

- Capture all accelerated traffic from eth2 to a file called `mycaps.pcap`:

  ```
  # onload_tcpdump -ieth2 -wmycaps.pcap
  ```

- If no file is specified onload_tcpdump will direct output to stdout:

  ```
  # onload_tcpdump -ieth2
  ```

- To capture accelerated traffic for a specific Onload stack (by name):

  ```
  # onload_tcpdump -ieth4 -o stackname
  ```

- To capture accelerated traffic for a specific Onload stack (by ID):

  ```
  # onload_tcpdump -o 7
  ```

- To capture accelerated traffic for Onload stacks where name begins with “abc”

  ```
  # onload_tcpdump -o 'abc*'
  ```

- To capture accelerated traffic for onload stack 1, stack named “stack2” and all onload stacks with name beginning with “ab”:

  ```
  # onload_tcpdump -o 1 -o 'stack2' -o 'ab*'
  ```

VLAN Examples

- Capture all UDP VLAN tagged traffic from the specified interface:

  ```
  # onload_tcpdump -nn -i eth3 udp and vlan
  ```

- Capture all UDP non-VLAN tagged traffic from the specified interface:

  ```
  # onload_tcpdump -nn -i eth3 udp and not vlan
  ```

Dependencies

The `onload_tcpdump` application requires `libpcap` and `libpcap-devel` to be installed on the server. If `libpcap` is not installed the following message is reported when `onload_tcpdump` is invoked:

```
./onload_tcpdump
ci Onload was compiled without libpcap development package installed. You
need to install libpcap-devel or libpcap-dev package to run onload_tcpdump.
tcpdump: truncated dump file; tried to read 24 file header bytes, only got 0
Hangup
```

If `libpcap` is missing it can be downloaded from http://www.tcpdump.org/.

Untar the compressed file on the target server and follow build instructions in the `INSTALL.txt` file. The `libpcap` package must be installed before Onload is built and installed.

Limitations

- Using multiple onload_tcpdump instances to capture from the same onload stack is not a supported configuration.
- Currently `onload_tcpdump` captures only packets from Onload stacks and not from kernel stacks.
- `onload_tcpdump` delivers timestamps with microsecond resolution. `onload_tcpdump` does not support nanosecond precision.
- The onload_tcpdump application monitors stack creation events and will attach to newly created stacks however, there is a short period (normally only a few milliseconds) between stack creation and the attachment during which packets sent/received will not be captured.

Known Issues

- Users might observe that the packets sent when the destination address is not in the host ARP table causes the packets to appear in both `onload_tcpdump` and (Linux) tcpdump.

  CAUTION! *Users should not attempt to accelerate onload_tcpdump. The following command should not be used:*

  ```
  onload onload_tcpdump -i <interface>
  ```

- onload_tcpdump will also be accelerated if LD_PRELOAD is exported in the Onload environment, so the following methods should not be used:

  ```
  # export LD_PRELOAD=libonload.so
  # onload_tcpdump -i <interface>
  ```

SolarCapture

SolarCapture is a packet capture application for Solarflare network adapters. It is able to capture received packets from the wire at line rate, assigning accurate nanosecond precision timestamps to each packet. Packets are captured to PCAP file or forwarded to user-supplied logic for processing. For details see the *SolarCapture User Guide* (SF-108469-CD).

Appendix H

ef_vi

The ef_vi API is a layer 2 API that grants an application direct access to the Solarflare network adapter datapath to deliver lower latency and reduced per message processing overheads. ef_vi is the internal API used by Onload for sending and receiving packets. It can be used directly by applications that want the very lowest latency send and receive API and that do not require a POSIX socket interface.

- ef_vi is packaged with the Onload distribution.
- ef_vi is an OSI level 2 interface which sends and receives raw Ethernet frames.
- ef_vi supports a zero-copy interface because the user process has direct access to memory buffers used by the hardware to receive and transmit data.
- An application can use both ef_vi and Onload at the same time. For example, use ef_vi to receive UDP market data and Onload sockets for TCP connections for trading.
- The ef_vi API can deliver lower latency than Onload and incurs reduced per message overheads.
- ef_vi is free software distributed under a LGPL license.
- The user application wishing to use the layer 2 ef_vi API must implement the higher layer protocols.

Components

All components required to build and link a user application with the Solarflare ef_vi API are distributed with Onload. When Onload is installed all required directories/files are located under the Onload distribution directory.

Compiling and Linking

Refer to the `README.ef_vi` file in the Onload directory for compile and link instructions.

Documentation

The ef_vi documentation is distributed in doxygen format with the Onload distribution. Documents in HTML and LaTeX format are generated by running doxygen in the following directory:

```
# cd onload-<version>/src/include/etherfabric/doxygen
# doxygen doxyfile_ef_vi
```

Documents are generated in the `html` and `latex` sub-directories.

The *ef_vi User Guide* can be viewed in HTML format by opening the `html/index.html` file.

If TeX Live is installed (version 2014 or later is recommended), the *ef_vi User Guide* can be generated in PDF format by:

```
# cd latex
# make pdf
```

The *ef_vi User Guide* (SF-114063-CD) is also available in PDF format from the Solarflare download site.

Appendix I

onload_iptables

The Linux netfilter iptables feature provides filtering based on user-configurable rules with the aim of managing access to network devices and preventing unauthorized or malicious passage of network traffic. Packets delivered to an application via the Onload accelerated path are not visible to the OS kernel and, as a result, these packets are not visible to the kernel firewall (iptables).

The onload_iptables feature allows the user to configure rules which determine which hardware filters Onload is permitted to insert on the adapter and therefore which connections and sockets can bypass the kernel and, as a consequence, bypass iptables.

The onload_iptables command can convert a snapshot copy of the kernel iptables rules into Onload firewall rules.

***Note*:** Any changes to kernel iptables subsequent to the snapshot will not be reflected in the Onload firewall.

These Onload firewall rules are used to determine if sockets, created by an Onloaded process, are retained by Onload or handed off to the kernel network stack. Additionally, user-defined filter rules can be added to the Onload firewall on a per interface basis. The Onload firewall applies to the receive filter path only.

How it Works

Before Onload accelerates a socket it first checks the Onload firewall module. If the firewall module indicates the acceleration of the socket would violate a firewall rule, the acceleration request is denied and the socket is handed off to the kernel. Network traffic sent or received on the socket is not accelerated.

Onload firewall rules are parsed in ascending numerical order. The first rule to match the newly created socket - which can indicate to accelerate or decelerate the socket - is selected and no further rules are parsed.

If the Onload firewall rules are an exact copy of the kernel iptables, with no additional rules added by the Onload user, then a socket handed off to the kernel because of an iptables rule violation will be unable to receive data through either path.

Changing rules using onload_iptables will not interrupt existing network connections.

Note: Onload firewall rules will not persist over network driver restarts.

Note: The onload_iptables "IP rules" will only block hardware IP filters from being inserted and onload_iptables "MAC rules" will only block hardware MAC filters from being inserted. Therefore it is possible that if a rule is inserted to block a MAC address, the user is still able to accept traffic from the specified host by Onload inserting an appropriate IP hardware filter.

Files

When the Onload drivers are loaded, firewall rules exist in the Linux `proc` pseudo file system at:

```
/proc/driver/sfc_resource
```

Within this directory the `firewall_add`, `firewall_del` and `resources` files will be present. These files are writable only by a root user. *No attempt should be made to remove these files.*

Once rules have been created for a particular interface – and only while these rules exist – a separate directory exists which contains the current firewall rules for the interface:

```
/proc/driver/sfc_resource/ethN/firewall_rules
```

Features

To get help:

```
# onload_iptables -h
```

Rules

The general format of the rule is:

```
[rule=n] if=ethN protocol=(ip|tcp|udp) [local_ip=a.b.c.d[/mask]]
[remote_ip=a.b.c.d[/mask]] [local_port=a[-b]] [remote_port=a[-b]] [vlan=n]
action=(ACCELERATE|DECELERATE)
```

Note: Using the IP address rule form, the vlan identifier is effective only when using a Solarflare SFN8000 or X2 series adapter which is configured to use the full-featured firmware variant. On other Solarflare adapters the vlan identifier is ignored. The vlan identifier can only be specified with the vlan=n syntax and not on the interface.

```
[rule=n] if=ethN protocol=eth mac=xx:xx:xx:xx:xx:xx[/FF:FF:FF:FF:FF:FF]
[vlan=n] action=(ACCELERATE|DECELERATE)
```

***Note*:** Using the MAC address rule form, the vlan identifier is effective when specified for any Solarflare adapter.

Preview Firewall Rules

Before creating the Onload firewall, run the `onload_iptables -v` option to identify which rules will be adopted by the firewall and which will be rejected (a reason is given for rejection):

```
# onload_iptables -v
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:5201
=> if=None protocol=tcp local_ip=0.0.0.0/0 local_port=5201-5201
remote_ip=0.0.0.0/0 remote_port=0-65535 action=DECELERATE
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:5201
=> if=None protocol=tcp local_ip=0.0.0.0/0 local_port=5201-5201
remote_ip=0.0.0.0/0 remote_port=0-65535 action=DECELERATE
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpts:80:88
=> if=None protocol=tcp local_ip=0.0.0.0/0 local_port=80-88
remote_ip=0.0.0.0/0 remote_port=0-65535 action=
tcp  --  0.0.0.0/0            0.0.0.0/0           tcp spt:800
=> Error parsing: Insuffcient arguments in rule.
```

The last rule is rejected because the action is missing.

***Note*:** The -v option does not create firewall rules for any Solarflare interface, but allows the user to preview which Linux iptables rules will be accepted and which will be rejected by Onload.

