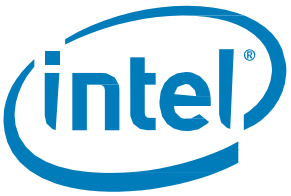


Intel® Memory Drive Technology for Redis

Performance Evaluation Guide

May 2018

Document Number: 337669-001US

Ordering Information

Contact your local Intel sales representative for ordering information.

Revision History

Revision Number	Description	Revision Date
001	• Initial release	May 2018

Contents

1 Introduction 5
1.1 Memory Use and the Need for More Memory per Server 5
1.2 The Scalability Challenge for In-memory Deployments 5
1.3 Intel® Memory Drive Technology Benefits and Capabilities 5
2 Redis Performance Benchmark 6
2.1 Benchmark Landscape 6
2.2 Intel® Memory Drive Technology Measured Performance 7
2.3 Cost Efficiency and Overall Savings 8
2.4 Summary 8
3 Benchmark Setup In details 9
4 System and Benchmark Installation and Configuration 10
4.1 Server System Installation and Configuration 10
4.2 Master Client (load) System Installation and configuration 10
4.3 Client (load) System Installation and configuration 18
4.4 System Connectivity Configuration 19
5 Running the Benchmark 20
5.1 Initialization the Benchmark on the master client 20
5.2 Collecting Results 20
5.3 Results obtained in the lab 21
6 Conclusions 22

Figures

Figure 1: DRAM-like Performance 5
Figure 2: Client and Server Configuration #1 6
Figure 3: Client and Server Configuration #2 7
Figure 4: Performance Comparison: DRAM vs Intel® Memory Drive Technology 7
Figure 5: DRAM vs Intel® Memory Drive Technology Cost Structures 8

Tables

Table 1: Test Results 21

Terms and Acronyms

Term	Definition
AHCI	Advanced Host Controller Interface
API	Application Programming Interface
ATA	Advanced Technology Attachment
DIPM	Device Initiated Power Management
GB	Gigabyte
HDD	Hard Disk Drive
KB	Kilobytes
I/O	Input/Output (the typical block used in specifications is 4 kB)
IOPS	Input/Output Operations Per Second
MB	Megabytes
NCQ	Native Command Queuing
PCH	Platform Controller Hub
RAID	Redundant Array of Independent Disks
SATA	Serial Advanced Technology Attachment
SSD	Solid State Drive

Sources

Additional information for topics discussed in this guide:

- https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html

1 Introduction

1.1 Memory Use and the Need for More Memory per Server

In-memory data-stores and caching engines, such as Redis and Memcached, are widely used in a variety of application domains, such as Ad-Tech, Financial Services, Gaming, Healthcare, and IOT. In-memory data-stores can improve application performance as well as reduce scale-out costs.

Such in-memory caching engines improve application performance by storing frequently accessed data in the main memory, so they can be retrieved quickly without the need to access the persistent data store.

In order to achieve the highest performance, the entire dataset is stored in-memory. If the data being handled is larger than the available memory in a single server, these caching engines allow for scale-out to multiple nodes (using sharding) according to the maximum amount of memory available per node.

1.2 The Scalability Challenge for In-memory Deployments

The amount of memory available on a single server, as well as prohibitive DRAM pricing, can limit in-memory computing deployments. Due to modern server architecture, the maximum amount of memory per node is capped at a maximum of 12 DIMMs per socket, and in many cases, with high-density servers, as few as 6-8 DIMMs per socket.

These limitations, combined with the fact that the per GB cost of DRAM increases exponentially for DIMMs larger than 64GiB, leads to a practical limit of 768GiB to 1.5TiB per dual socket node.

As a result, the number of nodes required for a large-scale solution (of a single tenant and/or multi-tenants on the same infrastructure), is determined by the amount of memory in each node, rather than by the compute capacity of the node. This is evident in typical deployment scenarios where CPU cluster utilization is commonly 10%-20%.

In order to implement large-scale solutions, organizations are forced to either use expensive high-density DIMMs, or increase the number of nodes. Both of these options result in much higher infrastructure costs per application, as well as an increase to the data-center footprint.

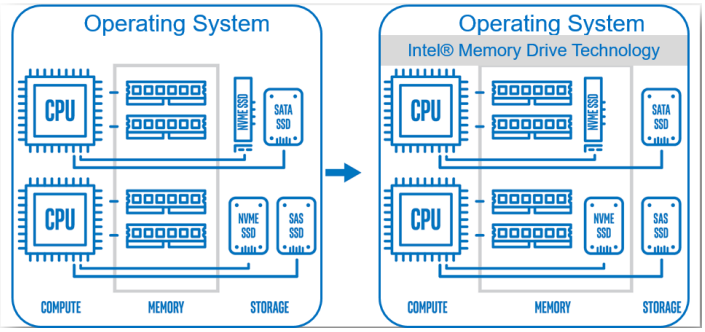
1.3 Intel® Memory Drive Technology Benefits and Capabilities

Increasing the total memory per node, or reducing the cost per GiB of memory, can significantly improve the cost structure of such in-memory caching engines, and improve their adaptability to additional use-cases.

The Intel® Memory Drive Technology is a revolutionary software-defined memory (SDM), which transparently integrates Intel® Optane™ SSDs into the memory subsystem and makes it appear like DRAM to the OS and applications.

Intel® Memory Drive Technology increases memory capacity beyond DRAM limitations in a significantly more cost-effective way, and delivers DRAM-like performance to the operating system and applications in a completely transparent manner. In addition, no changes are required to the OS or applications.

