SOLARFLARE®
A XILINX COMPANY

Cloud Onload® Netty.io Cookbook

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

A list of patents associated with this product is at http://www.solarflare.com/patent

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2019 Xilinx, Inc. Xilinx, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

SF-121964-CD

Issue 2

Table of Contents

1 Introduction 1
1.1 About this document 1
1.2 Intended audience 2
1.3 Registration and support 2
1.4 Download access 2
1.5 Further reading 2

2 Overview 3
2.1 Netty.io overview 3
2.2 Wrk2 overview 4
2.3 Cloud Onload overview 4

3 Summary of benchmarking 6
3.1 Overview of Netty.io benchmarking 6
3.2 Architecture for Netty.io benchmarking 7
3.3 Netty.io benchmarking process 8

4 Evaluation 10
4.1 General server setup 10
4.2 wrk2 client 11
4.3 Netty-based HTTP server 11
Static files for HTTP servers 12
4.4 Graphing the benchmarking results 12

5 Benchmark results 13
5.1 Results 14
25GbE with 16 byte payload 14
25GbE with 32 byte payload 15
25GbE with 64 byte payload 16
25GbE with 128 byte payload 17
25GbE with 256 byte payload 18
25GbE with 512 byte payload 19
5.2 Analysis 20

 © Copyright 2019 Xilinx, Inc

A Cloud Onload profiles 21
A.1 The wrk-profile Cloud Onload profile 21
A.2 The Netty Cloud Onload profiles 22
The nettyio-performance profile 23
The nettyio-balanced profile 24

© Copyright 2019 Xilinx, Inc

1 Introduction

This chapter introduces you to this document. See:

- About this document on page 1
- Intended audience on page 2
- Registration and support on page 2
- Download access on page 2
- Further reading on page 2.

1.1 About this document

This document is the *Netty.io Cookbook* for Cloud Onload. It gives procedures for technical staff to configure and run tests, to benchmark Netty.io utilizing Solarflare's Cloud Onload and Solarflare NICs.

This document contains the following chapters:

- Introduction on page 1 (this chapter) introduces you to this document.
- Overview on page 3 gives an overviews of the software distributions used for this benchmarking.
- Summary of benchmarking on page 6 summarizes how the performance of Netty.io has been benchmarked, both with and without Cloud Onload, to determine what benefits might be seen.
- Evaluation on page 10 describes how the performance of the test system is evaluated.
- Benchmark results on page 13 presents the benchmark results that are achieved.

and the following appendixes:

- Cloud Onload profiles on page 21 contains the Cloud Onload profiles used for this benchmarking.

© Copyright 2019 Xilinx, Inc

1.2 Intended audience

The intended audience for this *Netty.io Cookbook* are:

- software installation and configuration engineers responsible for commissioning and evaluating this system
- system administrators responsible for subsequently deploying this system for production use.

1.3 Registration and support

Support is available from support@solarflare.com.

1.4 Download access

Cloud Onload can be downloaded from: https://support.solarflare.com/.

Solarflare drivers, utilities packages, application software packages and user documentation can be downloaded from: https://support.solarflare.com/.

The scripts and Cloud Onload profiles used for this benchmarking are available on request from support@solarflare.com.

Please contact your Solarflare sales channel to obtain download site access.

1.5 Further reading

For advice on tuning the performance of Solarflare network adapters, see the following:

- *Solarflare Server Adapter User Guide* (SF-103837-CD).

  This is available from: https://support.solarflare.com/.

For more information about Cloud Onload, see the following:

- *Onload User Guide* (SF-104474-CD).

  This is available from: https://support.solarflare.com/.

© Copyright 2019 Xilinx, Inc

2 Overview

This chapter gives an overview of the software distributions used for this benchmarking. See:

- Netty.io overview on page 3
- Wrk2 overview on page 4
- Cloud Onload overview on page 4.

2.1 Netty.io overview

Netty.io is an asynchronous event-driven network application framework written in Java for rapid development of maintainable high performance protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server. Netty.io has the following performance features:

- Better throughput, lower latency
- Less resource consumption
- Minimized unnecessary memory copy.

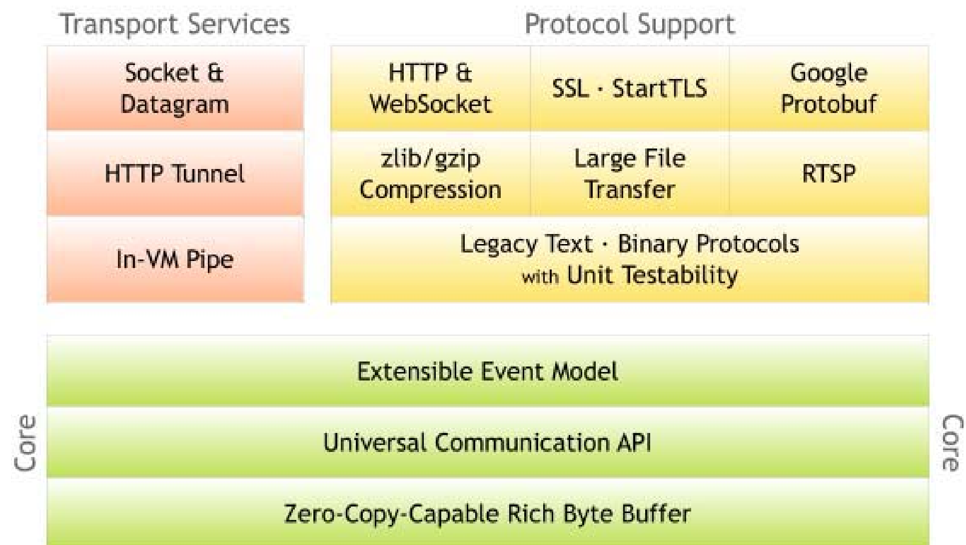

Figure 1: Netty.io architecture

© Copyright 2019 Xilinx, Inc

2.2 Wrk2 overview

Wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems such as epoll and kqueue. An optional LuaJIT script can perform HTTP request generation, response processing, and custom reporting.

