Onload User Guide

SF-104474-CD

Issue 15

Table of Contents

Chapter 1: What's New

This section identifies new features introduced in the OpenOnload 201310 release, the OpenOnload 201310-u1 release and latest additions to this user guide.

This version of the user guide introduces the Solarflare Flareon™ Ultra SFN7122F and SFN7322F Dual-Port 10GbE PCIe 3.0 Server I/O Adapters.

This user guide is also applicable to EnterpriseOnload v2.1.0.3 - refer to Change History on page 85 to confirm feature availability in the Enterprise release.

For a complete list of features and enhancements refer to the *release notes* and the *release change log* available from: http://www.openonload.org/download.html.

New Features 201310-u1

Receive Packet Hardware Timestamping

The SO_TIMESTAMPING socket option allow an application to recover hardware generated timestamps for received packets. This is supported for Onload applications running on the Solarflare SFN7000 series adapters. For details refer to SO_TIMESTAMPING (hardware timestamps) on page 50.

The Onload environment variable EF_RX_TIMESTAMPING controls whether hardware timestamps are enabled on all received packets. Refer to Appendix A: Parameter Reference on page 93 for details.

Low Latency Transmit

The Onload environment variable EF_TX_PUSH_THRESHOLD is used together with EF_TX_PUSH to improve the performance of the low-latency transmit feature. This feature is supported on SFN7000 series adapters. For details of this variable refer to Appendix A: Parameter Reference on page 93.

Onload Extensions API - Java Native Interface

The JNI wrapper provides access to the Onload extensions API from within Java. Users should refer to the README file and Java source files in `/onload-<version>/src/tools/jni` directory.

Onload Extensions API - MSG WARM Detection

Onload versions before onload-201310 do not support the ONLOAD_MSG_WARM flag and using the flag on TCP send calls will cause the 'warm' packet to be actually sent on the wire.

The `onload_fd_check_feature()` function can be called before using the ONLOAD_MSG_WARM flag on TCP send calls to ensure the Onload socket being used does actually support the ONLOAD_MSG_WARM feature. For details refer to onload_fd_check_feature() on page 114.

Adapter Net Driver

The OpenOnload 201310-u1 release includes an updated net driver. The **v4.0.2.6628** driver supports all Solarflare adapters.

New Features 201310

The OpenOnload 201310 release includes new features to support the Solarflare Flareon Ultra SFN7122F and SFN7322F adapters.

Multicast Replication

The SFN7000 series adapters support Multicast Replication where 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 a copy of the received packets without the need to share Onload stacks. See Multicast Replication on page 56 for details.

Large Buffer Table Support

The SFN7000 series adapters support large buffer table entries, which for many applications, can help eliminate limitations on the number of packet buffers that existed on previous generation Solarflare adapters. Refer to Large Buffer Table Support on page 62 for further details.

PIO

The SFN7000 series adapters support programmed input/output on the transmit path where packets are pushed directly to the adapter by the CPU. See “Programmed I/O” on page 70.

Templated Sends

Building on top of PIO, the templated sends extensions API can further improve transmit latency using a packet template which contains the bulk of the data to be sent. The template is created on the adapter and populated with the remaining data immediately before sending. For more information about templated sends and the use of packet templates see Templated Sends on page 71.

ONLOAD_MSG_WARM

The ONLOAD_MSG_WARM flag, when used on the TCP send calls, will ’exercise’ the TCP fast send path to maintain low latency performance on TCP connections that rarely send data. See ONLOAD_MSG_WARM on page 46 for details.

Onload Handling dup3()

A system call to dup3( ) will now be intercepted in Onload-201310.

Adapter Net Driver

The OpenOnload 201310 release includes an updated net driver. The **v4.0.0.6585** driver supports all Solarflare adapters.

Change History

The Change History section 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. Refer to Change History on page 85.

Documentation Changes

Appendix J: Solarflare efpio Test Application on page 180 describes the Solarflare layer 2 benchmark efpio with command line examples and options.

Chapter 2: Low Latency Quickstart Guide

Introduction

This section demonstrates how to achieve very low latency coupled with minimum jitter on a system fitted with the Solarflare SFN7122F network adapter and using Solarflare's kernel-bypass network acceleration middleware, OpenOnload.

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

Please read the Solarflare LICENSE file regarding the disclosure of benchmark test results.

Software Installation

Before running Low Latency benchmark tests ensure that correct driver and firmware versions are installed e.g. (minimum driver and firmware versions are shown):

```
[root@server-N]# ethtool -i eth3
driver: sfc
version: 4.0.0.6585
firmware-version: 4.0.0.6585
```

Netperf

Netperf can be downloaded from http://www.netperf.org/netperf/

Unpack the compressed tar file using the tar command:

```
[root@system-N]# tar -zxvf netperf-<version>.tar.gz
```

This will create a sub-directory called `netperf-<version>` from which the configure and make commands can be run (as root):

```
./configure
make install
```

Following installation the `netperf` and `netserver` applications are located in the `src` subdirectory.

Solarflare sfnettest

Download the `sfnettest-<version>.tgz` source file from www.openonload.org

Unpack the tar file using the tar command:

```
[root@system-N]# tar -zxvf sfnettest-<version>.tgz
```

Run the `make` utility from the `sfnettest-<version>/src` subdirectory to build the `sfnt-pingpong` application.

Solarflare 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 in the Onload User Guide for more details.

Download the `Onload-<version>.tgz` file from www.openonload.org

Unpack the tar file using the tar command:

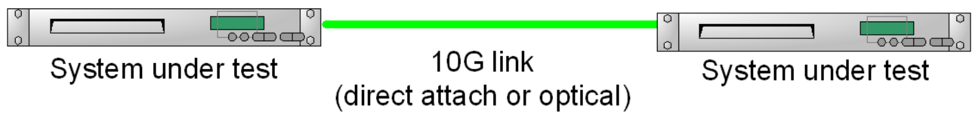
```
[root@system-N]# tar -zxvf onload-<version>.tgz
```

Run the `onload_install` command from the `Onload-<version>/scripts` subdirectory:

```
[root@system-N]# ./onload_install
```

Test Setup

The diagram below identifies the required physical configuration of two servers equipped with Solarflare network adapters connected back-to-back in order to measure the latency of the adapter, drivers and acceleration middleware. If required, tests can be repeated with a 10G switch on the link to measure the additional latency delta using a particular switch.



Requirements:

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

Pre-Test Configuration

On both machines:

1. Isolate the CPU cores that will be used from the general SMP balancing and scheduler algorithms. Add the following option to the kernel line in /boot/grub/grub.conf:

   ```
   isolcpus=<comma separated cpu list>
   ```

2. Stop the cpuspeed service to prevent power saving modes from reducing CPU clock speed.

   ```
   [root@system-N]# service cpuspeed stop
   ```

3. Stop the irqbalance service to prevent the OS from rebalancing interrupts between available CPU cores.

   ```
   [root@system-N]# service irqbalance stop
   ```

4. Stop the iptables service to eliminate overheads incurred by the firewall. Solarflare recommend this step on RHEL6 for improved latency when using the kernel network driver.

   ```
   [root@system-N]# service iptables stop
   ```

5. Disable interrupt moderation.

   ```
   [root@system-N]# ethtool -C eth<N> rx-usecs 0 adaptive-rx off
   ```

   Where N is the identifier of the Solarflare adapter ethernet interface e.g. `eth4`

6. Refer to the Reference System Specification below for BIOS features.

Reference System Specification

The following latency measurements were recorded on twin Intel® Sandy Bridge servers. The specification of the test systems is as follows:

- DELL PowerEdge R210 servers equipped with Intel® Xeon® CPU E3-1280 @3.60GHz, 2 x 2GB DIMMs.
- BIOS: Turbo mode ENABLED, cstates DISABLED, IOMMU DISABLED.
- Red Hat Enterprise Linux V6.4 (x86_64 kernel, version 2.6.32-358.el6.x86_64).
- Solarflare SFN7122F NIC (driver and firmware – see Software Installation) Direct attach cable at 10G.
- OpenOnload distribution: openonload-201310.

It is expected that similar results will be achieved on any Intel based, PCIe Gen 3 server or compatible system.

UDP Latency: Netperf

Run the netserver application on system-1:

```
[root@system-1]# pkill -f netserver
[root@system-1]# onload --profile=latency taskset -c 0 ./netserver
```

Run the netperf application on system -2:

```
[root@system-2]# onload --profile=latency taskset -c 0 ./netperf -t UDP_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
229376 229376 32       32      10.00    294170.83
```

294170 transactions/second means that each transaction takes 1/294170 seconds resulting in a RTT/2 latency of (1/294170)/2 or **1.69µs**

UDP Latency: sfnt-pingpong

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

```
[root@system-1]# onload --profile=latency taskset -c 0 ./sfnt-pingpong
[root@system-2]# onload --profile=latency taskset  -c 0 ./sfnt-pingpong --affinity
"0;0"  udp  <system1-ip>


 #      size    mean    min     median  max     %ile    stddev  iter
        0       1648    1603    1642    12340   1742    48      904000
        1       1643    1598    1638    14532   1732    46      907000
        2       1644    1600    1638    9641    1734    42      907000
        4       1643    1598    1638    13210   1731    43      907000
        8       1646    1600    1640    8707    1736    44      905000
        16      1635    1591    1629    9753    1724    41      911000
        32      1672    1612    1666    11198   1756    48      891000
        64      1706    1649    1701    9638    1796    42      873000
        128     1824    1744    1820    8620    1938    53      817000
        256     1981    1911    1975    8238    2090    48      753000
```

The output identifies mean, minimum, median and maximum (nanosecond) RTT/2 latency for increasing TCP packet sizes including the 99% percentile and standard deviation for these results. A message size of 32 bytes has a mean latency of **1.67µs** with a 99%ile latency under **1.8µs**.

TCP Latency: Netperf

Run the netserver application on system-1:

```
[root@system-1]# pkill -f netserver
[root@system-1]# onload --profile=latency taskset -c 0 ./netserver
```

Run the netperf application on system-2:

```
[root@system-2]# onload --profile=latency taskset  -c  0 ./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    271057.10
```

271057 transactions/second means that each transaction takes 1/271057 seconds resulting in a RTT/2 latency of (1/271057)/2 or **1.84µs**.

TCP Latency: sfnt-pingpong

Run the sfnt-pingpong application on both systems:

```
[root@system-1]# onload --profile=latency taskset -c 0 ./sfnt-pingpong
[root@system-2]# onload --profile=latency taskset -c 0 ./sfnt-pingpong --affinity
"0;0"  tcp  <system1-ip>
#
 #       size     mean     min      median   max      %ile     stddev   iter
         1        1810     1744     1795     15775    1959     62       823000
         2        1809     1744     1793     11080    1961     60       824000
         4        1812     1743     1797     12595    1964     60       822000
         8        1810     1745     1794     11773    1959     58       824000
         16       1816     1746     1801     9081     1967     56       821000
         32       1835     1767     1820     8049     1989     58       812000
         64       1900     1804     1886     9764     2065     67       785000
         128      2000     1907     1985     8440     2167     67       746000
         256      2139     2057     2123     482577   2304     578      698000
```

The output identifies mean, minimum, median and maximum (nanosecond) RTT/2 latency for increasing TCP packet sizes including the 99% percentile and standard deviation for these results. A message size of 32 bytes has a mean latency of **1.83µs** with a 99%ile latency under **2.0µs.**

Layer 2 ef_vi Latency

The `efpio` UDP test application, supplied with the onload-201310 package, can be used to measure latency of the Solarflare `ef_vi` layer 2 API. efpio uses PIO.

Using the same back-to-back configuration described above, `efpio` latency tests were recorded on DELL PowerEdge R210 servers.

```
# ef_vi_version_str: 201306-7122preview2
# udp payload len: 28
# iterations: 100000
# frame len: 70
round-trip time: 2.65 µs  (1.32 RTT/2)
```

Appendix J: Solarflare efpio Test Application on page 180 describes the `efpio` application, command line options and provides example command lines.

Comparative Data

Dual Package Server

Latency tests recorded in this document, were conducted on a single CPU package server. Results may differ between different server types and may be different from servers having multiple NUMA nodes. Many factors influence the latency on a server so some experimentation is required to identify the CPU core producing the lowest latency.

To enable comparison, the latency benchmark tests were repeated with a pair of dual CPU package servers having the following specification:

- DELL PowerEdge R620 servers equipped with Intel® Xeon® CPU E5-2690 @2.90GHz, 4 x 8GB DIMMs.
- BIOS: Turbo mode ENABLED, cstates DISABLED, IOMMU DISABLED.
- Red Hat Enterprise Linux V6.4 (x86_64 kernel, version 2.6.32-358.el6.x86_64).
- Solarflare SFN7122F NIC (see Software Installation above) Direct attach cable at 10G.
- OpenOnload distribution: openonload-201310.

Table 1: Dual CPU Server Data

Application	UDP µs	TCP µs
Netperf	1.83	1.99
efpio	1.42	N/A
sfnt-pingpong	1.78	1.97

Adapter Comparison

The following table shows a comparison between latency tests conducted on the SFN6122F and the SFN7122F adapters - values shown are the RTT/2 value in microseconds.

Table 2: Latency Tests - Comparative Data

Test	SFN6122F	SFN7122F	Latency gain
UDP	2.2	1.6	27%
TCP	2.4	1.8	25%
ef_vi UDP	efpingpong - 2.0	efpio - 1.3	40%

Testing Without Onload

The benchmark performance tests can be run without Onload using the regular kernel network drivers. To do this remove the `onload --profile=latency` part from the command line.

To get the best response and comparable latency results using kernel drivers, Solarflare recommend setting interrupt affinity such that interrupts and the application are running on different CPU cores but on the same processor package - examples below.

Use the following command to identify receive queues created for an interface e.g:

```
# cat /proc/interrupts | grep eth2

33: 0 0 0 0 IR-PCI-MSI-edge eth2-0
34: 0 0 0 0 IR-PCI-MSI-edge eth2-1
```

Direct IRQ 33 to CPU core 0 and IRQ 34 to CPU core 1:

```
# echo 1 > /proc/irq/33/smp_affinity
# echo 2 > /proc/irq/34/smp_affinity
```

Latency figures are shown in microseconds.

RHEL 6.4 (TCP)		**RHEL 6.4 (UDP)**	
Mean	*99%ile*	*Mean*	*99%ile*
6.3	7.0	5.4	5.8

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) available from https://support.solarflare.com/

The Onload feature set and detailed performance tuning information can be found in the Onload User Guide (SF-104474-CD) available from https://support.solarflare.com/

Questions regarding Solarflare products, Onload and this user guide can be emailed to support@solarflare.com.

Chapter 3: Background

3.1 Introduction.

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

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

Onload is the Solarflare 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
- 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

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 Figure 1.

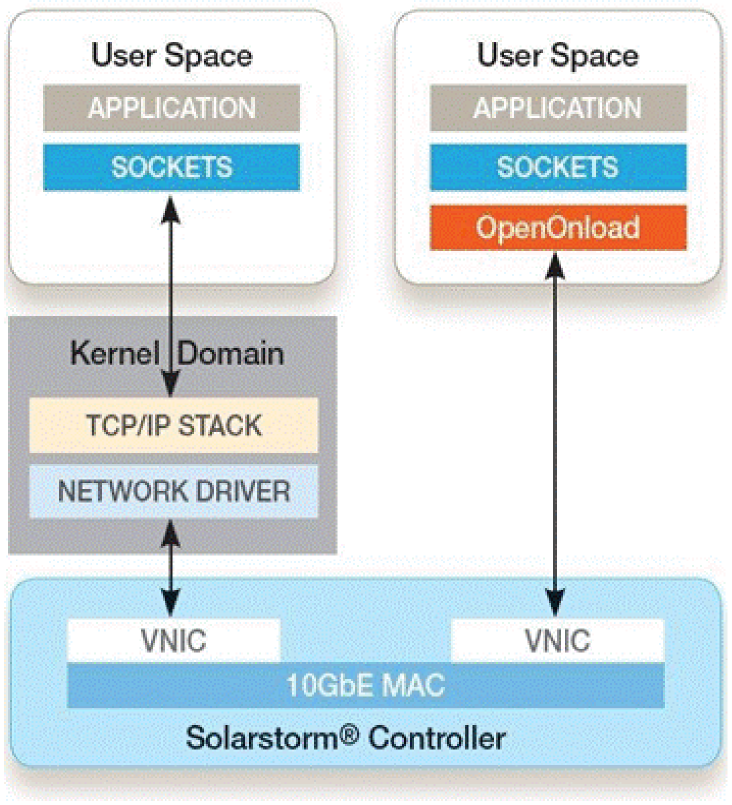


Figure 1: Contrast with Conventional Networking.

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 Solarflare NIC allows optimum performance without compromising security or system integrity.

In summary, Onload can significantly reduce network processing overheads. As an example, the table below shows application to application latency (1/2 RTT) being reduced by 60% over the regular kernel stack.

Back to Back 1/2 RTT latency	**TCP (us)**	**UDP (us)**
Solarflare (Onload)	1.83	1.67
Solarflare (Kernel)	6.3	5.4

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, i.e. 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 Figure 2.

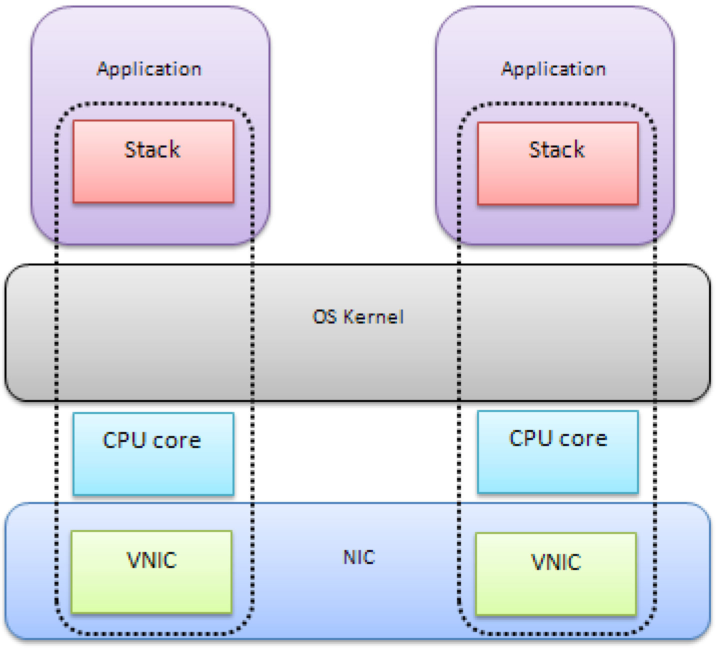


Figure 2: Onload Partitioned Network Adapter

Further Information

For detailed information about Onload operation and functionality refer to Features & Functionality on page 34.

Chapter 4: Installation

4.1 Introduction

This chapter covers the following topics:

4.2 Onload Distributions

Onload is available in two distributions

- "OpenOnload" is a free version of Onload available from http://www.openonload.org/ distributed as a source tarball under the GPLv2 license. OpenOnload is subject to a linear development cycle where major releases every 3-4 months 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.

The Solarflare product range offers a flexible and broad range of support options, users should consult their reseller for details and refer to the Solarflare Enterprise Service and Support information at http://www.solarflare.com/Enterprise-Service-Support.

4.3 Hardware and Software Supported Platforms

- Onload is supported for all Solarflare Flareon Adapters, Onload Network Adapters, Solarflare mezzanine adapters and the SFA6902F ApplicationOnload™ Engine. Refer to the Solarflare Server Adapter User Guide 'Product Specifications' for details.
- Onload can be installed on the following x86, 32 bit and 64 bit platforms:

  Red Hat Enterprise Linux 5, 6 and Red Hat MRG, MRG 2 update 3

  SUSE Linux Enterprise Server 10, 11 and SLERT

  Whilst the Onload QA test cycle predominantly focuses on the Linux OS versions documented above, Solarflare are not aware of any issues preventing Onload installation on other Linux variants such as Ubuntu, Centos, Gentoo, Fedora and Debian variants.
- All Intel and AMD x86 processors.

4.4 Onload and the Network Adapter Driver

The Solarflare network adapter driver, the "net driver", is generally available from three sources:

- Download as source RPM from support.solarflare.com.
- Packaged 'in box' in many Linux distributions e.g Red Hat Enterprise Linux.
- Packaged in the OpenOnload distribution.

When using Onload you must use the adapter driver distributed with that version of Onload.

4.5 Pre-install Notes

NOTE: If Onload is to accelerate a 32bit application on a 64bit architecture, the 32bit libc development headers should be installed before building Onload. Refer to Appendix C for install instructions.

NOTE: You must remove any existing Solarflare RPM driver packages before installing Onload.

NOTE: When migrating between Onload versions or between OpenOnload and EnterpriseOnload, a previously installed version must first be removed using the `onload_uninstall` command.

NOTE: The Solarflare drivers are currently classified as unsupported in SLES11, the certification process is underway. To overcome this add '`allow_unsupported_modules 1`' to the `/etc/modprobe.d/unsupported-modules` file

4.6 EnterpriseOnload - Build and Install from SRPM

The following steps identify the procedures to build and install EnterpriseOnload. SRPMs can be built by the 'root' or 'non-root' user, but the user must have superuser privileges to install RPMs. Customers should contact their Solarflare customer representative for access to the EnterpriseOnload SRPM resources.

Build the RPM

NOTE: Refer to Appendix C 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 2 binary RPM files which can be found in the

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

for example the EnterpriseOnload user-space components:

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

and the EnterpriseOnload kernel components:

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

Install the EnterpriseOnload RPM

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

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

NOTE: EnterpriseOnload 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.

Installing the EnterpriseOnload Kernel Module

This will load the EnterpriseOnload kernel driver and other driver dependencies and create any device nodes needed for EnterpriseOnload drivers and utilities. The command should be run as root.

```
/etc/init.d/openonload start
```

Following successful execution this command produces no output, but the ’onload’ script will identify that the kernel module is now loaded.

```
onload


EnterpriseOnload <version>
Copyright 2006-2013 Solarflare Communications, 2002-2005 Level 5 Networks
Built: Oct 15 2013 09:19:23 12:23:12 (release)
Kernel module: <version>
```

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

4.7 OpenOnload DKMS Installation

OpenOnload from version 201210 is available in DKMS RPM format. The OpenOnload DKMS distribution package is not available directly from www.openonload.org, users who wish to install from DKMS should send an email to support@solarflare.com.

4.8 Build OpenOnload Source RPM

A source RPM can be built from the OpenOnload distribution tar file.

1 Download the required tar file from the following location:

http://www.openonload.org/download.html

Copy the file to a directory on the machine where the source RPM is to be created.

2 As root, execute the following command:

```
rpmbuild -ts openonload-<version>.tgz*

 x86_64 Wrote: /root/rpmbuild/SRPMS/openonload-<version>.src.rpm
```

The output identifies the location of the source RPM. Use the `-ta` option to get a binary RPM.

4.9 OpenOnload - Installation

The following procedure demonstrates how to download, untar and install OpenOnload.