To Convert Linux iptables to Onload Firewall Rules

The Linux iptables can be applied to all or individual Solarflare interfaces.

Onload iptables are only applied to the receive filter path. The user can select the INPUT CHAIN or a user defined CHAIN to parse from the iptables. The default CHAIN is INPUT. To adopt the rules from iptables even though some rules will be rejected enter the following command identifying the Solarflare interface the rules should be applied to:

```
# onload_iptables -i ethN -c
# onload_iptables -a -c
```

Running the onload_iptables command will overwrite existing rules in the Onload firewall when used with the -i (interface) or -a (all interfaces) options.

***Note*:** Applying the Linux iptables to a Solarflare interface is optional. The alternatives are to create user-defined firewall rules per interface or not to apply any firewall rules per interface (default behavior).

***Note*:** `onload_iptables` will import all rules to the identified interface - even rules specified on another interface. To avoid importing rules specified on 'other' interfaces using the --use-extended option.

To View Rules for a Specific Interface

When firewall rules exist for a Solarflare interface, and only while they exist, a directory for the interface will be created in:

```
/proc/driver/sfc_resource
```

Rules for a specific interface will be found in the firewall_rules file. For example:

```
cat /proc/driver/sfc_resource/eth3/firewall_rules
if=eth3 rule=0 protocol=tcp local_ip=0.0.0.0/0.0.0.0
remote_ip=0.0.0.0/0.0.0.0 local_port=5201-5201 remote_port=0-65535
action=DECELERATE
if=eth3 rule=1 protocol=tcp local_ip=0.0.0.0/0.0.0.0
remote_ip=0.0.0.0/0.0.0.0 local_port=5201-5201 remote_port=0-65535
action=DECELERATE
if=eth3 rule=2 protocol=tcp local_ip=0.0.0.0/0.0.0.0
remote_ip=0.0.0.0/0.0.0.0 local_port=5201-5201 remote_port=72-72
action=DECELERATE
if=eth3 rule=3 protocol=tcp local_ip=0.0.0.0/0.0.0.0
remote_ip=0.0.0.0/0.0.0.0 local_port=80-88 remote_port=0-65535
action=DECELERATE
```

To Add a Rule for a Selected Interface

```
echo "rule=4 if=eth3 action=ACCEPT protocol=udp local_port=7330-7340" \
     > /proc/driver/sfc_resource/firewall_add
```

Rules can be inserted into any position in the table and existing rule numbers will be adjusted to accommodate new rules. If a rule number is not specified the rule will be appended to the existing rule list.

Note: Errors resulting from the add/delete commands will be displayed in dmesg.

To Delete a Rule from a Selected Interface

To delete a single rule:

```
# echo "if=eth3 rule=2" > /proc/driver/sfc_resource/firewall_del
```

To delete all rules:

```
echo "eth2 all" > /proc/driver/sfc_resource/firewall_del
```

When the last rule for an interface has been deleted the interface `firewall_rules` file is removed from `/proc/driver/sfc_resource`. The interface directory will be removed only when completely empty.

Error Checking

The onload_iptables command does not log errors to stdout. Errors arising from add or delete commands will logged in dmesg.

Interface and Port

Onload firewall rules are bound to an interface and not to a physical adapter port. It is possible to create rules for an interface in a configured/down state.

Virtual/Bonded Interface

On virtual or bonded interfaces firewall rules are only applied and enforced on the 'real' interface.

Error Messages

Error messages relating to onload_iptables operations will appear in dmesg.

Table 62: **Error messages for onload_iptables**

Error Message	Description
`Internal error`	Internal condition - should not happen.
`Unsupported rule`	Internal condition - should not happen.
`Out of memory allocating new rule`	Memory allocation error.
`Seen multiple rule numbers`	Only a single rule number can be specified when adding/ deleting rules.
`Seen multiple interfaces`	Only a single interface can be specified when adding/ deleting rules.
`Unable to understand action`	The action specified when adding a rule is not supported. ***Note*:** There should be no spaces, like this: action=ACCELERATE.
`Unable to understand protocol`	Non-supported protocol.
`Unable to understand remainder of the rule`	Non-supported parameters/syntax.
`Failed to understand interface`	The interface does not exist. Rules can be added to an interface that does not yet exist, but cannot be deleted from an non-existent interface.
`Failed to remove rule`	The rule does not exist.
`Error removing table`	Internal condition - should not happen.

Table 62: **Error messages for onload_iptables** *(cont'd)*

Error Message	Description
`Invalid local_ip rule`	Invalid address/mask format. Supported formats: a.b.c.d a.b.c.d/n a.b.c.d/e.f.g.h where a.b.c.d.e.f.g.h are decimal range 0-255, n = decimal range 0-32.
`Invalid remote_ip rule`	Invalid address/mask format.
`Invalid rule`	A rule must identify at least an interface, a protocol, an action and at least one match criteria.
`Invalid mac`	Invalid mac address/mask format. Supported formats: xx:xx:xx:xx:xx:xx xx:xx:xx:xx:xx:xx/xx:xx:xx:xx:xx:xx where x is a hex digit.

Note: A Linux limitation applicable to the `/proc/` filesystem restricts a write operation to 1024 bytes. When writing to `/proc/driver/sfc_resource/firewall_[add|del]` files the user is advised to flush the write between lines which exceed the 1024 byte limit.

eflatency

The OpenOnload distribution includes the command line eflatency test application to measure latency of the ef_vi layer 2 API.

eflatency is a single thread ping/pong application. When all iterations are complete the client side will display the round-trip time.

eflatency determines the lowest latency mode that it is possible to use, from the following:

- TX alternatives
- PIO
- DMA.

By default, eflatency sends 10000 warm-up packets to fill caches and stabilize the system, before measuring statistics over 100000 iterations of packets with no payload. Payload size and numbers of iterations can be configured.

With the Onload distribution installed, eflatency will be present in the following directory:

```
~/onload-<version>/build/gnu_x86_64/tests/ef_vi
```

eflatency

```
./eflatency -help
usage:
  eflatency [options] <ping|pong> <interface>
options:
  -n <iterations>          - set number of iterations
  -s <message-size>        - set udp payload size
  -w <iterations>          - set number of warmup iterations
```

Table 63: **eflatency Parameters**

Parameter	Description
`interface`	the local interface to use, for example eth2

To Run eflatency

The eflatency must be started on the server (pong side) before the client (ping side) is run. Command line examples are shown below.

1. On the server side (server1)

```
taskset -c <M> ./eflatency -s 28 pong eth<N>
# ef_vi_version_str: <onload version>
# udp payload len: 28
# iterations: 100000
# warmups: 10000
# frame len: 70
# mode: Alternatives
```

   where:

   - `<M>` is the CPU core
   - `<N>` is the Solarflare adapter interface.

2. On the client side (server2)

```
taskset -c <M> ./eflatency -s 28 ping eth<N>
# ef_vi_version_str: <onload version>
# udp payload len: 28
# iterations: 100000
# warmups: 10000
# frame len: 70
# mode: Alternatives
mean round-trip time: <n.nnn> usec
```

   where:

   - `<M>` is the CPU core
   - `<N>` is the Solarflare adapter interface
   - `<n.nnn>` is the reported mean round-trip time for a 28 byte payload.

Appendix K

Management Information Base

The Onload Management Information Base utility, `onload_mibdump` introduced in OpenOnload 201710, provides state information from the onload control plane MIB tables.

In previous versions of Onload this information was provided via the tables in:

```
/proc/driver/onload_cplane/mib-*
```

Note: This utility is designed primarily to aid Solarflare support when investigating Onload support issues.

Host

When the 'onload_mibdump all' command is run in the host environment, tables are generated for all onload_cp_server objects visible from all namespaces and all containers.

Container

When the 'onload_mibdump all' command is run within a container, tables are generated only for onload_cp_server objects visible within the container namespace.

Namespaces

When the 'onload_mibdump all' command is run within a specific namespace, tables are generated only for onload_cp_server objects visible within the namespace.

List Available Options

```
# onload_mibdump
onload_mibdump: No tables specified.
onload_mibdump:
onload_mibdump: usage:
onload_mibdump:   onload_mibdump [options] [table...]
onload_mibdump:
onload_mibdump: options:
onload_mibdump:   -a --all                  -- Dump all visible control
planes
onload_mibdump:   -n --namespace            -- Dump the control plane for
the specified namespace
onload_mibdump:
onload_mibdump: Options can also be given with the environment variable
CI_OPTS
onload_mibdump:
onload_mibdump: Available tables are:
onload_mibdump:   'usage' - amount of used and free space in each table
onload_mibdump:   'version' - MIB table versions
onload_mibdump:   'hwport' - mapping from hwports to interfaces
onload_mibdump:   'llap' - status of all known interfaces
onload_mibdump:   'ipif' - local IP address configuration
onload_mibdump:   'ip6if' - local IPv6 address configuration
onload_mibdump:   'fwd' - routing table
onload_mibdump:   'stats' - statistics
onload_mibdump:   'internal' - all the internal state
onload_mibdump:   'int_base' - base of the internal state
onload_mibdump:   'int_dst' - destination prefixes from the internal state
onload_mibdump:   'int_src' - source prefixes from the internal state
onload_mibdump:   'int_dst6' - IPv6 destination prefixes from the internal
state
onload_mibdump:   'int_src6' - IPv6 source prefixes from the internal state
onload_mibdump:   'int_llap' - llap private of the internal state
onload_mibdump:   'int_team' - team table of the internal state
onload_mibdump:   'int_mac' - mac IP table of the internal state
onload_mibdump:   'int_mac6' - mac IPv6 table of the internal state
onload_mibdump:   'int_fwd' - fwd private of the internal state
onload_mibdump:   'int_stats' - stats of the internal state
onload_mibdump:   'int_stat_doc' - documentation for internal statistic
counters
onload_mibdump:
onload_mibdump: Or use 'all' to dump all tables.
```

Tables

Also refer to User-space Control Plane Server and Changing Onload Control Plane Table Sizes.

