Cloud Onload® Redis Cookbook

SF-121461-CD

Issue 3

Table of Contents

1 Introduction

This chapter introduces you to this document. See:

1.1 About this document

This document is the *Redis Cookbook* for Cloud Onload. It gives procedures for technical staff to configure and run tests, to benchmark Open Source Redis 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 Redis software distribution, and of Solarflare Cloud Onload.
- Summary of benchmarking on page 5 summarizes how the performance of Redis has been benchmarked, both with and without Cloud Onload, to determine what benefits might be seen.
- Redis installation and configuration on page 8 describes how to install and configure Redis.
- Evaluation on page 12 describes how the performance of the test system is evaluated.
- Benchmark results on page 20 presents the benchmark results that are achieved.

1.2 Intended audience

The intended audience for this *Redis 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/.

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/.

2 Overview

This chapter gives an overviews of the Redis software distribution, and of Solarflare Cloud Onload. See:

2.1 Redis overview

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports a wide variety of data structures, such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperlog logs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, different levels of on-disk persistence, and many other capabilities. It provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

Redis is heavily network dependent by design, so its performance can be significantly improved through enhancements to the underlying networking layer.

2.2 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”.

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.

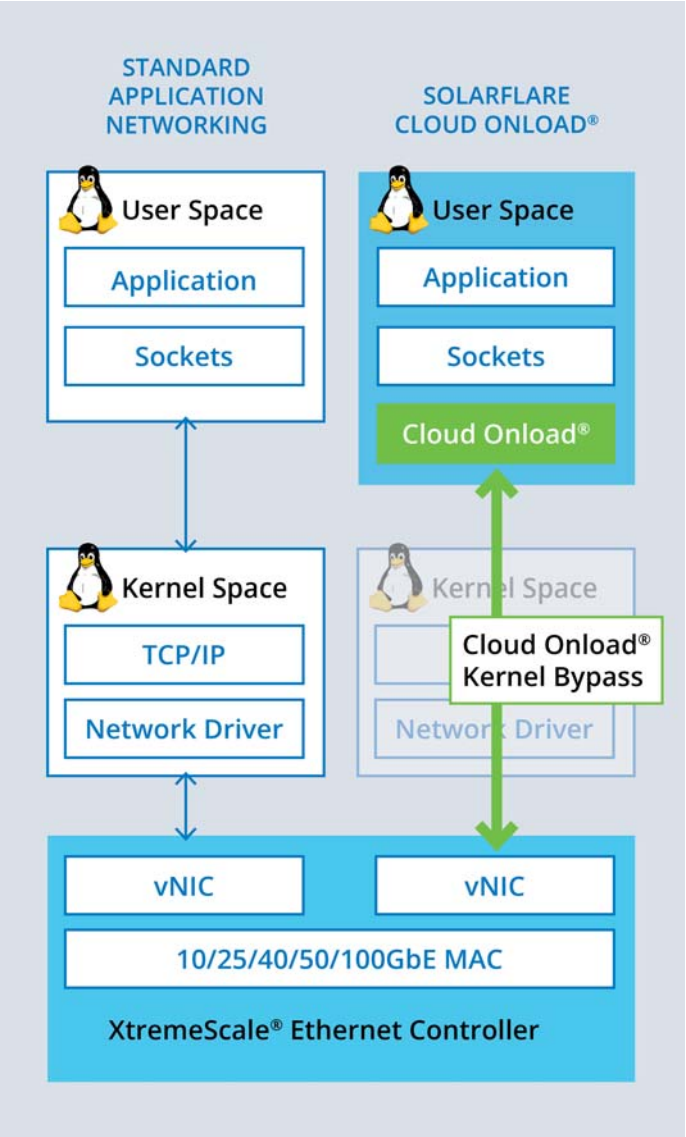


Figure 1: Cloud Onload architecture

3 Summary of benchmarking

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

- Architecture for Redis benchmarking on page 6
- Redis benchmarking process on page 7.

3.1 Architecture for Redis benchmarking

Benchmarking was performed with two Dell R740 servers, with the following specification:

Server	Dell R740XD
Memory	192GB (12 × 16384 MB)
NICs	SFN8522 (dual port 10G) X2522-25G (dual port 25G) X2541 (single port 100G)
CPU	sfocr740a (used for `redis-benchmark`): 2 × Intel® Xeon® Platinum 8153 CPU @ 2.00GHz sfocr740b (used for `redis-server`): Intel® Xeon® Gold 5120 CPU @ 2.20GHz
OS	Red Hat Enterprise Linux Server release 7.5 (Maipo)
Redis	Redis 5.0.2

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

All high-volume test traffic is routed through a dedicated switch that provides 10, 25, and 100GbE ports.

This enables testing at three different network speeds (10GbE, 25GbE and 100GbE) to determine when applications become bottlenecked as a result of network traffic.