`Wrk2` is `wrk` modified to produce a constant throughput load, and accurate latency details to the high 9s (it can produce an accurate 99.9999 percentile when run long enough). In addition to `wrk`'s arguments, `wrk2` takes a required throughput argument (in total requests per second) via either the `--rate` or `-R` parameters.

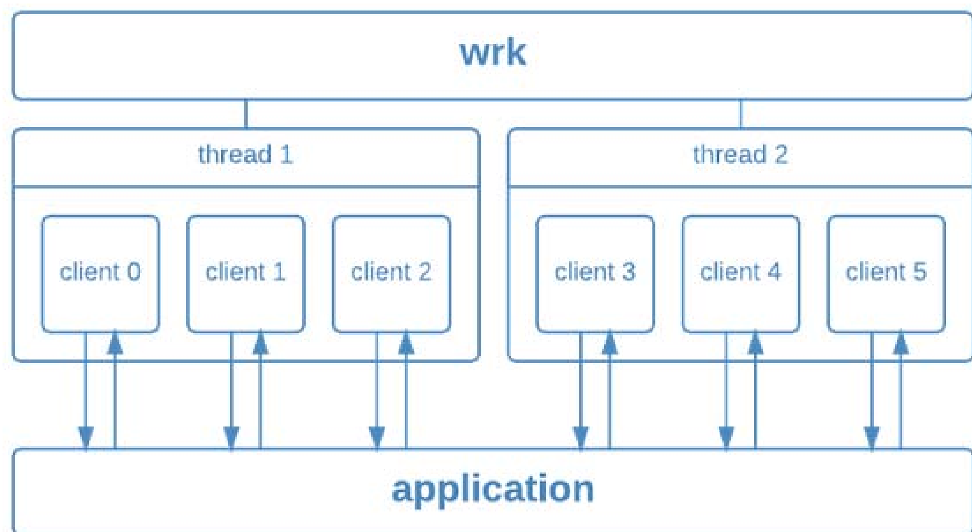

Figure 2: Wrk/wrk2 architecture

2.3 Cloud Onload overview

Cloud Onload is a high performance network stack from Solarflare (https://www.solarflare.com/) that dramatically reduces latency, improves CPU utilization, eliminates jitter, and increases both message rates and bandwidth. Cloud Onload runs on Linux and supports the TCP network protocol with a POSIX compliant sockets API and requires no application modifications to use. Cloud Onload achieves performance improvements in part by performing network processing at user-level, bypassing the OS kernel entirely on the data path.

Cloud Onload is a shared library implementation of TCP, which is dynamically linked into the address space of the application. Using Solarflare network adapters, Cloud Onload is granted direct (but safe) access to the network. The result is that the application can transmit and receive data directly to and from the network, without any involvement of the operating system. This technique is known as “kernel bypass”.

© Copyright 2019 Xilinx, Inc

When an application is accelerated using Cloud Onload it sends or receives data without access to the operating system, and it can directly access a partition on the network adapter.

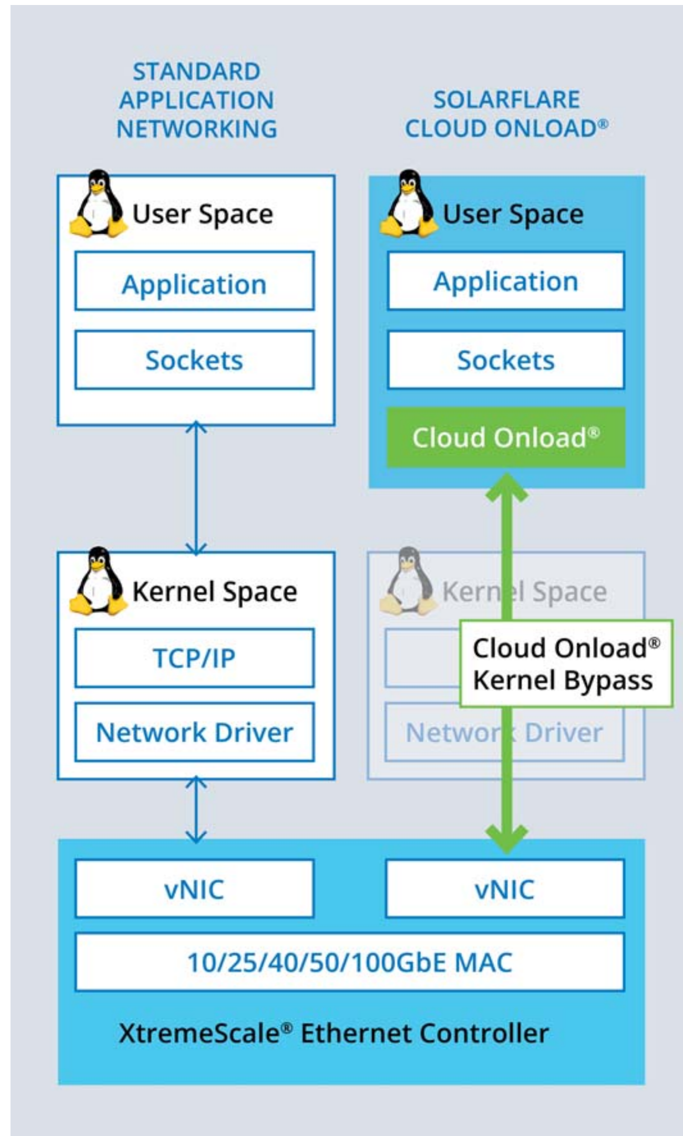

Figure 3: Cloud Onload architecture

© Copyright 2019 Xilinx, Inc

3 Summary of benchmarking

This chapter summarizes how the performance of Netty.io has been benchmarked, both with and without Cloud Onload, to determine what benefits might be seen. See:

- Overview of Netty.io benchmarking on page 6
- Architecture for Netty.io benchmarking on page 7
- Netty.io benchmarking process on page 8.

