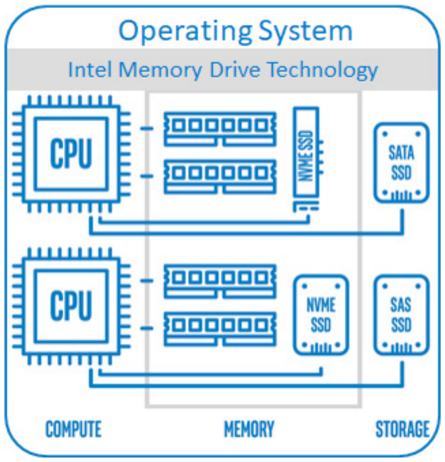
TECHNOLOGY BRIEF

Cloud
Virtualization

Intel® Optane™ DC SSDs with Intel® Memory Drive Technology: Increases Server's VM Capacity

Enabling more affordable or bigger memory solutions for data centers.



Server VM Capacity and the Need for More Memory

Running a large number of VMs on a single server (by either single or multi-tenant) is an established best practice for increasing the utilization of servers. The ability to maximize the number of VMs per server has a direct influence on the economics of an infrastructure. With the increase in core count and CPU capabilities, capacity planning in VM deployments is now more focused on memory requirements, as the system could run out of the memory resource without maximizing the computing power of the CPUs.

This storage capacity limitation has driven the use of less powerful CPUs and curb the memory of workloads running VMs. This use of insufficient amount of physical memory results in the host using Linux* Swap (even with high-end SSDs), typically producing poor performance. In most cases, customers consider this option as a last resort to prevent out-of-memory crash, and carefully plan capacity to avoid using Linux Swap.

Memory-induced Scalability Challenge for VMs Deployment

A server's VMs capacity is typically limited by the amount of memory available on that single server. Due to the high cost of DRAM, organizations seek the "sweet-spot" of $/GB; it is cost-prohibitive to purchase high-density DIMMS. To compound these difficulties, the maximum amount of memory per server is capped y modern server architecture: generally 12 DIMMS per processor socket, however, in high-density servers, the cap is as low as 6-8 DIMMS per socket.

These financial limitations are significant. The per GB cost of DRAM increases exponentially with size for high-density DIMMs. Currently, in mid-2018, DIMMs larger than 64GB are effectively cost-prohibitive, leading to a practical system memory limit of 768GB to 1.5TB for a standard dual socket server.

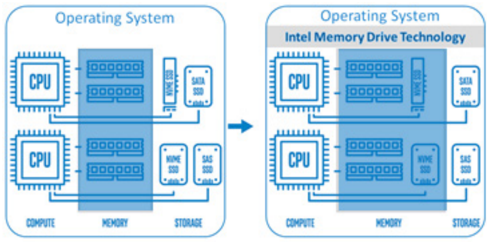
As a result, the number of server nodes required for a large-scale virtualized infrastructure, for public or private clouds, is typically determined by the amount of memory in each node, rather than by the compute capacity available with the latest processor models. This is evident in typical deployment scenarios where CPU utilization is far from maxing out, which results in the customer's choosing lower grade parts.

To support a large number of VMs, organizations are forced to use expensive high-density DIMMs, or to increase the number of nodes for a large-scale infrastructure, resulting in much higher costs for the infrastructure, as well as a larger data center footprint and subsequent cost.

Author
Frank Ober
Enterprise Architect

Intel® Memory Drive Technology Benefits and Capabilities

Increasing the total memory per node, or reducing the cost per GB of memory, can significantly improve the cost efficiency of a VM infrastructure.



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

Unlike Linux Swap, Intel® Memory Drive Technology increases memory capacity beyond DRAM limitations in a significantly greater, more cost-effective way, while delivering DRAM-like performance in a completely transparent manner to the operating system and applications.

It is a plug-in addition/upgrade to an existing server, and no changes are required to the OS, VMs or applications.

Linux* KVM + Redis Performance Benchmark–Based on Customer Use-Case

Based on a real use-case, a customer was experiencing low CPU utilization due to the limited number of virtual machines able to run on a single server. We tested the option of expanding memory with Intel® Optane™ DC SSD P4800X with Intel Memory Drive Technology to increase a server's capacity to run more VMs without a significant impact on performance, as compared to the performance drawback of Linux Swap. We used a set of VMs based on the Linux* Kernel Virtual Machine (KVM), each VM running a redis server and a redis client, performing a mixture of set and get operations.

The benchmark was first run on a bare-metal, DRAM-only system with 192GB DDR4 DRAM, to validate that it can fit no-more than 31 VMs due to the amount of memory required (~5.7GB/VM), and then re-tested with two configurations providing 4x more memory - 768GB each:

1. A system with 768GB of virtual memory and Linux Swap, using 192GB of DDR4 plus the best available Intel® Optane™ SSDs as the Linux Swap device
2. A system with 768 GB of memory using 192GB of DDR4 augmented with Intel Memory Drive Technology and Intel® Optane™ SSDs.

Workload Characteristics of Each VM (All claims based on the following configuration.[2])