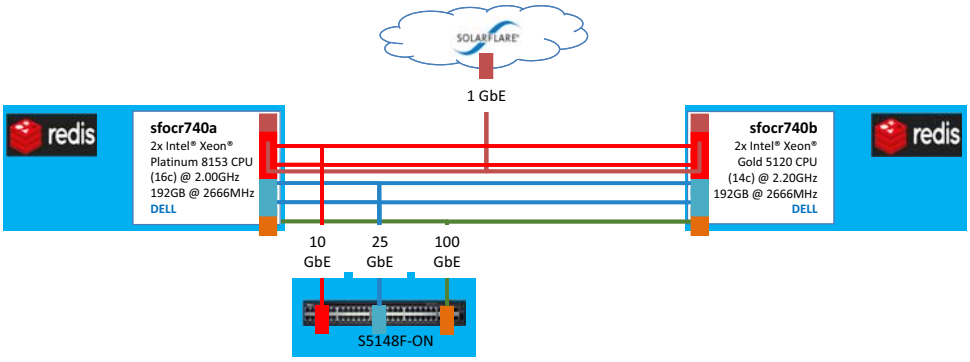


Figure 2: Architecture for Redis benchmarking

3.2 Redis benchmarking process

These are the high-level steps we followed to complete benchmarking with Redis:

- Install Redis on both servers

  The Redis installation contains the `redis-server` and the `redis-benchmark` tool.

- Start `redis-server`(s) on one node (sfoc740b).

  To try and saturate the NICs we will run multiple Redis servers, starting each server on a separate port.

- Start `redis-benchmark` test(s) on the other node (sfoc740a)

  `Redis-benchmark` is a utility to check the performance of a Redis server by running a synthetic Redis workload. Testing focuses on its SET and GET tests.

  One instance of `redis-benchmark` is started per instance of `redis-server`, using the same port.

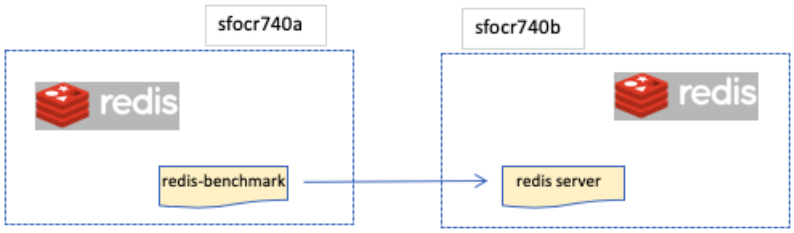


Figure 3: Redis software usage

- Record the response rate of the Redis server, as the number of requests per second.
- Repeat the test across all interfaces available on the server.
- Repeat all tests, accelerating Redis with Cloud Onload.

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

4 Redis installation and configuration

This chapter describes how to install and configure Redis. See:

4.1 Installing Redis

To install Redis:

1. Download the tarball for the open source version of Redis:

```
$ wget http://download.redis.io/releases/redis-5.0.2.tar.gz
```

2. Unpack the tarball:

```
$ tar xvzf redis-5.0.2.tar.gz
```

3. Check the README file:

```
$ less README.md
```

4. Build Redis and its tests:

```
$ cd redis-5.0.2/
$ make
$ make test
```

5. Run the tests:

```
$ ./runtest
...
\o/ All tests passed without errors!
Cleanup: may take some time... OK
```

4.2 Running redis-server with default options

If no configuration file is specified, Redis uses its default configuration. Redis recognizes any issues, such as settings that are not appropriate for the server in use, and outputs tips on how to improve performance:

```
# ./redis-server
43910:C 05 Dec 2018 20:12:12.240 # oO0OoO0OoO0Oo Redis is starting
oO0OoO0OoO0Oo
43910:C 05 Dec 2018 20:12:12.240 # Redis version=5.0.2, bits=64,
commit=00000000, modified=0, pid=43910, just started
43910:C 05 Dec 2018 20:12:12.240 # Warning: no config file specified,
using the default config. In order to specify a config file use ./redis-
server /path/to/redis.conf

43910:M 05 Dec 2018 20:12:12.240 * Increased maximum number of open files
to 10032 (it was originally set to 1024).
...

43910:M 05 Dec 2018 20:12:12.241 # WARNING: The TCP backlog setting of 511
cannot be enforced because /proc/sys/net/core/somaxconn is set to the
lower value of 128.
43910:M 05 Dec 2018 20:12:12.241 # WARNING overcommit_memory is set to 0!
Background save may fail under low memory condition. To fix this issue add
'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the
command 'sysctl vm.overcommit_memory=1' for this to take effect.
43910:M 05 Dec 2018 20:12:12.241 # WARNING you have Transparent Huge Pages
(THP) support enabled in your kernel. This will create latency and memory
usage issues with Redis. To fix this issue run the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your
/etc/rc.local in order to retain the setting after a reboot. Redis must be
restarted after THP is disabled.
```

4.3 Implementing performance tips

The performance tips output by Redis require various settings to be made. To ensure these settings persist after a reboot, create a redis.sh profile script to hold them:

```
# echo "#!/bin/bash" > /etc/profile.d/redis.sh
# chmod a+x /etc/profile.d/redis.sh
```

Create a redis.conf file

```
43910:C 05 Dec 2018 20:12:12.240 # Warning: no config file specified,
using the default config. In order to specify a config file use ./redis-
server /path/to/redis.conf
```

Make a copy of the example `redis.conf` shipped in the root of the Redis install.