Figure 1: DRAM-like Performance



§

2 Redis Performance Benchmark

The Redis benchmark uses high concurrency SET/GET operations of small (1kB) and large (100kB) messages (reflecting small values and large objects), in which the Redis server and client/load system are connected over 10GbE.

Since it is impossible to benchmark against a 6TB DRAM-only server, our comparison was performed using a 768GB DDR4 DRAM-only server, compared to a server with 192GB DDR4, augmented with Intel® Memory Drive Technology, for a total of 768GB.[1] The amount of memory consumed by Redis was ~700GB.

2.1 Benchmark Landscape

This benchmark testing is using a server machine with Redis server software installed, and client systems (one or many, depends on your lab environment) which is used to generate a high load of "SET" and "GET" keys and values on/from the Redis server. In order to be able to generate substantial load to test the performance of the system running the Redis server, the client system needs to have a large number of cores (recommended: use double the number of sockets of your Redis server to generate the load), and the clients and server machines need to be connected with at-least 10GbE Ethernet connection between them.

In this benchmark, the first (or only) client will also serve as the "master client" which will orchestrate the launch of Redis servers on the system and Redis clients on the client machines.

- The Redis server is started with multiple instances as requested (typically a multiple of the number of logical CPUs on the system in order to have enough concurrency for optimal performance).
- As Redis is typically configured to not use persistent storage, the instances are 'empty" at the start of each scenario.
- Once all Redis instances are started, Redis cache is dropped, and the client starts to SET keys on all instances in parallel.
- Once SET is done, the client starts GET all keys from all instances in parallel.
- In this benchmark, the 2 steps above are repeated 3 times and detailed statistics and histograms are collected and calculated.

Figure 2: Client and Server Configuration #1

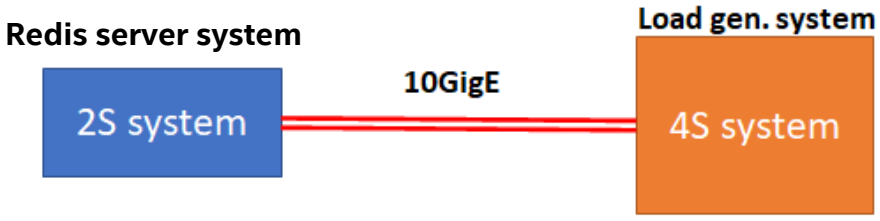
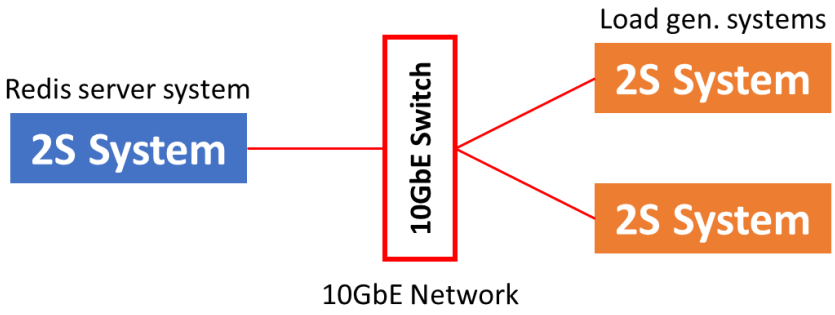
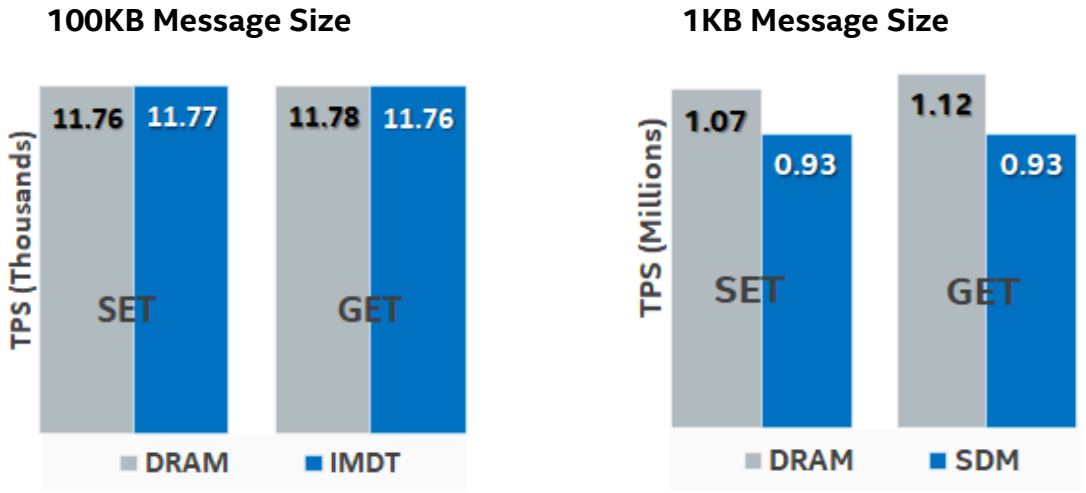


Figure 3: Client and Server Configuration #2



2.2 Intel® Memory Drive Technology Measured Performance

Figure 4: Performance Comparison: DRAM vs Intel® Memory Drive Technology



99 percentile latency: <17ms

99 percentile latency: <300µs

Deliver 83%-99% of DRAM-only performance for a dataset 4x larger than DRAM

NOTES:

Test and System Configuration:

Redis Server 1# (DRAM Mode):
CPU: Intel® Xeon® E5-2687W v4 3.0GHz 30MB 160W 12 cores, CPU Sockets: 2, RAM Capacity: 32Gx24, RAM Model: DDR4, RAM Stuffing: NA, DIMM Slots Populated: 24 slots, PCIe Attach: CPU (not PCH lane attach), Chipset: Intel C610 chipset, BIOS: SE5C610.86B.01.01.0019.101220160604, Switch/ReTimer Model/Vendor: Intel A2U44X25NVMEDK, OS: CentOS 7.3.1611, Kernel: 3.0.10-693, NVMe Driver: Inbox, C-states: Disabled, Hyper Threading: Disabled, CPU Governor (through OS): Performance Mode (Default Mode: Balanced)

Redis Server 2# (Intel® Memory Drive Technology Mode):
CPU: Intel® Xeon® E5-2687W v4 3.0GHz 30MB 160W 12 cores, CPU Sockets: 2, RAM Capacity: 32Gx6, RAM Model: DDR4, RAM Stuffing: NA, DIMM Slots Populated: 6 slots, PCIe Attach: CPU (not PCH lane attach), Chipset: Intel C610 chipset, BIOS: SE5C610.86B.01.01.0019.101220160604, Switch/ReTimer Model/Vendor: Intel A2U44X25NVMEDK, OS: CentOS 7.3.1611, Kernel: 3.0.10-693, NVMe Driver: Inbox, C-states: Disabled, Hyper Threading: Disabled, CPU Governor (through OS): Performance Mode (Default Mode: Balanced), 2 x 375GB Intel® Optane™ SSD DC P4800X U.2

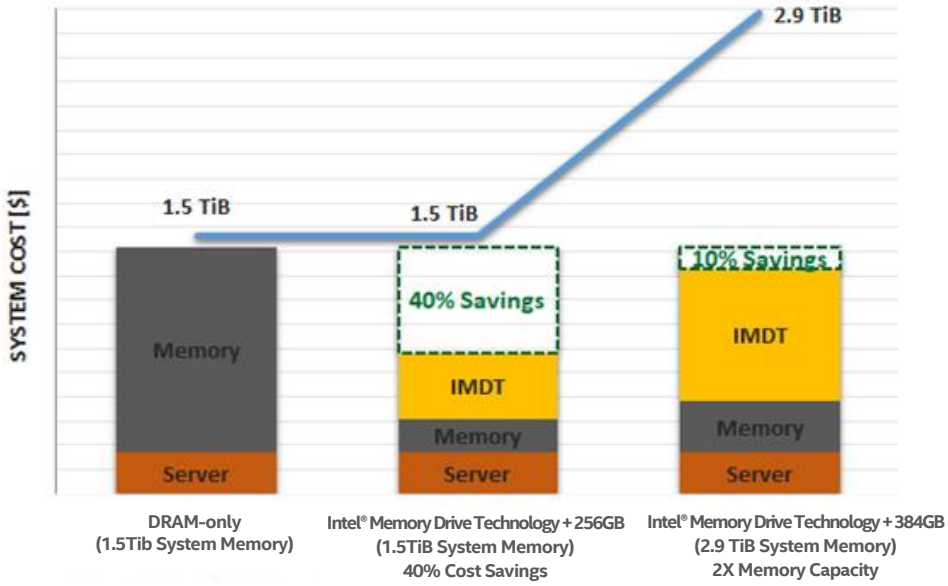
Redis Clients:
CPU: Intel® Xeon® E5-2687W v4 3.0GHz 30MB 160W 12 cores, CPU Sockets: 2, RAM Capacity: 32Gx6, RAM Model: DDR4, RAM Stuffing: NA, DIMM Slots Populated: 6 slots, PCIe Attach: CPU (not PCH lane attach), Chipset: Intel C610 chipset, BIOS: SE5C610.86B.01.01.0019.101220160604, Switch/ReTimer Model/Vendor: Intel A2U44X25NVMEDK, OS: CentOS 7.3.1611, Kernel: 3.0.10-693, NVMe Driver: Inbox, C-states: Disabled, Hyper Threading: Disabled, CPU Governor (through OS): Performance Mode (Default Mode: Balanced)

2.3 Cost Efficiency and Overall Savings

For a significantly lower cost than DRAM, Intel® Memory Drive Technology enables reaching the same memory configuration while allowing for larger amounts of memory, much larger than the practical limitations of a given server.

The chart below compares the different cost structures.

Figure 5: DRAM vs Intel® Memory Drive Technology Cost Structures



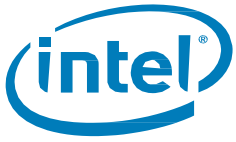
- DRAM-only server with 1.5 TiB of memory
- Server configured with 256GiB RAM + Intel® Memory Drive Technology to reach the same overall amount of memory (1.5 TiB), at 40% lower cost than DRAM-only.
- Server configured with 384GiB RAM + Intel® Memory Drive Technology to reach almost double the total amount of memory (2.9 TiB total) for a similar cost.

Additional cost savings are achieved due to the Intel® Memory Drive Technology solution requiring a smaller datacenter footprint, and reduced energy and maintenance costs.

2.4 Summary

Leveraging Intel® Memory Drive Technology provides a cost-effective way to support nodes with up to 8x more memory than server's specifications' limits, thereby enabling a cost-effective infrastructure for In-Memory data-stores and caching engines, with minimal impact on performance.

§



3 Benchmark Setup In details

For the workload in our example, at least two client systems are required:

- One will be running Redis server
- One (or more) will be the load generator which will be using a Redis client program

If using just one client system, we recommended the following in order to generate the load on the server system:

  - A large (high core count) system
  - Only use a high speed Ethernet connection to the server system (at least 10GE)

In this document, we will focus on a configuration using Centos* (7.3 or 7.4). Those steps can be applied to other distributions, with minor changes in the installation process. We assume that the OS has been pre-installed.