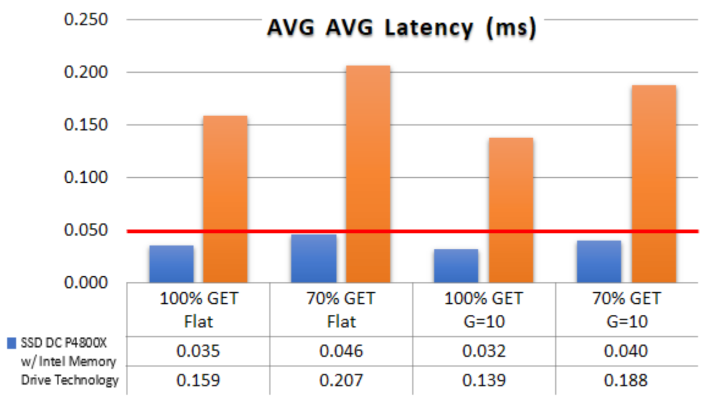
- Maximum throughput per VM was throttled to: 5,000 requests/sec (operations/sec) to simulate real use-case workload, to reach an average latency target of 50µs, and a 99% latency of 300µs (which is considered a good practice for such scenarios)
- Each Redis* server instance (one per VM) was loaded with 3.3M records of 1KB each, resulting in a memory footprint of 5.7GB for each VM
- The operations were tested with different set/get ratios as well as different random Gaussian distributions (bell curve) with different influence factors[1] (Gaussian width)
    - o 100% Random, flat distribution – influence=0: 100% GET, 70% GET
    - o Random, Gaussian distribution – influence=10: 100% GET, 70% GET
- 4 iterations were performed for each test:
    - o 1st iteration was used for warmup
    - o 3 measured iterations from which data was derived for results
- Software stack:
    - o CentOS Linux release 7.4.1708 (Core)
    - o kernel: 4.15.12-1.el7.elrepo.x86_64 (el7.x86_64)
    - o VMs: Ubuntu* 16.04.2 LTS (GNU/Linux 4.4.0-62-generic x86_64)
    - o Intel Memory Drive Technology 8.5.2401.0
- Hardware setup:
    - o Dual-socket server: 2 x Intel® Xeon®Gold 6154 CPU @ 3.00 GHz
    - o Intel Memory Drive Technology:
        - – 192GB DDR4 DRAM + Intel Memory Drive Technology: 2 x 750GB (capped to 768GB)
    - o Linux Swap:
        - – 192GB DDR4 DRAM + Linux Swap: 2 x 750GB Intel® Optane™ SSDs
        - – Intel best known methods (BKMs) for Linux Swap applied (see appendix)

Intel® Optane™ SSD DC P4800X with Intel® Memory Drive Technology Measured Performance

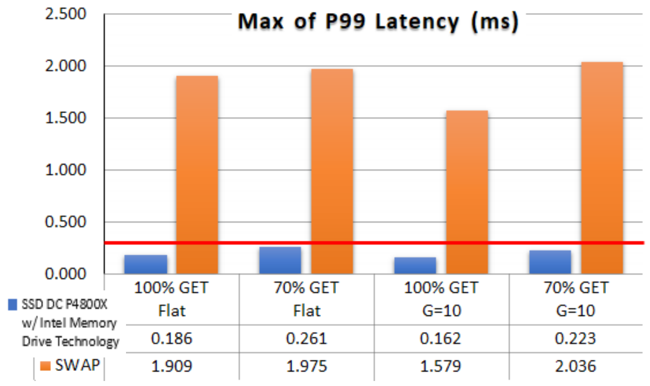
The following charts show the performance of a system running 108 VMs concurrently, and compares the two configurations previously discussed:

1. Linux Swap: A system using 192GB DDR4 DRAM + SWAP on Intel® Optane™ SSDs
2. Intel Memory Drive Technology: A system with 192GB DDR4 DRAM + Intel Optane SSD DC P4800X with Intel Memory Drive Technology to achieve a 768GB of memory in a cost-effective setup

Target average latency levels (50µs average and a 99% latency of 300µs are marked in red).



	100% GET Flat	70% GET Flat	100% GET G=10	70% GET G=10
■ SSD DC P4800X w/ Intel Memory Drive Technology	0.035	0.046	0.032	0.040
	0.159	0.207	0.139	0.188



	100% GET Flat	70% GET Flat	100% GET G=10	70% GET G=10
■ SSD DC P4800X w/ Intel Memory Drive Technology	0.186	0.261	0.162	0.223
■ SWAP	1.909	1.975	1.579	2.036

Average Latency with Intel Memory Drive Technology is below 50µs

Latency for 99% of operations is below 300µs

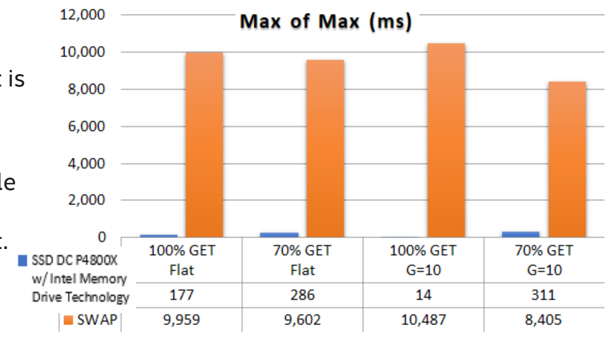
Average Latency with Intel Memory Drive Technology is 2x-4x better than Linux Swap

Latency for 99% is 7x to 10x better than Linux Swap

Overall Quality of Service

When evaluating the quality of service, check the maximum latency that is achieved from each option, indicated on the worst-case performance.

As shown on the chart to the right, the maximum latency achieved with SSD DC P4800X with Intel Memory Drive Technology is within acceptable range, while with Linux Swap we see that peak latency is 30x higher and reaches levels which are unacceptable for such a workload environment.



	100% GET Flat	70% GET Flat	100% GET G=10	70% GET G=10
■ SSD DC P4800X w/ Intel Memory Drive Technology	177	286	14	311
■ SWAP	9,959	9,602	10,487	8,405

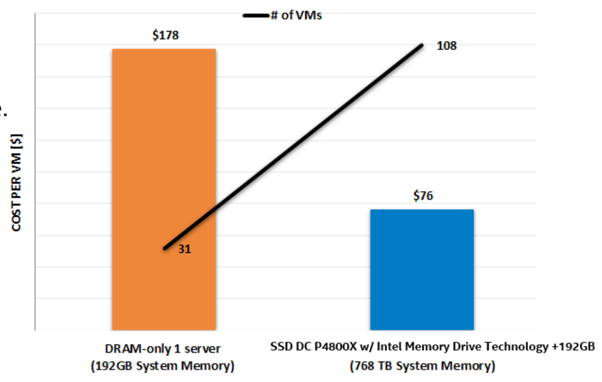
Cost Efficiency and Overall Savings

Intel Optane DC SSD with Intel Memory Drive Technology enables the cost-effective expansion of a server's memory, enabling 3.5x the number of concurrent VMs (108 vs. 31) within the required 99% latency acceptable range.