3.1 Overview of Netty.io benchmarking

The Netty.io benchmarking uses two servers:

- The *load server* runs multiple instances of wrk2 to generate requests.
- The *proxy server* runs multiple instances of Netty-based HTTP servers to service the requests.

Various benchmark tests are run, with the HTTP servers using the Linux kernel network stack.

The tests are then repeated, using Cloud Onload to accelerate the HTTP servers. Two different Cloud Onload profiles are used, that have different priorities:

- The balanced profile gives excellent throughput, with low latency. It has reduced CPU usage at lower traffic rates.
- The performance profile is latency focused. It constantly polls for network events to achieve the lowest latency possible, and so has higher CPU usage.

The results using the kernel network stack are compared with the results using the two different Cloud Onload profiles.

© Copyright 2019 Xilinx, Inc

3.2 Architecture for Netty.io benchmarking

Benchmarking was performed with two Dell R640 servers, with the following specification:

Server	Dell R640
Memory	96GB
NICs	1 × X2541 (single port 100G) 1 × X2522-25G (dual port 25G): • Each NIC is affinitized to a separate NUMA node.
CPU	2 × Intel® Xeon® Gold 6148 CPU @ 2.40GHz: • Each CPU is on a separate NUMA node • There are 20 cores per CPU • Hyperthreading is enabled to give 40 hyperthreads per NUMA node
OS	Red Hat Enterprise Linux Server release 7.5 (Maipo)
Software	Netty.io wrk2

Each server is configured to leave as many CPUs as possible available for the application being benchmarked.

Each server has 2 NUMA nodes. 2 Solarflare NICs are fitted, each affinitized to a separate NUMA node, and connected directly to the corresponding NIC in the other server:

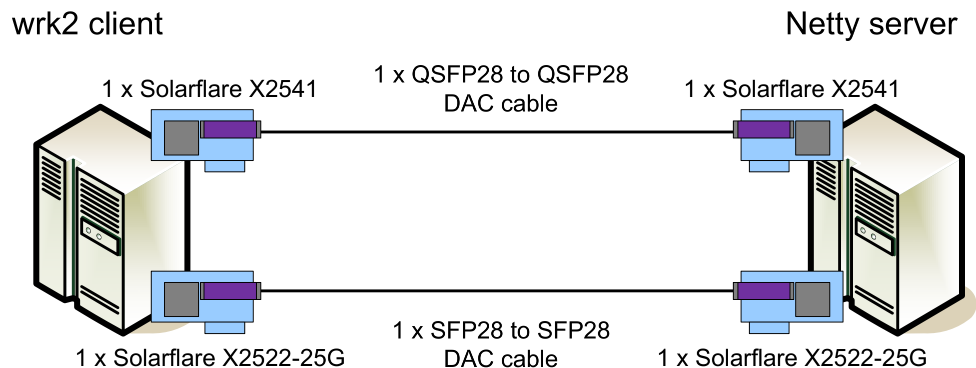

Figure 4: Architecture for Netty.io benchmarking

© Copyright 2019 Xilinx, Inc

3.3 Netty.io benchmarking process

These are the high-level steps we followed to complete benchmarking with Netty.io:

- Install Cloud Onload on both machines.

  Refer to the *Onload User Guide*.

- Install OpenJDK 1.8 on the Netty server machine.

  Refer to https://openjdk.java.net/install/.

- Run a shell script on the wrk2 client machine.

  The script uses ssh to install the following on the Netty server machine:

  - the Netty.io framework library
  - a Netty-based HTTP server
  - the Cloud Onload tuning profiles for Netty.io
  - a shell script to invoke the Netty-based HTTP server in different test modes.

  The script installs the following on the wrk2 client machine:

  - `wrk2`
  - a Python program to perform the benchmarking
  - an `ini` file containing settings for the Python program.

  The script then invokes the Python program, passing the `ini` file as a parameter.

The Python program then performs the benchmarking as follows:

- Start Netty-based HTTP servers on the Netty server machine.
  - Each HTTP server process is assigned to a dedicated CPU, distributed across the NUMA nodes.
  - Each HTTP server process uses a dedicated port.
  - The first iteration of the test uses a single HTTP server process.
- Start `wrk2` on the first server to generate load.
  - Each `wrk2` process is assigned to a dedicated CPU, distributed across the NUMA nodes.
  - Each `wrk2` process uses a dedicated port.
  - The first iteration of the test uses a single `wrk2` process.
- Measure the response rate of the Netty-based HTTP server, as the number of requests per second.
- Repeat the test 5 times, and record the median response rate.
- Repeat the test for varying payloads.

 © Copyright 2019 Xilinx, Inc

- Increase the number of `wrk2` and HTTP server processes on each server, and repeat the test.

  Continue doing this until the number of `wrk2` or HTTP server processes is the same as the number of CPUs on the server. For the setup used, this is 40 processes.