Raise somaxconn above 511

```
43910:M 05 Dec 2018 20:12:12.241 # WARNING: The TCP backlog setting of 511
cannot be enforced because /proc/sys/net/core/somaxconn is set to the
lower value of 128.
```

To fix this, add the following line to `/etc/profile.d/redis.sh`:

```
sysctl -w net.core.somaxconn=65535
```

After the next reboot, the new setting will be to allow 65535 connections instead of 128 as before.

Set vm.overcommit_memory to 1

```
43910:M 05 Dec 2018 20:12:12.241 # WARNING overcommit_memory is set to 0!
Background save may fail under low memory condition. To fix this issue add
'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the
command 'sysctl vm.overcommit_memory=1' for this to take effect.
```

To fix this, add the following line to `/etc/profile.d/redis.sh`:

```
sysctl vm.overcommit_memory=1
```

Disable transparent huge pages

```
43910:M 05 Dec 2018 20:12:12.241 # WARNING you have Transparent Huge Pages
(THP) support enabled in your kernel. This will create latency and memory
usage issues with Redis. To fix this issue run the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your
/etc/rc.local in order to retain the setting after a reboot. Redis must be
restarted after THP is disabled.
```

To fix this, add the following line to `/etc/profile.d/redis.sh`:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

Reboot

Reboot, to make the above changes happen.

4.4 Enabling remote access to redis-server

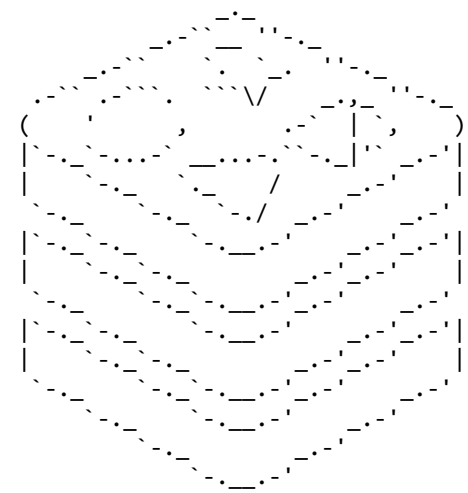
To enable remote access to `redis-server` from another server, edit your copy of `redis.conf`:

```
# vi <location>/redis.conf
...
#bind 127.0.0.1
...
#protected-mode yes
protected-mode no
...
```

4.5 Running redis-server after performance tips

Now run `redis-server`, passing it the path to your copy of `redis.conf`:

```
# <location>/redis-server <location>/redis.conf
382566:C 16 Dec 2018 12:17:40.721 # oO0OoO0OoO0Oo Redis is starting
oO0OoO0OoO0Oo
382566:C 16 Dec 2018 12:17:40.721 # Redis version=5.0.2, bits=64,
commit=00000000, modified=0, pid=382566, just started
382566:C 16 Dec 2018 12:17:40.721 # Configuration loaded
382566:M 16 Dec 2018 12:17:40.721 * Increased maximum number of open files
to 10032 (it was originally set to 1024).
```




```
382566:M 16 Dec 2018 12:17:40.722 # Server initialized
382566:M 16 Dec 2018 12:17:40.722 * Ready to accept connections
```

NOTE: There are now no performance tips.

5 Evaluation

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

5.1 Choosing options for the redis-benchmark utility

The following options are available for running `redis-benchmark`.

Table 1: Redis-benchmark options

Option	Description	Default value
`-h`	Server host name	127.0.0.1
`-p`	Server port	6379
`-s`	Server socket	
`-a`	Password for Redis Auth	
`-c`	Number of parallel connections	50
`-n`	Total number of requests	10000
`-d`	Data size of SET/GET value in bytes	2
`-k`	1=keep alive, 0=reconnect	1
`-r`	Use random keys for SET/GET/INCR, random values for SADD	
`-P`	Pipeline <numreq> requests	1
`-q`	Quiet. Just shows query/sec values	
`--csv`	Output in CSV format	
`-l`	Loop. Run the tests forever	
`-t`	Only runs the comma-separated list of tests	
`-I`	Idle mode. Just opens N idle connections and wait	

Of the above options, the following have been changed:

- `-h`: Server host name.

  The server in the test system has multiple interfaces that offer different speeds. Each has its own IP address.

  The IP address offering the desired speed was used for testing:

  - 100GbE: 192.168.105.35
  - 25GbE: 192.168.101.35
  - 10GbE: 192.168.103.35

- `-p`: Server port.

  Where multiple instances are run on a single host, each instance must use its own port number.

  Consecutive values starting with 6379 (the default) were used for testing.

- `-n`: Total number of requests.

  This value was increased until the test duration was sufficiently long for the benchmark to reach a steady state operating rate.

  A value of 50,000,000 was used for testing.

- `-P`: Number of requests to pipeline.

  Redis supports pipelining, so it is possible to send multiple requests at once. This feature is often exploited by real world applications. The pipelining was scaled up until the requests per second smoothed out, and larger pipelines had negligible effect.

  A value of 128 was used for testing.

- `-c`: Number of parallel connections.

  No significant performance change was recorded above the default value of 50.

  A value of 200 was used for testing, to match previous Solarflare Redis benchmark tests.

- `-d`: Data size.

  No significant performance change was recorded above the default value of 2.

  A value of 128 was used for testing, to match previous Solarflare Redis benchmark tests.

- `-t`: Comma separated list of tests.