Usage

Identifies the amount of used/max entries in each table.

```
# onload_mibdump -a usage
Control plane state for server 21745:

<Version info - see Version>

Table space usage:

hwport: 2/8
llap:   7/32
ipif:   5/256
ip6if:  0/0     FULL
fwd:    2/1024
```

Version numbers are for internal use only. To increase the size of cplane tables, refer to Changing Onload Control Plane Table Sizes.

Version

Displays MIB tables version numbers. These values are for internal use only.

```
# onload_mibdump -a version
Control plane state for server 21745:

Table version number: 28
LLAP version number: 9
Dump version number: 1903528
Idle version number: 2679954
OOF version number: 26
```

hwport

Displays mappings of hardware ports to interfaces.

```
# onload_mibdump -a hwport
Control plane state for server 21745:

<Version info - see Version>

Hwport table (licensed f, unlicensed 0):

hwport[000]:
        flags LICENSED-ONLOAD (84)
        oo_vi_flags_mask=ffffffff efhw_flags_extra=00000000 pio_len_shift=0
        ctpio_start_offset=00000000
hwport[001]:
        flags LICENSED-ONLOAD (84)
        oo_vi_flags_mask=ffffffff efhw_flags_extra=00000000 pio_len_shift=0
        ctpio_start_offset=00000000
hwport[002]:
        flags LICENSED-ONLOAD LICENSED-TCP-DIRECT (8c)
        oo_vi_flags_mask=ffffffff efhw_flags_extra=00000000 pio_len_shift=0
```

```
        ctpio_start_offset=00000000
hwport[003]:
        flags LICENSED-ONLOAD LICENSED-TCP-DIRECT (8c)
        oo_vi_flags_mask=ffffffff efhw_flags_extra=00000000 pio_len_shift=0
        ctpio_start_offset=00000000
```

- flags

  Onload features with an activation key on the adapter

- oo_vi_flags_mask

  flag definitions in `/src/include/ci/efhw/common.h`

- efhw_flags_extra

  flag definitions in `/src/include/ci/efhw/common.h`

- pio_len_shift

  internal PIO value

- ctpio_start_offset

  Internal CTPIO value.

llap

The llap command provides some of the data available from the `ip link show` command.

Link state and link characteristics data is displayed for all layer 2 interfaces which have a loaded driver. Interfaces are sequentially numbered [nnn], this value is dynamically calculated and should not be used to refer to the interface.

```
# onload_mibdump -a llap
Control plane state for server 21745:

<Version info - see Version>

LLAP table:

llap[000]: enp4s0f1 (650) UP mtu 1500 arp_base 30000ms
         TX hwports 1
         RX hwports 1
         mac 00:0f:53:01:45:49
llap[001]: enp4s0f0 (649) UP mtu 1500 arp_base 30000ms
         TX hwports 0
         RX hwports 0
         mac 00:0f:53:01:45:48
llap[002]:     eth0 (652) UP mtu 1500 arp_base 30000ms
         TX hwports 2
         RX hwports 2
         mac 00:0f:53:21:9b:b0
llap[003]: enp5s0f1 (653) UP mtu 1500 arp_base 30000ms
         TX hwports 3
         RX hwports 3
         mac 00:0f:53:21:9b:b1
llap[004]:       lo (1) UP mtu 65535 arp_base 30000ms
```

```
          encap LOOP
          no TX hwports
          no RX hwports
llap[005]:      eno1 (2) UP mtu 1500 arp_base 30000ms
          no TX hwports
          no RX hwports
llap[006]:      eno2 (3) UP mtu 1500 arp_base 30000ms
          no TX hwports
          no RX hwports
llap[007]:      eno3 (4) UP mtu 1500 arp_base 30000ms
          no TX hwports
          no RX hwports
llap[008]:      eno4 (5) UP mtu 1500 arp_base 30000ms
          no TX hwports
          no RX hwports
```

- arp_base

  is the `/proc/sys/net/ipv4/neigh/<iface>/base_reachable_time_ms` (milliseconds).

- no TX/RX hwports

  is a mask mapping kernel interfaces to Onload interfaces.

ipif

Displays IP configuration for local interfaces.

```
# onload_mibdump ipif

<Version info - see Version>

IPIF table:

ipif[000]:        lo (1) 127.0.0.1/8 bcast 0.0.0.0 scope host
ipif[001]:      eno1 (2) 10.17.130.253/21 bcast 10.17.135.255 scope univ
ipif[002]:    virbr0 (10) 192.168.122.1/24 bcast 192.168.122.255 scope univ
ipif[003]: enp4s0f0 (12) 172.16.130.253/21 bcast 172.16.135.255 scope univ
ipif[004]: enp4s0f1 (13) 172.16.138.253/21 bcast 172.16.143.255 scope univ
ipif[005]: enp5s0f0 (14) 172.16.154.253/21 bcast 172.16.159.255 scope univ
ipif[006]: enp5s0f1 (15) 172.16.162.253/21 bcast 172.16.167.255 scope univ
```

fwd

Will identify and list all interface routes and metrics assigned to each interface. The fwd option is the equivalent of using the' ip route show' command.

```
# onload_mibdump fwd

<Version info - see Version>

FWD table:

Source prefix length in use: 5 32
Destination prefix length in use: 32
```

```
fwd[001]: from 172.16.154.253/32 to 172.16.154.252/32 via any tos 0
        from 172.16.154.253 via 172.16.154.252 enp5s0f0 (14)
        mtu 1500 type NORMAL arp valid
        hwports 4 from 00:0F:53:25:3A:20 to 00:0F:53:21:9B:B0
        last used: 2682 ms ago
        in use: 1       verinfo: 1-2
fwd[003]: from 10.17.130.253/32 to 10.17.135.251/32 via any tos 0
        from 10.17.130.253 via 10.17.135.251 (0)
        mtu 0 type ALIEN arp invalid
        hwports 0 from 00:00:00:00:00:00 to 00:00:00:00:00:00
        last used: 66111 ms ago
        in use: 1       verinfo: 3-e
fwd[218]: from 0.0.0.0/5 to 172.16.154.252/32 via any tos 0
        from 172.16.154.253 via 172.16.154.252 enp5s0f0 (14)
        mtu 1500 type NORMAL arp valid
        hwports 4 from 00:0F:53:25:3A:20 to 00:0F:53:21:9B:B0
        last used: 326092 ms ago
        in use: 1       verinfo: da-2
```

stats

These stats are counts of instances when Onload requests routing resolution from the kernel.

```
# onload_mibdump -a stats
Control plane state for server 21745:

<Version info - see Version>

Control Plane statistics:

Route requests (non-waiting):   2
Route requests (waiting):       15
Route requests queue depth:     0
Filter engine requests (non-waiting):   28
ARP confirmations (tried):      29
ARP confirmations (successful): 29
Dropped IP packets routed via OS:       0
```

internal

```
# onload_mibdump -a internal
Control plane state for server 21745:

<Version info - see Version>

Requesting dump of internal state...
cp_session_print_state(0x0):
  flags=1043 license_threads=0
  state=0 prev_state=0 seen[0]=ffffffffffffff
  user_hz=10 khz=3695993

Destinations in routes and rules:
  allocated/used/sorted: 32 / 20 / 20
  [0] 192.168.122.255/32
  [1] 192.168.122.1/32
  [2] 192.168.122.0/32
  [3] 172.16.143.255/32
```

```
  [4] 172.16.137.206/32
  [5] 172.16.136.0/32
  [6] 172.16.135.255/32
  [7] 172.16.129.206/32
  [8] 172.16.128.0/32
  [9] 127.255.255.255/32
  [10] 127.0.0.1/32
  [11] 127.0.0.0/32
  [12] 10.17.135.255/32
  [13] 10.17.129.206/32
  [14] 10.17.128.0/32
  [15] 192.168.122.0/24
  [16] 172.16.136.0/21
  [17] 172.16.128.0/21
  [18] 10.17.128.0/21
  [19] 127.0.0.0/8
Source rules:
  allocated/used/sorted: 8 / 5 / 5
  [0] 192.168.122.1/32
  [1] 172.16.137.206/32
  [2] 172.16.129.206/32
  [3] 127.0.0.1/32
  [4] 10.17.129.206/32
IPv6 destinations in routes and rules:
IPv6 support disabled
IPv6 source rules:
IPv6 support disabled
cp_llap_print:
llap[000]: LOOP  arp_base 30000ms

llap[001]:  arp_base 30000ms

llap[002]:  arp_base 30000ms

llap[003]:  arp_base 30000ms

llap[004]:  arp_base 30000ms

llap[005]:  arp_base 30000ms

llap[006]:  arp_base 30000ms

cp_team_print:
cp_mac_print:
mac[357]: if 2 ip 10.17.135.251 mac 00:50:56:9C:F7:55 reachable (1 refs)

        to be re-confirmed after 0 msec

mac[383]: if 1 ip 127.0.0.1 mac 00:00:00:00:00:00 noarp (1 refs)

mac[597]: if 1 ip 10.17.129.206 mac 00:00:00:00:00:00 noarp (1 refs)

mac[610]: if 2 ip 10.17.135.252 mac 00:50:56:9C:3F:FF stale (1 refs)

mac[760]: if 10 ip 172.16.129.207 mac 00:0F:53:65:17:40 reachable (1 refs)

        to be re-confirmed after 0 msec

mac[855]: if 2 ip 10.17.129.207 mac 50:9A:4C:6D:49:F0 stale (1 refs)

mac[871]: if 2 ip 10.17.128.254 mac 70:CA:9B:52:CC:4C stale (1 refs)

cp_mac6_print:
```

```
IPv6 support disabled
cp_fwd_print:
[769]: macid=760 used 13828 ms ago
[819]: macid=357 used 20828 ms ago
Statistics:
nlmsg_error.link_nodev: 0
nlmsg_error.link: 0
nlmsg_error.addr: 0
nlmsg_error.neigh: 0
nlmsg_error.route: 0
nlmsg_error.rule: 0
nlmsg_error.other: 0
fwd.collision: 0
fwd.hash_loop: 0
fwd.full: 0
fwd.req_complete: 15
fwd.nlmsg_mismatch: 0
fwd.error_mismatch: 0
mac.collision: 0
mac.hash_loop: 0
mac.full: 0
llap.unsupported_ifi_type: 0
llap.unsupported_info_kind: 1903626
llap.unsupported_vlan: 0
llap.full: 0
ipif.full: 0
notify.llap_mod: 201
notify.llap_update_filters: 58
notify.ip_mod: 119
notify.ready: 81
license.onload: 2
license.scaleout: 0
license.non_onload: 4
license.tcp_direct: 2
license.siena: 0
license.ef10: 0
license.medford: 2
license.noname: 0
license.rename: 0
license.too_many_renames: 0
license.sfc_driver: 2
license.non_sfc_driver: 4
Succeeded.
```