All scripts are written such that you can use more than one client; however it will also work if you configure only one client.

§

4 System and Benchmark Installation and Configuration

4.1 Server System Installation and Configuration

This procedure assumes that the system used has the latest CentOS (7.3 or 7.4) installed, and updated to latest version by executing "**`yum -y update`**".

1. Install a recent copy of Centos (CentOS 7.3 or 7.4) or upgrade your system
2. Login to the server system as "root"
3. Make sure it is updated:

```
# yum update -y
```

4. Create a folder named "redis":

```
# mkdir /root/redis
```

5. Step into the "redis" folder:

```
# cd /root/redis
```

6. Obtain the latest Redis software:

```
# wget http://download.redis.io/releases/redis-4.0.2.tar.gz
```

7. Open the tar file:

```
# tar xzf redis-4.0.2.tar.gz
```

8. Step into the extracted folder:

```
# cd redis-4.0.2
```

9. Build the Redis software:

```
# make
```

4.2 Master Client (load) System Installation and configuration

This procedure assumes that the system used has the latest CentOS (7.3 or 7.4) installed, and updated to latest version by executing "**`yum -y update`**".

1. Install a recent copy of Centos (CentOS 7.3 or 7.4) or upgrade your system
2. Login to the server system as "root"
3. Make sure it is updated:

```
# yum update -y
```

4. Ensure the "screen" packdge is installed

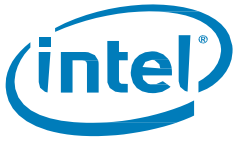
```
# yum install -y screen
```

5. Create a folder named "redis":

```
# mkdir /root/redis
```

6. Step into the "redis" folder:

```
#cd /root/redis
```



7. Put the runscript below in “do_it.sh”.

   Please make sure that REDISROOT is defined correctly in the script (in case you are using a directory different than /root/redis/).

   Please note that this script assumes password-less connection is configured between master-client, clients and server machines (as is outlined in more details in section 4).

   Configuring the invocation script (do_it.sh):

   (a) *SERVER- IP* Address of server system (10/25GB ethernet connection)

   (b) *CLIENT_IP* list of client(s) IP addresses

   (c) *SERVER_CORES* Number of locical CPUs on server system

   (d) *SRV_OVERIDE* Number of Redis servers to run (Recommended to run a multiple of the number of logical CPUs on the server machine)