  Only the SET and GET tests are required.

  Values of `set` and `get` were used for testing.

- `-q`: Quiet. Just shows query/sec values.

  Only the results are required.

  Quiet mode is enabled for testing.

The resulting command lines used during the testing are:

```
redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t set -q
redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t get -q
```

5.2 Starting redis-server for kernel benchmarking

The benchmarking uses multiple instances of `redis-server`. Each instance:

- uses a separate port
- records its pid in a separate file
- is pinned to an isolated CPU.

To achieve this:

- A custom configuration file is created for each instance, that sets the correct port and pid file location.
- The `taskset` command is used to pin each instance to its own CPU.

Custom configuration files

Each custom configuration file is named `redis_<port>.conf`, where `<port>` is a consecutive number, starting with 6379. Each file contains the settings for running redis-server on the given port. The files differ only in their settings for `port` and `pidfile`. Below are excerpts from `redis_6380.conf`:

```
...
# Accept connections on the specified port, default is 6379 (IANA #815344).
# If port 0 is specified Redis will not listen on a TCP socket.
port 6380
...
# Creating a pid file is best effort: if Redis is not able to create it
# nothing bad happens, the server will start and run normally.
pidfile /var/run/redis_6380.pid
```

Starting redis-server

The following command is used to start each instance of `redis-server`:

```
# tasket -c <cpu> <location>/redis-server <location>/redis_<port>.conf
```

where:

- `<port>` is incremented from 6379 onwards, to reference each of the custom configuration files
- `<cpu>` is incremented so as to evenly spread the instances of `redis-server` over all isolated CPUs on the server.

5.3 Running redis-benchmark for kernel benchmarking

The benchmarking uses multiple instances of the `redis-benchmark` utility, each running on the same port as one of the instances of `redis-server`.

SET test

Multiple iterations of the SET test are made, using this command line:

```
# redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t set -q &
```

- The first iteration runs `redis-benchmark` once, for port 6379.
- The second iteration runs `redis-benchmark` twice in parallel, for ports 6379 and 6380.
- This continues until the final iteration, which runs multiple instances of `redis-benchmark` in parallel (one per instance of `redis-server`), for ports 6379 upwards.

Each instance of `redis-benchmark` outputs the total requests per second:

```
SET: 1223121.88 requests per second
```

So, for the *n*th iteration, there are *n* totals output. These totals are stored in files.

GET test

The same number of iterations of the GET test are then made, using this command line:

```
# redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t get -q &
```

Each instance of `redis-benchmark` outputs the total requests per second:

```
GET: 1705669.75 requests per second
```

So, for the *n*th iteration, there are *n* totals output. These totals are stored in files.

5.4 Graphing the kernel benchmarking results

The results from each iteration of `redis-benchmark` 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.

5.5 Cloud Onload benchmarking

The benchmarking is then repeated using Cloud Onload to accelerate Redis. This uses the following Cloud Onload profiles:

- Cloud Onload `redis-balanced.opf`
- Cloud Onload `redis-performance.opf`

Cloud Onload redis-balanced.opf

The Cloud Onload `redis-balanced.opf` profile is designed to get maximum performance, while avoiding the trade-offs associated with using 'performance' tuning settings. Specifically, it does not rely on application threads having exclusive use of physical CPU cores. CPU cores may be shared with other applications, hyperthreads may be used and CPU utilization metrics will continue to be able to indicate system load. When compared against a 'performance' profile, this profile should get as good a throughput and transaction rate, under high load conditions and given equivalent CPU resources. However response times might not be as good. When compared against the same application not running Onload, this profile should get better average and 99-percentile transaction response times.

```
# 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 2 as will provide the best scalability and speed
# and be compatible with redis process forking
# EPOLL can be multithread safe, as redis poll architecture is single threaded
onload_set EF_UL_EPOLL 2
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
```

Cloud Onload redis-performance.opf

The Cloud Onload `redis-performance.opf` profile is designed to get maximum performance, giving the best throughput and transaction rate, as well as best average and 99-percentile transaction response times. However this profile assumes that the application threads have exclusive use of physical CPU cores. To get best performance, the user may need to explicitly pin applications threads to physical cores (i.e. avoid threads sharing hyperthreaded CPU cores), and ensure there are enough unused CPU cores for scheduling other applications. Because this profile may use busy polling (also known as spinning), CPU utilization metrics do not provide a usable indication of system load.

```
# redis 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
# and be compatible with redis process forking
# EPOLL can be multithread safe, as redis poll architecture is single threaded
onload_set EF_UL_EPOLL 2
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
```

Use Cloud Onload to accelerate Redis

Repeat the testing using Cloud Onload to accelerate `redis-server` and `redis-benchmark`, first with the `redis-balanced` profile, then with the `redis-performance` profile.