Download and untar OpenOnload

1 Download the required tar file from the following location:

http://www.openonload.org/download.html

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 openonload-<version>.tgz
```

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

Building and Installing OpenOnload

NOTE: Refer to Appendix C for details of build dependencies.

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

```
./onload_install
```

Successful installation will output the following 3 lines from the `onload_install` process:

```
onload_install: Build complete.
onload_install: Installing OpenOnload.
onload_install: Install complete.
```

NOTE: The onload_install script does not create RPMs.

Load Onload Drivers

Following installation it is necessary to load the Onload drivers:

```
onload_tool reload
```

When used with OpenOnload this command will replace any previously loaded network adapter driver with the driver from the OpenOnload distribution.

Check that Solarflare drivers are loaded using the following commands:

```
lsmod | grep sfc
lsmod | grep onload
```

An alternative to the reload command is to reboot the system to load Onload drivers.

Confirm Onload Installation

When the Onload installation is complete run the `onload` command from the `openonload-<version>/scripts` subdirectory to confirm installation of Onload software and kernel module:

```
[root@server1 scripts] onload
```

Will display the Onload product banner and usage:

```
OpenOnload 201310
Copyright 2006-2012 Solarflare Communications, 2002-2005 Level 5
Networks
Built: Oct 15 2013 09:19:23 (release)
Kernel module: 201310

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
```

4.10 Onload Kernel Modules

To identify Solarflare drivers already installed on the server:

```
modprobe -l | grep sfc (grep onload)
```

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.
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 behaviour.
sfc_resource.ko	Manages the virtualization resources of the adapter and shares the resources between other drivers.
onload.ko	The kernel component of Onload.

To unload any loaded drivers:

```
onload_tool unload
```

To remove the installed files of a previous Onload:

```
onload_uninstall
```

To load the Solarflare and Onload drivers (if not already loaded):

```
modprobe sfc
```

Reload drivers following upgrade or changed settings:

```
onload_tool reload
```

4.11 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 e.g. system-config-network (Red Hat) or yast (SUSE).

4.12 Installing Netperf

Refer to the Low Latency Quickstart Guide on page 4 for instructions to install Netperf and Solarflare sfnettest applications.

4.13 How to run Onload

Once Onload has been installed there are different ways to accelerate applications. Exporting LD_PRELOAD will mean that all applications started in the same environment will be accelerated.

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

Pre-fixing the application command line with the onload command will accelerate the application.

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

4.14 Testing the Onload Installation

The the Low Latency Quickstart Guide on page 4 demonstrates testing of Onload with Netperf and the Solarflare sfnettest benchmark tools.

4.15 Apply an Onload Patch

Occasionally, the Solarflare Support Group may issue a software 'patch' which is applied to onload to resolve a specific bug or investigate a specific issue. The following procedure describes how a patch should be applied to the **installed OpenOnload software**.

1 Copy the patch to a directory on the server where onload is already installed.

2 Go to the onload directory and apply the patch e.g.

```
cd openonload-<version>
[openonload-<version>]$ patch -p1 < ~/<path>/<name of patch file>.patch
```

3 Uninstall the old onload drivers

```
[openonload-<version>]$ onload_uninstall
```

4 Build and re-install the onload drivers

```
[openonload-<version>]$ ./scripts/onload_install
[openonload-<version>]$ onload_tool reload
```

The following procedure describes how a patch should be applied to the **installed EnterpriseOnload RPM**. (This example patches EnterpriseOnload version 2.1.0.3).

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

```
rpm2cpio enterpriseonload-2.1.0.3-1.src.rpm | cpio –id
tar -xzf enterpriseonload-2.1.0.3.tgz
cd enterpriseonload-2.1.0.3
patch -p1 < $PATCHNAME
```

2 This can now be installed directory from this directory:

```
./scripts/onload_install
```

3 Or it can be repackaged as a new RPM:

```
cd ..
tar cz fenterpriseonload-2.1.0.3.tgz enterpriseonload-2.1.0.3
rpmbuild -ts enterpriseonload-2.1.0.3.tgz
```

4 The rpmbuild procedure will display a ’Wrote’ line identifying the location of the built RPM e.g

```
Wrote: /root/rpmbuild/SRPMS/enterpriseonload-2.1.0.3-1.el6.src.rpm
```

5 Install the RPM in the usual way:

```
rpm -ivh /root/rpmbuild/SRPMS/enterpriseonload-2.1.0.3-1.el6.src.rpm
```

Chapter 5: Tuning Onload

5.1 Introduction

This chapter documents the available tuning options for Onload, and the expected results. The options can be split into the following categories:

- System Tuning
- Standard Latency Tuning.
- Advanced Tuning driven from analysis of the Onload stack using `onload_stackdump.`

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 Appendices A and B of this manual. Examples throughout this guide assume the use of the *bash* or *sh* shells; other shells may use different methods to export variables into the applications environment.

Section 5.2 describes tools and commands which can be used to tune the server and OS.

Section 5.3 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.

Section 5.5 introduces advanced tuning options using `onload_stackdump`. There are worked examples to 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.

5.2 System Tuning

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

Sysjitter

The Solarflare 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 www.openonload.org

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 4 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.

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
`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

Field	Description
`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

CPU Power Saving Mode

Modern processors utilize 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 signalled to return to the fully-on power state. These changes in CPU core power states create additional network latency and jitter.

Solarflare therefore recommend 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. If this is not possible, as an alternative, Solarflare provide a software mechanism to prevent each CPU core entering these lower power states. This is achieved by using the following command (as root):

```
onload_tool disable_cstates [persist]
```

Use the persist option to retain the setting following system restarts.

This command prevents Linux from indicating to the CPU core it is idle, but can result in the processor using additional power compared with disabling the lower power states in the BIOS. It can therefore cause the processor to operate at a higher temperature; this is why disabling processor power states in the BIOS is recommended where available.

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

NOTE: The `onload_tool disable_cstates` command relies on the idle_enable=2 option in the onload.conf file. In Linux 3.x kernels this does not function and the user should replace the line options sfc_tune idle_enable=2 line in /etc/modprobe.d/onload.conf with intel_idle.max_cstate=0 idle=poll.

Customers should consult their system vendor and documentation for details concerning the disabling of C1E, C states or CPU power saving states.

5.3 Standard Tuning

This section details standard tuning steps for Onload.

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 behaviour. 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 Appendix B: Meta Options on page 121 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()`. 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 behaviour 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=EF_POLL_USEC
EF_SOCK_LOCK_BUZZ=1
```

```
EF_STACK_LOCK_BUZZ=1
```

Turn off adaptive moderation and set interrupt moderation to 60 microseconds. Use the following command:

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

See Appendix B: Meta Options on page 121 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 (aka 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 (e.g. 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 (i.e. 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.

5.4 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 Low Latency Quickstart Guide on page 4 - 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 may 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 may indicate that one thread is being held off the CPU core by another thread.
- 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.
- 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. If separate stacks are not an option then it may 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.
- Jitter may 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.

- When applications only need to send and receive occasionally it may 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.
- 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 Solarflare sysjitter utility can be used to identify and measure jitter on all cores of an idle system - refer to Sysjitter on page 22 for details.

Using Onload Tuning Profiles

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

```
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 e.g

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

Following Onload installation, profiles provided by Solarflare 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 e.g.

```
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 See "Appendix A: Parameter Reference" on page 93.

Benchmark Testing

Benchmark procedures using Onload, netperf and sfnt_pingpong are described in the Low Latency Quickstart Guide on page 4.

5.5 Advanced Tuning

Advanced tuning requires closer examination of the application performance. The application should be tuned to achieve the following objectives:

- To have as much processing at user-level as possible.
- To have as few interrupts as possible.
- To eliminate drops.
- To minimize lock contention.

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

The following sections demonstrate the use of `onload_stackdump` to examine aspects of the system performance and set environment variables to achieve the tuning objectives.

For further examples and use of `onload_stackdump` refer to Appendix E: onload_stackdump on page 137.

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

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 poll
```

Output:

```
$ onload_stackdump lots | grep poll
  time: netif=52a6fc7 poll=52a6fc7 now=52a6fd8  (diff=0.017sec)
                     k_polls: 673
                     u_polls: 2
              periodic_polls: 655
            evq_wakeup_polls: 12
              deferred_polls: 0
           evq_timeout_polls: 4
        $
```

The output identifies many more k_polls than u_polls indicating that the stack is operating mainly at kernel-level and not achieving optimal performance.

Solution

Terminate the application and set the EF_POLL_USEC parameter to 100000. Re-start the application and re-run onload_stackdump:

```
export EF_POLL_USEC=100000
onload_stackdump lots | grep polls
```

```
$ onload_stackdump lots | grep polls
  time: netif=52debb1 poll=52debb1 now=52debb1  (diff=0.000sec)
                     k_polls: 18
                     u_polls: 181272
              periodic_polls: 35
            evq_wakeup_polls: 3
              deferred_polls: 0
           evq_timeout_polls: 1
$
```

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.

As Few Interrupts as Possible

All applications achieve better performance when subject to as few interrupts as possible. The level of interrupts is reported as the evq_interrupt_wakes value reported by onload_stackdump. For example:

```
onload_stackdump lots | grep _evs
```

Output:

```
$ onload_stackdump lots | grep _evs
  evq: cap=2048 current=70 is_32_evs=0 is_ev=0
                        rx_evs: 77274
                        tx_evs: 6583
                  periodic_evs: 0
                evq_interrupt_wakes: 52
               evq_timeout_evs: 3
$
```

The output identifies the number of packets sent and received `tx_evs` and `rx_evs`, together with the number of interrupts, `evq_interrupt_wakes`.

Solution

If an application is observed taking lots of interrupts it may 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`.

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 eth2 | egrep "(nodesc)|(bad)"
     tx_bad_bytes: 0
     tx_bad: 0
     rx_bad_bytes: 0
     rx_bad: 0
     rx_bad_lt64: 0
     rx_bad_64_to_15xx: 0
     rx_bad_15xx_to_jumbo: 0
     rx_bad_gtjumbo: 0
     rx_nodesc_drop_cnt: 0
$
```

The `rx_nodesc_drop_cnt` increasing over time is an indication that packets are being dropped by the adapter due to a lack of Onload-provided receive buffering.

Packets can also be dropped with UDP due to datagrams arriving when the socket buffer is full i.e. traffic is arriving faster than the application can consume. To check for dropped packets at the socket level, enter:

```
onload_stackdump lots | grep drop
```

Output:

```
$ onload_stackdump lots | grep drop
   rcv: drop=0(0%) eagain=0 pktinfo=0
$
```

The output identifies that packets are not being dropped on the system.

Solution

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 may also be worth experimenting with socket buffer sizes (see EF_UDP_RCVBUF). Setting the EF_EVS_PER_POLL variable to a higher value may also improve efficiency - refer to Appendix A for a description of this variable.

Minimizing Lock Contention

Lock contention can greatly affect performance. Use `onload_stackdump` to identify instances of lock contention:

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

Output:

```
$ onload_stackdump lots | egrep "(lock_)|(sleep)"
  sleep_seq=1 wake_rq=TxRx flags=
                   sock_sleeps: 1
                   unlock_slow: 0
                    lock_wakes: 0
               stack_lock_buzz: 0
              sock_lock_sleeps: 0
                sock_lock_buzz: 0
     tcp_send_ni_lock_contends: 0
     udp_send_ni_lock_contends: 0
   getsockopt_ni_lock_contends: 0
   setsockopt_ni_lock_contends: 0
$
```

The output identifies that very little lock contention is occurring.

Solution

If high values are observed for any of the lock variables, try increasing the value of EF_BUZZ_USEC to reduce the 'sleeps' value. If stacks are being shared across processes, try using a different stack per process.

Chapter 6: Features & Functionality

6.1 Introduction

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

6.2 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 e.g. `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 Solarflare interface - which are processed by the Onload stack, whilst those not using a Solarflare interface are transparently passed to the kernel stack.

6.3 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 on page 55), and for a single application to have more than one Onload stack. Refer to Appendix D: Onload Extensions API on page 112.

6.4 Virtual Network Interface (VNIC)

The Solarflare network adapter supports 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 the VNIC client i.e. 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.

6.5 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 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 - since when the application is making socket calls, Onload is busy processing the event queue picking up new network events.

6.6 Onload with Mixed Network Adapters

A server may be equipped with Solarflare network interfaces and non-Solarflare network interfaces. When an application is accelerated, Onload reads the Linux kernel routing table (Onload will only consider the kernel default routing table) to identify which network interface is required to make a connection. If a non-Solarflare 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 by looking in the IP route table.

6.7 Maximum Number of Network Interfaces

Onload supports up to 6 Solarflare network interfaces by default. If an application requires more Solarflare interfaces the following values can be altered in the source code: `src/include/ci/internal/transport_config_opt.h` header file:

`CI_CFG_MAX_INTERFACES and CI_CFG_MAX_REGISTER_INTERFACES`.

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

6.8 Onloaded PIDs

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 on page 138 for details.

6.9 Onload and 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 e.g.

```
# 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.

6.10 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`

6.11 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. These defaults can be overridden, for a specified processes or socket, using socket options or with Onload environment variables. The following tables identify the default Linux values and how Onload tuning parameters can override the Linux settings.

Kernel Value	`tcp_slow_start_after_idle`
Description	controls congestion window validation as per RFC2861. This is "off" by default in Onload, as it's not usually useful in modern switched networks
Onload value	`#define CI_CFG_CONGESTION_WINDOW_VALIDATION`
Comments	in transport_config_opt.h - recompile after changing.

Kernel Value	`tcp_congestion_control`
Description	determines what congestion control algorithm is used by TCP. Valid settings include reno, bic and cubic
Onload value	no direct equivalent - see the section on TCP Congestion Control

Comments	see EF_CONG_AVOID_SCALE_BACK

Kernel Value	`tcp_adv_win_scale`
Description	defines how quickly the TCP window will advance
Onload value	no direct equivalent - see the section on TCP Congestion Control
Comments	see EF_TCP_ADV_WIN_SCALE_MAX

Kernel Value	`tcp_rmem`
Description	the default size of sockets' receive buffers (in bytes)
Onload value	defaults to the currently active Linux settings
Comments	can be overriden with the SO_RCVBUF socket option. can be set with EF_TCP_RCVBUF

Kernel Value	`tcp_wmem`
Description	the default size of sockets' send buffers (in bytes)
Onload value	defaults to the currently active Linux settings
Comments	EF_TCP_SNDBUF overrides SO_SNDBUF which overrides `tcp_wmem`

Kernel Value	`tcp_dsack`
Description	allows TCP to send duplicate SACKS
Onload value	uses the currently active Linux settings
Comments	

Kernel Value	`tcp_fack`
Description	enables fast retransmissions
Onload value	fast retransmissions are always enabled
Comments	

Kernel Value	`tcp_sack`
Description	enable TCP select acknowledgements, as per RFC2018
Onload value	enabled by default
Comments	clear bit 2 of EF_TCP_SYN_OPTS to disable

Kernel Value	`tcp_max_syn_backlog`
Description	the maximum size of a listening socket's backlog
Onload value	set with EF_TCP_BACKLOG_MAX
Comments	

Refer to the Appendix A: Parameter Reference on page 93 for details of environment variables.

6.12 Changing Onload Control Plane Table Sizes

Onload supports the following runtime configurable options which determine the size of control plane tables:

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.	50
max_neighs	Sets the maximum number of rows in the Onload ARP/neighbour table. The value is rounded up to a power of two.	1024
max_routes	Sets the maximum number of entries in the Onload route table.	256

The table above identifies the default values for the Onload control plane tables. The default values are normally sufficient for the majority of applications and creating larger tables may impact application performance. If non-default values are needed, the user should create a file in the /etc/modprobe.d directory. The file must have a .conf extension and Onload options can be added to the file, a single option per line, in the following format:

```
options onload max_neighs=512
```

Following changes Onload should be restarted using the reload command:

```
onload_tool reload
```

6.13 TCP Operation

The table below identifies the 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
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, SYNACK

During the TCP connection establishment 3-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 on page 93 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.

Option	Description
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.
SO_CONNECT_TIME	number of seconds a connection has been established. (Only valid as a `getsockopt()`).
SO_DEBUG	enable protocol debugging.
SO_DONTROUTE	outgoing data should be sent on whatever interface the socket is bound to and not routed via another interface.
SO_ERROR	the errno value of the last error occurring on the socket. (Only valid as a `getsockopt()`).
SO_EXCLUSIVEADDR USE	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-bound 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 may be processed first depending on the selected device queueing discipline.
SO_RCVBUF	sets or gets the maximum socket receive buffer in bytes. The value set is doubled by the kernel and by Onload to allow for bookkeeping overheads when it is set by the `setsockopt()` function call. Note that EF_TCP_RCVBUF overrides this value.
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 i.e. another socket can bind to the same port except when there is an active listening socket bound to the 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. Note that EF_TCP_SNDBUF and EF_TCP_SNDBUF_MODE override 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/disable receiving the SO_TIMESTAMP control message.
SO_TIMESTAMPNS	enable/disable receiving the SO_TIMESTAMP control message.
SO_TIMESTAMPING	enable/disable hardware timestamps for received packets. See SO_TIMESTAMPING (hardware timestamps) on page 50.
SO_TYPE	returns the socket type (SOCK_STREAM or SOCK_DGRAM). (Only valid as a `getsockopt()`).

TCP Level Options

Onload TCP supports the following TCP options which can be used in the `setsockopt()` and `getsockopt()` function calls

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_THRESHHOLD	how long to try to produce a successful keepalive before giving up.
TCP_KEEPALIVE_THRESHHOLD	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 i.e. it is a one time option. Default value is 1 (enabled).

TCP File Descriptor Control

Onload supports the following options in `socket()` and `accept()` calls.

Option	Description
`SOCK_CLOEXEC`	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_NONBLOCK`	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 on page 93 for details.

During Slow Start, the congestion window is initially set to 2 x 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 x MSS bytes. This allows Onload to transmit the minimum of the congestion window and advertised window size i.e.

```
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 x 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 3 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 behaviour of the congestion window and recovery algorithms Refer to Appendix A: Parameter Reference on page 93.:

`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.

The congestion variables should be used with caution so as to avoid violating TCP protocol requirements and degrading TCP performance.

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 3-way handshake. Refer to RFC 2018 for further information.

TCP QUICKACK

TCP will generally aim to defer the sending of ACKs in order to minimize the number of packets on the network. Onload supports the standard `TCP_QUICKACK` socket option which allows some control over this behaviour. 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 on page 93 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 on page 93 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.

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 on page 93 for a description of the loopback variables.

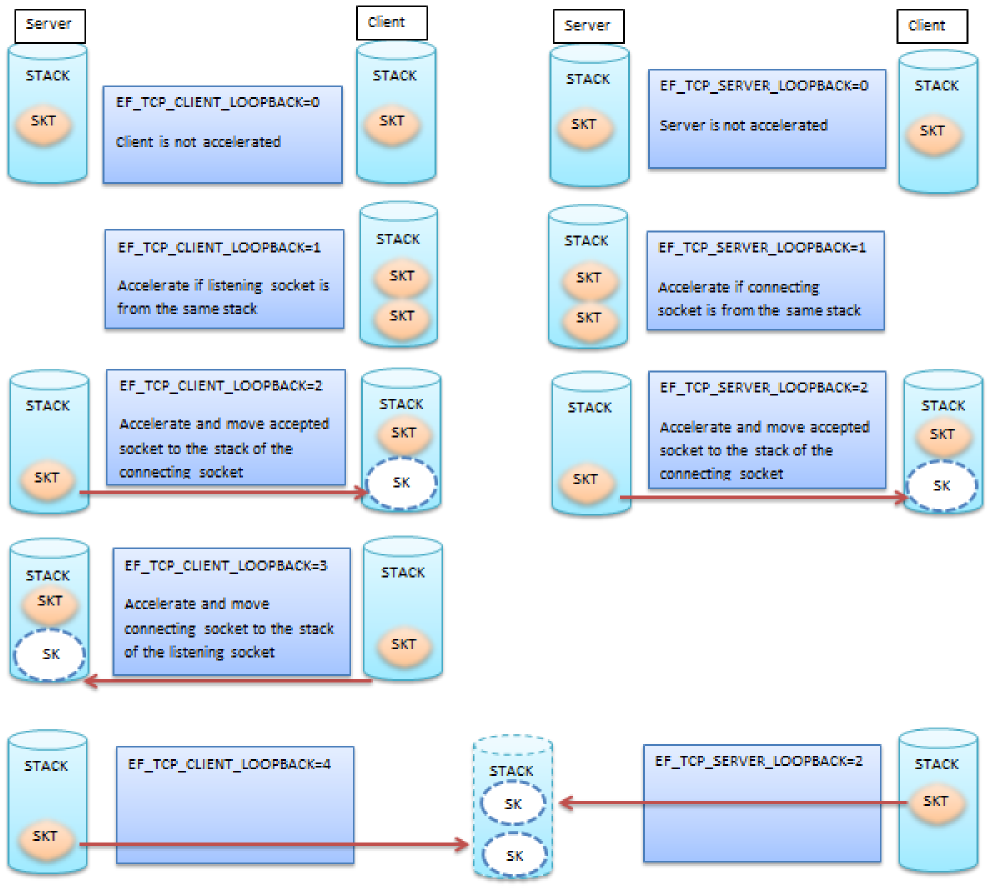

Server
Client
STACK
SKT
EF_TCP_CLIENT_LOOPBACK=0
Client is not accelerated
EF_TCP_SERVER_LOOPBACK=0
Server is not accelerated
EF_TCP_CLIENT_LOOPBACK=1
Accelerate if listening socket is from the same stack
EF_TCP_SERVER_LOOPBACK=1
Accelerate if connecting socket is from the same stack
EF_TCP_CLIENT_LOOPBACK=2
Accelerate and move accepted socket to the stack of the connecting socket
SK
EF_TCP_SERVER_LOOPBACK=2
Accelerate and move accepted socket to the stack of the connecting socket
EF_TCP_CLIENT_LOOPBACK=3
Accelerate and move connecting socket to the stack of the listening socket
EF_TCP_CLIENT_LOOPBACK=4
EF_TCP_SERVER_LOOPBACK=2

Figure 3: 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.

TCP Striping

Onload supports a Solarflare 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 may 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 may 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 therefore supports applications repeatedly calling 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 to 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 new counters to indicate the level of message warm use:

`tx_msg_warm_try` 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.

`tx_msg_warm` is a count of the number of times a message warm send function was called when the send path was exercised.

NOTE: If the ONLOAD_MSG_WARM flag is used on sockets which are not accelerated - including those handed off to the kernel by Onload, it may 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() on page 114).

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.

6.14 UDP Operation

The table below identifies the Onload 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 on page 50

Socket Options

Onload UDP supports the following socket options which can be used in the `setsockopt()` and `getsockopt()` function calls.

Option	Description
`SO_BINDTODEVICE`	bind this socket to a particular network interface. See SO_BINDTODEVICE below.
`SO_BROADCAST`	when enabled datagram sockets can send and receive packets to/from a broadcast address.
`SO_DEBUG`	enable protocol debugging.
`SO_DONTROUTE`	outgoing data should be sent on whatever interface the socket is bound to and not routed via another interface.
`SO_ERROR`	the errno value of the last error occurring on the socket. (Only valid as a `getsockopt()`).
`SO_EXCLUSIVEADDR USE`	prevents other sockets using the SO_REUSEADDR option to bind to the same address and port.
`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 may be processed first depending on the selected device queueing discipline.
`SO_RCVBUF`	sets or gets the maximum socket receive 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. Note that EF_UDP_RCVBUF overrides this value.
`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 i.e. another socket can bind to the same port number except when there is an active listening socket bound to the 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. Note that 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 SO_TIMESTAMP and SO_TIMESTAMPNS (software timestamps) below.
SO_TIMESTAMPNS	enable or disable receiving the SO_TIMESTAMP control message (nanosecond resolution). See SO_TIMESTAMP and SO_TIMESTAMPNS (software timestamps) below.
SO_TIMESTAMPING	enable/disable hardware timestamps for received packets. See SO_TIMESTAMPING (hardware timestamps) on page 50.
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 release notes for Onload specific behaviour regarding these 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.

SO_TIMESTAMP and SO_TIMESTAMPNS (software timestamps)

The `setsockopt()` function call may be passed an option SO_`TIMESTAMP` to enable timestamping on the specified socket. Functions such as `cmesg() recvmsg() and recvmmsg()` can then extract timestamp data for packets received at the socket.

Onload implements a software timestamping mechanism, providing microsecond resolution. Onload software timestamps avoid the need for a per-packet system call and thereby reduce the normal timestamp overheads.

The Solarflare adapter 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 buffer. This implies that the timestamps will reflect the order the packets arrive from the network, but there may be a delay between the packet being received by the hardware and the software timestamp being applied.