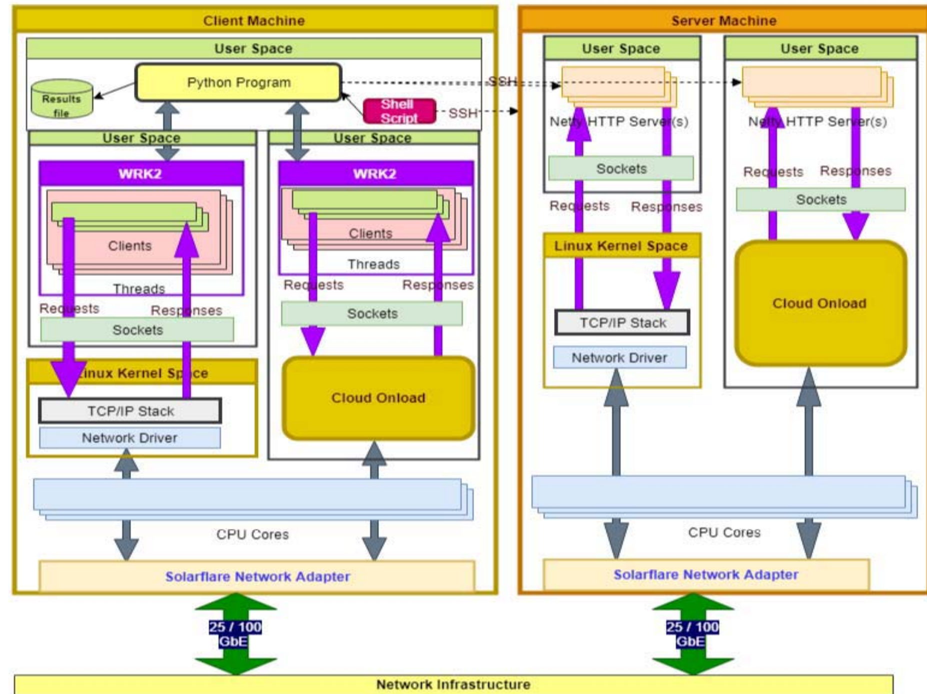

Figure 5: Netty.io software usage

- Repeat all tests, accelerating the Netty-based HTTP server with Cloud Onload.

These steps are detailed in the remaining chapters of this *Cookbook*.

The scripts and Cloud Onload profiles used for this benchmarking, that perform the above steps, are available on request from support@solarflare.com.

© Copyright 2019 Xilinx, Inc

4 Evaluation

This chapter describes how the performance of the test system is evaluated. See:

- General server setup on page 10
- wrk2 client on page 11
- Netty-based HTTP server on page 11
- Graphing the benchmarking results on page 12.

4.1 General server setup

Each server is setup as follows:

1 Ensure the following BIOS settings are made.

- enable Turbo mode
- enable C-States
- disable Hyper-threading
- disable Intel Virtualization Technology.

2 Stop various services:

```
systemctl stop cpupower
systemctl stop cpuspeed
systemctl stop cpufreqd
systemctl stop powerd
systemctl stop irqbalance
systemctl stop firewalld
systemctl stop iptables
```

3 Disable interrupt moderation

```
ethtool --coalesce <interface-name> adaptive-rx off rx-usecs 0
```

4 Allocate huge pages.

For example, to configure 1024 huge pages:

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

Update the `/etc/sysctl.conf` file to make this change persistent. For example:

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

 © Copyright 2019 Xilinx, Inc

4.2 wrk2 client

The wrk2 command line is generated by the Python program. An example for the first instance (core 1) is below:

```
taskset -c 1 \
        onload -p wrk-profile.opf \
        /opt/wrk2/wrk \
        -R 50M \
        -c 1000 \
        -d 30s \
        -t 1 \
        -s wrk.lua
        http://192.168.0.101:8081/256
```

This example runs a *Requests per second* test using a payload size of 1024 bytes (HTTP GET with keepalive).

- The `taskset -c` parameter is changed for each instance, to use cores 1 to 40.
- The `wrk.lua` LUA script file creates threads, assigns a server, and generates concurrent requests.
- The port number in the URL (`8081` above) is incremented for each instance.
- The filename in the URL (`256` above) is set to the desired payload size.

4.3 Netty-based HTTP server

The Netty-based HTTP server command line is generated by the `netty.sh` shell script. Example command lines for the first instance (core 1) are below:

- To start the HTTP server with the kernel network stack, use the following `netty.sh` command:

  ```
  netty.sh 1 kernel
  ```

  which generates the following command line:

  ```
  taskset -c 1 java HttpSnoopServer
  ```

- To start the HTTP server with an Onload-accelerated network stack, use one of the following `netty.sh` commands, for the two different Onload profiles under test:

  ```
  netty.sh 1 onload-performance
  netty.sh 1 onload-balanced
  ```

  which generate the corresponding command lines:

  ```
  taskset -c 1 onload -p nettyio-performance java HttpSnoopServer
  taskset -c 1 onload -p nettyio-balanced java HttpSnoopServer
  ```


© Copyright 2019 Xilinx, Inc

Note the following:

- The `taskset -c` parameter is changed for each instance, to use cores 1 to 40.
- The HTTP responses obey the following guidelines:.
  - the response content type is set to `text/plain`
  - both the header and the response body are composed dynamically
  - the response header includes Content-Length, Server and Date
  - HTTP keep-alive is used
  - TCP persistent connections. are used
  - GET requests are used.

Static files for HTTP servers

Each HTTP server serves static files from the server root directory.

The static files used range from 16 bytes to 512 bytes. They were generated using `dd`. The example below creates the necessary files for a server:

```
# for payload in 16 32 64 128 256 512
> do
> dd if=/dev/urandom of=$server_root/$payload \
>    bs=$payload count=1 > /dev/null 2>&1
> done
```

4.4 Graphing the benchmarking results

The results from each pass of `wrk2` are now gathered and summed, so that they can be further analyzed. They are then transferred into an Excel spreadsheet, to create graphs from the data.

 © Copyright 2019 Xilinx, Inc

5 Benchmark results

This chapter presents the benchmark results that are achieved. See:

- Results on page 14
- Analysis on page 20.