The cost chart on the right, compares the cost per VM and number of concurrent VMs of: [3]

- DRAM-only server with 192GB of memory
- A server configuration with 192GB RAM + SSD DC P4800X with Intel Memory Drive Technology to reach the same overall amount of memory (768GB)

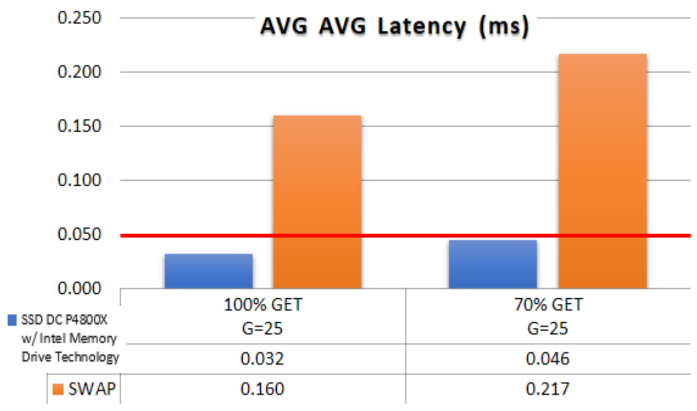
Additional cost savings are achieved as the solution requires a smaller data center footprint, energy savings, and reduced maintenance costs.



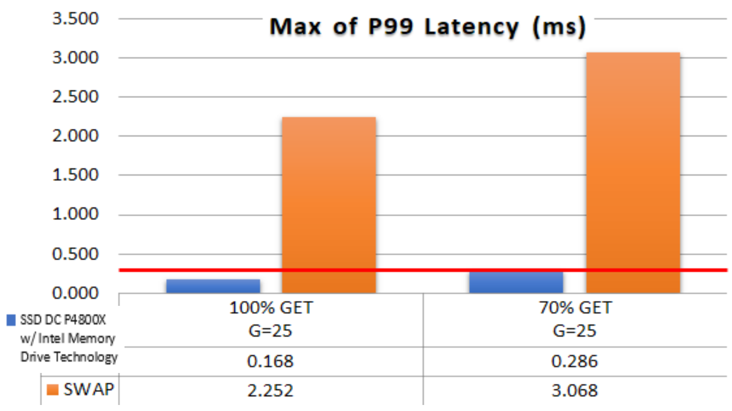
Upgrading an existing 192GB server with Intel® Optane™ DC SSD with Intel® Memory Drive Technology provides additional capacity of 350%, and reduces the cost per VM by up to 60%!

Higher Hit Ratio Allows for Greater Throughput

In addition to the scenarios above, we have tested with a Gaussian distribution influence factor of 25. This simulates a narrower bell curve (and as a result a higher hit ratio), which requires less memory bandwidth to the Intel® Optane™ DC SSDs with NVMe*. This allows the ability to run up to 126 VMs, while meeting the latency targets.



	100% GET G=25	70% GET G=25
🟦 SSD DC P4800X w/ Intel Memory Drive Technology	0.032	0.046
🟧 SWAP	0.160	0.217



	100% GET G=25	70% GET G=25
🟦 SSD DC P4800X w/ Intel Memory Drive Technology	0.168	0.286
🟧 SWAP	2.252	3.068

Intel® Optane™ DC SSD with Intel® Memory Drive Technology are a true memory replacement technology

Summary

Leveraging Intel Optane DC SSDs with Intel Memory Drive Technology provides a cost-effective way, as well as an in-place transparent upgrade, to support server nodes with up to 8x[4] more memory than server's specifications' limits (or current memory configuration), enabling a cost-effective infrastructure for VMs, with minimal impact on performance. Unlike Linux Swap, Intel Optane DC SSDs with Intel Memory Drive Technology enables all this while maintaining the commonly acceptable latency requirements.