Applications preferring timestamps with nanosecond resolution can use SO_`TIMESTAMPNS` in place of the normal (micro second resolution) SO_`TIMESTAMP` value.

SO_TIMESTAMPING (hardware timestamps)

The setsockopt() function call may be passed an option SO_TIMESTAMPING to enable hardware timestamps to be generated by the adapter for each received packet. Functions such as cmesg(), recvmsg() and recvmmsg() can then extract timestamp data for packets received at the socket. This support is only available on SFN7000 series adapters. Before receiving hardware timestamps:

- A valid AppFlex license for hardware timestamps must be installed on the Solarflare Flareon Ultra adapter. The PTP/timestamping license is installed on the SFN7322F during manufacture, such a license can be installed on the SFN7122F adapter by the user.
- The Onload stack for the socket must have the environment variable EF_RX_TIMESTAMPING set - see Appendix A: Parameter Reference on page 92 for settings.

Received packets are timestamped when they enter the MAC on the SFN7000 series adapter.

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:

http://lxr.linux.no/linux/Documentation/networking/timestamping.

SO_BINDTODEVICE

In response to the `setsockopt()` function call with `SO_BINDTODEVICE`, sockets identifying non-Solarflare interfaces will be handled by the kernel and all sockets identifying Solarflare interfaces will be handled by Onload. All sends from a socket are sent via the bound interface and all TCP, UDP and Multicast packets received via the bound interface are delivered only to the socket bound to the interface.

UDP Send and Receive Paths

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.

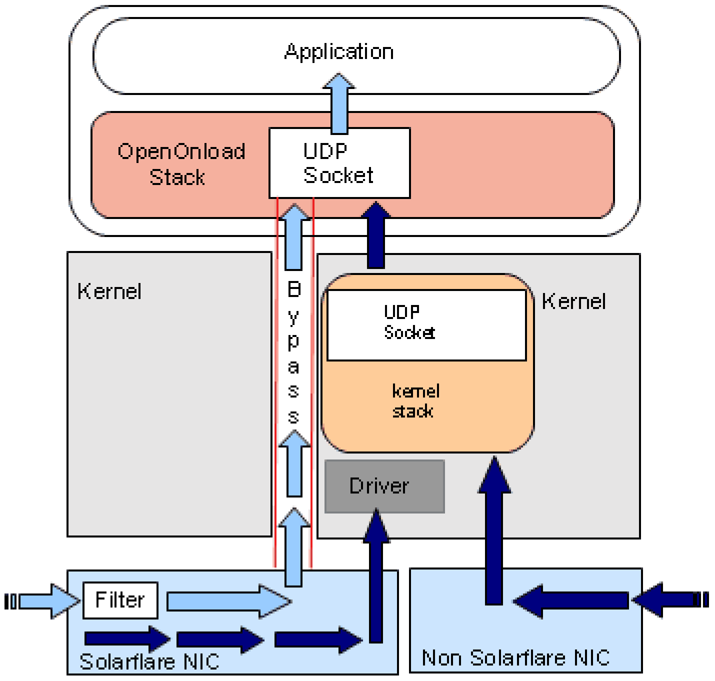


Figure 4: UDP Send and Receive Paths

Figure 4 illustrates the UDP send and receive paths. Lighter arrows indicate the accelerated ’kernel bypass’ path. Darker arrows identify fragmented UDP packets received by the Solarflare adapter and

UDP packets received from a non-Solarflare adapter. UDP packets arriving at the Solarflare 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 a non-Solarflare 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.

6.15 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.

6.16 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.

6.17 Multiplexed I/O

Linux supports various common methods for handling multiplexed I/O operation:

- poll(), ppoll()
- select(), pselect()
- the epoll set of functions.

The general behaviour of the `poll(), ppoll(), select(), pselect(), epoll_wait() and epoll_pwait()`functions with Onload is as follows:

- If there are operations ready on any Onload accelerated file descriptor these functions will return immediately. Refer to the relevant subsections below for specific behaviour details.
- If there are no Onload accelerated file descriptors ready, and spinning is not enabled these functions will enter the kernel and block.
- If the set contains file descriptors that are not accelerated sockets, Onload must make a system call to determine the readiness of these sockets. 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 behaviour 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:

Value	Behaviour
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 in Appendix A: Parameter Reference on page 93.

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:

Value	Epoll Behaviour
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 in Appendix A: Parameter Reference on page 93.

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 behaviour.

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:

Value	Epoll Behaviour
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.
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.
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.

The relative performance of epoll options 1 and 2 depends on the details of application behaviour as well as the number of accelerated file descriptors in the epoll set. Behaviour may 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 may result in performance (latency spikes). A kernel-level CPU intensive process is less likely to be de-scheduled resulting in better performance. Solarflare recommend the user evaluate both options for an applications that manages many file descriptors.

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 in Appendix A: Parameter Reference on page 93.

Refer to epoll - Known Issues on page 78.

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

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 may impact the other, or one process can gain access to the other's privileged information (i.e. 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 on page 93 for a description of the EF_NAME and EF_SHARE_WITH variables.

Processes sharing an Onload stack should also not 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.

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 on page 112 for details.

6.19 Multicast Replication

The Solarflare SFN7000 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. 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.

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.

6.20 Multicast Operation and Stack Sharing

To illustrate shared stacks, the following examples describe Onload behaviour when two processes, on the same host, subscribe to the same multicast stream:

- Multicast Receive Using Different Onload Stacks...Page 57
- Multicast Transmit Using Different Onload Stacks...Page 57
- Multicast Receive Sharing an Onload Stack...Page 58
- Multicast Transmit Sharing an Onload Stack...Page 58
- Multicast Receive - Onload Stack and Kernel Stack...Page 58.

NOTE: The following subsections use two processes to demonstrate Onload behaviour. 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 Receive Using Different Onload Stacks

Onload will notice if two Onload stacks on the same host subscribe to the same multicast stream and will respond by redirecting the stream to go through the kernel. Handing the stream to the kernel, though still using Onload stacks, allows both subscribers to receive the datagrams, but user-space acceleration is lost and the receive rate is lower that it could otherwise be. Figure 5 below illustrates the configuration. Arrows indicate the receive path and fragmented UDP path.

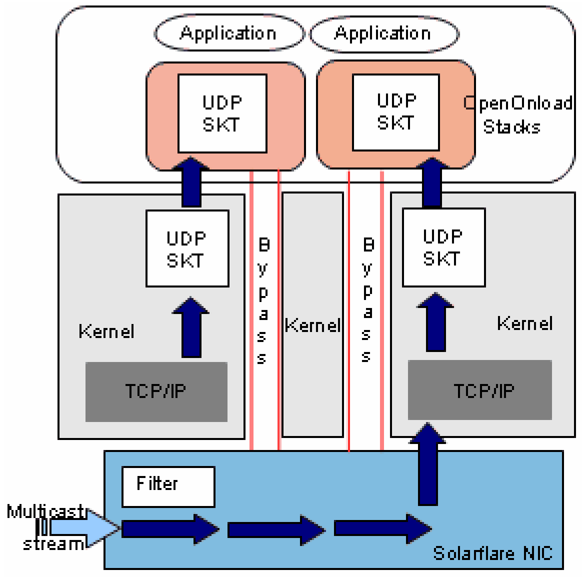


Figure 5: Multicast Receive Using Different Onload Stacks.

The reason for this behaviour is because the Solarflare NIC will not deliver a single received multicast packet multiple times to multiple stacks – the packet is delivered only once. If a received packet is delivered to kernel-space, then the kernel TCP/IP stack will copy the received data multiple times to each socket listening on the corresponding multicast stream. If the received packet were delivered directly to Onload, where the stacks are mapped to user-space, it would only be delivered to a single subscriber of the multicast stream.

Multicast Transmit Using Different Onload Stacks

Referring to Figure 5, 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.

Onload by default changes the default state of the `IP_MULTICAST_LOOP` socket option to `0` rather than `1`. This change allows Onload to accelerate multicast transmit for most applications, but means that multicast traffic is not delivered to other applications on the same host unless the subscriber sockets are in the same stack. The normal behaviour can be restored by setting

EF_FORCE_SEND_MULTICAST=0, but this limits multicast acceleration on transmit to sockets that have manually set the IP_MULTICAST_LOOP socket option to zero.

Multicast Receive Sharing an Onload Stack

Setting the EF_NAME environment variable to the same string in both processes means they can share an Onload stack. The stream is no longer redirected through the kernel resulting in a much higher receive rate than can be observed with the kernel TCP/IP stack (or with separate Onload stacks where the data path is via the kernel TCP/IP stack). This configuration is illustrated in Figure 6 below. Lighter arrows indicate the accelerated (kernel bypass) path. Darker arrows indicate the fragmented UDP path.

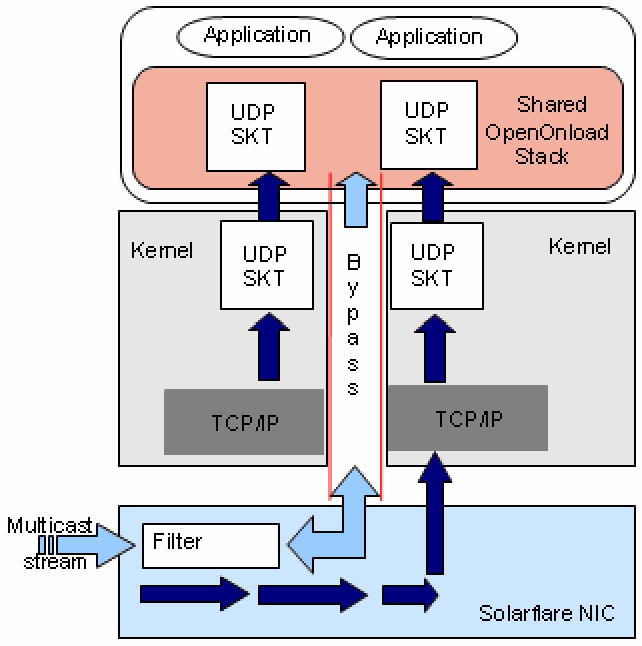


Figure 6: Sharing an Onload Stack

Multicast Transmit Sharing an Onload Stack

Referring to Figure 6, datagrams transmitted by one process would be received by the second process because both processes share the Onload stack.

Multicast Receive - Onload Stack and 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.

To enable multiple applications accelerated with Onload to subscribe to the same multicast stream, the applications must share the same Onload stack. Stack sharing is achieved by using the EF_NAME environment variable.

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.

6.21 Multicast Loopback

The socket option IP_MULTICAST_LOOP controls whether multicast traffic sent on a socket can be received locally on the machine. With Onload, the default value of the IP_MULTICAST_LOOP socket option is 0 (the kernel stack defaults IP_MULTICAST_LOOP to 1). Therefore by default with Onload multicast traffic sent on a socket will not be received locally.

As well as setting IP_MULTICAST_LOOP to 1, 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. If one local process is to receive the data sent by a sending local process, EF_MULTICAST_LOOP_OFF must be disabled (set to zero) on the sender.

6.22 Bonding, Link aggregation and Failover

Bonding (aka teaming) allows for improved reliability and increased bandwidth by combining physical ports from one or more Solarflare 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.

e.g. 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.

To disable Onload acceleration of bonds set `CI_CFG_TEAMING=0` in the file `transport_config_opt.h` at compile time.

Refer to the Limitations section, Bonding, Link aggregation on page 75 for further information.

6.23 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.

e.g. to add an interface for VLAN 5 over an SFC interface (eth2)

```
onload_loaddrivers
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 on page 75 for further information.

6.24 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 2Kbyte socket buffers and expanded as size requires to a maximum size of 64Kbytes.

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 on page 93.

NOTE: Only anonymous pipes created with the `pipe()` or pipe2() function calls will be accelerated.

6.25 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 on page 123 for detailed descriptions and example source code of the API.

6.26 Receive Filtering

The Onload Extensions API supports a Receive Filtering API which allows user-defined filters to determine if data received on a UDP socket should be discarded before it enters the socket receive buffer. Refer to the Receive Filtering API Overview on page 134 for a detailed description and example source code of the API.

6.27 Packet Buffers

Introduction

Packet buffers describe the memory used by the Onload stack (and Solarflare 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.

While the adapter's buffer table is sufficient for the majority of applications, on adapters prior to the SFN7000 series, it is limited to approximately 120,000 x 2Kbyte buffers which have to be shared between all Onload stacks.

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 SFN7000 series adapters alleviate the packet buffer limitations of previous generation Solarflare adapters and 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 SFN7000 series adapter can describe a 4kbyte, 64kbyte, 1Mbyte or 4Mbyte block of memory where each table entry is the page size as directed by the operating system.

Scalable Packet Buffer Mode

Scalable Packet Buffer Mode is an alternative packet buffer mode which allows a much higher number of packet buffers to be used by Onload. Using the Scalable Packet Buffer Mode Onload stacks employ Single Root I/O Virtualization (SR-IOV) virtual functions (VF) to provide memory protection and translation. This mechanism removes the 120K buffers limitation imposed by the Network Adapter Buffer Table Mode.

For deployments where using SR-IOV and/or the IOMMU is not an option, Onload also supports an alternative Scalable Packet Buffer Mode scheme called Physical Addressing Mode. Physical addressing also removes the 120K packet buffer limitation, however physical addressing does not provide the memory protection provided by SR-IOV and an IOMMU. For details of Physical Addressing Mode see Physical Addressing Mode on page 70.

NOTE: Enabling SR-IOV, which is needed for Scalable Packet Buffer Mode, has a latency impact which depends on the adapter model. For the SFN5000 adapter series, latency increases by approximately 50ns for the 1/2 RTT latency. The SFN6000 adapter series has equivalent latency to the SFN5000 adapter series when operating in this mode.

NOTE: SR-IOV and therefore Scalable Packet Buffer Mode is not supported on the Solarflare SFN7122F network adapter at this time, but will be made available in a future revision.

NOTE: SR-IOV and therefore Scalable Packet Buffer Mode is not supported on the Solarflare SFN4112F network adapter.

NOTE: MRG users should refer to Red Hat MRG 2 and SR-IOV on page 83.

For further details on SR-IOV configuration refer to Configuring Scalable Packet Buffers on page 66.

How Packet Buffers Are Used by Onload

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 receive and transmit socket buffers.
4. TCP sockets can also hold packet buffers in the socket's retransmit queue and in the reorder queue.

NOTE: User-level pipes do not 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 e.g 10
- each stack has multiple sockets e.g 6
- and each socket uses many packet buffers e.g 2000

This would require a total of 120000 packet buffers

Example 2:

If on a stack the TCP receive queue is 1 Mbyte 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 200 Kbytes where received datagrams are each 200 bytes would hold 1000 packet buffers.

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 120K 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.

For large scale applications the Scalable Packet Buffer Mode removes the limitations imposed by the network adapter buffer table. See Configuring Scalable Packet Buffers on page 66 for details.

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 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 that `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 on page 93 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 i.e. the Onload stack is under memory pressure. Only 9 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 as a result of memory pressure.

```
memory_pressure: 1
memory_pressure_drops: 22096
```

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 on page 93 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 may 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.

Configuring Scalable Packet Buffers

NOTE: SR-IOV and therefore Scalable Packet Buffer Mode is not currently supported on the SFN7000 series adapter but will be available in a future release.

Using the Scalable Packet Buffer Mode Onload stacks are bound to virtual functions (VFs) and provide a PCI SR-IOV compliant means to provide memory protection and translation. VFs employ the kernel IOMMU.

Refer to Chapter 7 and Scalable Packet Buffer Mode on page 83 for 32-bit kernel limitations.

Procedure:

Step 1. Platform Support

Scalable Packet Buffer Mode is implemented using SR-IOV, support for which is a relatively recent addition to the Linux kernel. There were several kernel bugs in early incarnations of SR-IOV support, up to and including kernel.org 2.6.34. The fixes have been back-ported to recent Red Hat kernels. Users are advised to enable scalable packet buffer mode on Red Hat kernel 2.6.32-131.0.15 or later, or kernel.org 2.6.35 or later. In other distributions, it is recommended that the most recent patched kernel version is used

- The system hardware must have an IOMMU and this must be enabled in the BIOS.
- The kernel must be compiled with support for IOMMU and kernel command line options are required to select the IOMMU mode.
- The kernel must be compiled with support for SR-IOV APIs (`CONFIG-PCI-IOV`).
- SR-IOV must be enabled on the network adapter using the `sfboot` utility.
- When more than 6 VFs are needed, the system hardware and kernel must support PCIe Alternative Requester ID (ARI) - a PCIe Gen 2 feature.
- Onload options EF_PACKET_BUFFER_MODE=1 must be set in the environment.

NOTE: The Scalable Packet Buffer feature can be susceptible to known kernel issues observed on RHEL6 and SLES 11. (See http://www.spinics.net/lists/linux-pci/msg10480.html for details. The condition can result in an unresponsive server if `intel_iommu` has been enabled in the `grub.conf` file, as per the procedure at Step 2. BIOS and Linux Kernel Configuration on page 67, and if the Solarflare sfc_resource driver is reloaded. This issue has been addressed in newer kernels

Step 2. BIOS and Linux Kernel Configuration

To use SR-IOV, hardware virtualization must be enabled. Refer to RedHat Enabling Intel VT-x and AMD-V Virtualization in BIOS for more information. Take care to enable VT-d as well as VT on an Intel platform.

To verify that the extensions have been correctly enabled refer to RedHat Verifying virtualization extensions.For best kernel configuration performance and to avoid kernel bugs exhibited when IOMMU is enabled for all devices, Solarflare recommend the kernel is configured to use the IOMMU in pass-through mode - append the following lines to kernel line in the `/boot/grub/grub.conf` file:

On an Intel system:

```
intel_iommu=on iommu=on,pt
```

On an AMD system:

```
amd_iommu=on, iommu=on,pt
```

In pass-through mode the IOMMU is bypassed for regular devices. Refer to Red Hat: PCI passthrough for more information.

NOTE: On Linux Red Hat 5 servers (2.6.18) it is necessary to also use the `iommu_type=2` option.

NOTE: EnterpriseOnload v2.1.0.0 users and OpenOnload v201109-u2 (onwards) users:

Recent kernels are compiled with support for IOMMUs by default, but unfortunately the realtime (-rt) kernel patches are not currently compatible with IOMMUs (Red Hat MRG kernels are compiled with CONFIG_PCI_IOV disabled). It is possible to use scalable packet buffer mode on some systems without IOMMU support, but in an insecure mode. In this configuration the IOMMU is bypassed, and there is no checking of DMA addresses provided by Onload in user-space. Bugs or mis-behaviour of user-space code can compromise the system.

To enable this insecure mode, set the Onload module option `unsafe_sriov_without_iommu=1` for the sfc_resource kernel module.

Linux MRG users are urged to use MRGu2 and kernel 3.2.33-rt50.66.el6rt.x86_64 or later to avoid known issues and limitations of earlier versions.

The `unsafe_sriov_without_iommu` option is obsoleted in OpenOnload 201210. It is replaced by physical addressing mode - see Physical Addressing Mode on page 70 for details.

Step 3. Update adapter firmware and enable SR-IOV

1 Download and install the Solarflare Linux Utilities RPM from support.solarflare.com and unzip the utilities file to reveal the RPM:

2 Install the RPM:

```
# rpm -Uvh sfutils-<version>.rpm
```

3 Identify the current firmware version on the adapter:

```
# sfupdate
```

4 Upgrade the adapter firmware with `sfupdate`:

```
# sfupdate --write
```

Full instructions on using `sfupdate` can be found in the Solarflare Network Server Adapter User Guide.

5 Use `sfboot` to enable SR-IOV and enable the VFs. You can enable up to 127 VFs per port, but the host BIOS may only be able to support a smaller number. The following example will configure 16 VFs on each Solarflare port:

```
# sfboot sriov=enabled vf-count=16 vf-msix-limit=1
```

Option	Default Value	Description
`sriov=<enabled | disabled>`	Disabled	Enable/Disable hardware SRIOV support
`vf-count=<n>`	127	Number of virtual functions advertised per port. *See the note below.*
`vf-msix-limit=<n>`	1	Number of MSI-X interrupts per VF

6 It is necessary to reboot the server following changes using sfboot and sfupdate.

NOTE: Enabling all 127 VFs per port with more than one MSI-X interrupt per VF may not be supported by the host BIOS. If the BIOS doesn't support this then you may get 127 VFs on one port and no VFs on the other port. You should contact your BIOS vendor for an upgrade or reduce the VF count.

NOTE: On Red Hat 5 servers the vf-count should not exceed 32.

NOTE: VF allocation must be symmetric across all Solarflare interfaces.

Step 4. Enable VFs for Onload

```
#export EF_PACKET_BUFFER_MODE=1
```

Refer to Appendix A: Parameter Reference on page 93 for other values.

Step 5. Check PCIe VF Configuration

The network adapter sfc driver will initialize the VFs, which can be displayed by the `lspci` command:

```
# lspci -d 1924:

05:00.0 Ethernet controller: Solarflare Communications SFC9020
[Solarflare]
05:00.1 Ethernet controller: Solarflare Communications SFC9020
[Solarflare]
05:00.2 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
```

```
05:00.3 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:00.4 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:00.5 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:00.6 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:00.7 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:01.0 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
05:01.1 Ethernet controller: Solarflare Communications SFC9020 Virtual
Function [Solarflare]
```

The lspci example output above identifies one physical function per physical port and the virtual functions (four for each port) of a single Solarflare dual-port network adapter.

Step 6. Check VFs in onload_stackdump

The `onload_stackdump netif` command will identify VFs being used by Onload stacks as in the following example:

```
# onload_stackdump netif
ci_netif_dump: stack=0 name=
  ver=201109 uid=0 pid=3354
  lock=10000000 UNLOCKED   nics=3 primed=3
  sock_bufs: max=1024 n_allocated=4
  pkt_bufs: size=2048 max=32768 alloc=1152 free=128 async=0
  pkt_bufs: rx=1024 rx_ring=1024 rx_queued=0
  pkt_bufs: tx=0 tx_ring=0 tx_oflow=0 tx_other=0
  time: netif=3df7d2 poll=3df7d2 now=3df7d2 (diff=0.000sec)
ci_netif_dump_vi: stack=0 intf=0 vi=67 dev=0000:05:01.0 hw=0C0
  evq: cap=2048 current=8 is_32_evs=0 is_ev=0
  rxq: cap=511 lim=511 spc=15 level=496 total_desc=0
  txq: cap=511 lim=511 spc=511 level=0 pkts=0 oflow_pkts=0
  txq: tot_pkts=0 bytes=0
ci_netif_dump_vi: stack=0 intf=1 vi=67 dev=0000:05:01.1 hw=0C0
  evq: cap=2048 current=8 is_32_evs=0 is_ev=0
  rxq: cap=511 lim=511 spc=15 level=496 total_desc=0
  txq: cap=511 lim=511 spc=511 level=0 pkts=0 oflow_pkts=0
  txq: tot_pkts=0 bytes=0
```

The output above corresponds to VFs advertised on the Solarflare network adapter interface identified using the `lspci` command - Refer to Step 5 above.

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. OpenOnload versions earlier than onload-201210 and EnterpriseOnload-2.1.0.0 are limited to 1 million packet buffers. This limit was raised to 2 million packets buffers in 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 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 on page 93 for a complete description of all EF_PACKET_BUFFER_MODE options.

6.28 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 7000 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 utilisation. 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.

PIO buffers on the adapter are limited to a maximum of 8 Onload stacks. 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 on page 138 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.

When both accelerated and non-accelerated sockets are using PIO, the number of PIO buffers available to Onload stacks can be calculated from the total 16 available PIO regions:

	description	example value
piobuf_size	driver module parameter	256
rss_cpus	driver module parameter	4
region	a chunk of memory 2048 bytes	2048 bytes

Using the above example values, each port on the adapter requires:

piobuf_size * rss_cpus / 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.

6.29 Templated Sends

“Templated sends” is another SFN7000 series adapter feature that builds on top of TX PIO to provide further transmit latency improvements. This 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 may publish packets that vary only in the specific value of certain fields, possibly different symbols and price information, but are otherwise identical. Templated sends involve creating 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.

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.

The API details are available in the Onload 201310 distribution at `/src/include/onload/extensions_zc.h`

Refer to Appendix D: Onload Extensions API for further information on the use of packet templates including code examples of using this feature.

6.30 Debug and Logging

Onload support 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`.

Log Levels:

- `EF_UNIX_LOG` - refer to Appendix A: Parameter Reference on page 93 for details.
- `TP_LOG` (bitmask) - useful for stack debugging. See Onload source code `/src/include/ci/internal/ip_log.h` for bit values.
- Onload module options:
  - oo_debug_bits=[bitmask] - useful for kernel logging and events not involving an onload stack. See `src/include/onload/debug.h` for bit values.
  - 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.

Chapter 7: Limitations

Users are advised to read the latest release_notes distributed with the Onload release for a comprehensive list of Known Issues.

7.1 Introduction

This chapter outlines configurations that Onload does not accelerate and ways in which Onload may change behaviour of the system and applications. It is a key goal of Onload to be fully compatible with the behaviour of the regular kernel stack, but there are some cases where behaviour deviates.

7.2 Changes to Behavior

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.

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 Solarflare supplied `onload_tcpdump` does support capture of UDP and TCP packets from Onload stacks - Refer to Appendix G: onload_tcpdump on page 164 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 on page 174.

NOTE: Hardware filtering on the network adapter will ensure that accelerated applications receive traffic only on ports to which they are bound.

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 Onload and File Descriptors, Stacks and Sockets on page 36 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 on page 137.

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 may cause TCP connections to time-out.

A signal which terminates an application can prevent threads from exiting cleanly. Refer to Multithreaded Applications Termination on page 73 for more information.

Undefined content may 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.

7.3 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 on page 52 for a description of Onload behaviour.

Broadcast Traffic

Broadcast sends and receives function as normal but will not be accelerated. Multicast traffic can be accelerated.

IPv6 Traffic

IPv6 traffic functions as normal but will not be accelerated.

Raw Sockets

Raw Socket sends and receives function as normal but will not be accelerated.

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 Solarflare network devices. Adding a non-Solarflare network device to a bond that is currently accelerated by Onload may 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.

In cases where Onload will not accelerate the traffic it will continue to work via the OS network stack.

For more information and details of configuration options refer to the Solarflare Server Adapter User Guide section 'Setting Up Teams'.

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. i.e. 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 interface is part of a bond (team) and also on a VLAN, network traffic will not be accelerated on this interface or any other interface on the same adapter and same VLAN.
- Hardware filters installed by Onload on the Solarflare adapter will only consider 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.

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 Solarflare Server Adapter User Guide section 'Setting Up VLANs'.

NOTE: The `onload_tool reload` command will unload then reload the adapter driver removing all physical devices and associated VLAN devices.

TCP RTO During Overload Conditions

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 and results in RTOs for the unacknowledged packets.

Enabling the per stack environment variable `EF_INT_DRIVEN` can reduce the likelihood of this behaviour 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.

Transmission Path - Packet Loss

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 3 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. Solarflare recommend setting 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.

7.4 epoll - Known Issues

Onload supports different implementations of epoll controlled by the EF_UL_EPOLL environment variable - see Multiplexed I/O on page 52 for configuration details.

- When using EF_UL_EPOLL=1, 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 TCP listening sockets and UDP sockets, but not for TCP connected 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.
- 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 or 2.

  If the application is monitoring the state of the epoll file descriptor indirectly, e.g. by monitoring the epoll fd with poll, then EF_EPOLL_CTL_FAST can cause issues and should be set to zero.
- A socket should removed from an epoll set only when all references to the socket are closed.

  With EF_UL_EPOLL=1 (default) 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()`. 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 */
  ```

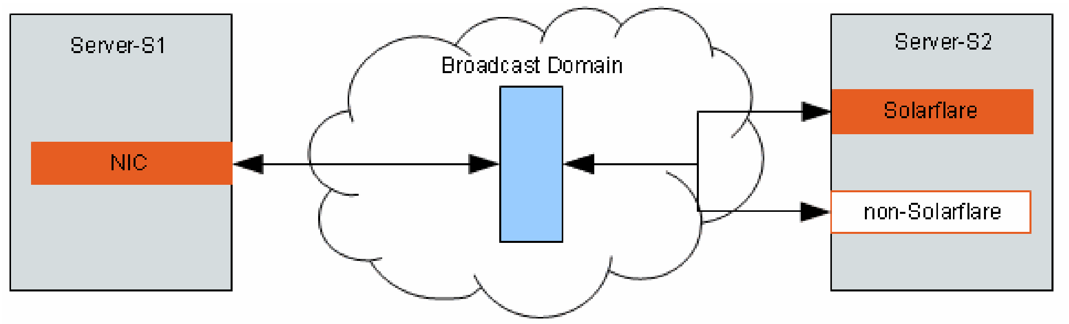
  Workaround is set EF_UL_EPOLL=2.
- When Onload is unable to accelerate a connected socket, e.g. because no route to the destination exists which uses a Solarflare interface, the socket will be handed off to the kernel and is removed from the epoll set. Because the socket is no longer in the epoll set, 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.
- 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. The problem exists for both implementations of epoll.
- When using the receive filtering API (see Receive Filtering API Overview on page 134) I/O multiplex functions such as `poll() and select()` may return that a socket is readable, but a subsequent call to read() can fail because the filter has rejected the packets. Affected system calls are `poll(), epoll(), select()` and `pselect()`.

7.5 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[1] as depicted by the following diagram.



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 may 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 may or may 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 either (i) 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, or (ii) a switch within the broadcast domain broadcasts the TCP packets to all interfaces.

1.A Broadcast domain can be a local network segment or VLAN.

Virtual Memory on 32 Bit Systems

On 32 bit Linux systems the amount of allocated virtual address space defaults, typically, to 128Mb which limits the number of Solarflare interfaces that can be configured. Virtual memory allocation can be identified in the `/proc/meminfo` file e.g.

```
grep Vmalloc /proc/meminfo
VmallocTotal: 122880 kB
VmallocUsed:   76380 kB
VmallocChunk:  15600 kB
```

The Onload driver will attempt to map all PCI Base Address Registers for each Solarflare interface into virtual memory where each interface requires 16Mb.

Examination of the kernel logs in `/var/log/messages` at the point the Onload driver is loading, would reveal a memory allocation failure as in the following extract:

```
allocation failed: out of vmalloc space - use vmalloc=<size> to increase
size.
[sfc efrm] Failed (-12) to map bar (16777216 bytes)
[sfc efrm] efrm_nic_add: ERROR: linux_efrm_nic_ctor failed (-12)
```

One solution is to use a 64 bit kernel. Another is to increase the virtual memory allocation on the 32 bit system by setting vmalloc size on the 'kernel line' in the `/boot/grub/grub.conf` file to 256, for example,

```
kernel /vmlinuz-2.6.18-238.el5 ro root=/dev/sda7 vmalloc=256M
```

The system must be rebooted for this change to take effect.

Hardware 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. These physical resources include:

1. 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 run out.
2. Filters. Filters are used to demultiplex packets received from the wire to the appropriate application. When these are exhausted it is not possible to create new accelerated sockets. Solarflare recommend that applications do not allocate more than 4096 filters.
3. Buffer table entries. The buffer table provides address protection and translation for DMA buffers. When these are exhausted it is not possible to create new Onload stacks, and existing stacks are not able to allocate more DMA buffers.

When any of these resources are exhausted, normal operation of the system should continue, but it will not be possible to accelerate new sockets or applications.

Under severe conditions, after resources are exhausted, it may not be possible to send or receive traffic resulting in applications getting 'stuck'. The `onload_stackdump` utility should be used to monitor hardware resources.

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 may 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.

Scalable Packet Buffer Mode

Support for SR-IOV is disabled on 32-bit kernels, therefore the following features are not available on 32-bit kernels.

- Scalable Packet Buffer Mode (EF_PACKET_BUFFER_MODE=1)
- ef_vi with VFs

On some recent kernel versions, configuring the adapter to have a large number of VFs (via sfboot) can cause kernel panics. This problem affects kernel versions in the range 3.0 to 3.3 inclusive and is due to the netlink messages that include information about network interfaces growing too large.

The problem can be avoided by ensuring that the total number of physical network interfaces, including VFs, is no more than 30.

Scalable Packet Buffer Mode on SFN7122F and SFN7322F

SR-IOV and therefore Scalable Packet Buffer Mode is not currently supported on the SFN7000 series adapters. This feature will be supported in a future release.

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 may result in a segfault.

Huge Pages with Shared Stacks

Processes which share an Onload stack should not attempt to use huge pages. Refer to Stack Sharing on page 55 for limitation details.

Huge Pages - Size

When using huge pages, it is recommended to avoid setting the page size greater than 2 Mbyte. A failure to observe this could lead to Onload unable to allocate further buffer table space for packet buffers.

Red Hat MRG 2 and SR-IOV

EnterpriseOnload from version 2.1.0.1 includes support for Red Hat MRG2 update 3 and the 3.6.11-rt kernel. Solarflare do not recommend the use of SR-IOV or the IOMMU when using Onload on these systems due to a number of known kernel issues. The following Onload features should not be used on MRG2u3:

- Scalable packet buffer mode (EF_PACKET_BUFFER_MODE=1)
- ef_vi with VFs

PowerPC Architecture

- 32 bit applications are known not to work correctly with onload-201310. This has been corrected in onload-201310-u1.
- SR-IOV is not supported by onload-201310 on PowerPC systems.
- PowerPC architectures do not currently support PIO for reduced latency. EF_PIO should be set to zero.

Java 7 Applications - use of vfork()

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.

Chapter 8: Change History

This chapter provides a brief history of changes, additions and removals to Onload releases affecting Onload behaviour 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**)

8.1 Features

Feature	OOL	EOL	Description/Notes
SO_TIMESTAMPING	201310-u1	NS	Socket option to receive hardware timestamps for received packets.
onload_fd_check_feature()	201310-u1	NS	onload_fd_check_feature() on page 114
4.0.2.6628 net driver	201310-u1	NS	Net driver supporting SFN5xxx, 6xxx and 7xxx series adapters introducing hardware packet timestamps and PTP on 7xxx series adapters.
4.0.0.6585 net driver	201310	NS	Net driver supporting SFN5xxx, 6xxx and 7xxx series adapters and Solarflare PTP and hardware packet timestamps.
Multicast Replication	201310	NS	Multicast Replication on page 56
TX PIO	201310	NS	Programmed I/O on page 70
Large Buffer Table Support	201310	NS	Large Buffer Table Support on page 62
Templated Sends	201310	NS	Templated Sends on page 71
ONLOAD_MSG_WARM	201310	NS	ONLOAD_MSG_WARM on page 46
SO_TIMESTAMP SO_TIMESTAMPNS	201310	NS	Supported for TCP sockets
dup3()	201310	NS	Onload wil intercept calls to create a copy of a file descriptor using dup3().
3.3.0.6262 net driver	NS	2.1.0.1	Support Solarflare Enhanced PTP (sfptpd).
IP_ADD_SOURCE_MEMBERSHIP	201210-u1	NS	Join the supplied multicast group on the given interface and accept data from the supplied source address.

Feature	OOL	EOL	Description/Notes
IP_DROP_SOURCE_MEMBERSHIP	201210-u1	NS	Drops membership to the given multicast group, interface and source address.
MCAST_JOIN_SOURCE_GROUP	201210-u1	NS	Join a source specific group.
MCAST_LEAVE_SOURCE_GROUP	201210-u1	NS	Leave a source specific group.
3.3.0.6246 net driver	201210-u1	NS	Support Solarflare Enhanced PTP (sfptpd).
Huge pages support	201210	NS	Packet buffers use huge pages. Controlled by EF_USE_HUGE_PAGES Default is 1 - use huge pages if available See Limitations on page 73
onload_iptables	201210	NS	Apply Linux iptables firewall rules or user-defined firewall rules to Solarflare interfaces
onload_stackdump processes onload_stackdump affinities onload_stackdump env	201210	NS	Show all accelerated processes by PID Show CPU core accelerated process is running on Show environment variables - `EF_VALIDATE_ENV`
Physical addressing mode	201210	NS	Allows a process to use physical addresses rather than controlled I/O addresses. Enabled by EF_PACKET_BUFFER_MODE 2 or 3
UDP `sendmmsg()`	201210	NS	Send multiple msgs in a single function call
I/O Multiplexing	201210	NS	Support for `ppoll()`, `pselect()` and `epoll_pwait()`
DKMS	201210	NS	OpenOnload available in DKMS RPM binary format
3.2.1.6222B net driver	201210	NS	OpenOnload only
3.2.1.6110 net driver	NS	2.1.0.0	EnterpriseOnload only
3.2.1.6099 net driver	201205-u1	NS	
Removing zombie stacks	201205-u1	2.1.0.0	`onload_stackdump -z` kill will terminate stacks lingering after exit
Compatibility	201205-u1	2.1.0.0	Compatibility with RHEL6.3 and Linux 3.4.0
TCP striping	201205	2.1.0.0	Single TCP connection can use the full bandwidth of both ports on a Solarflare adapter
TCP loopback acceleration	201205	2.1.0.0	`EF_TCP_CLIENT_LOOPBACK` & `EF_TCP_SERVER_LOOPBACK`
TCP delayed acknowledgments	201205	2.1.0.0	`EF_DYNAMIC_ACK_THRESH`
TCP reset following RTO	201205	2.1.0.0	`EF_TCP_RST_DELAYED_CONN`

Configure control plane tables	201205	2.1.0.0	`max_layer_2_interface`
			`max_neighs`
			`max_routes`

Onload adapter support	201109-u2	2.0.0.0	Onload support for SFN5322F & SFN6x22F

Feature	OOL	EOL	Description/Notes
Accelerate `pipe2()`	201109-u2	2.0.0.0	Accelerate `pipe2()` function call
`SOCK_NONBLOCK` `SOCK_CLOEXEC`	201109-u2	2.0.0.0	TCP socket types
Extensions API	201109-u2	2.0.0.0	Support for `onload_thread_set_spin()`
3.2 net driver	201109-u1	2.0.0.0	
Onload_tcpdump	201109	2.0.0.0	
Scalable Packet Buffer	201109	2.0.0.0	`EF_PACKET_BUFFER_MODE=1`
Zero-Copy UDP RX	201109	2.0.0.0	
Zero-Copy TCP TX	201109	2.0.0.0	
Receive filtering	201109	2.0.0.0	
`TCP_QUICKACK`	201109	2.0.0.0	`setsockopt()` option
Benchmark tool sfnettest	201109	2.0.0.0	Support for sfnt-stream
3.1 net driver	201104		
Extensions API	201104	2.0.0.0	Initial publication
`SO_BINDTODEVICE` `SO_TIMESTAMP` `SO_TIMESTAMPNS`	201104	2.0.0.0	`setsockopt()` and `getsockopt()` options
Accelerated `pipe()`	201104	2.0.0.0	Accelerate `pipe()` function call
UDP `recvmmsg()`	201104	2.0.0.0	Deliver multiple msgs in a single function call
Benchmark tool sfnettest	201104	2.0.0.0	Supports only sfnt-pingpong

8.2 Environment Variables

Variable	OOL	EOL	Changed	Notes
`EF_TX_PUSH_THRESHOLD`	201310_u1	NS		Improve EF_TX_PUSH low latency transmit feature.
`EF_RX_TIMESTAMPING`	201310_u1	NS		Control of receive packet hardware timestamps.
`EF_RETRANSMIT_THRESHOLD_SYNACK`	201104	1.0.0.0	201310-u1	Default changed from 4 to 5.
`EF_PIO`	201310	NS		Enable/disable PIO Default value 1.

Variable	OOL	EOL	Changed	Notes
EF_PIO_THRESHOLD	201310	NS		Identifies the largest packet size that can use PIO. Default value is 1514.
EF_VFORK_MODE	201310	NS		Dictates how vfork() intercept should work.
EF_FREE_PACKETS_LOW_WATERMARK	201310	NS		Level of free packets to be retained during runtime.
EF_TCP_SNDBUF_MODE	201310	2.0.0.6		Limit TCP packet buffers used on the send queue and retransmit queue.
EF_TXQ_SIZE			201310	Limited to 2048 for SFN7000 series.
EF_MAX_ENDPOINTS	201104	1.1.0.3	201310	Default changed to 1024 from 10.
EF_SO_TIMESTAMP_RESYNC_TIME	201104	2.1.0.1	201310	Removed from OOL.
EF_SIGNALS_NOPOSTPONE	201210-u1	2.1.0.1		Prevent the specified list of signals from being postponed by onload.
EF_FORCE_TCP_NODELAY	201210	NS		Force use of TCP_NODELAY.
EF_USE_HUGE_PAGES	201210	NS		Enables huge pages for packet buffers.
EF_VALIDATE_ENV	201210	NS		Will warn about obsolete or misspelled options in the environment Default value 1.
EF_PD_VF	201205-u1	2.1.0.0	201210	Allocate VIs within SR-IOV VFs to allocate unlimited memory. Replaced with new options on EF_PACKET_BUFFER_MODE
EF_PD_PHYS_MODE	201205_u1	2.1.0.0	201210	Allows a VI to use physical addressing rather than protected I/O addresses Replaced with new options on EF_PACKET_BUFFER_MODE
EF_MAX_PACKETS	20101111	1.0.0.0	201210	Onload will round the specified value up to the nearest multiple of 1024.
EF_EPCACHE_MAX	20101111	1.0.0.0	201210	Removed from OOL
EF_TCP_MAX_SEQERR_MSGS		NS	201210	Removed
EF_STACK_LOCK_BUZZ	20101111	1.0.0.0	201210	OOL Change to per_process, from per_stack. EOL is per stack.
EF_RFC_RTO_INITIAL	20101111	1.0.0.0	201210 2.1.0.0	Change default to 1000 from 3000
EF_DYNAMIC_ACK_THRESH	201205	2.1.0.0	201210	Default value changed to 16 from 32 in 201210

Variable	OOL	EOL	Changed	Notes
EF_TCP_SERVER_LOOPBACK EF_TCP_CLIENT_LOOPBACK	201205	2.1.0.0	201210	TCP loopback acceleration Added option 4 for client loopback to cause both ends of a TCP connection to share a newly created stack. Option 4 is not supported in EnterpriseOnload.
EF_TCP_RST_DELAYED	201205	2.1.0.0		Reset TCP connection following RTO expiry
EF_SA_ONSTACK_INTERCEPT	201205	2.1.0.0		Default value 0
EF_SHARE_WITH	201109-u2	2.0.0.0		
EF_EPOLL_CTL_HANDOFF	201109-u2	2.0.0.0		Default value 1
EF_CHECK_STACK_USER		NS	201109-u2	Renamed EF_SHARE_WITH
EF_POLL_USEC	201109-u1	1.0.0.0		
EF_DEFER_WORK_LIMIT	201109-u1	2.0.0.0		Default value 32
EF_POLL_FAST_LOOPS	20101111	1.0.0.0	201109-u1 2.0.0.0	Renamed EF_POLL_FAST_USEC
EF_POLL_NONBLOCK_FAST_LOOPS	201104	2.0.0.0	201109-u1 2.0.0.1	Renamed EF_POLL_NONBLOCK_FAST_USEC
EF_PIPE_RECV_SPIN	201104	2.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_PKT_WAIT_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_PIPE_SEND_SPIN	201104	2.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_TCP_ACCEPT_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_TCP_RECV_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_TCP_SEND_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_UDP_RECV_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_UDP_SEND_SPIN	20101111	1.0.0.0	201109-u1	Becomes per-process, was previously per-stack
EF_EPOLL_NONBLOCK_FAST_LOOPS	201104-u2	2.0.0.0	201109-u1	Removed
EF_POLL_AVOID_INT	20101111	1.0.0.0	201109-u1	Removed
EF_SELECT_AVOID_INT	20101111	1.0.0.0	201109-u1	Removed

Variable	OOL	EOL	Changed	Notes
EF_SIG_DEFER	20101111	1.0.0.0	201109-u1	Removed
EF_IRQ_CORE	201109	2.0.0.0	201109-u2	Non-root user can now set it when using scalable packet buffer mode
EF_IRQ_CHANNEL	201109	2.0.0.0		
EF_IRQ_MODERATION	201109	2.0.0.0		Default value 0
EF_PACKET_BUFFER_MODE	201109	2.0.0.0	201210	In 201210 options 2 and 3 enable physical addressing mode. EOL only supports option 1. Default - disabled
EF_SIG_REINIT	201109	NS	201109-u1	Default value 0. Removed in 201109-u1
EF_POLL_TCP_LISTEN_UL_ONLY	201104	2.0.0.0	201109	Removed
EF_POLL_UDP	20101111	1.0.0.0	201109	Removed
EF_POLL_UDP_TX_FAST	20101111	1.0.0.0	201109	Removed
EF_POLL_UDP_UL_ONLY	201104	2.0.0.0	201109	Removed
EF_SELECT_UDP	20101111	1.0.0.0	201109	Removed
EF_SELECT_UDP_TX_FAST	20101111	1.0.0.0	201109	Removed
EF_UDP_CHECK_ERRORS	20101111	1.0.0.0	201109	Removed
EF_UDP_RECV_FAST_LOOPS	20101111	1.0.0.0	201109	Removed
EF_UDP_RECV_MCAST_UL_ONLY	20101111	1.0.0.0	201109	Removed
EF_UDP_RECV_UL_ONLY	20101111	1.0.0.0	201109	Removed
EF_TX_QOS_CLASS	201104-u2	2.0.0.0		Default value 0
EF_TX_MIN_IPG_CNTL	201104-u2	2.0.0.0		Default value 0
EF_TCP_LISTEN_HANDOVER	201104-u2	2.0.0.0		Default value 0
EF_TCP_CONNECT_HANDOVER	201104-u2	2.0.0.0		Default value 0
EF_EPOLL_NONBLOCK_FAST_LOOPS	201104-u2	2.0.0.0	201109-u1	Default value 32 Removed in 201109-u1
EF_TCP_SNDBUF_MODE		2.0.0.6		Default value 0
EF_UDP_PORT_HANDOVER2_MAX	201104-u1	2.0.0.0		Default value 1
EF_UDP_PORT_HANDOVER2_MIN	201104-u1	2.0.0.0		Default value 2
EF_UDP_PORT_HANDOVER3_MAX	201104-u1	2.0.0.0		Default value 1
EF_UDP_PORT_HANDOVER3_MIN	201104-u1	2.0.0.0		Default value 2
EF_STACK_PER_THREAD	201104-u1	2.0.0.0		Default value 0

Variable	OOL	EOL	Changed	Notes
EF_PREFAULT_PACKETS	20101111	1.0.0.0	201104-u1	Enabled by default, was previously disabled
EF_MCAST_RECV	201104-u1	2.0.0.0		Default value 1
EF_MCAST_JOIN_BINDTODEVICE	201104-u1	2.0.0.0		Default value 0
EF_MCAST_JOIN_HANDOVER	201104-u1	2.0.0.0		Default value 0
EF_DONT_ACCELERATE	201104-u1	2.0.0.0		Default value 0
EF_MULTICAST	20101111	1.0.0.0	201104-u1	Removed
EF_TX_PUSH	20101111-u1	1.0.0.0	201104	Enabled by default, was previously disabled
			201109	No longer set by the latency profile script

8.3 Module Options

Option	OOL	EOL	Changed	Notes
epoll2_max_stacks	201210	NS		Identifies the maximum number of stacks that an epoll file descriptor can handle when EF_UL_EPOLL=2
phys_mode_gid	201210	NS		Enable physical addressing mode and restrict which users can use it
shared_buffer_table	201210	NS		This option should be set to enable ef_vi applications that use the ef_iobufset API. Setting shared_buffer_table=10000 will make 10000 buffer table entries available for use with ef_iobufset.
safe_signals_and_exit	201205	2.1.0.0		When Onload intercepts a termination signal it will attempt a clean exit by releasing resources including stack locks etc. The default is (1) enabled and it is recommended that this remains enabled unless signal handling problems occur when it can be disabled (0).