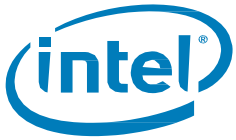
   The invocation script below includes examples of running the benchmark with long value (100,000) and short value (10,000) keys.

```
#!/bin/sh
set -x

#Logical CPU count of the machine running the redis server
SERVER_CORES=72
#IP address of the machine running the redis server, so it can be seamlessly accessed from the
client machine
SERVER=192.168.3.1
# Total Number of redis servers to run (Typically a multiple of the number of CPUs on the server
machine)
SRV_OVERIDE=288
VER=4.0.2
REDISTROOT=/root/redis/redis-$VER

REDISRUNS=3
#All clients IP addresses (including master-client)
CLIENT_IP=( 192.168.1.92 192.168.1.90 )

############ Do not edit below this line ###################################
#Address of master-client
CLIENT=${CLIENT_IP[0]}
#calculate the number clinets
CLIENT_NUM=${#CLIENT_IP[@]}
#execution log
SCRIPT_LOG=script_`date +%Y%m%d-%H%M%S`.log
#Configure utility prefix
for fname in `ssh $SERVER "echo -n /usr/local/bin/????version | sed
's/^\(.*\)\(\/usr\/local\/bin\/vsmpversion\)\(.*\)$/\1\3 \2/'"`; do
        cn=`ssh $SERVER "strings $fname 2>&1 | grep -io \"vsmpversion\"|wc -l"`
        if [ $cn != 0 ]; then UTILPX=`basename $fname | awk '{print substr($0,0,4)}'`; break; fi
done
# Compile test program
gcc -g -O2 -fopenmp -o test test.c -I$REDISTROOT/deps/hiredis -L$REDISTROOT/deps/hiredis -lhiredis
if [ $? -ne 0 ]; then
    echo Compilation of benchmark failed. exiting.
    exit 1
fi
#############################################
exec_on_server()
{
  while : ; do
    ssh $SERVER "$1"
    [ $? -le 1 ] && break
    sleep 1
  done
}
#############################################
echo "Total redis servers: $SRV_OVERIDE"
#Prepare server only once
        echo  "Preparing the server..."
        #Stop un-needed services on server
```

```
        echo 'for i in ksm.service ksmtuned.service irqbalance.service tuned.service firewalld
iptables; do echo $i;systemctl stop $i;done' > /tmp/run_on_server$$.sh
        ssh $SERVER 'bash -s' < /tmp/run_on_server$$.sh
        rm -f /tmp/run_on_server$$.sh
        #Set CPU power govoner to performance
        exec_on_server 'cpupower frequency-set -g performance; cpupower set -b 0'
        #Distribute NIC irq acroos CPU cores
        MAC=`arp $SERVER | grep ether | awk '{print $3}'`
        msi_dir=`ssh $SERVER "grep $MAC /sys/class/net/*/address" |sed 's/[^/]*$/device\/msi_irqs/'`
        #
        msis=`ssh $SERVER "ls -1 $msi_dir | wc -l" `

        step=$(($SERVER_CORES/msis))
        step=$((step<1?1:step))
        msi_list=`ssh $SERVER "ls $msi_dir"`
        j=0
        for msi in $msi_list; do
                current=`ssh $SERVER "cat /proc/irq/$msi/smp_affinity_list"`
                new="$j-$((j+step-1))"
                exec_on_server "echo $new > /proc/irq/$msi/smp_affinity_list"
                now=`ssh $SERVER "cat /proc/irq/$msi/smp_affinity_list"`
                j=$((j+step))
        j=$((j>=SERVER_CORES?j=0:j))
        done


VER=4.0.2
REDISTROOT=/root/redis/redis-$VER
REDISTMP=/tmp/redis
LOCALTMP=/tmp/redis.txt
BINDIR=$REDISTROOT/src
REDISBASE=6379
MAXTHREADS=256

##############################################
kill_all_redis_servers()
{
  echo "Removing any running redis servers"
  while : ; do
    ssh $SERVER 'killall -9 redis-server'
    [ $? -le 1 ] && break
    sleep 1
  done
}

get_server_process_list()
{
  while : ; do
    ssh $SERVER 'ps -efa' 2>/dev/null
    [ $? -eq 0 ] && break
    sleep 1
  done
}

get_server_free_memory()
{
  while : ; do
    ssh $SERVER "free -g" 2>/dev/null
    [ $? -eq 0 ] && break
    sleep 1
  done
}
##############################################

#calculate new start port


# Get memory information
memory=`get_server_free_memory | grep Mem: | head -1 | awk '{print $2}'`

echo Overiding $REDISNMBR with $SRV_OVERIDE
REDISNMBR=$SRV_OVERIDE
REDISKLEN=1000000
if [ "$#" -ge 2 ]; then
    REDISKLEN=$2
fi

#Parameters used to calculate the required amount of memory as base and needed by redis per key
RAMPCT=90
BASEMEM=8
```

```
KEYSIZEFACTOR=110

cat << EOF > /tmp/runclient.sh
#!/bin/bash
while : ; do
        killall -9 redis-server
        sleep 10
        num=\`ps -efa | grep redis | grep -v grep | wc -l\`
        echo "\$num servers are up.  We need 0."
        [ \$num -ne 0 ] || break
        echo "  servers are up (\$num != 0). retrying."
done

rm -rf $REDISTMP

sysctl -wq vm.overcommit_memory=1
sysctl -wq net.core.somaxconn=1024

cpus=\`cat /proc/cpuinfo | grep processor | wc -l\`
step=\$((cpus/$REDISNMBR))
step=\$((step<1?1:step))
j=0

mkdir -p /etc/redis/

for port in \`seq $REDISBASE $(($REDISBASE + $REDISNMBR - 1))\`;do
 mkdir -p $REDISTMP/redis-\$port
 chmod 777 $REDISTMP/redis-\$port

 rm -f /etc/redis/\$port.conf
 cp $REDISTROOT/redis.conf /etc/redis/\$port.conf
 sed -i "s/^bind .*/bind $SERVER/" /etc/redis/\$port.conf
 sed -i "s/^protected-mode yes/protected-mode no/" /etc/redis/\$port.conf
 sed -i "s/port 6379/port \$port/" /etc/redis/\$port.conf
 sed -i "s|dir ./|dir $REDISTMP/redis-\$port/|" /etc/redis/\$port.conf
 sed -i "s|logfile \"\"|logfile $REDISTMP/redis-\$port.log|" /etc/redis/\$port.conf
 sed -i "s/^save /# save/" /etc/redis/\$port.conf

 taskset -c \$j-\$((j+step-1)) $BINDIR/redis-server /etc/redis/\$port.conf &

 j=\$((j+step))
 j=\$((j>=cpus?j=0:j))
done
EOF

kill_all_redis_servers

  echo  "Getting proccess list from server (if stuck restart sshd on server)..."
while : ; do
  num=`get_server_process_list | grep redis | wc -l`
  [ $num -eq 0 ] && break
  echo "$num servers still up on $SERVER"
  sleep 10
done
echo "Done"
sleep 5

while : ; do
  ssh $SERVER "${UTILPX}ctl --nna=on ; echo never > /sys/kernel/mm/transparent_hugepage/enabled"
  [ $? -eq 0 ] && break
  sleep 2
done

cmd="uname -a; echo; free -g; echo; uptime; echo; grep ^
/sys/kernel/mm/transparent_hugepage/enabled; echo; lscpu; echo; if [ -x
/usr/local/bin/${UTILPX}version ]; then ${UTILPX}version -vvv; ${UTILPX}ctl --status; else echo
'Running NATIVE'; fi; echo"
rm -f setup.log
for host in SERVER CLIENT; do
        (echo "===>> $host system -"; ssh `eval echo "$"${host}` "$cmd") >> setup.log 2>&1
done

while : ; do
  echo "Starting $REDISNMBR servers on $SERVER"
  #ssh $SERVER 'bash -s' < /tmp/runclient.sh 2>/dev/null &
  ssh $SERVER 'bash -s' < /tmp/runclient.sh &
  PID=$!
  trap "{ echo -e \"\n===============\nQuit Requested.\n===============\" ; kill $PID ;
kill_all_redis_servers ; exit 1; }" INT
  sleep 10
```

```
  [ -e /proc/$PID ] && break
  echo "Failed... Retrying..."
done

echo  "Getting process list from server (if stuck restart sshd on server)..."
while : ; do
  num=`get_server_process_list | grep redis | wc -l`
  echo "$num servers are up.  We need $REDISNMBR."
  [ $num -ne $REDISNMBR ] || break
  echo "  servers are not up ($num != $REDISNMBR). waiting."
  sleep 2
done
sleep 3
echo REDIS setup finished

# Server is ready , lets run the clients
for len in 1000 100000 ; do
#Calculating required number of redis keys, based on available memory
REDISKEYS=$(((memory-BASEMEM)*1024*1024*1024*RAMPCT/KEYSIZEFACTOR/len))
REDISKEYS=$((REDISKEYS/10000))
REDISKEYS=$((REDISKEYS*10000))
#Ruild Redis servers start script
for j in `seq 1 $REDISRUNS`; do
 echo "Running instance $j of $REDISRUNS"
 MYSERVERS=$((REDISNMBR/CLIENT_NUM))
 client_num=0
 for ip in ${CLIENT_IP[*]}; do
        logts=redis.log.$ip.`date +%Y%m%d-%H%M%S`.$j
        cp setup.log $logts
        new_base=$((client_num*MYSERVERS))
        new_base=$((REDISBASE+new_base))
        echo "`date` Starting $MYSERVERS clients on $ip starting at port $new_base" >> $SCRIPT_LOG
        echo "`date` Running /root/redis/test $SERVER $new_base $len $((REDISKEYS/REDISNMBR))"  >>
$SCRIPT_LOG
        echo "#!/bin/bash" > /tmp/screen_$ip
        echo "ssh $ip \"export OMP_NUM_THREADS=$MYSERVERS ;cd /root/redis/; time /root/redis/test
$SERVER $new_base $len $((REDISKEYS/REDISNMBR))\"  >> $logts" >> /tmp/screen_$ip
        chmod 755  /tmp/screen_$ip
        screen -S Redis_client -d -m  /tmp/screen_$ip
        client_num=$((client_num+1))
 done
 # Wait for all screen to finish
 set +x
 echo "`date` Waiting for all screen to finish"  >> $SCRIPT_LOG
 while : ; do
        screen -ls | grep -q Redis_client
        if [ $? != 0 ]; then
                break
        fi
        sleep 5
        done
        echo "`date` All screen sessions have finished" >> $SCRIPT_LOG
        echo DONE >> $logts
done
set -x
done
exit
```