The internal command outputs onload_cplane_server internal statistics.

This command is for diagnostic purposes and might be requested by Solarflare support.

For documentation of the internal statistic counters, type:

```
onload_mibdump int_stat_doc
```

all

Running the following command will generate all MIB tables:

```
# onload_mibdump all
```

Appendix L

X2 Low Latency Quickstart

This appendix demonstrates how to achieve very low latency coupled with minimum jitter on a system fitted with an X2 series network adapter and using the OpenOnload kernel-bypass network acceleration middleware. These techniques also apply to other supported network adapters that use the `sfc` network driver, such as 8000 series adapters.

The procedure will focus on the performance of the network adapter for TCP and UDP applications running on Linux, using the AMD supplied open source sfnettest network benchmark test tools, and also the industry-standard Netperf network benchmark application.

The results of these tests can be found in Latency Test Results, and Latency against Payload.

Note: Please read the supplied `ONLOAD_LICENSE` file regarding the disclosure of performance test results.

Software Installation

Before running these benchmark tests ensure that correct driver and firmware versions are installed. For example, for the reference system described later in this chapter:

```
[root@server-N]# ethtool -i <interface>
driver: sfc
version: 4.15.0.1012
firmware-version: 7.5.0.1022 rx1 tx1
```

Firmware Variant

On SFN8000 and X2 series adapters, the adapter should use the *ultra-low-latency* firmware variant – as indicated by the presence of *rx1 tx1* as shown above. Firmware variants are selected with the `sfboot` utility from the Linux Utilities package (SF-107601-LS).

Onload

Before Onload network and kernel drivers can be built and installed the system must support a build environment capable of compiling kernel modules. Refer to Appendix C: Build Dependencies for more details.

1. Download the `onload-<version>.tgz` file from the NIC Software and Drivers web page.

2. Unpack the tar file using the tar command:

```
# tar -zxvf onload-<version>.tgz
```

3. Run the `onload_install` command from the `onload-<version>/scripts` subdirectory:

```
# ./onload-<version>/scripts/onload_install
```

Refer to Driver Loading - NUMA Node to ensure that drivers are affinitized to a core on the correct NUMA node.

Netperf

Netperf is available as a package for most OS distributions.

Netperf can also be downloaded from https://github.com/HewlettPackard/netperf

- Unpack the compressed zip file using the unzip command:

```
# unzip netperf-master.zip
```

- Refer to the `INSTALL` file within the distribution for instructions.

Following installation the `netperf` and `netserver` applications are typically located in the `/usr/local/bin` subdirectory.

Sfnettest

Download the `sfnettest` source from https://github.com/Xilinx-CNS/cns-sfnettest.

Unpack the downloaded source if necessary, for example:

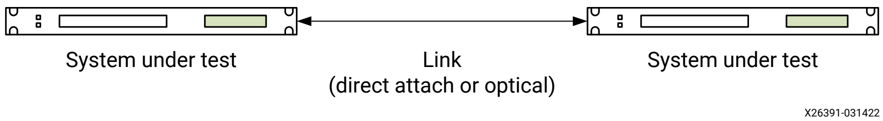
```
# unzip cns-sfnettest-master.tgz
```

Run the `make` utility from the `src` subdirectory to build the `sfnt-pingpong` and other test applications.

Test Setup

The following figure identifies the required physical configuration of two servers equipped with supported network adapters connected back-to-back.

Figure 42: **Test setup**



- Two servers are equipped with supported network adapters and connected with a single cable between the supported interfaces.
- The supported interfaces are configured with an IP address so that traffic can pass between them. Use `ping` to verify connection.
- Onload, sfnettest and netperf are installed on both machines.

If required, tests can be repeated with a switch on the link to measure the additional latency delta using a particular switch.

BIOS Settings

Make the following BIOS settings on both machines:

1. Enable Turbo Boost (sometimes called Turbo Mode).
2. Enable CStates.
3. Disable any of the following settings that are present:
   - Virtualization Technology (also called VT-d/VT-x)
   - IOMMU.

   These are similar in their effect, and typically only one will be present.

Pre-Test Configuration

The following configuration options are applicable to RHEL7 systems.

First, set some configuration options that decrease latency for Onload acceleration technologies. On both machines:

1. Add the following options to the kernel configuration line in `/boot/grub/grub.conf`:

```
isolcpus=<comma separated cpu list> nohz=off iommu=off intel_iommu=off
mce=ignore_ce nmi_watchdog=0
```

2. Stop the following services on the server:

```
systemctl stop cpupower
systemctl stop cpuspeed
systemctl stop cpufreqd
systemctl stop powerd
systemctl stop irqbalance
systemctl stop firewalld
```

3. Allocate huge pages. For example, to configure 1024 huge pages:

```
# sysctl -w vm.nr_hugepages=1024
```

   To make this change persistent, update `/etc/sysctl.conf`. For example:

```
# echo "vm.nr_hugepages = 1024" >> /etc/sysctl.conf
```

   For more information refer to Allocating Huge Pages.

4. Consider the selection of the NUMA node, as this affects latency on a NUMA-aware system. Refer to Onload Deployment on NUMA Systems.

5. Disable interrupt moderation.

```
# ethtool -C <interface> rx-usecs 0 adaptive-rx off
```

6. Enable PIO in the Onload environment.

   EF_PIO=1

Now perform the following configuration to improve latency without Onload.

Note: These configuration changes have minimal effect on the performance of Onload.

1. Set interrupt affinity such that interrupts and the application are running on different CPU cores but on the same processor package.

   a. Use the following command to identify the interrupts used by the receive queues created for an interface:

```
# cat /proc/interrupts | grep <interface>
```

   The output lists the IRQs. For example:

```
34:    ...   PCI-MSI-edge      p2p1-0
35:    ...   PCI-MSI-edge      p2p1-1
36:    ...   PCI-MSI-edge      p2p1-2
37:    ...   PCI-MSI-edge      p2p1-3
38:    ...   PCI-MSI-edge      p2p1-ptp
```

   b. Direct the listed IRQs to unused CPU cores that are on the same processor package as the application. For example, to direct IRQs 34-38 to CPU core 2 (where cores are numbered from 0 upwards), using `bash`:

```
# for irq in {34..38}
> do
> echo 04 > /proc/irq/$irq/smp_affinity
> done
```

2. Set an appropriate tuned profile:
   - The tuned network-latency profile produces better kernel latency results:

     ```
     # tuned-adm profile network-latency
     ```

   - If available, the cpu-partitioning profile includes the network-latency profile, but also makes it easy to isolate cores that can be dedicated to interrupt handling or to an application. For example, to isolate cores 1-3:

     ```
     # echo "isolated_cores=1-3" \
          > /etc/tuned/cpu-partitioning-variables.conf
     # tuned-adm profile cpu-partitioning
     ```

3. Enable the kernel "busy poll" feature to disable interrupts and allow polling of the socket receive queue. The following values are recommended:

   ```
   # sysctl net.core.busy_poll=50 && sysctl net.core.busy_read=50
   ```

Reference System Specification

The following measurements were recorded on Intel® Kaby Lake servers. The specification of the test systems is as follows:

- DELL PowerEdge R230 servers equipped with Intel Xeon CPU E3-1240 v6 @ 3.70 GHz, 16 GB RAM.
- BIOS configured as specified in BIOS Settings.
- Solarflare X2522-25G NIC (driver and firmware – see Software Installation).
- Direct attach cable linking the NICs:
  - 10 Gb cable for measurements at 10 Gb
  - 25 Gb cable for measurements at 25 Gb
- Red Hat Enterprise Linux 7.4 (x86_64 kernel, version 3.10.0-693.5.2.el7.x86_64).
- OS configured as specified in Pre-Test Configuration

  The `tuned` cpu-partitioning profile has been enabled, configured to isolate all cores except for core 0, to reduce jitter and remove outliers.
- OpenOnload distribution: openonload-201811.
- sfnettest version 1.5.0.
- netperf version 2.7.1.