Option	OOL	EOL	Changed	Notes
max_layer2_interfaces	201205	2.1.0.0		Maximum number of network interfaces (includes physical, VLAN and bonds) supported in the control plane.
max_routes	201205	2.1.0.0		Maximum number of entries in the Onload route table. Default is 256.
max_neighs	201205	2.1.0.0		Maximum number of entries in Onload ARP/neighbour table. Rounded up to power of two value. Default is 1024.
unsafe_sriov_without_iommu	201209-u2	2.0.0.0	201210	Removed, obsoleted by physical addressing modes and phys_mode_gid.
buffer_table_min buffer_table_max		2.0.0.0	201210	Obsolete - Removed

NOTE: The user should always refer to the Onload distribution *release notes* and *change log*. These are available from http://www.openonload.org/download.html.

Appendix A: Parameter Reference

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, min and max values.
- Whether the variable scope applies per-stack or per-process.
- Description.

EF_ACCEPTQ_MIN_BACKLOG

Name: `acceptq_min_backlog` default: `1` 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_NODELAY

Name: `accept_force_inherit_nodelay` default: `1` min: `0` max: `1` per-process

If set to 1, TCP sockets accepted from a listening socket inherit the TCP_NODELAY socket option from the listening socket.

EF_ACCEPT_INHERIT_NONBLOCK

Name: `accept_force_inherit_nonblock` default: `0` min: `0` max: `1` per-process

If set to 1, TCP sockets accepted from a listening socket inherit the O_NONBLOCK flag from the listening socket.

EF_BINDTODEVICE_HANDOVER

Name: `bindtodevice_handover` default: `0` min: `0` max: `1` 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` 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` 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_CONG_AVOID_SCALE_BACK

Name: `cong_avoid_scale_back` default: `0` 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_DEFER_WORK_LIMIT

Name: `defer_work_limit` default: `32` 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` min: `0` max: `65535` per-stack

This option controls the delayed acknowledgement algorithm. A socket may receive up to the specified number of TCP segments without generating an ACK. Setting this option to 0 disables delayed acknowledgements.NB. This option is overridden by EF_DYNAMIC_ACK_THRESH, so both options need to be set to 0 to disable delayed acknowledgements.

EF_DONT_ACCELERATE

Name: `dont_accelerate` default: `0` min: `0` max: `1` per-process

Do not accelerate by default. This option is usually used in conjuction with onload_set_stackname() to allow individual sockets to be accelerated selectively.

EF_DYNAMIC_ACK_THRESH

Name: `dynack_thresh` default: `16` min: `0` max: `65535` per-stack

If set to >0 this will turn on dynamic adapation 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_EPOLL_CTL_FAST

Name: `ul_epoll_ctl_fast` default: `1` min: `0` max: `1` 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. Following dup(), dup2(), fork() or exec(), some changes to epoll sets may be lost. 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.

EF_EPOLL_CTL_HANDOFF

Name: `ul_epoll_ctl_handoff` default: `1` min: `0` max: `1` per-process

Allow epoll_ctl() calls to be passed from one thread to another in order to avoid lock contention. This optimisation 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 may 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` min: `0` max: `1` 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 are concurrency safe.This option improves performance with EF_UL_EPOLL=1 and also with EF_UL_EPOLL=2 and EF_EPOLL_CTL_FAST=1.

EF_EPOLL_SPIN

Name: `ul_epoll_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `0x7fffffff` per-stack

Sets the number of hardware network events to handle before performing other work. The value chosen represents a trade-off: Larger values increase batching (which typically improves efficiency) but may also increase the working set size (which harms cache efficiency).

EF_FDS_MT_SAFE

Name: `fds_mt_safe` default: `1` min: `0` max: `1` 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() may change 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.Also concurrent calls may happen from signal handlers, so set this to 0 if your signal handlers may close sockets

EF_FDTABLE_SIZE

Name: `fdtable_size` default: `0` 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` min: `0` max: `1` 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` min: `0` max: `1` 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 those receivers are sharing an OpenOnload stack with the sender (see EF_NAME) and EF_MULTICAST_LOOP_OFF=0. See the OpenOnload manual for further details on multicast operation.

EF_FORCE_TCP_NODELAY

Name: tcp_force_nodelay default: 0 min: 0 max: 2 per-stack

This option allows the user to override the use of TCP_NODELAY. This may be useful in cases where 3rd-party software is (not) setting this value and the user would like to control its behaviour: 0 - do not override 1 - always set TCP_NODELAY 2 - never set TCP_NODELAY

EF_FORK_NETIF

Name: `fork_netif` default: `3` min: `CI_UNIX_FORK_NETIF_NONE` max: `CI_UNIX_FORK_NETIF_BOTH` per-process

This option controls behaviour after an application calls fork(). 0 - Neither fork parent nor child creates a new OpenOnload 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: 100 per-stack

Keep free packets number to be at least this value. EF_MIN_FREE_PACKETS defines initialisation behaviour; this value is about normal application runtime. This value is used if we can not allocate more packets at any time, i.e. in case of AMD IOMMU only.

EF_HELPER_PRIME_USEC

Name: `timer_prime_usec` default: `250` 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` 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_INT_DRIVEN

Name: `int_driven` default: `1` min: `0` max: `1` 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 in conjunction 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 (eg. 100us) to prevent too high a rate of interrupts.

EF_INT_REPRIME

Name: `int_reprime` default: `0` min: `0` max: `1` per-stack

Enable interrupts more aggressively than the default.

EF_IRQ_CHANNEL

Name: `irq_channel` default: `4294967295` min: `-1` max: SMAX 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.This option only takes effect EF_PACKET_BUFFER_MODE=0 (default) or 2.

EF_IRQ_CORE

Name: `irq_core` default: `4294967295` min: `-1` max: SMAX per-stack

Specify which CPU core interrupts for this stack should be handled on.With EF_PACKET_BUFFER_MODE=1 or 3, Onload creates dedicated interrupts for each stack, and the interrupt is assigned to the requested core.With EF_PACKET_BUFFER_MODE=0 (default) or 2, Onload interrupts are handled via net driver receive channel interrupts. The sfc_affinity driver is used to choose which net-driver receive channel is used. 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 that if the IRQ balancer service is enabled it may redirect interrupts to other cores.

EF_IRQ_MODERATION

Name: `irq_usec` default: `0` min: `0` max: `1000000` per-stack

Interrupt moderation interval, in microseconds.This option only takes effective with EF_PACKET_BUFFER_MODE=1 or 3. Otherwise the interrupt moderation settings of the kernel net driver take effect.

EF_KEEPALIVE_INTVL

Name: `keepalive_intvl` default: `75000` per-stack

Default interval between keepalives, in milliseconds.

EF_KEEPALIVE_PROBES

Name: `keepalive_probes` default: `9` per-stack

Default number of keepalive probes to try before aborting the connection.

EF_KEEPALIVE_TIME

Name: `keepalive_time` default: `7200000` per-stack

Default idle time before keepalive probes are sent, in milliseconds.

EF_LOAD_ENV

Name: `load_env` default: `1` min: `0` max: `1` per-process

OpenOnload 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_VIA_IOCTL

Name: `log_via_ioctl` default: `0` min: `0` max: `1` per-process

Causes error and log messages emitted by OpenOnload 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 OpenOnload stack.By default, OpenOnload 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: `1024` min: `0` max: `CI_CFG_NETIF_MAX_ENDPOINTS_MAX` 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 1 and 32,768.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: Multiple endpoint buffers are consumed by each accelerated pipe.

EF_MAX_EP_PINNED_PAGES

Name: `max_ep_pinned_pages` default: `512` per-stack

Not currently used.

EF_MAX_PACKETS

Name: `max_packets` default: `32768` min: `1024` per-stack

Upper limit on number of packet buffers in each OpenOnload 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. The SFN5x series adapters support approximately 120,000 packet buffers.

EF_MAX_RX_PACKETS

Name: `max_rx_packets` default: `24576` min: `0` max: `1000000000` 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` min: `0` max: `1000000000` 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` min: `0` max: `1` 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` min: `0` max: `2` 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` min: `0` max: `1` 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.See the OpenOnload manual for further details on multicast operation.

EF_MIN_FREE_PACKETS

Name: `min_free_packets` default: `100` per-stack

Minimum number of free packets to reserve for each stack at initialisation. 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` min: `0` max: `1` per-stack

When set, disables loopback of multicast traffic to receivers in the same OpenOnload stack.See the OpenOnload manual for further details on multicast operation.

EF_NAME

Name: `ef_name` per-process

Processes setting the same value for EF_NAME in their environment can share an OpenOnload stack.

EF_NETIF_DTOR

Name: `netif_dtor` default: `1` min: `0` max: `2` per-process

This option controls the lifetime of OpenOnload stacks when the last socket in a stack is closed.

EF_NONAGLE_INFLIGHT_MAX

Name: `nonagle_inflight_max` default: `50` min: `1` per-stack

This option affects the behaviour 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` min: `0` max: `1` 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 (ie. to find out when they are not).

EF_PACKET_BUFFER_MODE

Name: `packet_buffer_mode` default: `0` min: `0` max: `3` 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. 1 - SR-IOV with IOMMU. Each stack allocates a separate PCI Virtual Function. IOMMU guarantees that different stacks do not have any access to each other data. 2 - Physical address mode. Inherently unsafe; no address space separation between different stacks or net driver packets. 3 - SR-IOV with physical address mode. Each stack allocates a separate PCI Virtual Function. IOMMU is not used, so this mode is unsafe in the same way as (2).To use odd modes (1 and 3) SR-IOV must be enabled in the BIOS, OS kernel and on the network adapter. In these modes you also get faster interrupt handler which can improve latency for some workloads.For mode (1) you also have to enable IOMMU (also known as VT-d) in BIOSand in your kernel.For unsafe physical address modes (2) and (3), you should tune phys_mode_gid module parameter of the onload module.

EF_PIO

NAME: pio default: 1 min: 0 max: 2 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); 2 - use PIO for small packets and fail if PIO is not available.

Mode 1 will fall back to DMA if PIO is not currently 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.

EF_PIO_THRESHOLD

NAME: pio_thresh default: 1514 max: 0 per-stack

Sets a threshold for the size of packet that will use PIO, if turned on using EF_PIO. Packets up to the threshold will use PIO. Larger packets will not.

EF_PIPE

Name: `ul_pipe` default: 2 min: `CI_UNIX_PIPE_DONT_ACCELERATE` max: `CI_UNIX_PIPE_ACCELERATE_IF_NETIF` per-process

- disable pipe acceleration, 1 - enable pipe acceleration, 2 - acclerate pipes only if an Onload stack already exists in the process.

EF_PIPE_RECV_SPIN

Name: `pipe_recv_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `1` 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_PKT_WAIT_SPIN

Name: `pkt_wait_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `1` 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` 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_NONBLOCK_FAST_USEC

Name: `ul_poll_nonblock_fast_usec` default: `200` 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 behaviour of poll() calls, so could potentially cause an application to misbehave.

EF_POLL_ON_DEMAND

Name: `poll_on_demand` default: `1` min: `0` max: `1` 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` min: `0` max: `1` 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`   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.

EF_PREFAULT_PACKETS

Name: `prefault_packets` default: `1`   min: `0`   max: `1000000000`   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`   min: `0`   max: `1`   per-process

When set, file descriptors accessed following exec() will be 'probed' and OpenOnload sockets will be mapped to user-land so that they can be accelerated. Otherwise OpenOnload sockets are not accelerated following exec().

EF_RETRANSMIT_THRESHOLD

Name: `retransmit_threshold` default: `15`   min: `0`   max: SMAX   per-stack

Number of retransmit timeouts before a TCP connect is aborted.

EF_RETRANSMIT_THRESHOLD_SYN

Name: `retransmit_threshold_syn` default: `4`   min: `0`   max: SMAX   per-stack

Number of times a SYN will be retransmitted before a connect() attempt will be aborted.

EF_RETRANSMIT_THRESHOLD_SYNACK

Name: `retransmit_threshold_synack` default: `5` min: `0` max: SMAX per-stack

Number of times a SYN-ACK will be retransmitted before an embryonic connection will be aborted.

EF_RFC_RTO_INITIAL

Name: `rto_initial` default: `1000` 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` per-stack

Maximum retransmit timeout in milliseconds.

EF_RFC_RTO_MIN

Name: `rto_min` default: `200` per-stack

Minimum retransmit timeout in milliseconds.

EF_RXQ_LIMIT

Name: `rxq_limit` default: `65535` min: `CI_CFG_RX_DESC_BATCH` max: `65535` 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` min: `2 * CI_CFG_RX_DESC_BATCH + 1` 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` min: `512` max: `4096` 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 may reduce efficiency because the working set size is increased.

EF_RX_TIMESTAMPING

Name: `rx_timestamping` default: `0` min: `0` max: `3` 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_SA_ONSTACK_INTERCEPT

Name: `sa_onstack_intercept` default: `0` min: `0` max: `1` per-process

Intercept signals when signal handler is installed with SA_ONSTACK flag. 0 - Don't 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 may deadlock. (default) 1 - Intercept. There is no guarantee that SA_ONSTACK flag will really work, but OpenOnload library will do its best.

EF_SELECT_FAST

Name: `ul_select_fast` default: `1` min: `0` max: `1` 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_SPIN

Name: `ul_select_spin` default: `0` min: `0` max: `1` 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` min: `1` max: `65535` 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` min: `0` max: `65535` per-stack

Poll for network events after sending this many packets.Setting this to a larger value may improve transmit throughput for small messages by allowing batching. However, such batching may cause sends to be delayed leading to increased jitter.

EF_SHARE_WITH

Name: `share_with` default: `0` min: `-1` max: SMAX 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.WARNING: 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 misbehaviour and crashes in any application using the stack.

EF_SIGNALS_NOPOSTPONE

Name: `signals_no_postpone` default: `67109952` min: `0` max: `(ci_uint64)(-1)` 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 SIGBUS, 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_SOCK_LOCK_BUZZ

Name: `sock_lock_buzz` default: `0` min: `0` max: `1` 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_SPIN_USEC

Name: `ul_spin_usec` default: `0` per-process

Sets the timeout in microseconds for spinning options. Set this to to -1 to spin forever. The spin timeout may 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` min: `0` max: `1` 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` min: `0` max: `1` per-process

Create a separate Onload stack for the sockets created by each thread.

EF_TCP

Name: `ul_tcp` default: `1` min: `0` max: `1` per-process

Clear to disable acceleration of new TCP sockets.

EF_TCP_ACCEPT_SPIN

Name: `tcp_accept_spin` default: `0` min: `0` max: `1` per-process

Spin in blocking TCP accept() 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_ADV_WIN_SCALE_MAX

Name: `tcp_adv_win_scale_max` default: `14` min: `0` max: `14` per-stack

Maximum value for TCP window scaling that will be advertised.

EF_TCP_BACKLOG_MAX

Name: `tcp_backlog_max` default: `256` per-stack

Places an upper limit on the number of embryonic (half-open) connections in an OpenOnload stack.

EF_TCP_CLIENT_LOOPBACK

Name: `tcp_client_loopback` default: `0` min: `0` max: `CITP_TCP_LOOPBACK_TO_NEWSTACK` 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: Option 3 breaks some usecases with epoll, fork and dup calls.

EF_TCP_CONNECT_HANDOVER

Name: `tcp_connect_handover` default: `0` min: `0` max: `1` 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_FASTSTART_IDLE

Name: `tcp_faststart_idle` default: `65536` min: `0` per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so may 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` min: `0` per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so may 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` min: `0` per-stack

The FASTSTART feature prevents Onload from delaying ACKs during times when doing so may 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` per-stack

Time in seconds to wait for a FIN in the TCP FIN_WAIT2 state.

EF_TCP_INITIAL_CWND

Name: `initial_cwnd` default: `0` min: `0` max: SMAX 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 may result in Onload being unable to send traffic.

WARNING: Modifying this option may violate the TCP protocol.

EF_TCP_LISTEN_HANDOVER

Name: `tcp_listen_handover` default: `0` min: `0` max: `1` 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` min: `0` max: SMAX per-stack

Sets the minimum size of the congestion window for TCP connections following loss.WARNING: Modifying this option may violate the TCP protocol.

EF_TCP_RCVBUF

Name: `tcp_rcvbuf_user` default: `0` 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_RECV_SPIN

Name: `tcp_recv_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `1` 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).WARNING: 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` min: `0` max: `1` 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` 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_SPIN

Name: `tcp_send_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `CITP_TCP_LOOPBACK_ALLOW_ALIEN_IN_ACCEPTQ` 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_SNDBUF

Name: `tcp_sndbuf_user` default: `0` 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_MODE

Name: tcp_sndbuf_mode default: 0 min: 0 max: 1 per-stack

This option controls how the SO_SNDBUF limit is applied to TCP sockets. In the default mode the limit applies only to the send queue. When this option is set to 1, the limit applies to the size of the send queue and retransmit queue combined.

EF_TCP_SYN_OPTS

Name: `syn_opts` default: 7 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).

EF_TCP_TCONST_MSL

Name: `msl_seconds` default: 25 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_TXQ_LIMIT

Name: `txq_limit` default: `268435455` min: `16 * 1024` max: `0xfffffff` per-stack

Maximum number of bytes to enqueue on the transmit descriptor ring.

EF_TXQ_RESTART

Name: `txq_restart` default: `268435455` min: `1` max: `0xfffffff` per-stack

Level (in bytes) to which the transmit descriptor ring must fall before it will be filled again.

EF_TXQ_SIZE

Name: `txq_size` default: `512` min: `512` max: `4096` per-stack

Set the size of the transmit descriptor ring. Valid values: 512, 1024, 2048 or 4096. The max transmit queue size supported by the SFN7000 series adapter is 2048.

EF_TX_MIN_IPG_CNTL

Name: `tx_min_ipg_cntl` default: `0` min: `-1` max: `20` per-stack

Rate pacing value.

EF_TX_PUSH

Name: `tx_push` default: `1` min: `0` max: `1` per-stack

Enable low-latency transmit.

EF_TX_PUSH_THRESHOLD

Name: `tx_push_thresh` default: `100` min: `1` 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. This threshold is ignored, and assumed to be 1, on pre-SFN7000 series hardware. It makes sense to set this value similar to EF_SEND_POLL_THRESH.

EF_TX_QOS_CLASS

Name: `tx_qos_class` default: `0` min: `0` max: `1` 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 minimise latency by placing latency sensitive traffic into a separate QOS class from bulk traffic.

EF_UDP

Name: `ul_udp` default: `1` min: `0` max: `1` per-process

Clear to disable acceleration of new UDP sockets.

EF_UDP_CONNECT_HANDOVER

Name: `udp_connect_handover` default: `1` min: `0` max: `1` per-stack

When a UDP socket is connected to an IP address that cannot be accelerated by OpenOnload, hand the socket over to the kernel stack.When this option is disabled the socket remains under the control of OpenOnload. This may be worthwhile because the socket may subsequently be re-connected to an IP address that can be accelerated.

EF_UDP_PORT_HANDOVER2_MAX

Name: `udp_port_handover2_max` default: `1` 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` 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` 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` 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` 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` 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` 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` min: `0` max: `1` 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_SPIN

Name: `udp_send_spin` default: `0` min: `0` max: `1` 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` min: `0` max: `1` 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` 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 may increase lock contention in multi-threaded applications.

EF_UDP_SNDBUF

Name: `udp_sndbuf_user` default: `0` 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` min: `0` max: `2` 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)The default is the user-level implementation (1).

EF_UL_POLL

Name: `ul_poll` default: `1` min: `0` max: `1` per-process

Clear to disable acceleration of poll() calls at user-level.

EF_UL_SELECT

Name: `ul_select` default: `1` min: `0` max: `1` per-process

Clear to disable acceleration of select() calls at user-level.

EF_UNCONFINE_SYN

Name: `unconfine_syn` default: `1` min: `0` max: `1` per-stack

Accept TCP connections that cross into or out-of a private network.

EF_UNIX_LOG

Name: `log_level` default: `3` per-process

A bitmask determining which kinds of diagnostics messages will be logged. 0x1 errors 0x2 unexpected 0x4 setup 0x8 verbose 0x10 select() 0x20 poll() 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 V.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` min: `0` max: `1` per-stack

Choose between compliance with RFC1122 (1) or BSD behaviour (0) regarding the location of the urgent point in TCP packet headers.

EF_USE_DSACK

Name: `use_dsack` default: `1` min: `0` max: `1` per-stack

Whether or not to use DSACK (duplicate SACK).

EF_USE_HUGE_PAGES

Name: `huge_pages` default: `1` min: `0` max: `2` 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; syslog message is printed if the system is out of huge pages.Non-initial packets may 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` min: `0` max: `1` per-stack

When set this option validates Onload related environment variables (starting with EF_).

EF_VFORK_MODE

NAME: vfork_mode default: 1 min: 0 max: 2 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 may 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()

Appendix B: Meta Options

There are several environment variables which act as meta-options and set several of the options detailed in Appendix A. 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_UDP_RECV_SPIN=1
EF_UDP_SEND_SPIN=1
EF_TCP_RECV_SPIN=1
EF_BUZZ_USEC=EF_POLL_USEC
EF_SOCK_LOCK_BUZZ=1
EF_STACK_LOCK_BUZZ=1
```

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

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 explicity set EF_BUZZ_USEC value e.g.

```
    export EF_POLL_USEC=10000
    export EF_BUZZ_USEC=1000
```

Appendix C: Build Dependencies

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 - i.e. has gcc, make, libc and libc-devel.
- Can compile kernel modules - i.e. has the correct kernel-devel package for the installed kernel version.
- If 32 bit applications are to be accelerated on 64 bit architectures the machine must be able to build 32 bit applications.

Building Kernel Modules

The kernel must be built with `CONFIG_NETFILTER` enabled. Standard distributions will already have this enabled, but it must also be enabled when building a custom kernel. This option does 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 32bit Kernel):

If the system supports a 32 bit Kernel and the kernel is PAE then install `kernel-PAE-devel` otherwise install the following package:

```
yum -y install kernel-devel
```

- For SuSE:

```
yast -i kernel-source
```

Building 32 bit applications on 64 bit architecture platforms

The following commands can be used to install 32 bit libc development headers.

- Debian based Distributions - including Ubuntu:

```
apt-get install gcc-multilib libc6-dev-i386
```

- For RedHat/Fedora:

```
yum -y install glibc-devel.i586
```

- For SuSE:

```
yast -i glibc-devel-32bit
yast -i gcc-32bit
```

Appendix D: Onload Extensions API

The Onload Extensions API allows the user to customise 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.

This section covers the follows topics:

Source Code

The onload 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
```

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 to this library include the '-l' linker option on the compiler command line i.e.

  ```
  -lonload_ext
  ```

- `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	Returns 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	Returns: 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.

- onload_fd_check_feature()

```
int onload_fd_check_feature (int fd,
enum onload_fd_feature feature);
enum onload_fd_feature {
  /* Check whether this fd supports ONLOAD_MSG_WARM or not */
  ONLOAD_FD_FEAT_MSG_WARM
};
```

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 >0 if the fd is supported -ENOSYS if onload_fd_check_feature() is not supported.
Notes	none.

- `onload_thread_set_spin()`

Description	For each thread, specify which operations should spin.

Definition	`int onload_thread_set_spin(`
	`enum onload_spin_type type,`
	`unsigned spin)`

Formal Parameters	type: which operation to change the spin status of. Type must be one of the following:
	enum onload_spin_type{
	ONLOAD_SPIN_ALL
	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
	};
	spin: is a boolean which indicates whether the operation should spin or not.

Return Value	Return zero 0 on success or -EINVAL if unsupported type is specified.

Notes	Spin time (for all threads) is set using the `EF_SPIN_USEC` parameter.
	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);`

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 behaviour for threads, and the `onload_thread_set_spin` API overrides the default.