Using the redis-balanced profile

Precede each Redis command with:

```
onload --profile=redis-balanced
```

So, when starting `redis-server`:

```
# tasket -c <cpu> onload --profile=redis-balanced \
     <location>/redis-server <location>/redis_<port>.conf
```

and when starting `redis-benchmark`:

```
# onload --profile=redis-balanced \
     redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t set -q &

# onload --profile=redis-balanced \
     redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t get -q &
```

Using the redis-performance profile

Precede each Redis command with:

```
onload --profile=redis-performance
```

So, when starting `redis-server`:

```
# tasket -c <cpu> onload --profile=redis-performance \
     <location>/redis-server <location>/redis_<port>.conf
```

and when starting `redis-benchmark`:

```
# onload --profile=redis-performance \
     redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t set -q &

# onload --profile=redis-performance \
     redis-benchmark -h <host>  -p <port> -n 50000000 -P 128 -c 200 -d 128 -t get -q &
```

6 Benchmark results

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

- SET and GET performance on page 20
- Analysis on page 27.

6.1 SET and GET performance

This section shows the results of running different numbers of `redis-benchmark` tests in parallel, and accumulating the results of each parallel run:

- 50,000,000 total requests
- 128 tests pipelined
- 200 parallel connections
- data size of 128 bytes.

The results show both kernel and Cloud Onload runs across different Solarflare NICs.

The results utilizing Cloud Onload use its `redis-balanced.opf` and `redis-performance.opf` profiles. See Use Cloud Onload to accelerate Redis on page 19.

SET and GET performance at 25Gb/s

The graphs in Figure 4 and Figure 5 below show SET and GET performance at 25Gb/s.