It is expected that similar results will be achieved on any Intel based, PCIe Gen 3 server or compatible system.

Latency Tests

This section describes various latency tests.

Most of these tests use cut through PIO (CTPIO). This is a feature introduced in the X2 series of adapters, where packets to be sent are streamed directly over the PCIe bus to the network port, bypassing the main adapter transmit datapath. For more information refer to CTPIO.

The tests use different CTPIO modes, depending on the link speed:

- 10 Gb tests use cut-through CTPIO. This is supported only at 10 Gb.
- 25 Gb tests use variants of store and forward CTPIO.

Note: These different CTPIO modes require changes to the command lines, noted below.

The command lines given below use the `taskset` command to run the tests on core 1. Change this as necessary, to use an appropriate isolated core on your test system.

Layer 2 ef_vi Latency

ef_vi is a network layer 2 API.

ef_vi test applications can be found in:

```
onload-<version>/build/gnu_x86_64/tests/ef_vi
```

Run the `eflatency` UDP test application on both systems:

```
[sys-1]# taskset -c 1 eflatency <mode> -s <payload> pong <interface>
```

```
[sys-2]# taskset -c 1 eflatency <mode> -s <payload> ping <interface>
```

where:

- `<mode>` is `-p` only for 25 Gb, to force store and forward (no-poison) CTPIO
- `<payload>` is the payload size, in bytes
- `<interface>` is the interface to use.

The output gives various diagnostic information (ef_vi version, payload and frame length, number of iterations and warmups, and mode). It also identifies mean RTT, which is halved to give the mean ½ RTT latency.

Note: Appendix J: eflatency describes the `eflatency` application, command line options and provides example command lines.

TCPDirect Latency

TCPDirect is a feature available for the SFN8000, X2 series and X3 series adapters. SFN8000 and X2 series adapters must have Onload and TCPDirect activation keys installed.

TCPDirect test applications can be found in:

```
onload-<version>/build/gnu_x86_64/tests/zf_apps/static
```

Run the `zfudppingpong` application on both systems:

```
[sys-1]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zfudppingpong -s <payload> pong <sys-1_ip>:20000 <sys-2_ip>:20000
```

```
[sys-2]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zfudppingpong -s <payload> ping <sys-2_ip>:20000 <sys-1_ip>:20000
```

or run the `zftcppingpong` application on both systems:

```
[sys-1]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zftcppingpong -s <payload> pong <sys-1_ip>:20000
```

```
[sys-2]# ZF_ATTR="interface=<interface>;ctpio_mode=<mode>" taskset -c 1 \
   zftcppingpong -s <payload> ping <sys-1_ip>:20000
```

where:

- `<interface>` is the interface to use
- `<mode>` is the CTPIO mode to use, which is `ct` for 10 Gb, or `sf` for 25 Gb
- `<payload>` is the payload size, in bytes
- `<sys-1_ip>` is the IP address of `sys-1`
- `<sys-2_ip>` is the IP address of `sys-2`.

The output identifies mean RTT, which is halved to give the mean ½ RTT latency.

Onload Latency with netperf

You can measure Onload latency with standard tools. This test identifies how to use `netperf`.

Run the `netserver` application on system-1:

```
[sys-1]# pkill -f netserver
[sys-1]# onload --profile=<profile> taskset -c 1 netserver
```

and the `netperf` application on system-2:

```
[sys-2]# onload --profile=<profile> taskset -c 1 \
    netperf -t <test> -H <sys-1_ip> -l 10 -- -r <payload>
```

where:

- `<profile>` is `latency-best` for 10 Gb (which uses cut-through CTPIO), or `latency` for 25 Gb (which uses store and forward (no-poison) CTPIO)
- `<test>` is `UDP_RR` or `TCP_RR`, as appropriate
- `<payload>` is the payload size, in bytes
- `<sys-1_ip>` is the IP address of `sys-1`.

The output identifies the transaction rate per second, from which:

mean ½ RTT = (1 / transaction rate) / 2

Onload Latency with sfnt-pingpong

You can also measure Onload latency with the `sfnt-pingpong` application.

Note: The latencies measured with `sfnt-pingpong` are almost identical to the latencies measured with `netperf` in Onload Latency with netperf.

Run the `sfnt-pingpong` application on both systems:

```
[sys-1]# onload --profile=<profile> taskset -c 1 sfnt-pingpong
```

```
[sys-2]# onload --profile=<profile> taskset -c 1 sfnt-pingpong \
    --affinity "1;1" <protocol> <sys-1_ip>
```

where:

- `<profile>` is `latency-best` for 10 Gb (which uses cut-through CTPIO), or `latency` for 25 Gb (which uses store and forward (no-poison) CTPIO)
- `<protocol>` is `udp` or `tcp`, as appropriate
- `<sys-1_ip>` is the IP address of `sys-1`.

The output identifies mean, minimum, median and maximum (nanosecond) ½ RTT latency for increasing packet sizes, including the 99% percentile and standard deviation for these results.

Latency without CTPIO

The previous tests all use CTPIO. This test shows the result of disabling CTPIO, using UDP traffic with TCPDirect.

TCPDirect test applications can be found in:

```
onload-<version>/build/gnu_x86_64/tests/zf_apps/static
```

Run the `zfudppingpong` application on both systems:

```
[sys-1]# ZF_ATTR="interface=<interface>;ctpio=0" taskset -c 1 \
    zfudppingpong -s <payload> pong <sys-1_ip>:20000 <sys-2_ip>:20000
```

```
[sys-2]# ZF_ATTR="interface=<interface>;ctpio=0" taskset -c 1 \
    zfudppingpong -s <payload> ping <sys-2_ip>:20000 <sys-1_ip>:20000
```

where:

- `<interface>` is the interface to use
- `<payload>` is the payload size, in bytes
- `<sys-1_ip>` is the IP address of `sys-1`
- `<sys-2_ip>` is the IP address of `sys-2`.

The output identifies mean RTT, which is halved to give the mean ½ RTT latency.

Note: This can be compared with the result of the UDP test in TCPDirect Latency, which is identical except that CTPIO is enabled.

Kernel Latency

The benchmark performance tests can be run without Onload using the regular kernel network drivers. To do this remove the `onload --profile=…` part from the command line.

Run the `sfnt-pingpong` application on both systems:

```
[sys-1]# taskset -c 1 sfnt-pingpong
```

```
[sys-2]# taskset -c 1 sfnt-pingpong --affinity "1;1" \
    <connect> <protocol> <sys-1_ip>
```

where:

- `<connect>` is `--connect` only for UDP, to use `connect()`
- `<protocol>` is `udp` or `tcp`, as appropriate
- `<sys-1_ip>` is the IP address of `sys-1`.

The output identifies mean, minimum, median and maximum (nanosecond) ½ RTT latency for increasing packet sizes, including the 99% percentile and standard deviation for these results.

Latency Test Results

The table below shows the results of running the tests described in Latency Tests. The times given are ½ RTT latency for a 32 byte message

Table 64: **½ RTT Latency for a 32 Byte Message**

Acceleration	Protocol	25 Gb	10 Gb	Notes	Description
ef_vi	UDP	750 ns	819 ns	eflatency	Layer 2 ef_vi Latency
TCPDirect	UDP	783 ns	864 ns	zfudppingpong	TCPDirect Latency
		968 ns	1022 ns	No CTPIO	Latency without CTPIO
	TCP	795 ns	870 ns	zftcppingpong	TCPDirect Latency
Onload	UDP	1034 ns	1107 ns	netperf	Onload Latency with netperf
		1022 ns	1095 ns	sfnt-pingpong	Onload Latency with sfnt-pingpong
	TCP	1032 ns	1119 ns	netperf	Onload Latency with netperf
		1025 ns	1110 ns	sfnt-pingpong	Onload Latency with sfnt-pingpong
Kernel	UDP	2658 ns	2750 ns	sfnt-pingpong	Kernel Latency
	TCP	3124 ns	3257 ns	sfnt-pingpong	Kernel Latency

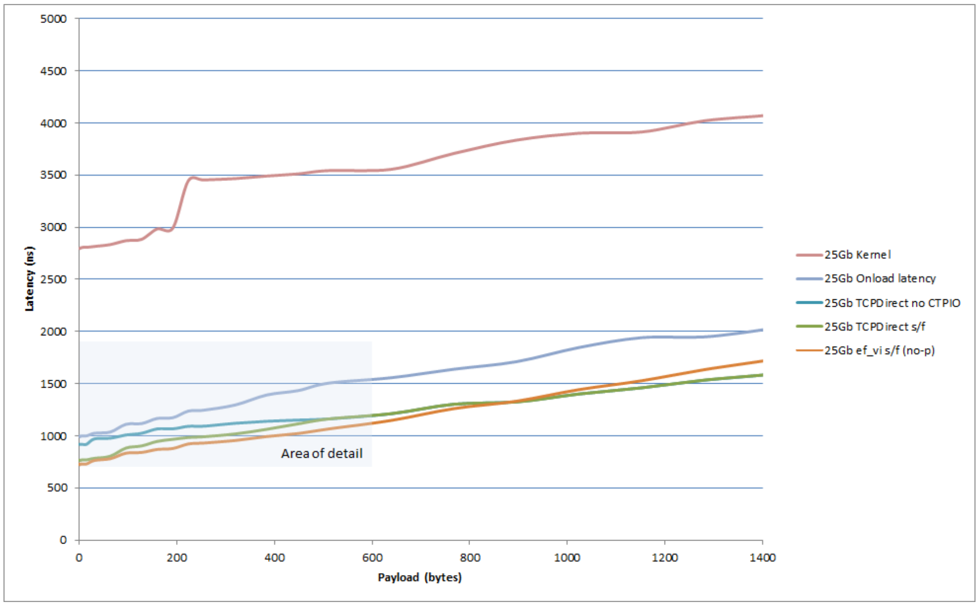
These tests have also been repeated with different payloads, to generate the graphs in Latency against Payload.