To 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()`.

To 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()`.

NOTE: If a thread is set to NOT spin and then blocks this may invoke an interrupt for the whole stack. Interrupts occurring on moderately busy threads may cause unintended and undesirable consequences.

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.

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 notes**. **name**: is the stack name up to 8 characters. or can be an empty string to set no stackname or can be 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.

Notes	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 i.e.
	
	`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.
	
	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.
	
	3. Scoping can be thought of as adding a suffix to the supplied name e.g.
	
	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.

Notes (continued)	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 behaviour 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 are:

- EF_DONT_ACCELERATE

  Default: 0 Min: 0 Max: 1 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 Min: 0 Max: 1 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

  The environment variable EF_NAME will be honoured to control Onlaod 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_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.
Definition	`int onload_stackname_restore (void)`
Formal Parameters	none
Return Value	0 on success non-zero if an error occurs.

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.

Stacks API Usage

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.

- `onload_stack_opt_set_int()`

Description	Set/modify per stack options that all subsequently created stacks will use instead of using the existing global stack options..

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(dont_accelerate, 1);`
Return Value	0 on success -1 with errno set to EINVAL if the requested option is not found.
Notes	Cannot be used to modify options on existing stacks - only for new stacks. Cannot be used to modify process options - only 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.

Stacks API - Examples

- 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:

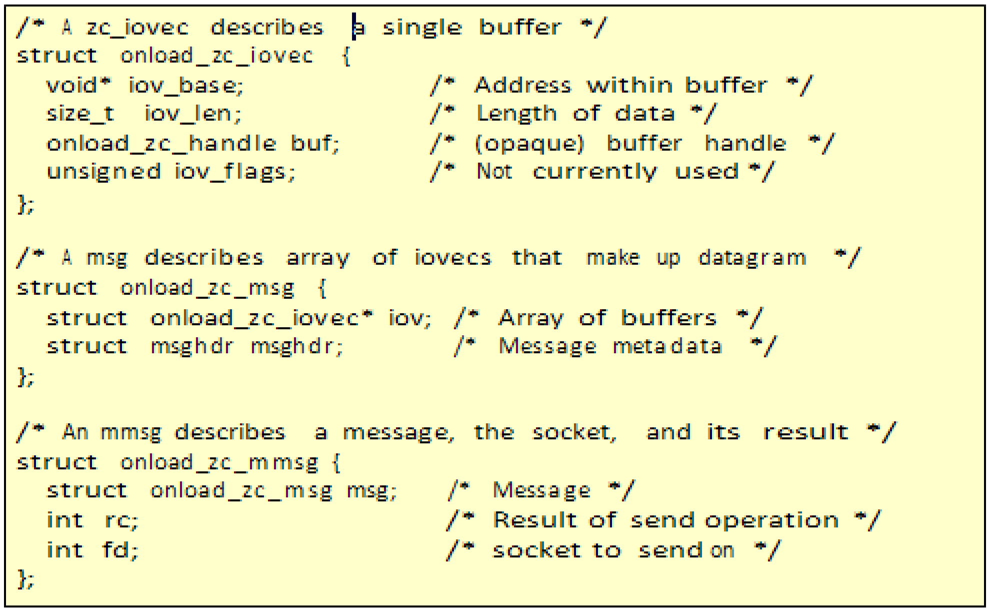
- `#include <onload/extensions_zc.h>`

  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.

  This file 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 may involve multiple messages using an array of `struct onload_zc_mmsg`.



```
/* 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 */
};
```

Figure 7: Zero-Copy Data Buffers

Zero-Copy UDP Receive Overview

Figure 8 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.

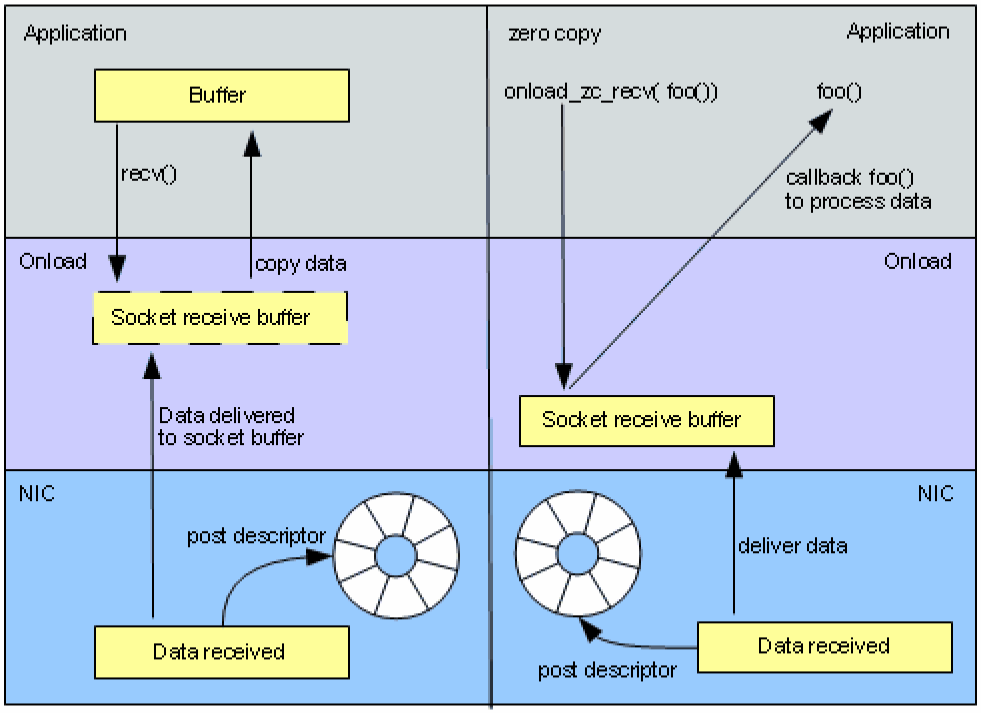


Figure 8: Traditional vs. Zero-Copy UDP Receive

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 does not require to look at all data received (i.e. is filtering) 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.

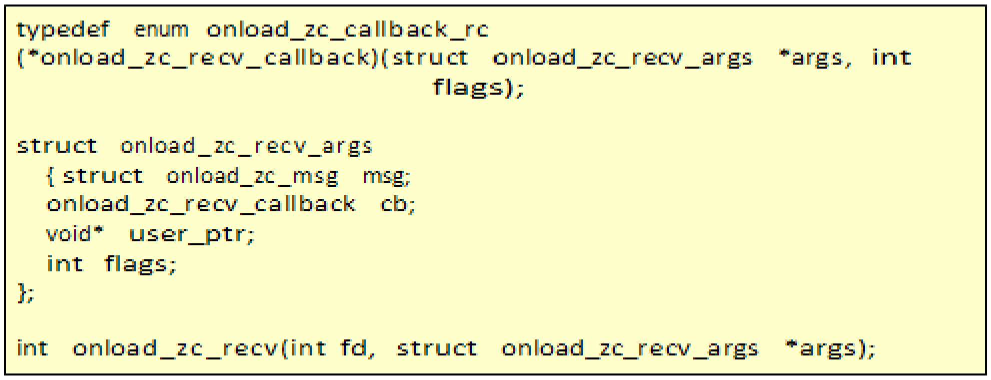
Zero-copy UDP Receive is implemented within the Onload Extensions API.

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()` (i.e. It blocks/spins waiting for data).

Before calling, the application must set the following in the `struct onload_zc_recv_args`:

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 (e.g. ONLOAD_MSG_DONTWAIT)



```
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);
```

Figure 9: Zero-Copy recv_args

The callback gets to examine the data, and can control what happens next: (i) whether or not the buffer(s) are kept by the callback or are immediately freed by Onload; and (ii) 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:_

`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
`ONLOAD_ZC_TERMINATE`	to insist that Onload immediately return from the `onload_zc_recv()`

Flags can also be set by Onload:

`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:

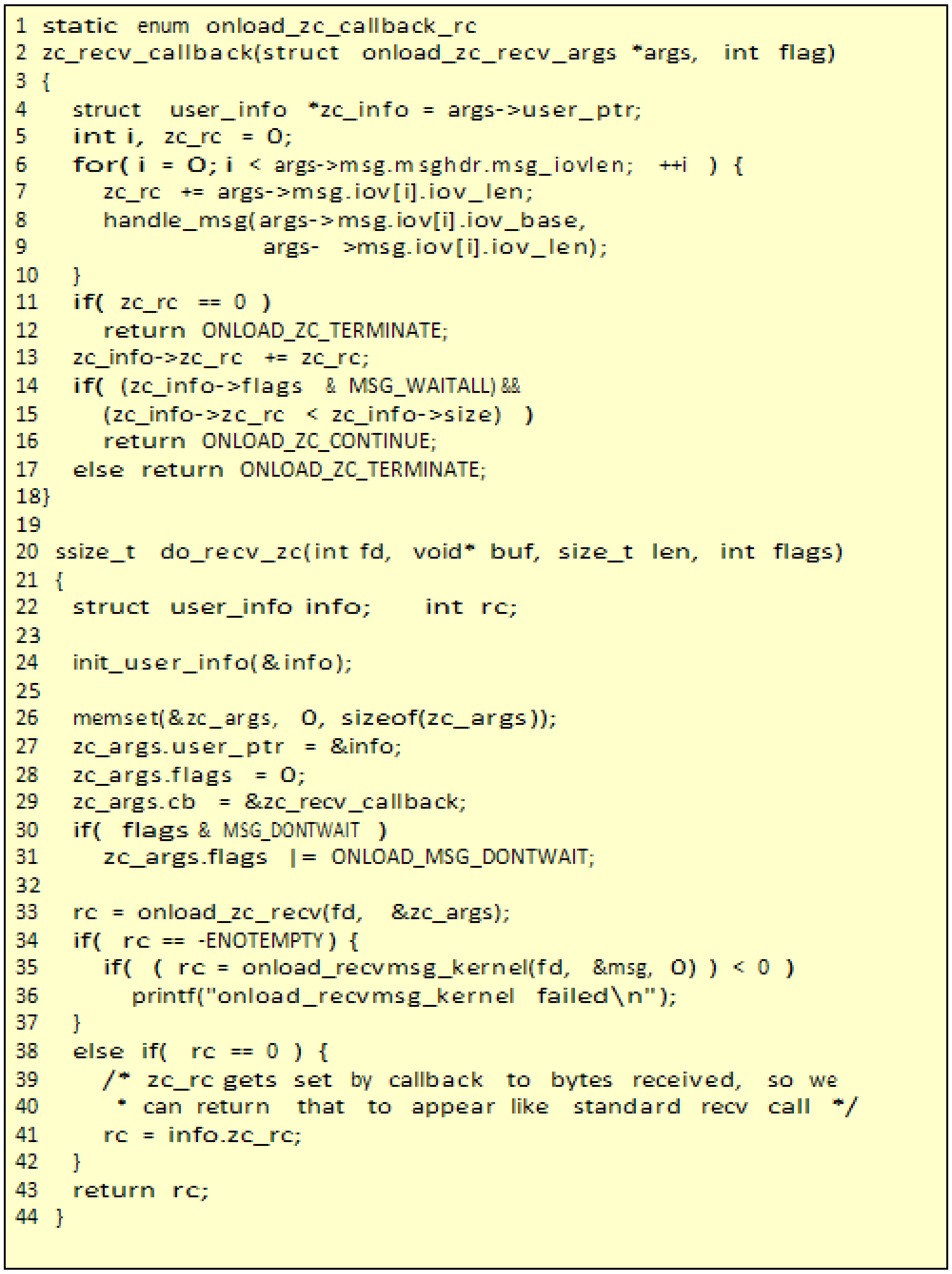
`ONLOAD_MSG_RECV_OS_INLINE`	set this flag when calling `onload_zc_recv()`. Onload will deal with the kernel data internally and pass it to the callback
check return code	check the return code from `onload_zc_recv()`. If it returns `ENOTEMPTY` then the application must call `onload_recvmsg_kernel()` to retrieve the kernel data.

Zero-Copy Receive Example #1

```
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;
}
```

Figure 10: Zero-Copy Receive -example #1

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;
}

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;
}
```

Figure 11: Zero-Copy Receive - example #2

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

Figure 12 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.

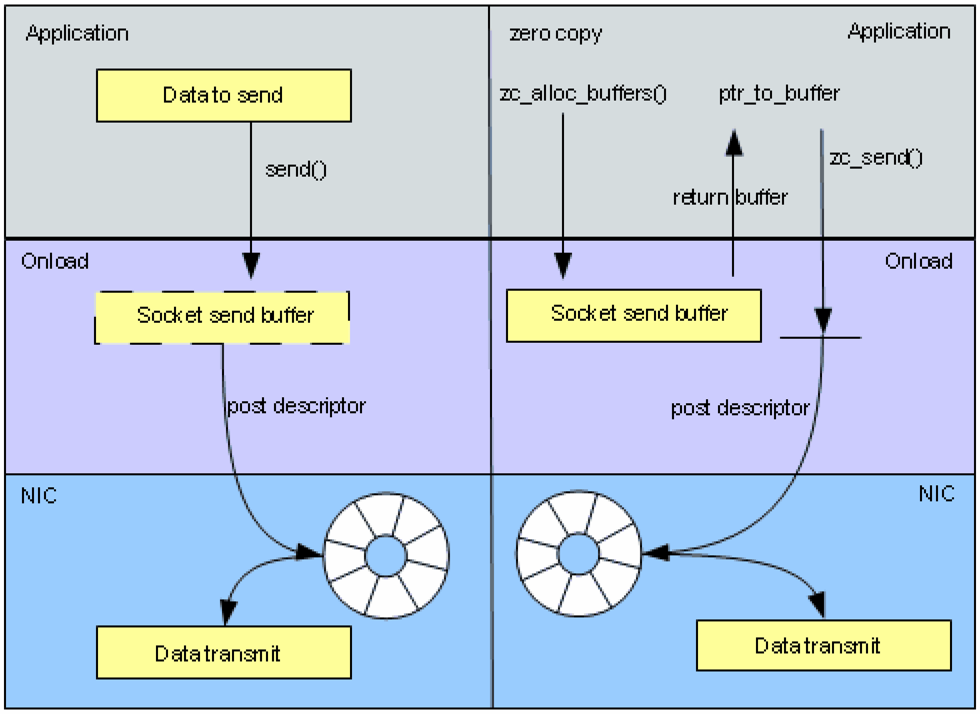


Figure 12: Traditional vs. Zero-Copy TCP Transmit

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.

```
int  onload_zc_send(struct onload_zc_mmsg* msgs,  int  mlen,  int
                          flags);
```

Figure 13: Zero-Copy send

```
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);
```

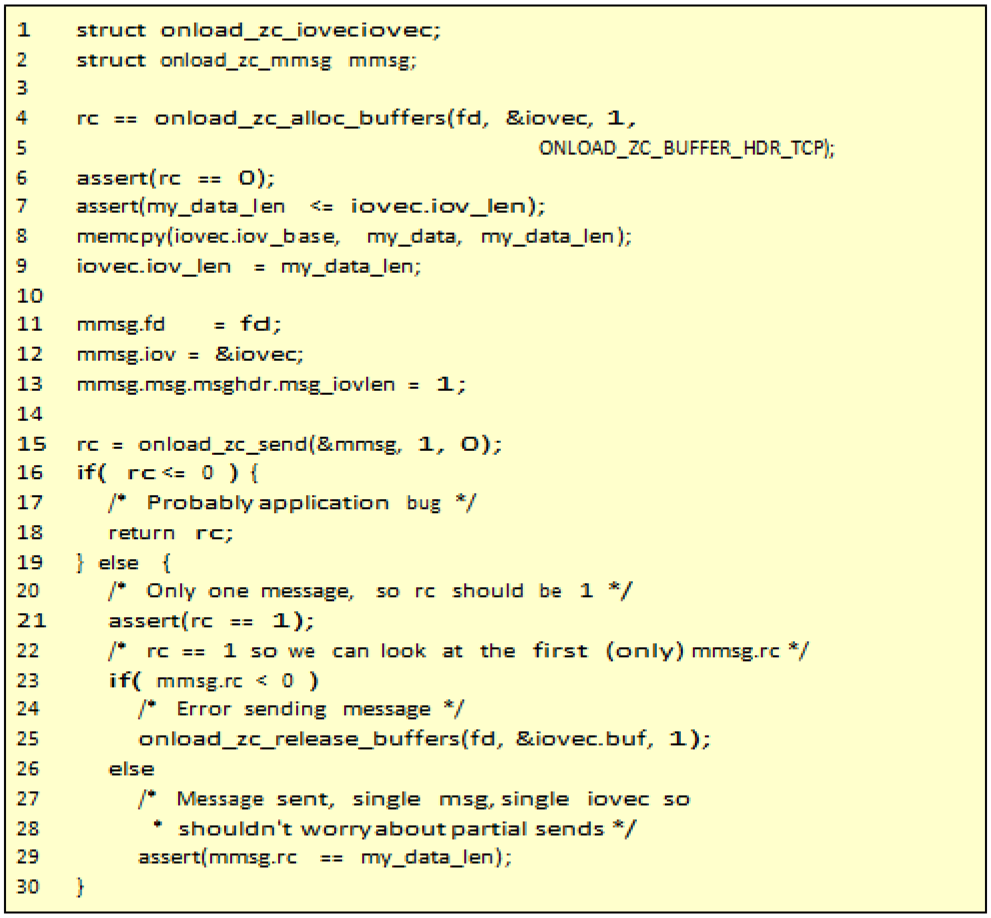
Figure 14: Zero-Copy allocate buffers

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.

Zero-Copy Send - Single Message, Single Buffer

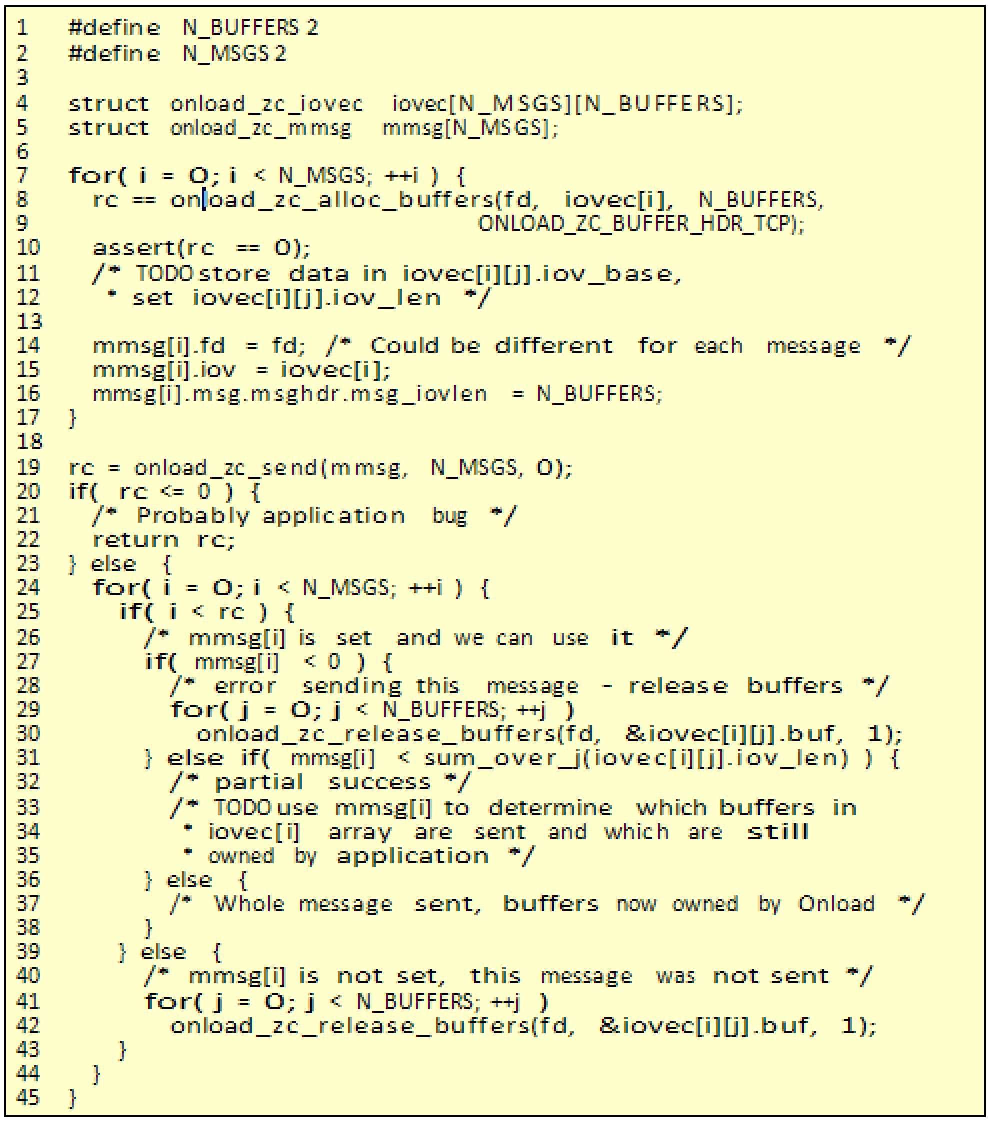


```
struct onload_zc_ioveciovec;
struct onload_zc_mmsg  mmsg;

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.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);
}
```

Figure 15: Zero-Copy - Single Message, Single Buffer Example

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 - Multiple Message, Multiple Buffers

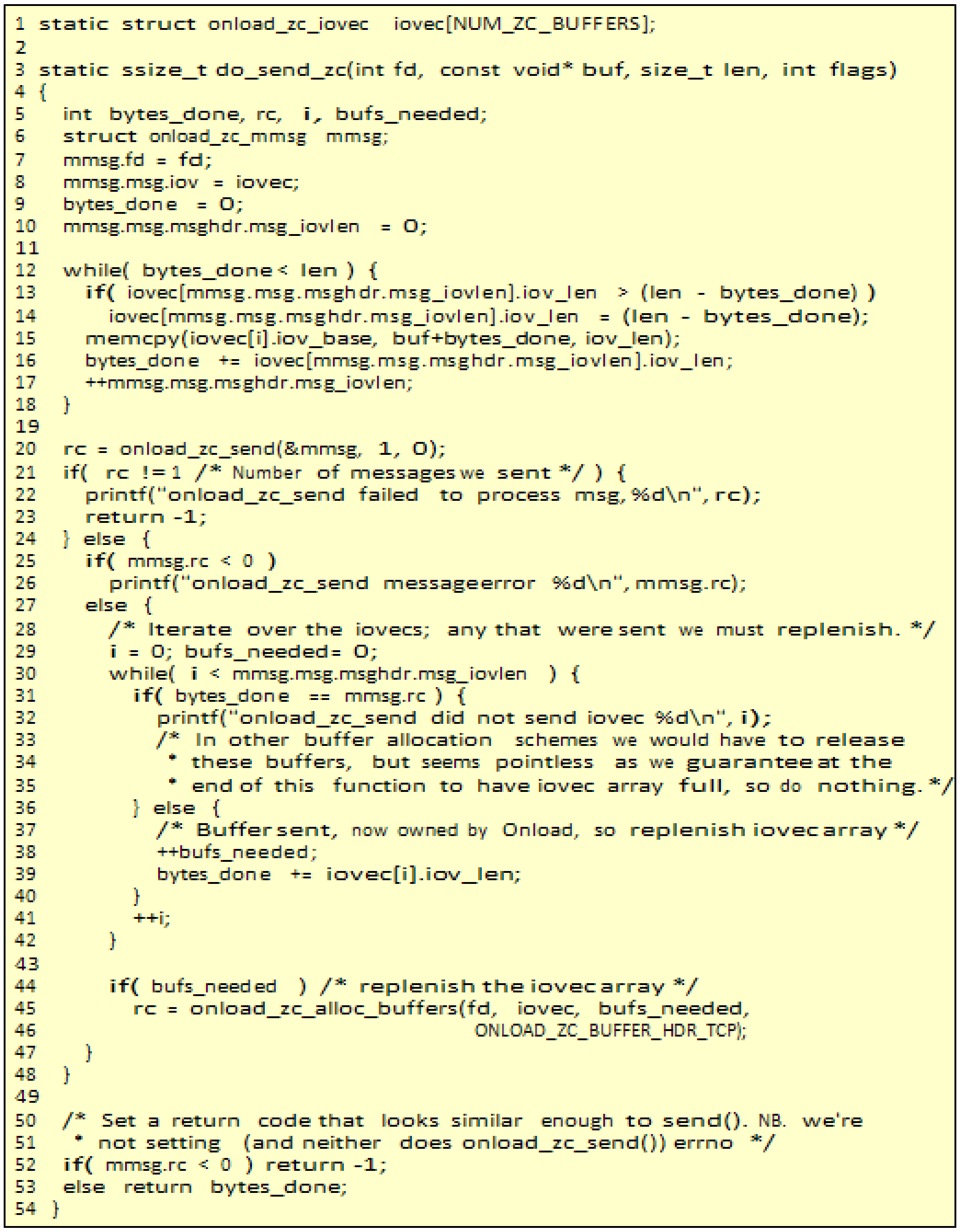