8. Put the code below in “test.c”

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hiredis.h"
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <omp.h>

// DEFAULTS:
#define DEF_HOST "127.0.0.1"
#define DEF_PORT 6789
#define DEF_SIZE 100
#define DEF_COUNT 200000000

// Histogram data
#define MAX_THREADS 256
#define MAX_HISTOGRAM 1000000
#define PAD (2*4096/sizeof(int))
long histogram[MAX_THREADS][MAX_HISTOGRAM + PAD];

char *value;
int threads = 1;
char hostname[256];
int port = DEF_PORT;
int val_size = DEF_SIZE;
long key_count = DEF_COUNT;

void save_and_reset_hist(char *histname)
{
    int m, i;
    FILE *hist;
    char nameBuff[64] = { 0 };

    strncpy(nameBuff, histname, 63);
    int filedes = mkstemp(nameBuff);

    if (NULL != (hist = fdopen(filedes, "w"))) {
        fprintf(hist, "us");
        for (m = 0; m < threads; m++)
            fprintf(hist, " %d", m);
        fprintf(hist, " total\n");

        int first = -1;
        int last = 0;
        for (i = 0; i <= MAX_HISTOGRAM; i++)
            for (m = 0; m < threads; m++) {
                if ((first == -1) && histogram[m][i]) first = i;
                if (histogram[m][i]) last = i;
            }

        if (first == -1) first = last + 1;

        long all = 0, all_acc = 0;
        for (i = first; i <= last; i++)
            for (m = 0; m < threads; m++)
                all += histogram[m][i];

        for (i = first; i <= last; i++) {
            fprintf(hist, "%d", i);
            long total = 0;
            for (m = 0; m < threads; m++) {
                fprintf(hist, " %d", histogram[m][i]);
                total += histogram[m][i];
            }
            all_acc += total;
            fprintf(hist, " %ld %.3f\n", total, (double) 100 * (double) all_acc / (double) all);
        }

        fclose(hist);
        printf("Histogram saved to file %s\n", nameBuff);
    } else
        printf("Unable to save histogram to file %s\n", nameBuff);

    memset(histogram, 0, sizeof(histogram));
}
```

```
long timestamp(void)
{
    static long start_time = -1;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    if (start_time == -1)
   start_time = tv.tv_sec;
    return ((long)(tv.tv_sec - start_time))*1000000L+tv.tv_usec;
}

char *datestr(void)
{
    time_t now;
    time(&now);
    return strtok(ctime(&now),"\n");
}

int main(int argc, char *argv[])
{
    // Arguments: HOST PORT SIZE COUNT
    // Number of threads will be taken from OMP_NUM_THEADS
    threads = omp_get_max_threads();
    if (threads > MAX_THREADS) {
   printf("Too many threads requested, maximum supported is %d\n", MAX_THREADS);
   exit(1);
    }

    strcpy(hostname, DEF_HOST);
    if (argc > 1) strcpy(hostname, argv[1]);
    if (argc > 2) port = atoi(argv[2]);
    if (argc > 3) val_size = atoi(argv[3]);
    if (argc > 4) key_count = atol(argv[4]);
    printf("threads = %d start_port = %d value_len = %d key_count = %ld\n", threads, port, val_size,
key_count);

#pragma omp parallel
    {
   redisContext *con;
   redisReply *reply;
   struct timeval timeout = { 1, 500000 };    // 1.5 seconds
   char line[1024];
   long i;
   int tid = omp_get_thread_num();
   char *value = (char*) malloc(val_size+1);

   if (NULL == value) {
       sprintf(line, "Out Of Memory in thread %d\n", tid);
       printf(line); fflush(stdout);
       exit(1);
   }

   for (i = 0; i < val_size; i++)
       value[i] = '0' + ((tid+i) % 75);
   value[val_size] = '\0';

   for (i = 0; i <= MAX_HISTOGRAM; i++)
       histogram[tid][i] = 0;