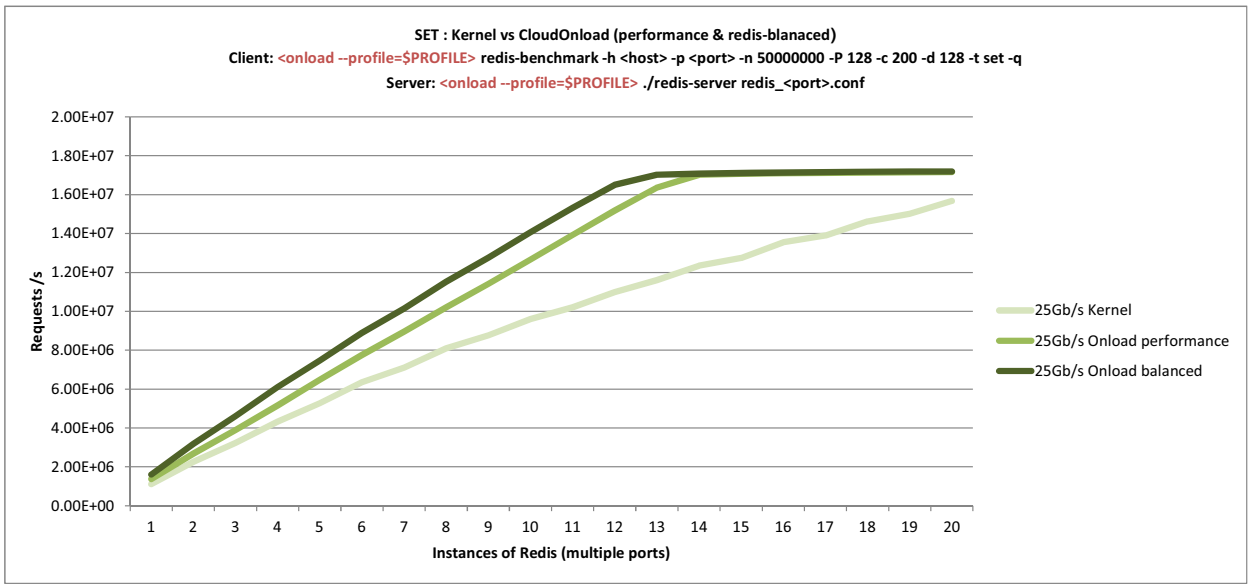


Figure 4: SET performance at 25Gb/s

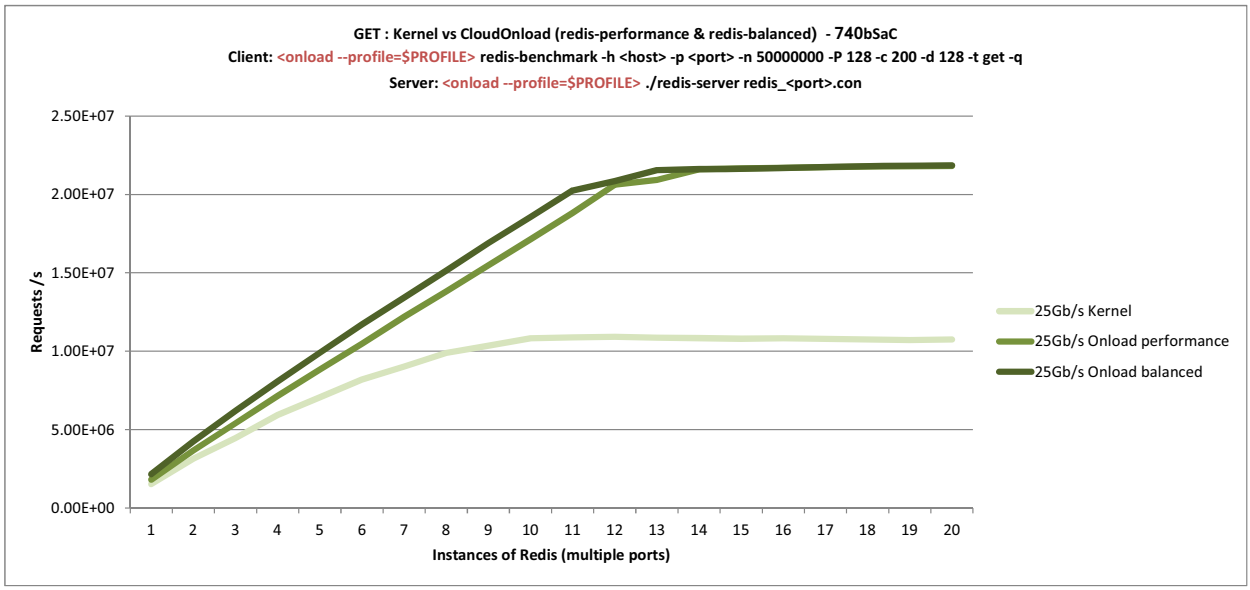


Figure 5: GET performance at 25Gb/s

Table 2 and Table 3 below show the results that were used to plot the graphs in Figure 4 and Figure 5 above.

Table 2: SET performance at 25Gb/s

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
1	1.12E+06	1.37E+06	1.61E+06	22.97%	43.98%
2	2.26E+06	2.68E+06	3.18E+06	18.79%	40.99%
3	3.23E+06	3.90E+06	4.61E+06	20.71%	42.74%
4	4.33E+06	5.17E+06	6.11E+06	19.42%	41.24%
5	5.28E+06	6.48E+06	7.47E+06	22.82%	41.65%
6	6.35E+06	7.76E+06	8.89E+06	22.23%	40.05%
7	7.11E+06	8.95E+06	1.01E+07	25.93%	42.70%
8	8.09E+06	1.02E+07	1.15E+07	26.04%	42.33%
9	8.77E+06	1.14E+07	1.28E+07	30.09%	45.47%
10	9.60E+06	1.27E+07	1.41E+07	31.85%	46.43%
11	1.02E+07	1.39E+07	1.53E+07	36.38%	50.09%
12	1.10E+07	1.52E+07	1.65E+07	38.13%	50.19%
13	1.16E+07	1.64E+07	1.70E+07	41.01%	46.76%
14	1.24E+07	1.70E+07	1.71E+07	37.80%	38.24%
15	1.27E+07	1.71E+07	1.71E+07	33.92%	34.28%
16	1.35E+07	1.71E+07	1.71E+07	26.21%	26.46%
17	1.39E+07	1.71E+07	1.72E+07	23.10%	23.40%
18	1.46E+07	1.71E+07	1.72E+07	17.15%	17.42%
19	1.50E+07	1.71E+07	1.72E+07	14.20%	14.51%
20	1.57E+07	1.72E+07	1.72E+07	9.50%	9.71%

Table 3: GET performance at 25Gb/s

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
1	1.52E+06	1.82E+06	2.16E+06	19.17%	41.75%
2	3.15E+06	3.67E+06	4.25E+06	16.72%	35.22%
3	4.46E+06	5.41E+06	6.20E+06	21.46%	39.16%
4	5.93E+06	7.15E+06	8.08E+06	20.57%	36.20%
5	7.07E+06	8.83E+06	9.90E+06	24.92%	40.05%
6	8.20E+06	1.05E+07	1.17E+07	27.89%	42.93%
7	9.03E+06	1.22E+07	1.34E+07	35.08%	48.75%
8	9.89E+06	1.38E+07	1.51E+07	39.92%	53.16%
9	1.03E+07	1.55E+07	1.69E+07	49.58%	63.43%
10	1.08E+07	1.71E+07	1.86E+07	58.44%	71.57%
11	1.09E+07	1.88E+07	2.03E+07	73.03%	86.09%
12	1.09E+07	2.06E+07	2.08E+07	89.01%	90.95%
13	1.09E+07	2.09E+07	2.16E+07	92.89%	98.54%
14	1.08E+07	2.16E+07	2.16E+07	99.30%	99.47%
15	1.08E+07	2.17E+07	2.16E+07	100.52%	100.22%
16	1.08E+07	2.17E+07	2.17E+07	100.49%	100.37%
17	1.08E+07	2.17E+07	2.17E+07	101.60%	101.67%
18	1.07E+07	2.18E+07	2.18E+07	102.66%	102.84%
19	1.07E+07	2.18E+07	2.18E+07	103.28%	103.61%
20	1.07E+07	2.18E+07	2.19E+07	103.04%	103.35%

SET and GET performance at 100Gb/s

The graphs in Figure 4 and Figure 5 below show SET and GET performance at 100Gb/s.