```
#define  N_BUFFERS 2
#define  N_MSGS 2

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);
    }
  }
}
```

Figure 16: Zero-Copy - Multiple Messages, Multiple Buffers Example

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



```c
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;
}
```

Figure 17: Zero-Copy Send

Receive Filtering API Overview

The Onload Extensions Receive API allows user-defined filters to determine if data received on a UDP socket should be discarded before it enters the socket receive buffer.

Receive filtering provides an alternative to the `onload_zc_recv()` function described in the previous sections. It allows the application to examine the received data inplace and direct Onload to discard a subset of the data if not required - thereby saving the overhead of having to copy unwanted data. Only the subset of datagrams that the application is interested in are copied into the application buffers

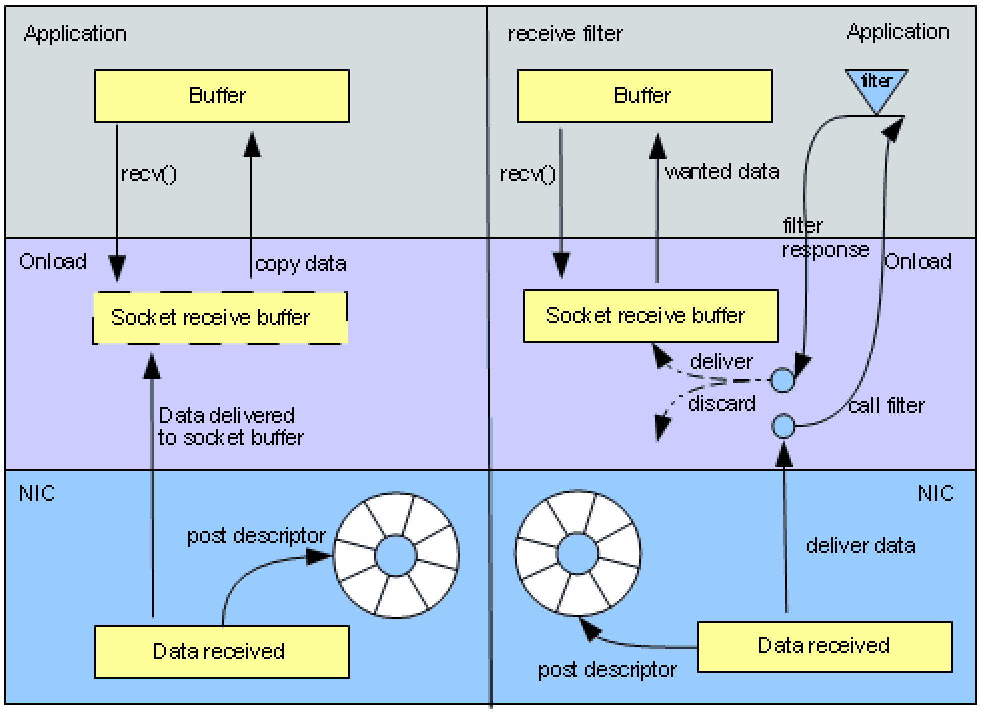


Figure 18: UDP Receive Filtering

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 e.g `select()` `poll()`, `epoll_wait()` or `recv()`, but must not use the `onload_zc_receive()` function.

Receive Filtering API

The Onload Extensions Receive Filtering API installs a user defined filter that determines whether received data is passed to the receive function or discarded by Onload.

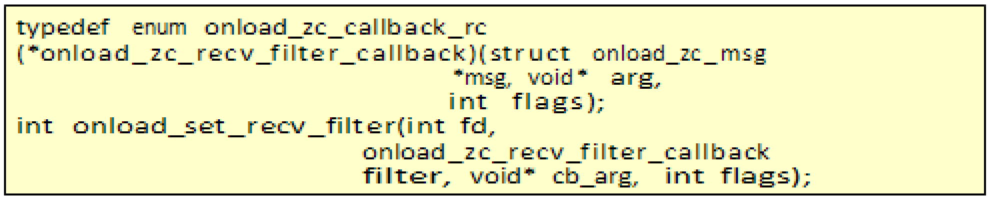
The API provides the following components:

- `#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.

  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 filter that is run on each datagram before it is received. The filter can elect to discard or even modify the received message before this is copied to the application.



```
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);
```

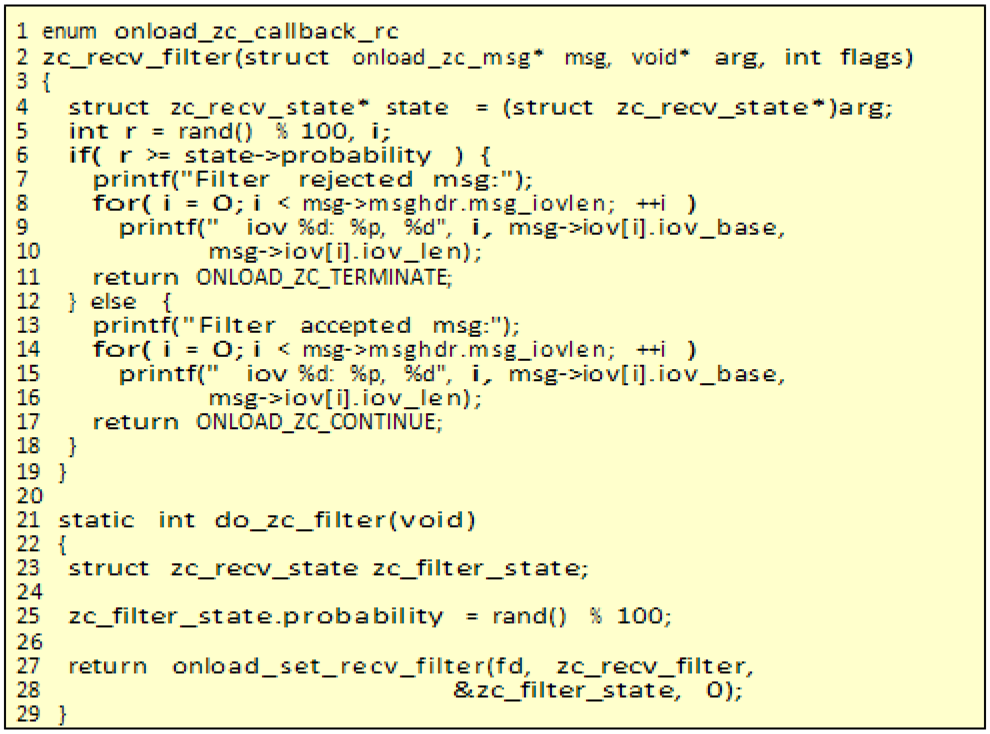
Figure 19: Receive Filter

The `onload_set_recv_filter()` function returns immediately and the filter callback is executed in the context of subsequent calls to `recv(), recvmsg()` etc.

NOTE: When using receive filtering, calls to `poll(), select()` etc, may return that a socket is readable, but if the filter then discards the datagram a call to `recv()` would not find it and block.

NOTE: The application should respect the `ONLOAD_ZC_MSG_SHARED` flag - this indicates that the datagram may be delivered to more than one socket and so should not be modified.

Receive Filter - Example



```
enum onload_zc_callback_rc
zc_recv_filter(struct onload_zc_msg* msg, void* arg, int flags)
{
  struct zc_recv_state* state = (struct zc_recv_state*)arg;
  int r = rand() % 100, i;
  if( r >= state->probability ) {
    printf("Filter rejected msg:");
    for( i = 0; i < msg->msghdr.msg_iovlen; ++i )
      printf(" iov %d: %p, %d", i, msg->iov[i].iov_base,
             msg->iov[i].iov_len);
    return ONLOAD_ZC_TERMINATE;
  } else {
    printf("Filter accepted msg:");
    for( i = 0; i < msg->msghdr.msg_iovlen; ++i )
      printf(" iov %d: %p, %d", i, msg->iov[i].iov_base,
             msg->iov[i].iov_len);
    return ONLOAD_ZC_CONTINUE;
  }
}

static int do_zc_filter(void)
{
  struct zc_recv_state zc_filter_state;

  zc_filter_state.probability = rand() % 100;

  return onload_set_recv_filter(fd, zc_recv_filter,
                                &zc_filter_state, 0);
}
```

Figure 20: Receive Filter - Example

.

NOTE: The `onload_set_recv_filter()` function should NOT be used together with the `onload_zc_recv()` function.

Templated Sends

For a description of the templates sends feature, refer to Templated Sends on page 71. For a description of the packet template to be used by the templated sends feature refer to the use notes and references to `onload_msg_template` in the `[onload]/src/include/onload/extensions_zc.h` file included with the Onload 201310 distribution.

Appendix E: onload_stackdump

Introduction

The Solarflare `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 and accessible to them via the EF_SHARE_WITH variable.

The following examples of `onload_stackdump` are demonstrated elsewhere in this user guide:

- Monitoring Using onload_stackdump on page 30
- Processing at User-Level on page 30
- As Few Interrupts as Possible on page 31
- Eliminating Drops on page 32
- Minimizing Lock Contention on page 33

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 -?
  ```

- To list and read environment variables descriptions:

  ```
  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 e.g (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 e.g.

```
# onload_stackdump processes
#pid  stack-id cmdline
25587    3     ./sfnt-pingpong
```

Onloaded proceseses which have not created a socket are not displayed, but can be identified using the lsof command.

Identify Onloaded Processes Affinities

The 'onload_stackdump affinities' command will identify the task affinity for an accelerated process e.g.

```
# onload_stackdump affinities
pid=25587
cmdline=./sfnt-pingpong
task25587: 80
```

The task affinity is identified from an 8 bit mask i.e. 01 is CPU core 0, 02 is CPU core 1, 80 is CPU core 7 etc.

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 e.g.

```
# 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 new counters to indicate how often TX PIO is being used - see Programmed I/O on page 70. 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.

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 honour 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 re-starting applications, for example the application may 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 may be 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
```

- 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 e.g.

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 e.g.

```
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=6
pkt_bufs: size=2048 max=32768 alloc=576 free=50 async=0
pkt_bufs: rx=525 rx_ring=522 rx_queued=3
pkt_bufs: tx=1 tx_ring=0 tx_oflow=0 tx_other=1
509: 0x8000 Rx
1: 0x4000 Nonb
n_zero_refs=66 n_freepkts=50 estimated_free_nonb=16
free_nonb=0 nonb_pkt_pool=a39ffff
```

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 576 buffers have been allocated and 50 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, 525 packet buffers have been allocated, 522 have been pushed to the receive ring - and are available for incoming packets, and 3 are currently in the receive queue for the application to process.

On the transmit side of the stack, only 1 packet buffer is currently allocated and because it is not currently in the transmit ring and is not in an overflow buffer it is counted as tx_other. The remaining values are calculations based on the packet buffer values.

TCP Application STATS

The following `onload_stackdump` commands can be used to monitor accelerated TCP connections:

```
onload_stackdump tcp_stats
```

Field	Description
`tcp_active_opens`	Number of socket connections initiated by the local end
`tcp_passive_opens`	Number of sockets connections accepted by the local end
`tcp_attempt_fails`	Number of failed connection attempts
`tcp_estab_resets`	Number of established connections which were subsequently reset
`tcp_curr_estab`	Number of socket connections in the established or close_wait states
`tcp_in_segs`	Total number of received segments - includes errored segments
`tcp_out_segs`	Total number of transmitted segments - excluding segments containing only retransmitted octets
`tcp_retran_segs`	Total number of retransmitted segments
`tcp_in_errs`	Total number of segments received with errors
`tcp_out_rsts`	Number of reset segments sent

```
onload_stackdump more_stats | grep tcp
```

Field	Description
`tcp_has_recvq`	Non zero if receive queue has data ready
`tcp_recvq_bytes`	Total bytes in receive queue
`tcp_recvq_pkts`	Total packets in receive queue
`tcp_has_recv_reorder`	Non zero if socket has out of sequence bytes
`tcp_recv_reorder_pkts:`	Number of out of sequence packets received
`tcp_has_sendq`	Non zero if send queues have data ready
`tcp_sendq_bytes`	Number of bytes currently in all send queues for this connection

`tcp_sendq_pkts`	Number of packets currently in all send queues for this connection
`tcp_has_inflight`	Non zero if some data remains unacknowledged
`tcp_inflight_bytes`	Total number of unacknowledged bytes
`tcp_inflight_pkts`	Total number of unacknowledged packets
`tcp_n_in_listenq`	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
`tcp_n_in_acceptq`	Number of sockets (summed across all listening sockets) that are currently queued waiting for the local application to call `accept()`

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 e.g.

```
onload_stackdump lots
onload_stackdump 2 lots
```

For descriptions of the statistics refer to the output from the following command:

```
onload_stackdump describe_stats
```

The following tables describe the output from the `onload_stackdump lots` command for:

- TCP stack
- TCP established connection socket
- TCP listening socket
- UDP socket

Within the tables the following abbreviations are used:

- *rx = receive (or receiver), tx = transmit (or send)*
- *pkts = packets, skts = sockets*
- *Max = maximum, num = number, seq = sequence number*

Stackdump Output: TCP Stack

`onload_stackdump lots`	Command entered
`ci_netif_dump: stack=7 name=`	Stack id and stack name as set by `EF_NAME`.
`ver=201310 uid=0 pid=21098`	Onload version, user id and process id of creator process
`lock=20000000 LOCKED   nics=3 primed=1`	Internal stack lock status nics = bitfield identifies adapters used by this stack e.g. 3 = 0x11 - so stack is using NICs 1 and 2. primed = 1 means the event queue will generate an interrupt when the next event arrives
`sock_bufs: max=1024 n_allocated=4`	Max number of sockets buffers which can be allocated, and number currently in use. Socket buffers are also used by pipes.
`pkt_bufs: size=2048 max=32768 alloc=576 free=57 async=0`	Packet buffers: A total of 32768 (each of 2048 bytes) pkt buffers are available to this stack. 576 have been allocated of which 57 are free and can be reused by either receive or transmit rings. async = buffers that are not free, not being used, not being reaped - i.e in a state waiting to be returned for reuse

`pkt_bufs: rx=517 rx_ring=514 rx_queued=3`	Receive packet buffers: A total of 517 pkt buffers are currently in use, 514 have been pushed to the receive ring, 3 are in the application's receive queue If the **CRITICAL** flag is displayed it indicates a memory pressure condition in which the number of packets in the receive socket buffers (rx=517) is approaching the EF_MAX_RX_PACKETS value. If the **LOW** flag is displayed it indicates a memory pressure condition when there are not enough packet buffers available to refill the RX descriptor ring.
`pkt_bufs: tx=2 tx_ring=1 tx_oflow=0 tx_other=1`	Transmit packet buffers: A total of 2 pkt buffers are currently in use, 1 remains in the transmit ring, 0 buffers have overflowed. tx_other = pkt buffers not in use by transmit and not in tx_ring or tx_oflow queue
`time: netif=5eb5c61 poll=5eb5c61 now=5eb5c61 (diff=0.000sec)`	Internal timer values
`ci_netif_dump_vi: stack=7 intf=0 vi_instance=87 hw=0C0`	Data describes the stack's virtual interface to the NIC
`evq: cap=2048 current=16de30 is_32_evs=0 is_ev=0`	Event queue data: cap - max num of events queue can hold current - current event queue location is_32_evs - is 1 if there are 32 or more events pending is_ev - is 1 if there are any events pending

`rxq: cap=511 lim=511 spc=1 level=510 total_desc=93666`	Receive queue data: cap - total capacity lim - max fill level for receive descriptor ring, specified by `EF_RXQ_LIMIT` spc - amount of free space in receive queue - how many descriptors could be added before the receive queue becomes full level - how full the receive queue currently is total_desc - 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: cap - total capacity lim - max fill level for transmit descriptor ring, specified by `EF_TXQ_LIMIT` spc - amount of free space in the transmit queue - how many descriptors could be added before the transmit queue becomes full level - how full the transmit queue currently is pkts - how many packets are represented by the descriptors in the transmit queue oflow - how many packets are in the overflow transmit queue (i.e. waiting for space in the NIC's transmit queue)
`txq: tot_pkts=93669 bytes=0`	Total number of packets sent and number of packet bytes currently in the queue
`ci_netif_dump_extra: stack=7`	Additional data follows
`in_poll=0 post_poll_list_empty=1 poll_did_wake=0`	Stack Polling Status: in_poll = process is currently polling post_poll_list_empty=1, (1=true, 0=false) tasks to be done once polling is complete poll_did_wake = while polling, the process identified a socket which needs to be woken following the poll

`rx_defrag_head=-1 rx_defrag_tail=-1`	Reassembly sequence numbers. -1 means no re-assembly has occurred
`tx_tcp_may_alloc=1 nonb_pool=1 send_may_poll=0 is_spinner=0`	TCP buffer data: tx_tcp_may_alloc=num pkt buffers tcp could use nonb_pool= number of pkt buffers available to tcp process without holding lock send_may_poll=0 is_spinner= TRUE if a thread is spinning
`send_may_poll=0`	0
`hwport_to_intf_i=0,-1,-1,-1,-1,-1 intf_i_to_hwport=0,0,0,0,0,0`	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

`ci_netif_dump_reap_list: stack=7`
`  7:2`
`  7:1`
	Identifies sockets that have buffers which can be freed e.g. 7:2 = stack 7 socket 2

Stackdump Output: TCP Established Connection Socket

`TCP 7:1 lcl=192.168.1.2:50773 rmt=192.168.1.1:34875 ESTABLISHED`	Socket Configuration. Stack:socket id, local and remote ip:port address, TCP connection is ESTABLISHED
`lock: 10000000 UNLOCKED`	Internal stack lock status
`rx_wake=0000b6f4(RQ) tx_wake=00000002 flags:`	Internal sequence values that are incremented each time a queue is ’woken’
`addr_spc_id=fffffffffffffffe s_flags: REUSE BOUND`	Address space identifier in which this socket exists and flags set on the socket Allow bind to reuse local addresses
`rcvbuf=129940 sndbuf=131072 rx_errno=0 tx_errno=0 so_error=0`	Socket receive buffer size, send buffer size, rx_errno = ZERO whilst data can still arrive, otherwise contains error code. tx_errno = ZERO if transmit can still happen, otherwise contains error code. so_error = current socket error (0 = no error)
`tcpflags: TSO WSCL SACK ESTAB`	TCP flags currently set for this sockets
`TCP state: ESTABLISHED`	State of the TCP connection
`snd: up=b554bb86 una-nxt-max=b554bb86-b554bb87-b556b6a6 enq=b554bb87`	TCP sequence numbers. up = (urgent pointer) sequence of byte following the OOB byte una-nxt-max = sequence number of first unacknowledged byte, sequence number of next byte we expect to be acknowledged and max = sequence of last byte in the current send window enq = sequence number of last byte currently queued for transmit

`send=0(0) pre=0 inflight=1(1) wnd=129824 unused=129823`	Send Data. send = number of pkts(bytes) sent pre = number of pkts in pre-send queue. A process can add data to the prequeue when it is prevented from sending the data immediately. The data will be sent when the current sending operation is complete inflight = number of pkts(bytes) sent but not yet acknowledged wnd = receiver’s advertised window size (bytes) and number of free (unused) space (bytes) in that window
`snd: cwnd=49733+0 used=0 ssthresh=65535 bytes_acked=0 Open`	Congestion window (cwnd). cwnd = congestion window size (bytes) used = portion of the cwnd currently in use slowstart thresh - number of bytes that have to be sent before process can exit slow start bytes_acked = number of bytes acknowledged - this value is used to calculate the rate at which the congestion window is opened current cwnd status = OPEN
`snd:Onloaded(Valid) if=6 mtu=1500 intf_i=0 vlan=0 encap=4`	Onloaded = can reach the destination via an accelerated interface. (Valid) = cached control plane information is up-to-date, can send immediately using this information. (Old) = cached control plane information may be out-of-date. On next send Onload will do a control plane lookup - this will add some latency.
`rcv: nxt-max=0e9251fe-0e944d1d current=0e944d92  FASTSTART FAST`	Receiver Data. nxt-max = next byte we expect to receive and last byte we expect to receive (because of window size) current = byte currently being processed

`rob_n=0 recv1_n=2 recv2_n=0 wnd adv=129823 cur=129940 usr=0`	Reorder buffer. Bytes received out of sequence are put into a reorder buffer awaiting further bytes before reordering can occur. rob_n = num of bytes in reorder buffer recv1_n = num of bytes in general reorder buffer recv2_n = num of bytes in urgent data reorder buffer wnd adv = receiver advertised window size cur = current receive window size usr = current tcp stack user
`async: rx_put=-1 rx_get=-1 tx_head=-1`	Asynchronous queue data.
`eff_mss=1448 smss=1460 amss=1460 used_bufs=2 uid=0 wscl s=1 r=1`	Max Segment Size. eff_mss = effective_mss smss = sender mss amss = advertised mss used_bufs = number of transmit buffers used user id that created this socket(0 = root) wscl s/r = parameters to window scaling algorithm

`srtt=01 rttvar=000 rto=189 zwins=0,0`	Round trip time (RTT) - all values are timer ticks.
	srtt = smothed RTT value
	rttvar = RTT variation
	rto = current RTO timeout value
	zwins = zero windows, times when advertised window has gone to zero size.

`retrans=0 dupacks=0 rtos=0 frecs=0 seqerr=0 ooo_pkts=0 ooo=0`	Re-transmissions. number of retrans which have occurred dupacks = number of duplicate acks received rtos = number of retrans timeouts frecs = number of fast recoverys seqerr = number of sequence errors number of out of sequence pkts number of out of order events
`timers:`	Currently active timers
`tx_nomac`	Number of TCP packets sent via the OS using raw sockets when up to date ARP data is not available.

Stackdump Output: TCP Stack Listen Socket

`TCP 7:3 lcl=0.0.0.0:50773 rmt=0.0.0.0:0 LISTEN`	Socket configuration. stack:socket id, LISTENING socket on port 50773 local and remote addresses not set - not bound to any IP addr
`lock: 10000000 UNLOCKED`	Internal stack lock status
`rx_wake=00000000 tx_wake=00000000    flags:`	Internal sequence values that are incremented each time a queue is 'woken'
`addr_spc_id=fffffffffffffffe s_flags: REUSE BOUND PBOUND`	Address space identifier in which this socket exists and flags set on the socket Allow bind to reuse local port
`rcvbuf=129940 sndbuf=131072 rx_errno=6b tx_errno=20 so_error=0`	Receive Buffer. socket receive buffer size, send buffer size, rx_errno = ZERO whilst data can still arrive, otherwise contains error code. tx_errno = ZERO if transmit can still happen, otherwise contains error code. so_error = current socket error (0 = no error)
`tcpflags: WSCL SACK`	Flags advertised during handshake
`listenq: max=1024 n=0`	Listen Queue. queue of half open connects (SYN received and SYNACK sent - waiting for final ACK) n - number of connections in the queue
`acceptq: max=5 n=0 get=-1 put=-1 total=0`	Accept Queue. queue of open connections, waiting for application to call accept(). max = max connections that can exist in the queue n = current number of connections get/put = indexes for queue access total = num of connections that have traversed this queue

`epcache: n=0 cache=EMPTY pending=EMPTY`	Endpoint cache. n = number of endpoints currently known to this socket cache = EMPTY or yes if endpoints are still cached pending = EMTPY or yes if endpoints stilll have to be cached
`defer_accept=0`	Number of times TCP_DEFER_ACCEPT kicked in - see TCP socket options
`l_overflow=0 l_no_synrecv=0 a_overflow=0 a_no_sock=0 ack_rsts=0 os=2`	l_overflow = number of times listen queue was full and had to reject a SYN request l_no_synrecv = number of times unable to allocate internal resource for SYN request a_overflow = number of times unable to promote connection to the accept queue which is full a_no_sock = number of times unable to create socket ack_rsts = number of times received an ACK before SYN so the connection was reset os=2 there are 2 sockets being processed in the kernel

Stackdump Output: UDP Socket:

`UDP 4:1 lcl=192.168.1.2:38142 rmt=192.168.1.1:42638 UDP`	Socket Configuration. stack:socket id, UDP socket on port 38142 Local and remote addresses and ports
`lock: 20000000 LOCKED`	Stack internal lock status
`rx_wake=000e69b0 tx_wake=000e69b1 flags:`	Internal sequence values that are incremented each time a queue is 'woken'
`addr_spc_id=fffffffffffffffe s_flags: REUSE`	Address space identifier in which this socket exists and flags set on the socket Allow bind to reuse local addresses
`rcvbuf=129024 sndbuf=129024 rx_errno=0 tx_errno=0 so_error=0`	Buffers. socket receive buffer size, send buffer size, rx_errno = ZERO whilst data can still arrive, otherwise contains error code. tx_errno = ZERO if transmit can still happen, otherwise contains error code. so_error = current socket error (0 = no error)
`udpflags: FILT MCAST_LOOP RXOS`	Flags set on the UDP socket

`mcast_snd: intf=-1 ifindex=0 saddr=0.0.0.0 ttl=1 mtu=1500`	Multicast.
	intf = multicast hardware port id (-1 means port was not set)
	ifindex = interface (port) identifier
	saddr = IP address
	tt1 = time to live (default for multicast =1)
	mtu = max transmission unit size

`rcv: q_bytes=0 q_pkts=0 reap=2 tot_bytes=30225920 tot_pkts=944560`	Receive Queue. q_bytes = num bytes currently in rx queue q_pkts = num pkts currently in rx queue tot_bytes = total bytes received tot_pkts = total pkts received

`rcv: oflow=0(0%) drop=0 eagain=0 pktinfo=0` `q_max=0`	Overflow Buffer. oflow = number of datagrams in the overflow queue when the socket buffer is full. drop = number of datagrams dropped due to running out of packet buffer memory. eagain = number of times the application tried to read from a socket when there is no data ready - this value can be ignored on the rcv side pktinfo = number of times IP_PKTINFO control message was received q_max = max depth reached by the receive queue (bytes)
`rcv: os=0(0%) os_slow=0 os_error=0`	Number of datagrams received via: os = operating system os_slow = operating system slow socket os_error = recv() function call via OS returned an error
`snd: q=0+0 ul=944561 os=0(0%) os_slow=0(0%)`	Send values. q = number of bytes sent to the interface but not yet transmitted ul = number of datagrams sent via onload os = number of datagrams sent via OS os_slow number of datagrams sent via OS slow path
`snd: cp_match=0(0%)`	Unconnected UDP send. cp_match = number dgrams sent via accelerated path and percent this is of all unconnected send dgrams
`snd: lk_poll=0(0%) lk_pkt=944561(100%) lk_snd=0(0%)`	Stack internal lock. lk_poll = number of times the lock was held while we poll the stack lk_pkt = number of pkts sent while holding the lock lk_snd = number of times the lock was held while sending data

`snd: lk_defer=0(0%) cached_daddr=0.0.0.0`	Sending deferred to the process/thread currently holding the lock
`snd: eagain=0 spin=0 block=0`	eagain = 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 may indicate transmit issues. spin = number of times process had to spin when the send queue was full block = number of times process had to block when the send queue was full
`snd: poll_avoids_full=0 fragments=0 confirm=0`	poll_avoids_full = number of times polling created space in the send queue fragments = number of (non first) fragments sent confirm = number of datagrams sent with MSG_CONFIRM flag
`snd: os_late=0 unconnect_late=0`	os_late = number of pkts sent via OS after copying unconnect_late = number of pkts silently dropped when process/thread becomes disconnected during a send procedure

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
```