 © Copyright 2019 Xilinx, Inc

5.1 Results

25GbE with 16 byte payload

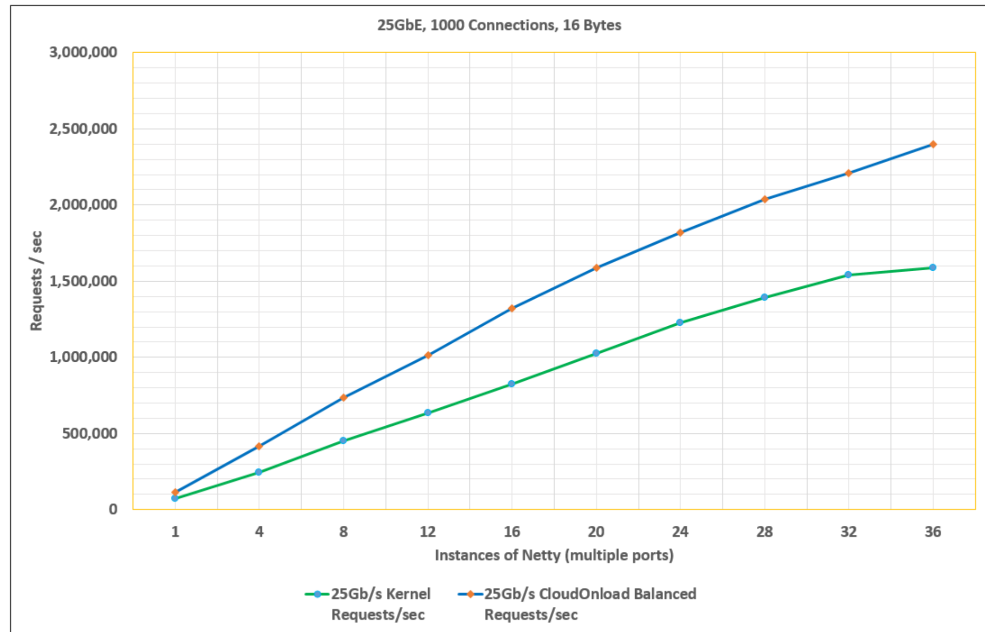

Figure 6: Netty.io requests per second

Table 1 below shows the results that were used to plot the graph in Figure 6 above.

Table 1: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	74,090	115,172	55%
4	247,057	416,578	69%
8	452,195	733,459	62%
12	632,339	1,015,292	61%
16	821,361	1,322,151	61%
20	1,023,532	1,586,738	55%
24	1,224,885	1,819,246	49%
28	1,390,398	2,036,244	46%
32	1,539,189	2,207,269	43%
36	1,587,437	2,398,869	51%

© Copyright 2019 Xilinx, Inc

25GbE with 32 byte payload

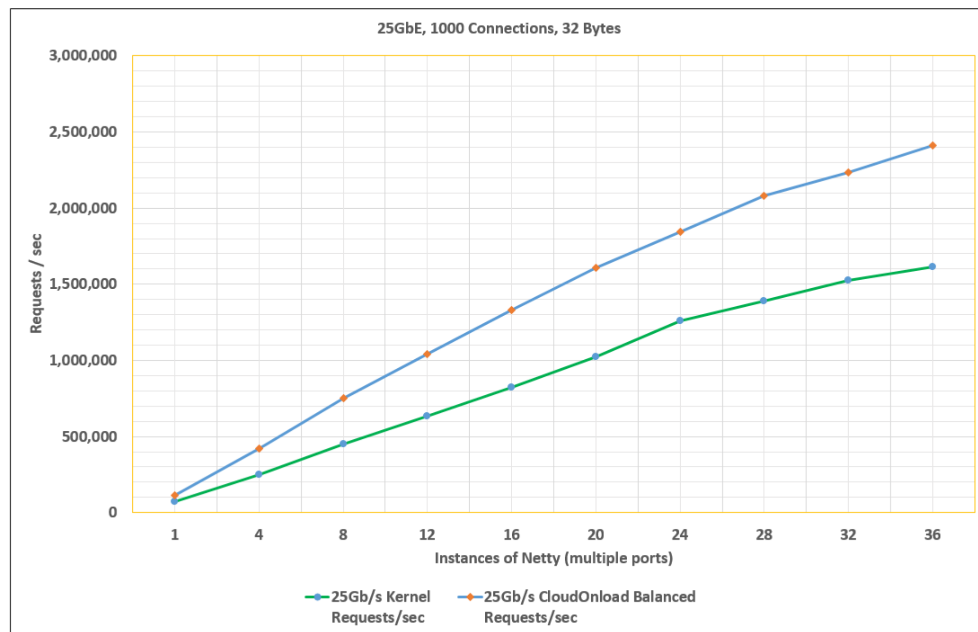

Figure 7: Netty.io requests per second

Table 2 below shows the results that were used to plot the graph in Figure 7 above.

Table 2: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	74,502	115,465	55%
4	247,456	423,847	71%
8	452,467	754,238	67%
12	635,024	1,042,664	64%
16	824,638	1,332,487	62%
20	1,024,068	1,610,808	57%
24	1,262,214	1,845,161	46%
28	1,392,240	2,079,610	49%
32	1,524,994	2,231,506	46%
36	1,612,211	2,414,067	50%