Appendix A: System and Benchmark Installation and Configuration

Benchmark Scripts and Code

Run script

```
#!/bin/bash
if [ “_$1” == “_” ]
then
  echo “Usage: $0 <output-directory>”
  exit 1
fi
BASE=”$PWD”
OUTDIR=”$PWD/$1”
if [ -e “$OUTDIR” ]
then
  echo “Output directory already exists.”
  exit 1
fi
mkdir “$OUTDIR”
exec &> >(tee -a $OUTDIR/log.txt)
./tune.sh
dmesg > $OUTDIR/dmesg
sleep 15
cd ansible
####### VMs setup ###############################################################
vms=96
echo “Clone VMs: $vms” | tee -a $OUTDIR/free.txt
free -m >> $OUTDIR/free.txt
../vms-clone.sh $vms
echo “Spawn VMs: $vms” | tee -a $OUTDIR/free.txt
free -m >> $OUTDIR/free.txt
../vms-spawn.sh $vms
echo “Run Benchmark”
ansible -i inventory all -m ping
################################################################################
sleep 15
####### fill ###################################################################
export REDIS_OP=fill
export REDIS_VALSIZE=1024
export REDIS_CUSTOMERS=3300000
json=$OUTDIR/fill_${REDIS_CUSTOMERS}_${REDIS_VALSIZE}.`date +%y%m%d-%H%M%S`.json
echo “Fill Redis - $REDIS_CUSTOMERS x $REDIS_VALSIZE Bytes - before - `date`” | tee -a $OUTDIR/free.txt
free -m >> $OUTDIR/free.txt
ansible-playbook -i inventory provisioning/benchmark-random-bias-mixed-redis.yml > $json
echo “Fill Redis - $REDIS_CUSTOMERS x $REDIS_VALSIZE Bytes - after - `date`” | tee -a $OUTDIR/free.txt
free -m >> $OUTDIR/free.txt
################################################################################
# Make sure the results are representitive, repeat each steps this numbre of times
ITERS=3
export REDIS_OP=test
export REDIS_RANDOM_BENCHMARK_RATE=5000
####### test ###################################################################
# we must run 0 first, to neutralize the sequential data fill
for inf in 0 10 25; do
  export REDIS_RANDOM_BENCHMARK_INFLUENCE=${inf}
  for mix in 100 70; do
    export REDIS_RANDOM_BENCHMARK_MIX=${mix}
    for ((iter=0; iter<=${ITERS}; iter++)); do
      sleep 10
      if [ $iter -eq 0 ]; then
        # warmup run
        json=$OUTDIR/wrup_random_bias_mixed_${REDIS_RANDOM_BENCHMARK_INFLUENCE}_${REDIS_RANDOM_BENCHMARK_
RATE}_${REDIS_RANDOM_BENCHMARK_MIX}.`date +%y%m%d-%H%M%S`.json
        echo “Random bias mixed - ${REDIS_RANDOM_BENCHMARK_INFLUENCE} ${REDIS_RANDOM_BENCHMARK_RATE} ${REDIS_
RANDOM_BENCHMARK_MIX} - warmup - `date`” | tee -a $OUTDIR/free.txt
      else
        # testing iteration
json=$OUTDIR/test_random_bias_mixed_${REDIS_RANDOM_BENCHMARK_INFLUENCE}_${REDIS_RANDOM_BENCHMARK_RATE}_${REDIS_
RANDOM_BENCHMARK_MIX}.`date +%y%m%d-%H%M%S`.json
        echo “Random bias mixed - ${REDIS_RANDOM_BENCHMARK_INFLUENCE} ${REDIS_RANDOM_BENCHMARK_RATE} ${REDIS_
RANDOM_BENCHMARK_MIX} - ${iter}/${ITERS} - `date`” | tee -a $OUTDIR/free.txt
      fi
    free -m >> $OUTDIR/free.txt
    ansible-playbook -i inventory provisioning/benchmark-random-bias-mixed-redis.yml > $json
    ./stats $json | grep Hosts -A5 | paste - - - - - - - - -
  done
    done
done
################################################################################
echo “Done”
echo
```

vms-clone.sh

```
#!/bin/bash
vms=$1
vm_user=”intel”
vm_master=”ubuntu1604-good-backup”
virsh destroy $vm_master
virsh undefine $vm_master
for vm_name in `virsh list --name --all`; do
  echo $vm_name
  virsh destroy $vm_name
  virsh undefine $vm_name --remove-all-storage
done
vms=$((vms-1))
virsh create /home/swapstream/$vm_master.xml
virsh suspend $vm_master
for i in `seq 0 $vms`; do
  echo $i
  vm_name=”ubuntu1604-$i”
  virt-clone -o $vm_master -n $vm_name --auto-clone
done
virsh destroy $vm_master
virsh undefine $vm_master
```

vms-spawn.sh

```
#!/bin/bash
vms=$1
vm_user=”intel”
vms=$((vms-1))
for i in `seq 0 $vms`; do
  echo $i
  vm_name=”ubuntu1604-$i”
  virsh start $vm_name
  sleep 1
done
echo “Sleeping”
sleep 15
../get_ips.sh $((vms+1))
echo “Sleeping”
sleep 15
```

get_ips.sh

```
#!/bin/bash
vms=$1
vm_user=”intel”
vms=$((vms-1))
while [ 1 ] ; do
  echo “Getting IPs”
  rm inventory
  for i in `seq 0 $vms`;
  do
    vm_name=”ubuntu1604-$i”
    ip=$(virsh domifaddr “$vm_name”|tail -2 | awk ‘{print $4}’| cut -d/ -f1)
    echo “$ip ansible_user=$vm_user” >> inventory
  done
  if [ `grep -c 192.168 inventory` -gt $vms ] ; then
    break;
  fi
  echo “Waiting for all VM IP addresses...”
  sleep 10
done
cat inventory
```

tune.sh

```
#!/bin/bash
echo “Set scaling governor”
for CPUFREQ in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor; do [ -f $CPUFREQ ] || continue; echo -n
performance > $CPUFREQ; done
echo “Show scaling governor”
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
echo
for fname in `echo -n /usr/local/bin/????version | sed ‘s/^\(.*\)\(\/usr\/local\/bin\/vsmpversion\)\(.*\)$/\1\3
\2/’`; do
  strings $fname 2>&1 | grep -iq “vsmpversion”
  if [ $? == 0 ]; then UTILPX=`basename $fname | awk ‘{print substr($0,0,4)}’`; break; fi
done
echo “Running environment”
NATIVE=0
${UTILPX}version --long
[ $? -ne 0 ] && NATIVE=1
echo
echo “Remove all swap devices”
swapoff -a
echo
if [ $NATIVE -eq 1 ] ; then
  if [ `free -g | grep Mem: | awk ‘{print $2}’` -le 500 ]; then
    ########### SWAP-SPECIFIC SETTINGS #####
    # Specify the NVME name-space to use.  Do not use /dev and
    # partitions will be created automatically per name-space.
    # WARNING: SET THIS CAREFULLY AS IT CAN DESTROY THE OS
    swapdevlist=”nvme0n1 nvme1n1”
    # For all values below USE -1 FOR DEFAULT
    partitions_per_device=-1
    cluster=-1
    watermark_scale_factor=400
    max_sectors_kb=-1
    io_queue_depth=-1
    nomerges=-1
    ########################################
    echo “Setting Optane as swap on $swapdevlist with $partitions_per_device per device”
    if [ ! -x /usr/sbin/partprobe -o ! -x /usr/sbin/parted ]; then
      echo “please install partprobe and parted”
      exit
    fi
    for dev in $swapdevlist; do echo $dev; dd if=/dev/zero of=/dev/${dev} bs=1M count=10 oflag=direct; done 2>&1
    /usr/sbin/partprobe; sleep 1; /usr/sbin/partprobe; sleep 1
    if [ $partitions_per_device -gt 1 ]; then
      pchunk=$((100/partitions_per_device))
      for dev in $swapdevlist; do
        /usr/sbin/parted /dev/${dev} mklabel msdos
        /usr/sbin/parted -a none /dev/${dev} mkpart extended 0 100%
        for p in `seq 1 $partitions_per_device`; do
          echo /dev/${dev} $(((p-1)*pchunk))% $((p*pchunk))%
          /usr/sbin/parted -a none /dev/${dev} mkpart logical $(((p-1)*pchunk))% $((p*pchunk))%
        done
      done 2>&1
      /usr/sbin/partprobe; sleep 1; /usr/sbin/partprobe; sleep 1
      for dev in $swapdevlist; do for part in /dev/${dev}p*; do
        if [[ $part =~ p1$ ]]; then echo “==== skipping ext partition $part”; continue; fi
        /usr/sbin/mkswap -f $part; /usr/sbin/swapon -p 0 $part
      done; done 2>&1
    else
      for dev in $swapdevlist; do /usr/sbin/mkswap -f $dev; /usr/sbin/swapon -p 0 $dev; done 2>&1
    fi
    swapon -s
    echo
    echo “Swap tunning”
    grep -H ^ /proc/sys/vm/page-cluster
    if [ $cluster -ge 0 ]; then
      echo $cluster > /proc/sys/vm/page-cluster
      grep -H ^ /proc/sys/vm/page-cluster
    fi
    grep -H ^ /proc/sys/vm/watermark_scale_factor
    if [ $watermark_scale_factor -ge 0 ]; then
      echo $watermark_scale_factor > /proc/sys/vm/watermark_scale_factor
      grep -H ^ /proc/sys/vm/watermark_scale_factor
    fi
    grep -H ^ /sys/block/nvme*/queue/max_sectors_kb
    if [ $max_sectors_kb -ge 0 ]; then
      for dev in $swapdevlist; do echo $max_sectors_kb > /sys/block/${dev}/queue/max_sectors_kb; done
      grep -H ^ /sys/block/nvme*/queue/max_sectors_kb
    fi
    grep -H ^ /sys/module/nvme/parameters/io_queue_depth
    if [ $io_queue_depth -ge 0 ]; then
      echo $io_queue_depth > /sys/module/nvme/parameters/io_queue_depth
      grep -H ^ /sys/module/nvme/parameters/io_queue_depth
    fi
```

```
        grep -H ^ /sys/block/nvme*/queue/nomerges
        if [ $nomerges -ge 0 ]; then
          for dev in $swapdevlist; do echo $nomerges > /sys/block/${dev}/queue/nomerges; done
          grep -H ^ /sys/block/nvme*/queue/nomerges
        fi
        echo
      fi
    fi
    echo “Disable hugepages”
    echo ‘never’ > /sys/kernel/mm/transparent_hugepage/enabled
    echo ‘never’ > /sys/kernel/mm/transparent_hugepage/defrag
    echo
    echo “Show NUMA Topology”
    numactl -H
    echo
    echo “Show Memory usage”
    free -g
    echo
    echo “Show /proc/meminfo”
    cat /proc/meminfo
    echo
    # Lastly, stop ksmtuned - as it is not effective with random data.
    echo “Stop ksmtuned and ksm”
    systemctl stop ksmtuned.service
    echo 0 > /sys/kernel/mm/ksm/run
    # If KSM is to be used, we have two options:
    # 1) Start ksm manually.  This is done to make sure KSM
    #    is running in both SWAP and IMDT, unrelated to memory size.
    # 2) Change /etc/ksmtuned.conf from KSM_THRES_COEF=20 to
    #    KSM_THRES_COEF=100-(DRAM/SYSTEM_MEMORY)*(100-20)
    #echo “Start ksm”
    #echo 1 > /sys/kernel/mm/ksm/run
    echo
    echo “Show kernel mm tuning”
    grep -r ^ /sys/kernel/mm/
```

ansible/provisioning/benchmark-random-bias-mixed-redis.yml

```
      - hosts: all
        become: true
        tasks:
          - name: Make sure we can connect
            ping:
          - name: Remove swap partition
            command: swapoff -a
          - name: copy efficient to VM
            copy:
              src: /root/newswapstream/solotest
              dest: /root/intel-bench
              mode: 0755
          - name: Benchmark Redis
            command: ./solotest $REDIS_OP 6379 $REDIS_VALSIZE $REDIS_CUSTOMERS $REDIS_RANDOM_BENCHMARK_INFLUENCE
      $REDIS_RANDOM_BENCHMARK_RATE $REDIS_RANDOM_BENCHMARK_MIX
            register: benchmark
            args:
              chdir: /root/intel-bench
            environment:
              REDIS_OP: “{{ lookup(‘env’,’REDIS_OP’) }}”
              REDIS_VALSIZE: “{{ lookup(‘env’,’REDIS_VALSIZE’) }}”
              REDIS_CUSTOMERS: “{{ lookup(‘env’,’REDIS_CUSTOMERS’) }}”
              REDIS_RANDOM_BENCHMARK_INFLUENCE: “{{ lookup(‘env’,’REDIS_RANDOM_BENCHMARK_INFLUENCE’) }}”
              REDIS_RANDOM_BENCHMARK_RATE: “{{ lookup(‘env’,’REDIS_RANDOM_BENCHMARK_RATE’) }}”
              REDIS_RANDOM_BENCHMARK_MIX: “{{ lookup(‘env’,’REDIS_RANDOM_BENCHMARK_MIX’) }}”
          - name: Output benchmark results
            debug: msg=”{{benchmark.stdout}}”
```

ansible/stats

```
#!/usr/bin/env node
var fs = require('fs')
var file = process.argv[2];
var json = fs.readFileSync(file).toString("utf-8");
if(json[0] != "{") {
  var lines = json.split("\n");
  lines.shift()
  json = lines.join("\n");
}
var contents = JSON.parse(json)
contents.plays.forEach(function(play){
  var tasks = play.tasks
  var benchmarkResults = tasks.filter(function(task){
    return task.task.name == "Benchmark Redis"
  })
  var hosts = benchmarkResults[0].hosts
  var totalHosts = 0;
  var totalAvg = 0, totalP95 = 0, totalP99 = 0, totalMin = 99999999, totalMax = 0;
  for(var host in hosts) {
    var results = hosts[host]
    if(totalHosts == 0) {
      console.log()
    }
    if(results.stdout == undefined) continue;
    ++totalHosts;
    console.log(results.stdout)
    var avg = parseFloat(results.stdout.split("\n")[1]) * 1000
    var min = parseFloat(results.stdout.split("\n")[2]) * 1000
    var max = parseFloat(results.stdout.split("\n")[3]) * 1000
    var P99 = parseFloat(results.stdout.split("\n")[4]) * 1000
    var P95 = parseFloat(results.stdout.split("\n")[5]) * 1000
    if(isNaN(avg)) {
      --totalHosts
    } else {
      totalAvg+=avg
      totalP95+=P95
      totalP99+=P99
      if (min < totalMin) {
        totalMin = min
      }
      if (max > totalMax) {
        totalMax = max
      }
    }
  }
  console.log()
  var avgAvg = totalAvg / totalHosts
  var avgP95 = totalP95 / totalHosts
  var avgP99 = totalP99 / totalHosts
  console.log("Hosts:", totalHosts)
  console.log("Avg:", avgAvg.toFixed(3))
  console.log("Min:", totalMin.toFixed(3))
  console.log("P95:", avgP95.toFixed(3))
  console.log("P99:", avgP99.toFixed(3))
  console.log("Max:", totalMax.toFixed(3))
})
```

Redis client code:

To compile:

```
gcc -O2 -o solotest solotest.c -I redis-4.0.2/deps/hiredis -L redis-4.0.2/deps/hiredis -lhiredis -lm
```

solotest.c

```
#define _GNU_SOURCE
#include <math.h>
#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 <sys/sysinfo.h>
// DEFAULTS:
#define DEF_PORT 6379
#define DEF_SIZE 1000
#define DEF_COUNT 20000000
#define DEF_BIAS (-1)
#define DEF_FREQ 1000000
#define DEF_READ 100
#define MAX_HISTOGRAM 1000000 // 1 sec
char *value;
char hostname[256] = { “127.0.0.1” };
int port = DEF_PORT;
int val_size = DEF_SIZE;
long key_count = DEF_COUNT;
long bias = DEF_BIAS;
long frequency = DEF_FREQ;
long readmix = DEF_READ;
unsigned int seed;
long *histogram;
int fill = 0;
int test = 0;
inline double gauss(double x, double D)
{
  double a = 1;
  double b = 50;
  double c = D;
  return a * exp(-(x - b) * (x - b) / (2 * c * c));
}
void select_mode(char *modestr)
{
  fill = strcasestr(modestr, “fill”) == NULL ? 0 : 1;
  test = strcasestr(modestr, “test”) == NULL ? 0 : 1;
}
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 get_seed()
{
  struct sysinfo s_info;
  int error = sysinfo(&s_info);
  if(error != 0)
    printf(“code error = %d\n”, error);
  return s_info.uptime * timestamp();
}
char *prep_value(char *value, int val_size)
{
  int i;
  for (i = 0; i < val_size / 4; i += 1)
    *((int *)value + i) = (rand_r(&seed) | 0x01010101);
  value[val_size] = ‘\0’;
}
int main(int argc, char *argv[])
{
  // Arguments: HOST PORT SIZE COUNT BIAS FREQ MIX
  if (argc > 1) select_mode(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]);
  if (argc > 5) bias = atol(argv[5]);
  if (argc > 6) frequency = atol(argv[6]);
```

```
    if (argc > 7) readmix = atol(argv[7]);
    fprintf(stderr, "%s: fill %d test %d port %d value_len %d key_count %ld bias %ld frequency %ld readmix %ld\n",
  datestr(), fill, test, port, val_size, key_count, bias, frequency, readmix);
    sleep (5);  // hold on for a while, so I can start on all VMs.
    seed = get_seed();
    redisContext *con;
    redisReply *reply;
    struct timeval timeout = { 1, 500000 }; // 1.5 seconds
    long i;
    char *value = (char *)malloc(((int)((val_size + 3) / 4)) * 4 + 1);
    long *histogram = calloc(MAX_HISTOGRAM + 1, sizeof(long));
    if (NULL == value) {
      fprintf(stderr, "Out Of Memory\n");
      exit(1);
    }
    fprintf(stderr, "%s : Value(s) allocated, connecting\n", datestr());
    con = redisConnectWithTimeout(hostname, port, timeout);
    if (NULL == con) {
      fprintf(stderr, "port %s:%d connection error: can't allocate redis context\n", hostname, port);
      exit(1);
    }
    if (con->err != 0) {
      fprintf(stderr, "port %s:%d connection error: %s\n", hostname, port, con->errstr);
      redisFree(con);
      exit(1);
    }
    fprintf(stderr, "port %s:%d connected\n", hostname, port);
    long start, total_elapse;
    if (fill) {
      fprintf(stderr, "%s: port %s:%d flushall - starting\n", datestr(), hostname, port);
      reply = redisCommand(con, "flushall");
      fprintf(stderr, "%s: port %s:%d flushall - %s\n", datestr(), hostname, port, reply->str);
      freeReplyObject(reply);
      fprintf(stderr, "%s: port %s:%d set - starting\n", datestr(), hostname, port);
      start = timestamp();
      for (i = 0; i < key_count; i++) {
        prep_value(value, val_size);
        reply = redisCommand(con, "SET %ld %s", i, value);
        if (strcmp(reply->str, "OK") != 0)
          fprintf(stderr, "port %s:%d SET %ld %s reply error! reply:%s\n", hostname, port, i, "<VALUE>", reply-
  >str);
        freeReplyObject(reply);
      }
      total_elapse = timestamp() - start;
      fprintf(stderr, "%s: port %s:%d set - %d keys, elapse %ld usecs\n", datestr(), hostname, port, key_count,
  total_elapse);
    }
    int set = 0;
    long max = 0, min = 99999999, total = 0, singlestart, elapsed, now;
    if (test) {
      fprintf(stderr, "%s: port %s:%d test - starting\n", datestr(), hostname, port);
      start = timestamp();
      for (i = 0; i < key_count; i++) {
        double random = (double)rand_r(&seed) / (((double)RAND_MAX) + 1.0);
        int k = (int)(random * (double)key_count);
        if (bias) {
          random = (double)rand_r(&seed) / (double)RAND_MAX;
          int influence = (int)(random * 100.0);
          k = (k > key_count / 2) ?
            (k + (int)(gauss(influence, bias) * (key_count / 2 - k))) :
            (k - (int)(gauss(influence, bias) * (k - key_count / 2)));
        }
        if (readmix < 100) {
          random = (double)rand_r(&seed) / (double)RAND_MAX;
          set = ((int)(random * 100.0) > readmix) ? 1 : 0;
        }
        if (set) {
          prep_value(value, val_size);
          singlestart = timestamp();
          reply = redisCommand(con, "SET %ld %s", k, value);
          elapsed = timestamp() - singlestart;
          if (strcmp(reply->str, "OK") != 0)
            fprintf(stderr, "port %s:%d SET %ld=%ld %s reply error! reply:%s\n", hostname, port, i, k, "<VAL-
  UE>", reply->str);
        } else {
          singlestart = timestamp();
          reply = redisCommand(con, "GET %ld", k);
          elapsed = timestamp() - singlestart;
          if (NULL == reply)
            fprintf(stderr, "port %s:%d GET %ld=%ld reply is NULL!\n", hostname, port, i, k);
        }
        freeReplyObject(reply);
        total += elapsed;
        histogram[elapsed > MAX_HISTOGRAM ? MAX_HISTOGRAM : elapsed]++;
```

```
          if (elapsed > max) {
            max = elapsed;
            fprintf(stderr, “port %s:%d test %s key %8ld (%d) - max time %6ld usecs\n”, hostname, port, set ?
    “SET” : “GET”, i, k, max);
          }
          if (elapsed < min) {
            min = elapsed;
            fprintf(stderr, “port %s:%d test %s key %8ld (%d) - min time %6ld usecs\n”, hostname, port, set ?
    “SET” : “GET”, i, k, min);
          }
          now = timestamp();
          elapsed = i * 1000000 / frequency - (now - start);  // Better throttle
          if (elapsed > 0) usleep(elapsed);
        }
        total_elapse = timestamp() - start;
        long pct = 0, pct95 = 0, pct99 = 0;
        for (i = 0; i <= MAX_HISTOGRAM; i++) {
          pct += histogram[i];
          if (!pct95 && (pct >= ((key_count * 95) / 100))) pct95 = i;
          if (!pct99 && (pct >= ((key_count * 99) / 100))) pct99 = i;
        }
        fprintf(stderr, “%s: port %s:%d test - %d keys, readmix %d%%, elapse %ld usecs, min %ld, max %ld, P95 %ld,
    P99 %ld\n”, datestr(), hostname, port, key_count, readmix, total_elapse, min, max, pct95, pct99);
        printf(“%ld\n%e\n%e\n%e\n%e\n%e\n”, key_count, (double)total / (double)key_count / 1000000, (double)min /
    1000000, (double)max / 1000000, (double)pct99 / 1000000, (double)pct95 / 1000000);
      }
      redisFree(con);
      free(value);
      fprintf(stderr, “%s : Complete.\n”, datestr());
```

Appendix B: Tuning

- VMs:
  - o Minimize latency by adding "**nohz=off highres=off lapic=notscdeadline**" to kernel (guest) command line
- Host:
  - o Scaling governor: performance
  - o Disable hugepages
  - o Disable KSM (random benchmark)
- Linux Swap:
  - o Equal priority devices
  - o Set watermark_scale_factor to 4%
  - o Increasing number of partitions does not improve performance

Appendix C: Intel® Best Practices for use of Linux* Swap

1. It is highly recommended to have at least one Intel® Optane™ SSD DC P4800X for each CPU socket in the system, with all SSDs divided equally across all sockets. Consult your system manual for PCIe to socket mapping.

   For example, in a dual socket system, two 375GB DC P4800X drives attached to each CPU socket would provide better performance than a single 750GB DC P4800X drive attached to one of the sockets.

   Avoid connecting an Intel® Optane™ SSD DC P4800X to a PCIe slot associated with a PCH.

2. Determine the device name(s) of your SSD(s).

   ```
   lsblk --output NAME,MODEL,SIZE,REV,VENDOR
   ```

   The device name of an NVMe SSD like the Intel® Optane™ SSD DC P4800X should be similar to `nvme0n1`. Use this name wherever the text `[devicename]` appears.

   Ensure that the block device(s) represents your Intel® Optane™ SSD DC P4800X(s) and not, for example, a data drive or your system boot drive.

3. For each Intel® Optane™ SSD DC P4800X:

   Note: These settings are not retained on reboot.

   Turn off all existing swap partitions.

   - `swapoff -a`

   Wipe all filesystems on the SSD.

   - **WARNING:** This will delete all data on the SSD.
   - `wipefs -a /dev/[devicename]`

   Make a swap area on the SSD.

   - `mkswap /dev/[devicename]`

   Enable the swap area with priority 10.

   - `swapon -p 10 /dev/[devicename]`
   - It is highly recommended that all DC P4800X swap devices use the same swap priority level, so that swap accesses are equally distributed across both devices.

When extending main memory using the Intel® Optane™ SSD DC P4800X and Linux Swap, it is highly recommended to set all CPUs in the system as No-Callbacks (No-CBs) CPUs. A No-CBs CPU is a CPU whose RCU callbacks are allowed to run on CPUs other than the CPU making the callback, which improves the page access quality of service (QoS) of the workload CPU but may lower average page access throughput.

Read-Copy-Update (RCU) is a Linux Kernel locking mechanism for shared memory. When a shared object in memory is updated, a new object is created in memory and all pointers to the old object are set to point to the new object. Old objects then need to be periodically cleaned out (garbage collected).

The RCU callback is an implementation of garbage collection. On a default Linux kernel, RCU callbacks are run in softirq context on the same CPU that performed the callback. This can cause jitter and poor page access QoS when RCU callbacks are run on workload CPUs, especially when the workload makes heavy use of swap space. CPUs which are set to be "No Callbacks

CPUs," or "No-CBs CPUs," are allowed to offload their RCU callbacks to threads which can then run on other (non-workload) CPUs. This allows for better page access QoS on the workload CPU.

1. Confirm that your existing Linux kernel configuration supports No-CBs CPUs.

   No-CBs CPU support was introduced in Linux Kernel 3.10, but not all Linux distribution kernels enable No-CBs CPU support.

   CentOS 7.3 supports No-CBs CPUs – skip to Step 4.

   Ubuntu 16.04.1 LTS does not support No-CBs CPUs – see Step 3.

   For all other distributions, check for the `CONFIG _ RCU _ NOCB _ CPU` config option in your kernel's config file.

   - `grep "CONFIG _ RCU _ NOCB _ CPU" /boot/config-`uname -r``

   If the command returns `CONFIG _ RCU _ NOCB _ CPU=y`, your kernel supports No-CBs CPUs. Skip to Step 4.

   - If `CONFIG _ RCU _ NOCB _ CPU _ ALL=y`, your kernel is already set to have all CPUs be No-CBs CPUs, and you do not need to follow this guide.

   Otherwise, if the command returns `# CONFIG _ RCU _ NOCB _ CPU` is not set or does not return any text, your kernel does not support No-CBs CPUs. See Step 3.

2. If your kernel does not support No-CBs CPUs, you will need to rebuild your kernel with enabled No-CBs support.

– Obtain the kernel source of your distribution, run make `oldconfig`, then edit the `.config` file and add the following lines:

   - `CONFIG _ RCU _ NOCB _ CPU=y`
   - `CONFIG _ RCU _ NOCB _ CPU _ NONE=y`

– Build and install the kernel.

– Set the `rcu _ nocbs` kernel boot parameter in the GRUB config.

   `rcu _ nocbs` specifies which CPUs are to be No-CBs CPUs at boot time.

   On a Red Hat/CentOS machine:

   o Use grubby to add `rcu _ nocbs=0-X` (where `X` is the highest CPU core number in the system) to the kernel boot arguments list.

   o For example, on a 32-core machine:

   ```
   grubby --update-kernel=ALL --args=rcu _ nocbs=0-31
   ```

   On an Ubuntu machine:

   o Edit `/etc/default/grub and append rcu _ nocbs=0-X` (where X is the highest CPU core number in the system) to `GRUB _ CMDLINE _ LINUX _ DEFAULT`.

   o For example, on a 32-core machine:

   ```
   GRUB _ CMDLINE _ LINUX _ DEFAULT="rcu _ nocbs=0-31"
   ```

   o Run `update-grub`, then reboot your machine.

`rcu _ nocbs` is a tunable boot parameter which can also specify only specific CPUs to be No-CBs CPUs, instead of all CPUs. Additionally, the RCU callback threads may be affinitized to specific CPUs. Such tweaks are beyond the scope of this guide.

The following configuration changes may further improve paging performance, but are not appropriate for all systems or workloads.

- If benchmarking, follow the steps in "System Tuning for Optimum Performance."
- Disable page clustering

  By default, when reading a page from swap, multiple consecutive pages are read into DRAM at once. On Intel® Optane™ Technology, page clustering may incur a latency penalty overhead when reading multiple random individual pages.

  ```
  echo '0' > /proc/sys/vm/page-cluster
  ```

- Disable transparent hugepages

This will remove the overhead of coalescing pages into hugepages and then breaking them up in swap.

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

- Disable NUMA balancing.

  Both swap and NUMA balancing frequently access LRU lists, which maintain the hot/cold pages in the system. Disabling NUMA balancing prevents it from contending with swap over the LRU lists, improving performance.

  ```
  echo '0' > /proc/sys/kernel/numa_balancing
  ```

- (Kernel 4.6 and later) Increase the watermark_scale_factor

  Watermark_scale_factor is a tunable variable which affects the memory thresholds at which the swap daemon wakes up or sleeps. We want kswapd to wake up earlier, so we make it wake up when there is 4% of available memory left, instead of the default 0.1% of available memory.

  ```
  echo '400' > /proc/sys/vm/watermark_scale_factor
  ```

- (Kernel 4.11 or later) Recompile the kernel with multi-queue deadline I/O scheduler support.

  This scheduler prevents write I/O request merges from blocking read requests for a long time.

  In the kernel .config, set `CONFIG_MQ_IOSCHED_DEADLINE`, then recompile the kernel. Use a short queue depth of 64 and small max_sectors_kb to 32 prevent write from blocking read in I/O for swap for better swap in latency.

[1] See appendix for solotest.c of Gaussian influence implementation.

[2] The benchmark results may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utlized in the testing and may not be applicable to any particular user's components, computer system or workloads. The results are no necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations. Implementation details: System BIOS: 00.01.0013 (Link); Kernel 4.15.12; Mitigation was validated for variants 1 through 3 using a checker script (Link – accessed June 21, 2018).

[3] Source – Intel: MSRP for Intel® Memory Drive Technology license with Intel® Optane™ SSD (April 2018). Trendforce* - Server DRAM Price Report Sep 2017 (Market Price)

[4] Source - Intel Memory Drive Technology Set Up and Configuration Guide, page 43. https://nsgresources.intel.com/asset-library/intel-memory-drive-technology-set-up-and-configuration-guide/