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 on page 93 or use the `onload_stackdump doc` command.

Appendix F: Solarflare sfnettest

Introduction

Solarflare sfnettest is a set of benchmark tools and test utilities supplied by Solarflare for benchmark and performance testing of network servers and network adapters. The sfnettest is available in binary and source forms from:

http://www.openonload.org/

Download the sfnettest-<version>.tgz source file and unpack using the tar command.

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

Run the `make` utility from the `/sfnettest-<version>/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:

Option	Description
--port	server port
--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

Option	Description
--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. **This option will override any value set by taskset on the same command line**.
--n-pings	number of ping messages

Option	Description
--n-pongs	number of pong messages
--nodelay	enable TCP_NODELAY

Standard options:

Option	Description
-? --help	this message
-q --quiet	quiet
-v --verbose	display more information

Example TCP latency command lines:

```
[root@server]# onload --profile=latency taskset -c 1 ./sfnt-pingpong
[root@client]# onload --profile=latency taskset -c 1 ./sfnt-pingpong --
maxms=10000 --affinity "1;1" tcp <server-ip>
```

Example UDP latency command lines:

```
[root@server]# onload --profile=latency taskset -c 9 ./sfnt-pingpong
[root@client]# onload --profile=latency taskset -c 9 ./sfnt-pingpong --
maxms=10000 --affinity "9;9" udp <server_ip>
```

Example output:

```
# version: 1.4.0-modified
# src: 13b27e6b86132da11b727fbe552e2293
# date: Sat Apr 21 11:56:22 BST 2012
# uname: Linux server4.uk.level5networks.com 2.6.32-220.el6.x86_64 #1 SMP Wed Nov
9 08:03:13 EST 2011 x86_64 x86_64 x86_64 GNU/Linux
# cpu: model name       : Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz
# 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) RTT/2 latency for increasing packet sizes including the 99% percentile and standard deviation for these results. A message size of 32 bytes has a mean latency of **2.4** microsecs with a 99%ile latency less than **2.7** microsecs.

sfnt-stream

The `sfnt-stream` application measures RTT latency (not 1/2 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.

Solarflare 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:

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)

Option	Description
--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 e.g. "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. **This option will override any value set by taskset on the same command line**.
--rtt-iter	iterations for RTT measurement

```
standard options:
```

Option	Description
-? --help	this message
-q --quiet	quiet
-v --verbose	display more information
--version	display version information

Example command lines client/server

```
# ./sfnt-stream  (server)
# ./sfnt-stream --affinity 1,1 udp <server-ip>  (client)
# ./taskset -c 1 ./sfnt-stream --affinity="3,5;3" --mcastintf=eth4 udp
<remote-ip>  (client)
```

Bonded Interfaces: sfnt-stream

The following example configures a single bond, having two slaves interfaces, on each machine. Both client and server machines use eth4 and eth5.

Client Configuration

```
[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
```

Server Configuration

```
[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
```

Output Fields:

All time measurements are nanoseconds unless otherwise stated.

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

Appendix G: onload_tcpdump

Introduction

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 dedicated 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.

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.

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*'
```

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

- Currently `onload_tcpdump` captures only packets from onload stacks and not from kernel stacks.
- 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 may notice 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.

SolarCapture

Solarflare's 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 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) available from https://support.solarflare.com/.

Appendix H: ef_vi

The Solarflare 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.

Characteristics

- 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.

Using ef_vi

Users of ef_vi must first allocate a virtual interface (VI), encapsulated by the type `"ef_vi"`. A VI includes:

- A receive descriptor ring (for receiving packets).
- A transmit descriptor ring (for sending packets).
- An event queue (to receive notifications from the hardware).

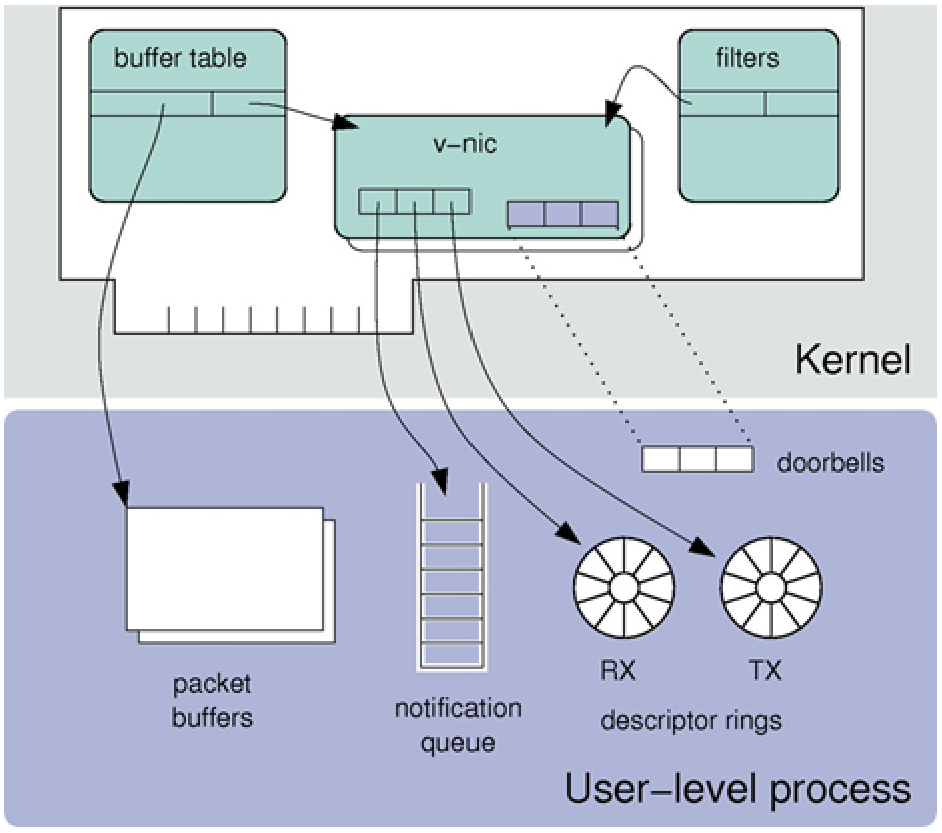


Figure 21: Virtual Interface Components

To transmit a packet, the application writes the packet contents (including all headers) into one or more packet buffers, and calls `ef_vi_transmit()`. One or more descriptors that describe the packet are queued in the transmit ring, and a doorbell is "rung" to inform the adapter that the transmit ring is non-empty.

To receive packets, descriptors, each identifying a buffer, are queued in the receive ring -- `ef_vi_receive_init()` and `_post()`(Refer to Handling Events on page 171). When packets arrive at the VI, they are written into the buffers in fifo order.

The event queue is a channel from the adapter to software which notifies software when packets arrive from the network, and when transmits complete (so that the buffers can be freed or reused). The application retrieves these events by calling `ef_eventq_poll()`.

The buffers used for packet data must be pinned so that they cannot be paged, and they must be registered for DMA with the network adapter. The type "`ef_iobufset`" encapsulates a set of buffers. The adapter uses a special address space to identify locations in these buffers, and such addresses are designated by the type "`ef_addr`".

Filters are the means by which the adapter decides where to deliver packets it receives from the network. By default all packets are delivered to the kernel network stack. Filters are added by the application to direct received packets to a given VI.

Protection Domain

The protection domain is a collection of VIs and memory regions tied to a single user interface. A memory region can be registered with different protection domains. This is useful for zero-copy forwarding. Refer also to ef_vi - Physical Addressing Mode on page 169 and ef_vi - Scalable Packet Buffer Mode on page 169.

Any memory region in a protection domain can be used with any VI in the protection domain.

```
int  ef_pd_alloc(ef_pd *pd,  ef_driver_handle  pd_dh,
                int  ifindex,  enum  ef_pd_flags  flags);
```

Figure 22: Create a Protection Domain

Virtual Interface

```
int  ef_vi_alloc_from_pd(ef_vi *vi,  ef_driver_handle  vi_dh,  ef_pd
                   *pd,  ef_driver_handle  pd_dh,  int
                   eventq_cap,int  rxq_cap,  int  txq_cap,  ef_vi
                   *opt_evq,
                   ef_driver_handle  opt_evq_dh,  enum  ef_vi_flags
                   flags));
```

Figure 23: Allocate a Virtual Interface

A virtual interface consists of a transmit queue, receive queue and event queue. A VI can be allocated with just one of the above three components, but if the event queue is omitted an optional VI that has an event queue must be specified.

Memory Region

```
int ef_memreg_alloc(ef_memreg* mr,  ef_driver_handle
                   mr_dh,  ef_pd*  pd,  ef_driver_handle
                   pd_dh, void*  p_mem,  int
                   len_bytes);
```

Figure 24: Allocate a Memory Region

The `ef_memreg_alloc()` function registers a block of memory to be used with VIs within a protection domain. Performance will be improved when cache lined memory is set to a use a minimum 4KB or aligning the memory region on a 2MB boundary to use huge pages.

Receive Packet Buffers

```
int ef_vi_receive_prefix_len(ef_vi* vi)
```

Returns the size (bytes) of the received packet meta data prefix.

```
int ef_vi_receive_buffer_len(ef_vi* vi)
```

A packet buffer should be at least as large as the value returned from ef_vi_receive_buffer_len.

Address Space

Three modes are possible for setting up the adapter address space; buffer table, SR-IOV, or no translation.

The buffer table is a block of memory on the adapter that does the translation from buffer ID to physical addresses. When using a SFN5000 or SFN6000 series adapter there are 120,000 entries in the buffer table. Each buffer is mapped in each adapter so, regardless of the number of NICs installed, there are a total of 120,000 packet buffers in the system. The SFN7000 series adapters can employ more than the 120K packet buffers without the need to use Scalable packet buffer mode - refer to Large Buffer Table Support on page 62 for details.

SR-IOV employs the IOMMU virtual addresses. The IOMMU removes the 120K buffer limitation of the buffer table. See Scalable Packet Buffer Mode on page 62 for details of how to configure SR-IOV and enable the system IOMMU.

The no translation mode requires the application to identify actual physical addresses to the adapter which means the application can direct the adapter to read/write any piece of memory. It is important to ensure that packet buffers are page aligned.

ef_vi - Physical Addressing Mode

An ef_vi application can use physical addressing mode, see Physical Addressing Mode on page 70. To enable physical addressing mode set the environment variable `EF_VI_PD_FLAGS=phys`.

ef_vi - Scalable Packet Buffer Mode

Using Scalable Packet Buffer Mode, packet buffers are allocated from the kernel IOMMU instead of from the buffer table. An ef_vi application can enable this mode by setting `EF_VI_PD_FLAGS=vf`. The caveats applicable to an Onload application will also apply i.e. SR-IOV must be enabled, the kernel must support an IOMMU. Refer to Scalable Packet Buffer Mode on page 62 for configuration details and caveats.

Filters

- The application is able to set multiple types of filters on the same VI.
- If a filter already exists, an error is returned.
- Cookies are used to remove filters.
- De-allocating a VI removes the filters set for the VI
- NOTE: IP filters do not match IP fragments, which are therefore received by the kernel stack. If this is an issue, layer 2 filters should be installed by the user.

```
void ef_filter_spec_init(ef_filter_spec* fs, enum
ef_filter_flags flags);


int ef_filter_spec_set_

{ip4_local,ip4_full,eth_local,unicast_all,multicast_all};


int ef_vi_filter_add(ef_vi* vi, ef_driver_handle dh, const
ef_filter_spec* fs, ef_filter_cookie* filter_cookie_out);


int ef_vi_filter_del(ef_vi* vi, ef_driver_handle dh,
ef_filter_cookie* filter_cookie);


int ef_filter_spec_set_block_kernel(ef_filter_spec* fs);
```

Figure 25: Creating Filters

An issue has been identified in Onload-201310 such that if two applications attempt to install the same unicast|multicast all filters, an error will be returned and either application could receive the packets (both not both). If either application then exits, the packets are returned through the kernel and the remaining application will not receive the packets. This will be addressed in a future Onload release.

Transmitting Packets

The packet buffer memory must have been previously registered with the protection domain. If the transmit queue is empty when the doorbell is rung, 'TX PUSH' is used. In 'TX_PUSH', the doorbell is rung and the address of the packet buffer is written in one shot improving latency. TX_PUSH can cause ef_vi to poll for events, to check if the transmit queue is empty, before sending which can lead to a latency versus throughput trade off in some scenarios.

```
// construct  packet  with  proper  headers

// Post  on  the  transmit  ring
ef_vi_transmit_init(&vi,  addr,  len,  id);
// Ring doorbell
ef_vi_transmit_push(&vi);
```

Figure 26: Transmit Packets

Handling Events

- Receive descriptors should be posted to the adapter receive ring in multiples of 8. When an application pushes 10 descriptors, ef_vi will push 8 and ev_vi will ignore descriptor batch sizes < 8. Users should beware that if the rx ring is empty and the application pushes < 8 descriptors before blocking on the event queue, the application will remain blocked as there are no descriptors available to receive packets so nothing gets posted to the event queue.
- The batch size for polling should be greater than the batch size for refilling to detect when the receive queue is going empty.
- Packets of a size smaller than the interface MTU but larger than packet buffer sizes are delivered in multiple buffers as jumbos.
- Since the adapter is cut-through, errors in receiving packets like multicast mismatch, CRC errors, etc. are delivered along with the packet. The software must detect these errors and recycle the associated packet buffers.

Using ef_vi Example

```
static void handle_poll(ef_vi *vi)
{
  ef_event  events[POLL_BATCH_SIZE];
  int  n_ev = ef_eventq_poll(&vi, events,  POLL_BATCH_SIZE);
  for( i = 0; i < n_ev;  ++i ) {
    switch(  EF_EVENT_TYPE(events[i])  ) {
    case  EF_EVENT_TYPE_RX:
      // Accumulate used
      buffer break;
    case  EF_EVENT_TYPE_TX:
      /* Each EF_EVENT_TYPE_TX can signal  multiple  completed sends */
      int num_completed  = ef_vi_transmit_unbundle(vi,  events[i],
      &dma_id);
      break;
    case  EF_EVENT_TYPE_RX_DISCARD:
    case  EF_EVENT_TYPE_RX_NO_DESC_TRUNC:
      /* Discard events also use up buffers */
      // Accumulate buffer in user
      space break;
    default:
      /* Other error types */
    }
  }
}

static  void refill_rx_ring(ef_vi  *vi)
{
  if( ef_vi_receive_space(&vi)  < REFILL_BATCH_SIZE  )
    return;
  int refill_count = REFILL_BATCH_SIZE;
  /* Falling  too low? */
  if( ef_vi_receive_space(&vi)  > ef_vi_receive_capacity(&vi)  / 2 )
    refill_count  = ef_vi_receive_space(&vi);
  /* Enough free buffers? */
  if(  refill_count > free_bufs_in_sw  )
    refill_count  = free_bufs_in_sw;
  /* Round down to batch size */
  refill_count  &= ~(REFILL_BATCH_SIZE - 1);
  if(  refill_count ) {
    while( refill_count  ) {
      ef_vi_receive_init(...);
      --refill_count;
    }
    ef_vi_receive_push(&vi);
  }
}

int  main(int argc,  char *argv[])  {
---
  while(  1 ) {
    poll_events(&vi);
    refill_rx_ring(&vi);
  }
}
```

Example Applications

Solarflare ef_vi comes with a range of example applications - including source code and make files.

Application	Description
efforward	Forward packets between two interfaces without modification.
efpingpong	Ping a simple message format between two interfaces.
efpio	Pingpong application that uses PIO.
efrss	Forward packets between two interfaces without modification, spreading the load over multiple VIs and threads.
efsink	Receives streams of packets on a single interface.

Building Example Applications

The ef_vi example applications are built along with the Onload installation and will be present in the `/Onload-<version>/build/gnu_x86_64/tests/ef_vi` subdirectory. In the build directory there will be gnu, gnu_x86_64, x86_64_linux-<kernel version> directories. Files under the gnu directory are 32bit (if these are built), files under the gnu_x86_64 are 64bit.

Source code files for the example applications exist in the `/Onload-<version>/src/tests/ef_vi` subdirectory.

To rebuild the example applications you must have the `Onload-<version>/scripts` subdirectory in your path and use the following procedure:

```
[root@server01 Onload-201109]# cd scripts/
[root@server01 scripts]# export PATH="$PWD:$PATH"
[root@server01 scripts]# cd ../build/_gnu_x86_64/tests/ef_vi/
[root@server01 ef_vi]# make clean
[root@serverr01 ef_vi]# make
```

Appendix I: onload_iptables

Description

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[1] copy of the kernel iptables rules into Onload firewall rules 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 may 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 i.e. 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.

1. Subsequent changes to kernel iptables will not be reflected in the Onload firewall.

Files

When the Onload drivers are loaded, firewall rules exist in the Linux `proc` psuedo file system at:

```
/proc/driver/sfc_resource
```

Within this directory the *firewall_add, firewall_del* and *resources* files will be present. These files are writeable 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]]
     action=(ACCELERATE|DECELERATE)

rule=n if=ethN protocol=eth mac=xx:xx:xx:xx:xx:xx[/FF:FF:FF:FF:FF:FF]
     action=(ACCELERATE|DECELERATE)
```

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 behaviour).

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 e.g.

```
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 & 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 3:

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 that there should be no spaces i.e. 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.
`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.

Table 3:

Error Message	Description
`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.

Appendix J: Solarflare efpio Test Application

The openonload-201310 distribution includes the command line efpio test application to measure latency of the Solarflare ef_vi layer 2 API with PIO. The efpio application is a single thread ping/pong. When all iterations are complete the client side will display the round-trip time.

By default efpio downloads a packet to the adapter at start of day and transmits this same packet on every iteration of the test. The **–c** option can be used to test the latency of ef_vi using PIO to transfer a new transmit packet to the adapter on every iteration.

With the onload distribution installed efpio will be present in the following directory:

```
~/openonload-201310/build/gnu_x86_64/tests/ef_vi
```

efpio Options

```
./efpio –help
usage:
  efpio [options] <ping|pong> <interface>
        <local-ip-intf> <local-port>
        <remote-mac> <remote-ip-intf> <remote-port>
```

Table 4: efpio Options

Parameter	Description
interface	the local interface to use e.g. eth2
local-ip-intf	local interface IP address/host name
local-port	local interface IP port number to use
remote-mac	MAC address of the remote interface
remote-ip-intf	remote server IP address/host name
remote-port	remote server port number

```
options:
  -n <iterations>         - set number of iterations
  -s <message-size>       - set udp payload size
  -w                      - sleep instead of busy wait
  -v                      - use a VF
  -p                      - physical address mode
  -t                      - disable TX push
  -c                      - copy on critical path
```

To run efpio

The efpio 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> ./efpio pong eth<N> <local-ip> 8001 <server2-mac> <server2-
     ip> 8001
# ef_vi_version_str: 201306-7122preview2
# udp payload len: 28
# iterations: 100000
# frame len: 70
```

2 On the client side (server2)

```
taskset –c <M> ./efpio ping eth<N> <local-ip> 8001 <server1-mac> <server1-
     ip> 8001
# ef_vi_version_str: 201306-7122preview2
# udp payload len: 28
# iterations: 100000
# frame len: 70
round-trip time: 2.848 µs

M = cpu core, N = Solarflare adapter interface.
```