© Copyright 2019 Xilinx, Inc

25GbE with 64 byte payload

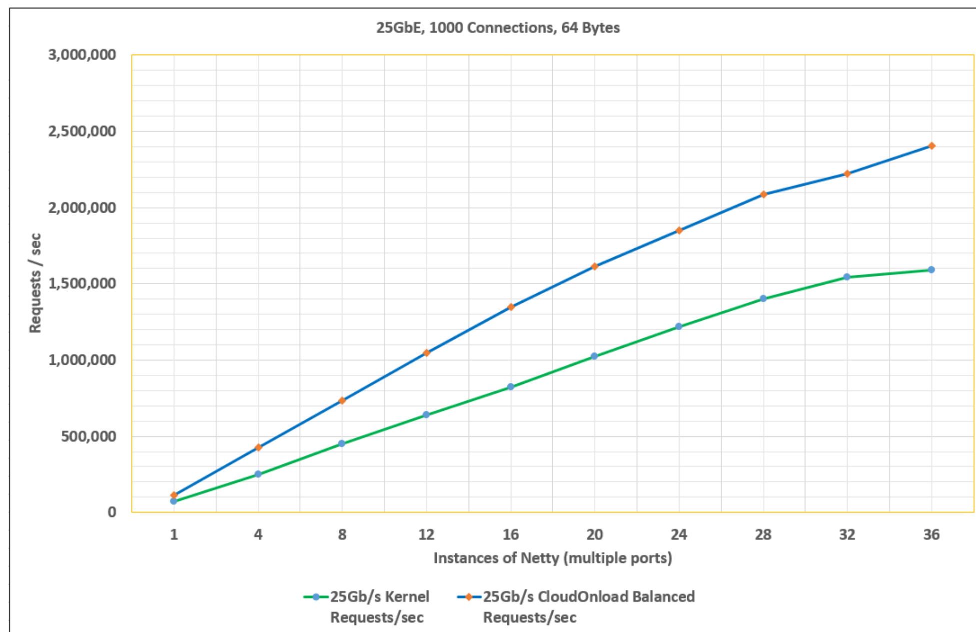

Figure 8: Netty.io requests per second

Table 3 below shows the results that were used to plot the graph in Figure 8 above.

Table 3: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	74,446	115,369	55%
4	248,993	425,108	71%
8	450,523	734,154	63%
12	640,035	1,047,015	64%
16	825,640	1,347,779	63%
20	1,022,246	1,612,354	58%
24	1,215,619	1,847,942	52%
28	1,398,708	2,086,071	49%
32	1,542,012	2,222,037	44%
36	1,587,569	2,407,833	52%

© Copyright 2019 Xilinx, Inc

25GbE with 128 byte payload

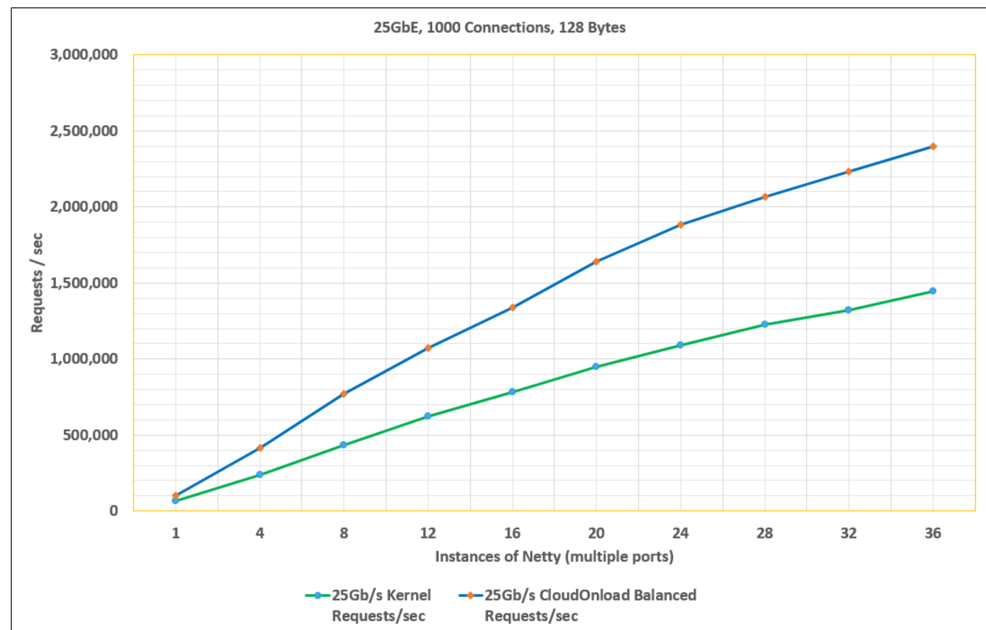

Figure 9: Netty.io requests per second

Table 4 below shows the results that were used to plot the graph in Figure 9 above.

Table 4: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	67,253	102,538	52%
4	236,525	417,489	77%
8	435,868	772,632	77%
12	621,379	1,073,886	73%
16	783,841	1,341,994	71%
20	946,056	1,641,211	73%
24	1,092,430	1,882,538	72%
28	1,226,745	2,067,025	68%
32	1,324,163	2,229,917	68%
36	1,443,811	2,400,253	66%

© Copyright 2019 Xilinx, Inc

25GbE with 256 byte payload

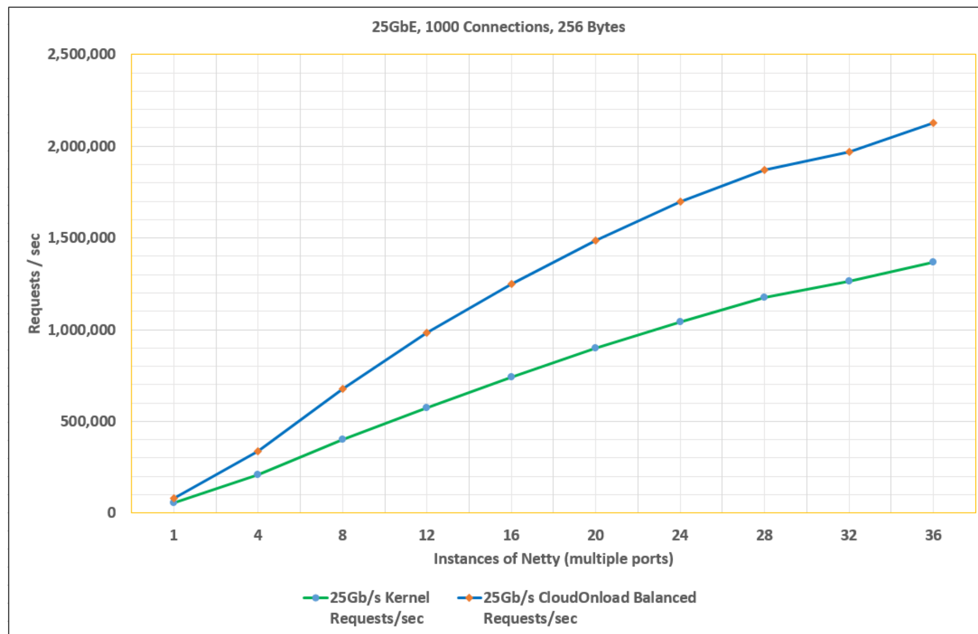

Figure 10: Netty.io requests per second

Table 5 below shows the results that were used to plot the graph in Figure 10 above.