#pragma omp barrier
   if (tid == 0) {
       sprintf(line, "%s : Value(s) allocated, connecting\n", datestr());
       printf(line); fflush(stdout);
   }

   con = redisConnectWithTimeout(hostname, port+tid, timeout);
   if (NULL == con) {
       sprintf(line, "Thread %3d: port %d connection error: can't allocate redis context\n", tid,
port+tid);
       printf(line); fflush(stdout);
       exit(1);
   }
   if (con->err != 0) {
       sprintf(line,"Thread %3d: port %d connection error: %s\n", tid, port+tid, con->errstr);
       printf(line); fflush(stdout);
       redisFree(con);
       exit(1);
   }
   sprintf(line, "Thread %3d: port %d connected\n", tid, port+tid);
   printf(line); fflush(stdout);
```

```
    reply = redisCommand(con, "flushall");
    sprintf(line, "Thread %3d: port %d flushall :%s\n", tid, port+tid, reply->str);
    printf(line); fflush(stdout);
    freeReplyObject(reply);
#pragma omp barrier
    if (tid == 0) {
        sprintf(line, "%s : %d threads started - set\n", datestr(), threads);
        printf(line); fflush(stdout);
    }

    // SET
    long maxtime = 0, av_time = 0, max_av_time = 0, total = 0, singlestart, elapsed, start,
total_elapse;
    start = timestamp();
    for (i = 0; i < key_count; i++) {
        singlestart = timestamp();
        reply = redisCommand(con, "SET %ld %s", i, value);
        if (strcmp(reply->str, "OK") != 0) {
        sprintf(line, "Thread %3d: port %d SET %ld %s reply error! reply:%s\n", tid, port+tid, i,
"<VALUE>", reply->str);
        printf(line); fflush(stdout);
        }
        elapsed = timestamp() - singlestart;
        histogram[tid][elapsed >= MAX_HISTOGRAM ? MAX_HISTOGRAM : elapsed]++;
        total = total + elapsed;
        av_time = total / (i+1);
        if (elapsed > maxtime || av_time > max_av_time) {
        if (elapsed > maxtime) {
            maxtime = elapsed;
        }
        if (av_time > max_av_time) {
            max_av_time = av_time;
        }
        sprintf(line, "Thread %3d: port %d set key %8d - max time %6ld usecs , max average %5ld\n",
tid, port+tid, i, maxtime, max_av_time);
        printf(line); fflush(stdout);
        }
        freeReplyObject(reply);
    }
    total_elapse = timestamp() - start;
    sprintf(line, "Thread %3d: port %d set %d keys - elapse %ld usecs , max time %ld , average %ld ,
max average %ld\n", tid, port+tid, key_count, total_elapse, maxtime, av_time, max_av_time);
    printf(line); fflush(stdout);
#pragma omp barrier
    if (tid == 0) {
        sprintf(line, "%s : %d threads finished - set\n", datestr(), threads);
        printf(line); fflush(stdout);
        save_and_reset_hist("histogram-set-XXXXXX");
    }
#pragma omp barrier
    if (tid == 0) {
        sprintf(line, "%s : %d threads started - get\n", datestr(), threads);
        printf(line); fflush(stdout);
    }
    /* Get */
    start = timestamp();
    maxtime = av_time = max_av_time = total = 0;

    for (i = 0; i < key_count; i++) {
        singlestart = timestamp();
        reply = redisCommand(con, "GET %ld", i);
        if (NULL == reply) {
        sprintf(line, "Thread %3d: port %d GET %ld reply is NULL!\n", tid, port+tid, i);
        printf(line); fflush(stdout);
        }
        else if (strcmp(reply->str, value) != 0) {
        sprintf(line, "Thread %3d: port %d GET %ld reply error! reply:%s\n", tid, port+tid, i,
reply->str);
        printf(line); fflush(stdout);
        }
        elapsed = timestamp() - singlestart;
        histogram[tid][elapsed >= MAX_HISTOGRAM ? MAX_HISTOGRAM : elapsed]++;
        total = total + elapsed;
        av_time = total / (i + 1);
        if (elapsed > maxtime || av_time > max_av_time) {
        if (elapsed > maxtime) {
            maxtime = elapsed;
        }
        if (av_time > max_av_time) {
            max_av_time = av_time;
```

```
        }
        sprintf(line, "Thread %3d: port %d get key %8ld - max time %6ld usecs , max average %5ld\n",
tid, port+tid, i, maxtime, max_av_time);
        printf(line); fflush(stdout);
        }
        freeReplyObject(reply);
    }
    total_elapse = timestamp() - start;
    sprintf(line, "Thread %3d: port %d get %ld keys - elapse %ld usecs , max time %ld , average %ld ,
max average %ld\n", tid, port+tid, key_count, total_elapse, maxtime, av_time, max_av_time);
    printf(line); fflush(stdout);
#pragma omp barrier
    if (tid == 0) {
            sprintf(line, "%s : %d threads finished - get\n", datestr(), threads);
            printf(line); fflush(stdout);
        save_and_reset_hist("histogram-get-XXXXXX");
    }
    redisFree(con);
    free(value);
     }
     printf("%s : Complete.\n", datestr());
}
```

9. Obtain the latest Redis software:

```
#wget http://download.redis.io/releases/redis-4.0.2.tar.gz
```

10. Open the tar file:

```
#tar xzf redis-4.0.2.tar.gz
```

11. Step into the extracted folder:

```
#cd redis-4.0.2
```

12. Build the redis software:

```
#make install
```

13. Build the redis client:

```
# cd /root/redis/redis-4.0.2/deps/hiredis
# make install
# gcc -g -O2 -fopenmp -o test test.c -I/root/redis/redis-4.0.2/deps/hiredis -L/root/redis/redis-
4.0.2/hiredis -lhiredis
```

4.3 Client (load) System Installation and configuration

This procedure assumes that the system used has the latest CentOS (7.3 or 7.4) installed, and updated to latest version by executing "**`yum -y update`**".

1. Install a recent copy of Centos (CentOS 7.3 or 7.4) or upgrade your system
2. Login to the master system as "root". And use passwordless connection to connect to each client (which is not the master client). Replace CLIENT below with the client IP address

```
# cd /root/redis
```

3. Make sure client is updated:

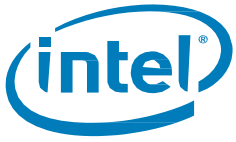
```
# ssh CLIENT "yum update -y"
```

4. Create a folder named "redis":

```
# ssh client "mkdir /root/redis"
```

5. Copy the test binary to the client macine:

```
#scp test CLIENT:/root/redis
```



4.4 System Connectivity Configuration

Make sure root can ssh from the master client machine (a) to itself and (b) to the other system without password (using keys), as the client run script is using ssh to connect to the server system.

This procedure assumes that the system used has the latest CentOS (7.3 or 7.4) installed, and updated to latest version by executing "**yum -y update**". You must also have the make and gcc packages installed.

1. Install a recent copy of Centos (CentOS 7.3 or 7.4) or upgrade your system
2. Login to the master client machine as "root"
3. Make sure it is updated:

```
# yum update -y
```

4. Generate a public/private RSA key pair.

```
# ssh-keygen -A -r rsa
```

5. Copy your public RSA key to each of the machines in the setup (including the current machine). Repeat this for the number of machines you have in your setup, replacing CLIENT with the IP address of the machine. **Please make sure to use >> below, and not >, as using > will overwrite any public keys that are stored**.

```
# cat /root/.ssh/id_rsa.pub | ssh root@CLIENT 'cat >> /root/.ssh/authorized_keys'
```

§

5 Running the Benchmark

5.1 Initialization the Benchmark on the master client

Run the invocation script on the master client:
Please note that the script assumes ssh without password is set between the masterclient, other clients and server systems.

```
#cd /root/redis
#./do_it.sh
```

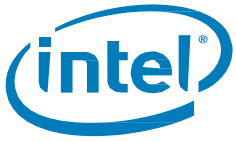
5.2 Collecting Results

Run the invocation script on the master client:

In order to make it easier to collect the results, the following script is provided (CSV format):

1. Summary2csv.sh

```
#!/bin/bash
echo "\"filename\",\"clients\",\"len\",\"servers\",\"keys\",\"max_time\",\"operation\""
for fname in `ls redis.log.*`;do
  clients=`grep key_count $fname|wc -l`
  servers=`grep key_count $fname|tail -1|awk '{print $3}'`
  len=`grep key_count $fname |tail -1 | awk '{print $9}'`
  IFS=$'\n'
  for line in `grep -B 1 "threads fin" $fname | grep "max average" `;do
        oper=`echo $line | awk '{print $5}'`
        keys=`echo $line | awk '{print $6}'`
        max_time=`echo $line | awk '{print $10}'`
        max_time=$((max_time / 1000000))
        echo "\"$fname\",\"$clients\",\"$len\",\"$servers\",\"$keys\",\"$max_time\",\"$oper\""

  done
done
```



5.3 Results obtained in the lab

The results below show the following permutations:

Using "value" size of 1,000 B and 100,000 B, running SET and then GET 3 times, measuring the transactions. The SETs are filling up the Redis store with 655 GB to use up substantially most memory of the node (768GB of memory).

Table 1: Test Results

Value-size	# keys	Servers	Operation	Native Tx Combined	Intel® Memory Drive Technology Tx Combined	% of Native	Max Time (ses)
1,000	2,275,590	288	SET	8,804,353,696	8,470,405,121	96.2%	594
1,000	2,275,590	288	GET	9,096,442,973	8,823,551,108	97.0%	585
1,000	2,275,590	288	SET	8,811,795,965	8,313,395,883	94.3%	593
1,000	2,275,590	288	GET	9,088,795,939	8,361,226,400	92.0%	586
1,000	2,275,590	288	SET	8,804,353,696	8,246,469,239	93.7%	594
1,000	2,275,590	288	GET	9,088,795,939	8,361,989,820	92.0%	586
100,000	22,743	288	SET	9,441,540,986	9,542,628,015	101.1%	553
100,000	22,743	288	GET	9,424,467,897	9,404,696,774	99.8%	557
100,000	22,743	288	SET	9,415,977,412	9,421,611,696	100.1%	556
100,000	22,743	288	GET	9,415,977,412	9,404,696,774	99.9%	556
100,000	22,743	288	SET	9,415,977,412	9,396,344,950	99.8%	556
100,000	22,743	288	GET	9,415,977,412	9,413,139,051	100.0%	556

§

6 Conclusions

We have shown that for users running Redis, in use-cases where the memory capacity per node is a limiting factor and thereby typically requiring the addition of servers, Intel® Memory Drive Technology provides an excellent alternative to adding nodes to the Redis cluster, enabling the same number of transactions per server while hosting more than 4x the data per server.

§