Latency against Payload

Latency for UDP Payloads at 25 Gb

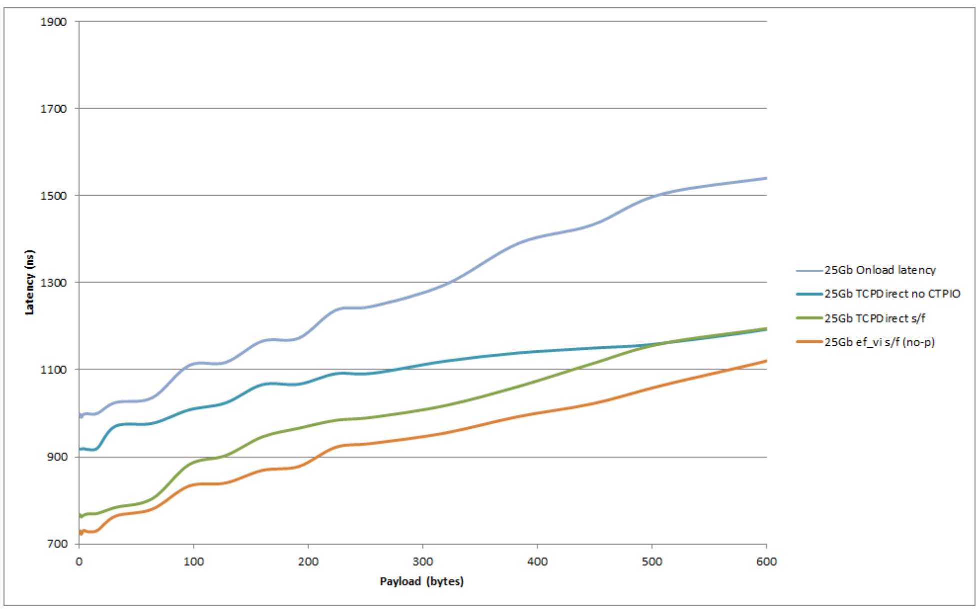
The following figure identifies latency for different UDP payloads, both without Onload, and with different Onload technologies.

Figure 43: **Latency for different UDP payloads at 25 Gb**



The following figure shows a detail of the preceding figure, for smaller payloads with different Onload technologies.

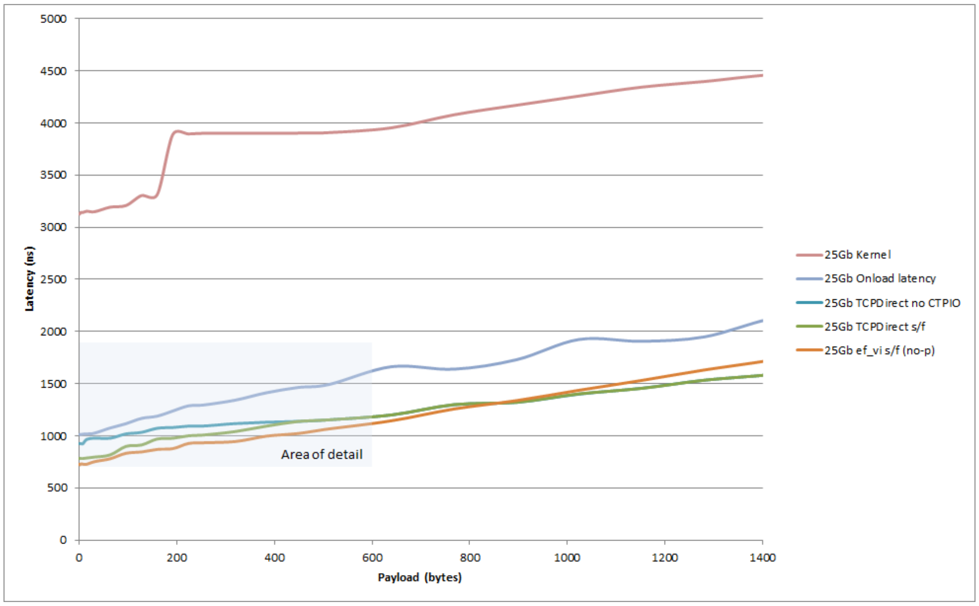
Figure 44: **Detail of latency for different UDP payloads at 25 Gb**



Latency for TCP Payloads at 25 Gb

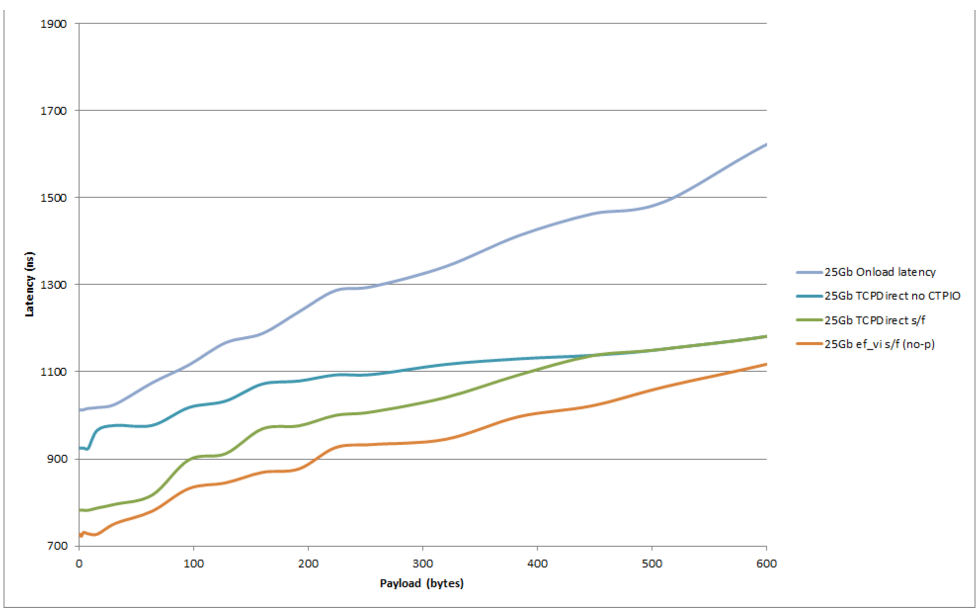
The following figure shows the latency for different TCP payloads, both without Onload, and with different Onload technologies.

Figure 45: **Latency for different TCP payloads at 25 Gb**



The following figure shows a detail of the preceding figure, for smaller payloads with different Onload technologies.

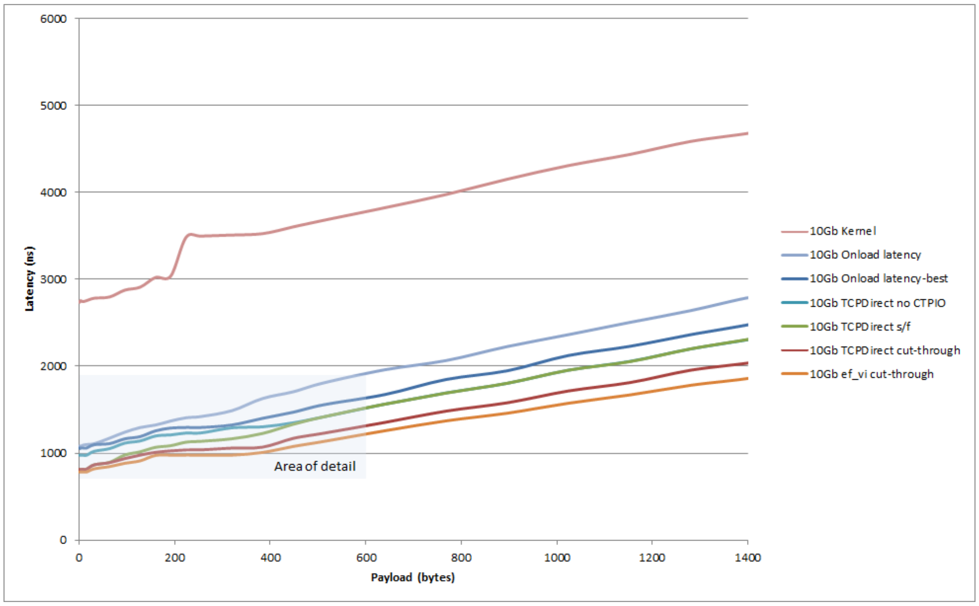
Figure 46: **Detail of latency for different TCP payloads at 25 Gb**



Latency for UDP Payloads at 10 Gb

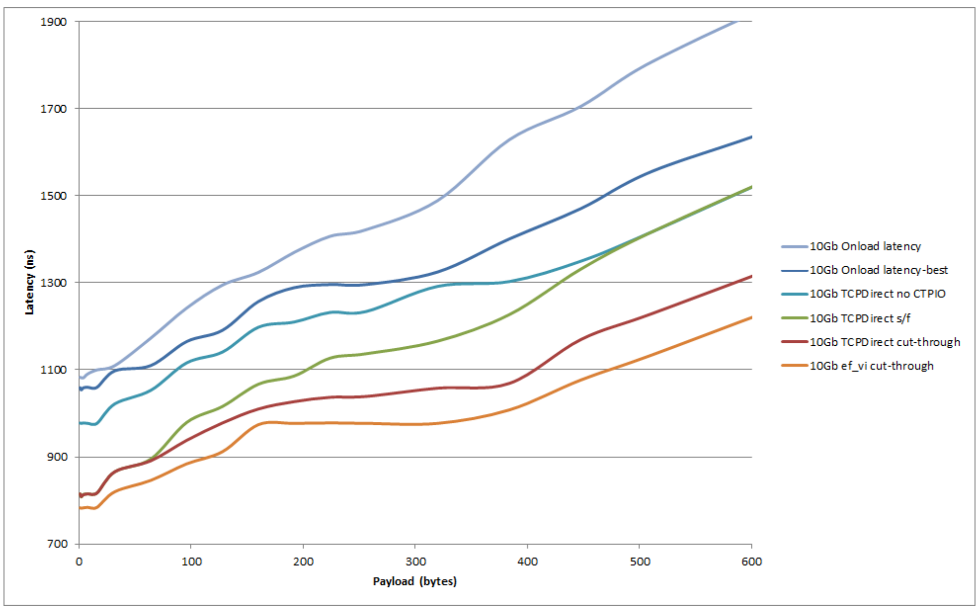
The following figure shows the latency for different UDP payloads, both without Onload, and with different Onload technologies.