Table 5: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	56,712	78,751	39%
4	209,992	336,575	60%
8	402,554	674,969	68%
12	575,605	982,092	71%
16	743,037	1,250,758	68%
20	899,196	1,484,873	65%
24	1,044,456	1,696,823	62%
28	1,173,183	1,871,371	60%
32	1,265,939	1,968,855	56%
36	1,365,344	2,128,143	56%

© Copyright 2019 Xilinx, Inc

25GbE with 512 byte payload

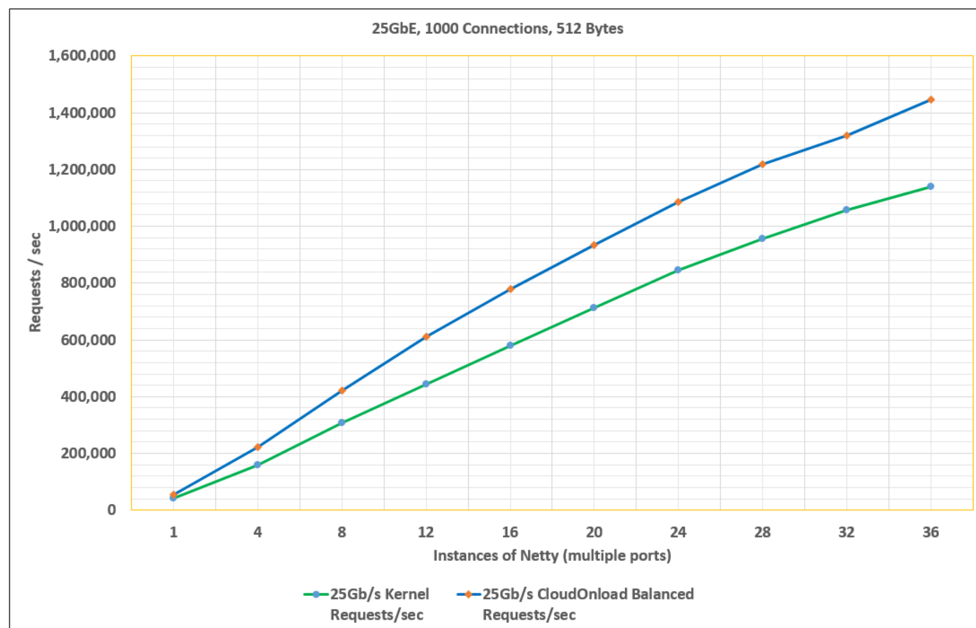

Figure 11: Netty.io requests per second

Table 6 below shows the results that were used to plot the graph in Figure 11 above.

Table 6: Requests per second

HTTP server instances	Kernel	Onload balanced	Onload balanced gain
1	42,530	54,234	28%
4	158,023	223,166	41%
8	306,457	422,328	38%
12	443,054	610,049	38%
16	581,166	777,353	34%
20	712,089	932,897	31%
24	844,865	1,084,682	28%
28	955,258	1,219,666	28%
32	1,057,640	1,319,236	25%
36	1,138,074	1,447,669	27%

© Copyright 2019 Xilinx, Inc

5.2 Analysis

From the data tables and graphs, one can see that Cloud Onload delivers an outstanding performance gain of up to 77% over the Linux kernel communications stack. It can also be seen that the Netty-based HTTP server with Cloud Onload serviced a maximum of 2.4 million GET requests/sec, whereas the same HTTP server with the kernel stack serviced a maximum of only 1.6 million GET requests/sec.

Since Netty.io is network intensive, every request includes network processing overhead. Whenever an application touches hardware other than the CPU or memory, such as the network, it must make one or more calls to the Linux kernel. Each call creates additional overhead, that requires both CPU cycles and processing time. Cloud Onload moves the network processing for Netty.io from the kernel into Netty.io's own application space in memory. This single modification is responsible for improving the performance of Netty.io.

Gains are seen with various numbers of Netty-based HTTP server instances (1 to 36) which are pinned to the corresponding number of unique CPU cores (1 to 36), and with various response payload sizes (16, 32, 64, 128, 256, and 512 bytes). The gains are achieved with Netty.io using NIO (Non-Blocking IO), and the `wrk2` client using the HTTP/1.1 persistent connection model which keeps connections opened between successive requests, and so reduces the time needed to open new connections.

Finally, one can see from the data tables and graphs that further performance gains might be possible by adding more instances of the accelerated HTTP server, until throughput is system limited.

© Copyright 2019 Xilinx, Inc

A Cloud Onload profiles

This appendix contains the Cloud Onload profiles used for this benchmarking. See:

- The wrk-profile Cloud Onload profile on page 21
- The Netty Cloud Onload profiles on page 22.

These profiles, along with the scripts used for this benchmarking, are available on request from support@solarflare.com.