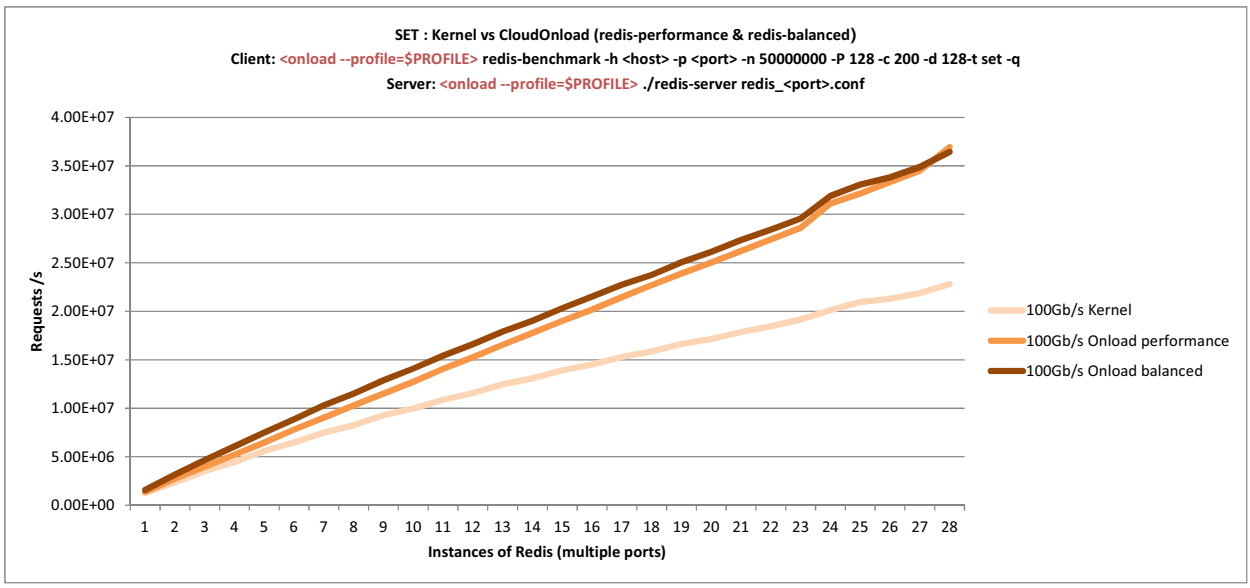


Figure 6: SET performance at 100Gb/s

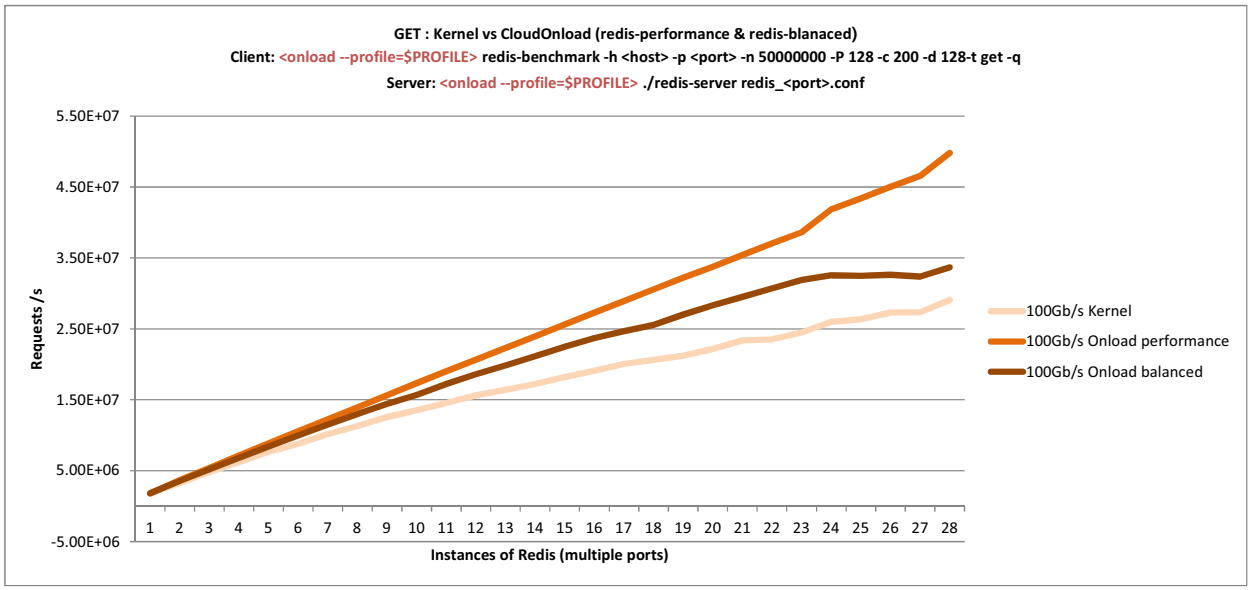


Figure 7: GET performance at 100Gb/s

Table 4 and Table 5 below show the results that were used to plot the graphs in Figure 6 and Figure 7 above.