Figure 47: **Latency for different UDP payloads at 10 Gb**



The following figure shows a detail of the preceding figure, for smaller payloads with different Onload technologies.

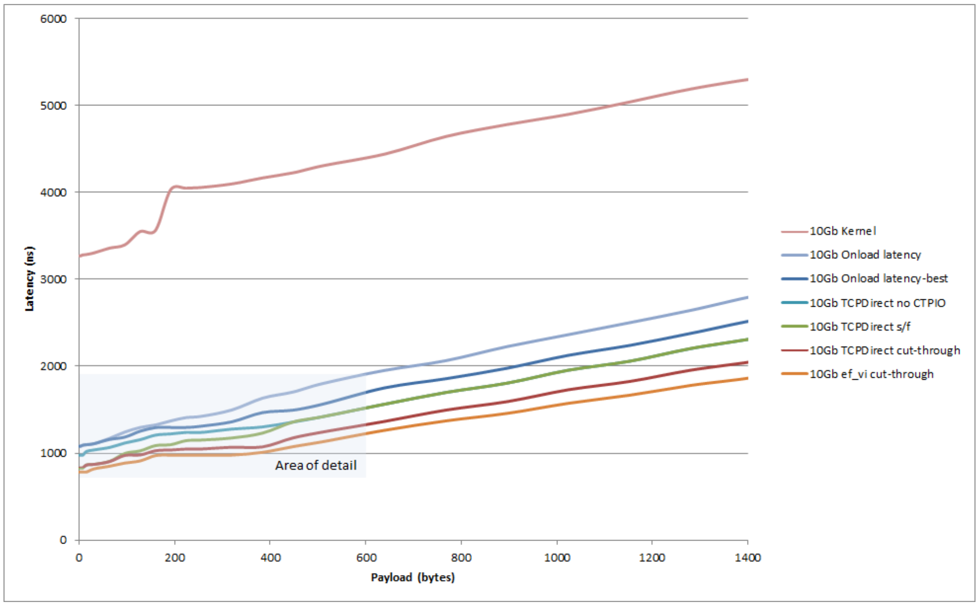
Figure 48: **Detail of latency for different UDP payloads at 10 Gb**



Latency for TCP Payloads at 10 Gb

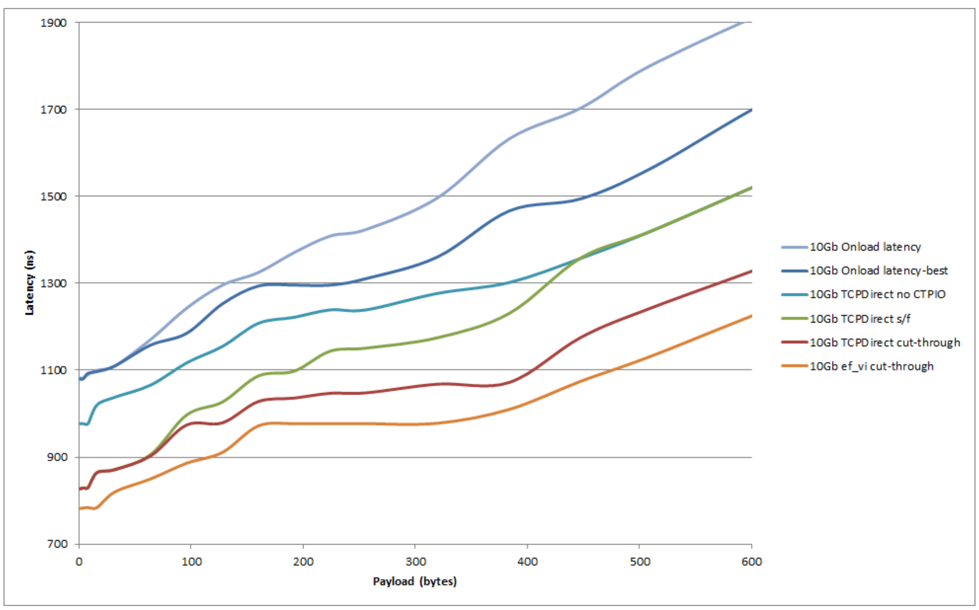
The following figure shows the latency for different TCP payloads, both without Onload, and with different Onload technologies.

Figure 49: **Latency for different TCP payloads at 10 Gb**



The following figure shows a detail of the preceding figure, for smaller payloads with different Onload technologies.

Figure 50: **Detail of latency for different TCP payloads at 10 Gb**



Notes on Latency and Payload Graphs

Note the following about the preceding latency and payload graphs:

- Latency is ordered as follows, starting with the lowest latency:
  1. ef_vi using CTPIO
  2. TCPDirect using cut-through CTPIO

     ***Note*:** Cut-through CTPIO is supported only for 10 Gb.
  3. TCPDirect using store and forward CTPIO
  4. TCPDirect without CTPIO
  5. Onload with the latency-best profile (which uses cut-through CTPIO)

     ***Note*:** Cut-through CTPIO is supported only for 10 Gb.
  6. Onload with the latency profile (which uses store and forward CTPIO, without poisoning).
  7. Kernel.
- The relative latency of CTPIO variants differs by payload:
  - for very small payloads, store and forward CTPIO has similar latency to cut-through CTPIO

    ***Note*:** Cut-through CTPIO is supported only for 10 Gb.
  - for intermediate payloads, the latency of store and forward CTPIO gradually increases relative to cut-through CTPIO, tending towards the latency for no CTPIO
  - for large payloads, store and forward CTPIO has similar latency to no CTPIO.
- The step in the graphs for a payload of 200+ bytes occurs when the packet size reaches the PIO threshold of 256 bytes.

Further Information

For installation of Solarflare adapters and performance tuning of the network driver when not using Onload refer to the *Solarflare Server Adapter User Guide* (SF-103837-CD).

Questions regarding Solarflare products, Onload and this *User Guide* can be emailed to support-nic@amd.com.

Appendix M

X2 Throughput Quickstart

This chapter demonstrates how to achieve efficient packet handling and throughput on a system fitted with an X2 series network adapter and using the OpenOnload kernel-bypass network acceleration middleware. These techniques also apply to other supported network adapters that use the `sfc` network driver, such as 8000 series adapters.

The procedure will focus on the performance of the network adapter for TCP applications running on Linux, using the industry-standard Netperf network benchmark application, and also the Nginx web server.

Note: Please read the supplied `ONLOAD_LICENSE` file regarding the disclosure of performance test results.

Software Installation

Before running these benchmark tests ensure that correct driver and firmware versions are installed. For example, for the reference system described later in this chapter:

```
[root@server-N]# ethtool -i <interface>
driver: sfc
version: 4.15.0.1012
firmware-version: 7.5.0.1002 rx1 tx1
```

Onload

Before Onload network and kernel drivers can be built and installed the system must support a build environment capable of compiling kernel modules. Refer to Appendix C: Build Dependencies for more details.

1. Download the `onload-<version>.tgz` file from the NIC Software and Drivers web page.
2. Unpack the tar file using the tar command:

   ```
   # tar -zxvf onload-<version>.tgz
   ```

3. Run the `onload_install` command from the `onload-<version>/scripts` subdirectory:

   ```
   # ./onload-<version>/scripts/onload_install
   ```

Refer to Driver Loading - NUMA Node to ensure that drivers are affinitized to a core on the correct NUMA node.

Netperf

Netperf is available as a package for most OS distributions.

Netperf can also be downloaded from https://github.com/HewlettPackard/netperf:

- Unpack the compressed zip file using the unzip command:

```
# unzip netperf-master.zip
```

- Refer to the `INSTALL` file within the distribution for instructions.

Following installation the `netperf` and `netserver` applications are typically located in the `/usr/local/bin` subdirectory.

Wrk

Wrk is available as a package for most OS distributions.

Wrk can also be downloaded from https://github.com/wg/wrk.git:

- Refer to the download website for instructions.

Following installation the `wrk` application is typically located in the `/usr/local/bin` subdirectory.

Nginx

Nginx is available as a package for most OS distributions.

Nginx can also be downloaded from https://nginx.org (free version without support) or from https://nginx.com (paid version with support):

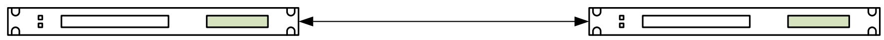
- Refer to the download website for instructions.