A.1 The wrk-profile Cloud Onload profile

The `wrk-profile.opf` Cloud Onload profile is as follows:

```
onload_set EF_SOCKET_CACHE_MAX 40000
onload_set EF_TCP_TCONST_MSL 1
onload_set EF_TCP_FIN_TIMEOUT 15
onload_set EF_HIGH_THROUGHPUT_MODE 1
onload_set EF_LOG_VIA_IOCTL 1
onload_set EF_NO_FAIL 1
onload_set EF_UDP 0

#ensure sufficient resources
onload_set EF_MAX_PACKETS 205000
onload_set EF_MAX_ENDPOINTS 400000
onload_set EF_FDTABLE_SIZE 8388608
onload_set EF_USE_HUGE_PAGES 2
onload_set EF_MIN_FREE_PACKETS 50000

#environment variable can overwrite
onload_set EF_LOAD_ENV 1

#spinning configuration
onload_set EF_POLL_USEC 100000
onload_set EF_SLEEP_SPIN_USEC 50
onload_set EF_EPOLL_SPIN 1

#scalable filters with clustering for outgoing connections
onload_set EF_SCALABLE_FILTERS 'any=rss:active'
onload_set EF_SCALABLE_FILTERS_ENABLE 1
onload_set EF_CLUSTER_NAME 'load'
onload_set EF_CLUSTER_SIZE 12 #needs to overwritten by environment

#shared local ports to improve rate of socket recycling
onload_set EF_TCP_SHARED_LOCAL_PORTS_MAX 28000
onload_set EF_TCP_SHARED_LOCAL_PORTS 28000
onload_set EF_TCP_SHARED_LOCAL_PORTS_PER_IP 1
onload_set EF_TCP_SHARED_LOCAL_PORTS_REUSE_FAST 1
onload_set EF_TCP_SHARED_LOCAL_PORTS_NO_FALLBACK 1
```


© Copyright 2019 Xilinx, Inc

```
#epoll configuration
onload_set EF_UL_EPOLL 3
onload_set EF_EPOLL_MT_SAFE 1

#reduce transmit CPU load
onload_set EF_TX_PUSH 0
onload_set EF_PIO 0
onload_set EF_CTPIO 0

# Adjustments for potentially-lossy network environment
onload_set EF_TCP_INITIAL_CWND 14600
onload_set EF_DYNAMIC_ACK_THRESH 4
onload_set EF_TAIL_DROP_PROBE 1
onload_set EF_TCP_RCVBUF_MODE 1
```

A.2 The Netty Cloud Onload profiles

There are two Netty Cloud Onload profiles.

- The *performance profile* is designed to get maximum performance, including best throughput and transaction rate, as well as best average and 99 percentile transaction response times.

  However this profile relies on the application threads having exclusive use of physical CPU cores. To get best performance, you must explicitly pin application threads to physical cores (avoiding threads sharing hyperthreaded CPU cores), and also ensure there are enough unused CPU cores for other applications to use.

  This profile constantly polls for network events to achieve the lowest latency possible, and so has higher CPU usage. CPU utilization metrics no longer provide a usable indication of system load.

- The *balanced profile* is designed also to get maximum performance, while avoiding the trade-offs associated with the performance profile.

  This profile does not rely on the application threads having exclusive use of physical CPU cores. CPU cores may be shared with other applications, hyper-threads may be used, and CPU utilization metrics indicate system load.

  Under high load conditions, this profile should deliver throughput and transaction rates that are equivalent to the performance profile. At lower traffic rates, CPU usage is reduced, but transaction response times might increase. However, response times will be better than when not running Onload.

 © Copyright 2019 Xilinx, Inc

The differences between these profiles are at the start of the profile files. See:

- The nettyio-performance profile on page 23
- The nettyio-balanced profile on page 24.

The nettyio-performance profile

The `nettyio-performance.opf` Cloud Onload profile is as follows:

```
# netty.io performance 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


# Use EPOLL mode 3 as will provide the best scalability and speed
# EPOLL can be multithread safe, as netty poll architecture is single threaded
onload_set EF_UL_EPOLL 3
onload_set EF_EPOLL_MT_SAFE 1

onload_set EF_RXQ_SIZE 4096
onload_set EF_CLUSTER_IGNORE 1

# Enable receive packet event batching, this adds a small latency
# cost, but improves transaction rate/efficiency
onload_set EF_HIGH_THROUGHPUT_MODE 1

# Disable CTPIO and PIO as these reduce CPU efficiency and don't
# for this class of application, bring major benefits.  If
# absolutely best latency is needed, then consider enabling them.
onload_set EF_CTPIO 0
onload_set EF_PIO 0
```


© Copyright 2019 Xilinx, Inc

The nettyio-balanced profile

The `nettyio-balanced.opf` Cloud Onload profile is as follows:

```
# netty.io balanced profile

# Enable small amount of polling / spinning.  When the application makes a blocking call
# such as recv() or poll(), this causes Onload to busy wait for up to 20us
# before blocking.
#
onload_set EF_INT_DRIVEN 0
onload_set EF_POLL_USEC 20

# Prevent spinning inside socket calls.
onload_set EF_PKT_WAIT_SPIN 0
onload_set EF_TCP_RECV_SPIN 0
onload_set EF_TCP_SEND_SPIN 0
onload_set EF_TCP_CONNECT_SPIN 0
onload_set EF_TCP_ACCEPT_SPIN 0
onload_set EF_UDP_RECV_SPIN 0
onload_set EF_UDP_SEND_SPIN 0


# Use EPOLL mode 3 as will provide the best scalability and speed
# EPOLL can be multithread safe, as netty poll architecture is single threaded
onload_set EF_UL_EPOLL 3
onload_set EF_EPOLL_MT_SAFE 1

onload_set EF_RXQ_SIZE 4096
onload_set EF_CLUSTER_IGNORE 1

# Enable receive packet event batching, this adds a small latency
# cost, but improves transaction rate/efficiency
onload_set EF_HIGH_THROUGHPUT_MODE 1

# Disable CTPIO and PIO as these reduce CPU efficiency and don't
# for this class of application, bring major benefits.
onload_set EF_CTPIO 0
onload_set EF_PIO 0
```


© Copyright 2019 Xilinx, Inc