Table 4: SET performance at 100Gb/s

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
1	1.29E+06	1.36E+06	1.57E+06	5.50%	21.23%
2	2.36E+06	2.74E+06	3.14E+06	16.36%	33.50%
3	3.52E+06	3.97E+06	4.65E+06	12.70%	32.03%
4	4.46E+06	5.21E+06	6.07E+06	16.78%	36.19%
5	5.59E+06	6.46E+06	7.49E+06	15.55%	33.94%
6	6.46E+06	7.80E+06	8.88E+06	20.75%	37.41%
7	7.49E+06	9.03E+06	1.03E+07	20.67%	37.57%
8	8.26E+06	1.03E+07	1.15E+07	24.57%	39.45%
9	9.28E+06	1.15E+07	1.29E+07	24.02%	38.76%
10	9.97E+06	1.27E+07	1.41E+07	27.69%	41.18%
11	1.09E+07	1.41E+07	1.54E+07	29.35%	41.72%
12	1.16E+07	1.53E+07	1.66E+07	31.91%	43.51%
13	1.25E+07	1.65E+07	1.79E+07	32.66%	43.63%
14	1.31E+07	1.77E+07	1.90E+07	35.74%	45.47%
15	1.39E+07	1.90E+07	2.03E+07	36.48%	45.87%
16	1.45E+07	2.02E+07	2.15E+07	39.06%	48.23%
17	1.53E+07	2.15E+07	2.27E+07	40.25%	48.58%
18	1.59E+07	2.27E+07	2.38E+07	43.09%	49.85%
19	1.66E+07	2.39E+07	2.51E+07	43.70%	50.68%
20	1.71E+07	2.50E+07	2.61E+07	46.10%	52.45%
21	1.79E+07	2.62E+07	2.73E+07	46.76%	53.04%
22	1.84E+07	2.74E+07	2.84E+07	48.56%	54.10%
23	1.91E+07	2.86E+07	2.96E+07	49.40%	54.38%
24	2.01E+07	3.11E+07	3.19E+07	54.52%	58.47%
25	2.09E+07	3.21E+07	3.31E+07	53.42%	57.88%

Table 4: SET performance at 100Gb/s (continued)

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
26	2.13E+07	3.33E+07	3.38E+07	56.38%	58.73%
27	2.19E+07	3.45E+07	3.49E+07	57.87%	59.61%
28	2.28E+07	3.69E+07	3.64E+07	61.95%	59.78%

Table 5: GET performance at 100Gb/s

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
1	1.82E+06	1.80E+06	1.81E+06	-1.09%	-0.02%
2	3.28E+06	3.68E+06	3.57E+06	12.42%	8.81%
3	4.84E+06	5.39E+06	5.19E+06	11.35%	7.18%
4	6.21E+06	7.12E+06	6.80E+06	14.69%	9.57%
5	7.66E+06	8.83E+06	8.42E+06	15.21%	9.89%
6	8.80E+06	1.06E+07	9.97E+06	20.24%	13.38%
7	1.02E+07	1.23E+07	1.15E+07	20.32%	12.98%
8	1.13E+07	1.39E+07	1.30E+07	23.14%	15.10%
9	1.26E+07	1.56E+07	1.44E+07	24.40%	14.80%
10	1.35E+07	1.73E+07	1.57E+07	28.07%	15.84%
11	1.46E+07	1.90E+07	1.72E+07	30.41%	18.30%
12	1.56E+07	2.07E+07	1.86E+07	32.47%	19.23%
13	1.64E+07	2.23E+07	1.99E+07	36.11%	21.04%
14	1.72E+07	2.39E+07	2.11E+07	38.88%	22.63%
15	1.82E+07	2.56E+07	2.25E+07	40.48%	23.26%
16	1.91E+07	2.73E+07	2.37E+07	43.21%	24.28%
17	2.01E+07	2.89E+07	2.47E+07	44.08%	23.04%
18	2.06E+07	3.05E+07	2.56E+07	48.01%	23.89%
19	2.12E+07	3.22E+07	2.70E+07	51.77%	27.36%
20	2.22E+07	3.38E+07	2.83E+07	52.34%	27.68%

Table 5: GET performance at 100Gb/s (continued)

Redis instances	Kernel	Onload performance	Onload balanced	Onload performance gain	Onload balanced gain
21	2.34E+07	3.54E+07	2.95E+07	51.35%	26.10%
22	2.35E+07	3.70E+07	3.07E+07	57.47%	30.52%
23	2.45E+07	3.86E+07	3.19E+07	57.77%	30.24%
24	2.60E+07	4.18E+07	3.25E+07	60.98%	25.24%
25	2.64E+07	4.34E+07	3.25E+07	64.69%	23.17%
26	2.73E+07	4.50E+07	3.26E+07	64.98%	19.54%
27	2.74E+07	4.66E+07	3.24E+07	70.19%	18.34%
28	2.91E+07	4.98E+07	3.37E+07	71.27%	15.83%

6.2 Analysis

Results from the 25Gbe database SET and GET performance tests show a gain in requests/sec of up to 50.19% and 103.28% respectively, using Cloud Onload (redis-balanced and redis-performance) vs kernel. Gains flatten out above 12 instances of `redis-server`, showing saturation of the traffic that can be pushed through a 25Gbe NIC. This flattening out was also present in Solarflare 10Gbe NICs (not graphed).

Results from the 100Gbe database SET and GET performance tests show a gain in requests/sec of up to 61.95% and 71.27% respectively, using Cloud Onload (redis-balanced and redis-performance) vs kernel.

Comparing Cloud Onload vs kernel, requests to the database are processed faster. This is because Redis receives data straight from the network directly into its memory, without a detour through the kernel. This direct path reduces memory copies, eliminates kernel context switches, and removes other system overhead. The result is a dramatic reduction in time, and CPU cycles. Similarly, when Redis fulfills a database request it can write that data directly to the network. This again saves more time, and reclaims more CPU cycles.

As more CPU cycles are freed up due to decreased latency, those compute resources go directly back into processing additional Redis database requests. The result is up to 36.9 million requests/second.