Following installation the `nginx` application is typically located in the `/usr/sbin` subdirectory.

Test Setup

The following figure identifies the required physical configuration of two servers equipped with supported network adapters connected back-to-back.

Figure 51: **Test setup**



- Two servers are equipped with supported network adapters and connected with a single cable between the supported interfaces.
- The supported interfaces are configured with an IP address so that traffic can pass between them. Use `ping` to verify connection.
- Onload and netperf are installed on both machines.
- Wrk is installed on one machine, and nginx on the other.

If required, tests can be repeated with a switch on the link to measure the additional latency delta using a particular switch.

Reference System Specification

The following measurements were recorded on Intel® Kaby Lake servers. The specification of the test systems is as follows:

- DELL PowerEdge R230 servers equipped with Intel Xeon CPU E3-1240 v6 @ 3.70 GHz, 16 GB RAM.
- BIOS configured as specified in BIOS Settings.
- Solarflare X2522-25G NIC (driver and firmware – see Software Installation).
- 25 Gb direct attach cable linking the NICs.
- Red Hat Enterprise Linux 7.4 (x86_64 kernel, version 3.10.0-693.5.2.el7.x86_64).
- OS configured as specified in Pre-Test Configuration

  The `tuned` cpu-partitioning profile has been enabled, configured to isolate all cores except for core 0, to reduce jitter and remove outliers.
- OpenOnload distribution: openonload-201811.
- netperf version 2.7.1.
- wrk version 4.1.0.
- nginx version 1.15.0.

It is expected that similar results will be achieved on any Intel based, PCIe Gen 3 server or compatible system.

Throughput

Pre-test Configuration

The EF_POLL_USEC environment variable is set to `100000`.

Throughput with Onload

The benchmark performance tests can be run with Onload using the Onload kernel bypass. To do this add `onload` to the start of each command line.

TCP Throughput with Netperf

Run the net-server application on system-1:

```
[system-1]# pkill -f netserver
[system-1]# onload taskset -c 1 netserver
```

Run the netperf application on system-2:

```
[system-2]# onload taskset -c 1 \
   netperf -t TCP_RR -H <system1-ip> -l 10 -- -r 32
Socket  Size    Request Resp.   Elapsed Trans.
Send    Recv    Size    Size    Time    Rate
bytes   Bytes   bytes   bytes   secs.   per sec
16384   87380   32      32      10.00   481130.34
```

This transaction rate is over 3 times the rate achieved without Onload (see Throughput without Onload below).

Throughput without Onload

The benchmark performance tests can be run without Onload using the regular kernel network drivers. To do this remove the `onload` part from the command line.

TCP Throughput with Netperf

Run the net-server application on system-1:

```
[system-1]# pkill -f netserver
[system-1]# taskset -c 1 netserver
```

Run the netperf application on system-2:

```
[system-2]# taskset -c 1 \
   netperf -t TCP_RR -H <system1-ip> -l 10 -- -r 32
Socket  Size    Request Resp.   Elapsed Trans.
Send    Recv    Size    Size    Time    Rate
bytes   Bytes   bytes   bytes   secs.   per sec
16384   87380   32      32      10.00   135806.82
```

Observe the reduced transaction rate compared to that achieved with Onload (see Throughput with Onload above).

HTTP connections

Pre-test Configuration

The `nginx` application is installed on `system-2` and is configured as follows.

The `/etc/nginx/nginx.conf` file is shown below. Changes from the distributed file are *highlighted* :

```
user                            nginx;
worker_processes                3;
worker_cpu_affinity             auto 1110;
error_log                       /var/log/nginx/error.log warn;
pid                             /var/run/nginx.pid;
events {
    worker_connections          1024;
}
http {
    include                     /etc/nginx/mime.types;
    default_type                application/octet-stream;
    log_format main             '$remote_addr - $remote_user [$time_local]
"$request" '
                                '$status $body_bytes_sent "$http_referer" '
                                '"$http_user_agent" "$http_x_forwarded_for"';
    access_log                  off;
    sendfile                    off;
    keepalive_timeout           300s;
    keepalive_requests          1000000;
    open_file_cache             max=1000 inactive=20s;
    open_file_cache_valid       30s;
    open_file_cache_errors      off;
    include                    /etc/nginx/conf.d/*.conf;
}
```

The `/etc/nginx/conf.d` directory contains only the `default.conf` file, shown below. Changes from the distributed file are *highlighted*:

```
server {
    listen        nn.nn.nn.nn:80 reuseport;
    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
    }
    error_page    500 502 503 504  /50x.html;
    location = /50x.html {
        root    /usr/share/nginx/html;
    }
}
```

where *nn.nn.nn.nn* is the IP address of the `system-2` adapter port that will receive the HTTP requests.

A zero-byte file is created in the root of the `nginx` server:

```
# touch /usr/share/nginx/html/0kb.bin
```

Connections with Onload

The benchmark performance tests can be run with Onload using the Onload kernel bypass. To do this add `onload` to the start of each command line.

HTTP Connections with nginx

Run the `nginx` application on system-2, increasing the number of file descriptors available, and accelerating it using the nginx_reverse_proxy Onload profile:

```
[system-2]# ulimit -n 1000000 && onload --profile=nginx_reverse_proxy nginx
```

Run multiple instances of the `wrk` application on system-1, so there is the capacity to generate more connection requests than can be handled. A zero-byte file is requested, and the "`Connection: close`" header is passed to close the connection immediately:

```
[system-1]# for i in {1..3}; do taskset -c $i wrk -t 1 -c 50 -d 10s
-H 'Connection: close' http://system-2/0kb.bin & done
```

Each instance outputs its own results. Aggregating the results gives the following totals:

```
Requests/sec: 111061.19
Transfer/sec:       25.81MB
```

The server handles 111061 connections per second (Requests/sec). This is over 3 times the rate achieved without Onload (see Connections without Onload below).

Note the following:

- the total throughput is much less than line rate, so any limit on capacity is in nginx rather than in the network.
- some instances have connect or timeout errors, or have reduced throughput, indicating that there are enough instances to reach full nginx capacity:

  ```
  Socket errors: connect 0, read 0, write 0, timeout 38
  ```

Connections without Onload

The benchmark performance tests can be run without Onload using the regular kernel network drivers. To do this remove the `onload` part from the command line.

HTTP Connections with nginx

Run the `nginx` application on system-2, increasing the number of file descriptors available:

```
[system-2]# ulimit -n 1000000 && nginx
```

Run multiple instances of the `wrk` application on system-1, so there is the capacity to generate more connection requests than can be handled. A zero-byte file is requested, and the "Connection: close" header is passed to close the connection immediately:

```
[system-1]# for i in {1..3}; do taskset -c $i wrk -t 1 -c 50 -d 10s
-H 'Connection: close' http://system-2/0kb.bin & done
```

Each instance outputs its own results. Aggregating the results gives the following totals:

```
Requests/sec: 34431.98
Transfer/sec:        8.02MB
```

The server handles only 34431 connections per second (Requests/sec). This rate is greatly reduced compared to that achieved with Onload (see Connections with Onload above).

Further Information

For installation of Solarflare adapters and performance tuning of the network driver when not using Onload refer to the *Solarflare Server Adapter User Guide* (SF-103837-CD).

Questions regarding Solarflare products, Onload and this user guide can be emailed to support-nic@amd.com.

Appendix N

Additional Resources and Legal Notices

Finding Additional Documentation

Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to https://docs.xilinx.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs webpage.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

XtremeScale Documents

1. *Solarflare Server Adapter User Guide* (SF-103837-CD)

Alveo Documents

1. *Alveo X3522 Data Sheet* (DS1002)
2. *Alveo X3522 Installation Guide* (UG1522)
3. *Alveo X3522 User Guide* (UG1523)

Onload Documents

1. *TCPDirect User Guide* (SF-116303-CD)
2. *ef_vi User Guide* (SF-114063-CD)
3. *Alveo X3 ef_vi Conversion Guide* (XN-201257-CD)

Precision Time Protocol Documents

1. *Enhanced PTP User Guide* (UG1602)

Additional AMD Resources

1. AMD licensing website: https://www.xilinx.com/getproduct
2. AMD Community Forums: https://forums.xilinx.com
3. *Xilinx Third-Party End User License Agreement*
4. *End-User License Agreement*

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/31/2023 Version 1.2	
New Features in OpenOnload-8.1.0	Added.
New Features in OpenOnload-8.0.2	Added.
New Features in OpenOnload-8.0.1	Added.
Supported Operating Systems	Updated for Onload 8.1.0.
Features	Added features that are new in Onload 8.1.0.
Environment Variables	Added change to EF_CHALLENGE_ACK_LIMIT.
Module Options	Inserted line breaks in examples.
Adapter Net Drivers	Added releases after Onload 8.0.0.34.
EF_CHALLENGE_ACK_LIMIT	Increased default and maximum values.
EF_TCP_RCVBUF	Added minimum and maximum values.
EF_TCP_RCVBUF_ESTABLISHED_DEFAULT	Added minimum and maximum values.
EF_TCP_SNDBUF	Added minimum and maximum values.
EF_TCP_SNDBUF_ESTABLISHED_DEFAULT	Added minimum and maximum values.
EF_UDP_RCVBUF	Added minimum and maximum values.
EF_UDP_SNDBUF	Added minimum and maximum values.
04/07/2023 Version 1.1	
Chapter 1: What's New	Updated download locations in first note.
Onload and Network Adapter Drivers	Retitled and rearranged. Updated to emphasize that X3 drivers must be built and installed before Onload.
Table 30: Number of PIO Buffers Available	Updated PIO buffer sizes.
10/18/2022 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright