

Configuration Used:

CPU	4 cores
Memory	512 MB
Port	Port 2222 on host mapped to port 22 virtual

Qemu command with above configuration:

```
qemu-system-aarch64 \
    -monitor stdio \
    -M virt,highmem=off \
    -accel hvf \
    -cpu host \
    -smp 4 \
    -m 512 \
    -bios QEMU_EFI.fd \
    -device virtio-gpu-pci \
    -display default,show-cursor=on \
    -device qemu-xhci \
    -device usb-kbd \
    -device usb-tablet \
    -device intel-hda \
    -device hda-duplex \
    -drive
    file=ubuntu-latest.raw,format=raw,if=virtio,cache=writethrough
```

Docker run command with above configuration

```
docker run -d --name ubuntu_container -p 2222:22 -m 512m --cpus=4
ubuntu
```

Steps to ENable Docker and QEMU

QEMU VM Steps:

1. Run the following command to install QEMU
`brew install qemu`
2. Download Ubuntu-server iso file
3. Run the following command to create an image
`qemu-img create -f raw ~/qemu/ubuntu-latest.raw 40G`

4. Download EDK2 UEFI image for QEMU and decompress it.
5. RUN the following command to boot from the image to install Ubuntu

```
qemu-system-aarch64 \
  -monitor stdio \
  -M virt,highmem=off \
  -accel hvf \
  -cpu host \
  -smp 4 \
  -m 3000 \
  -bios QEMU_EFI.fd \
  -device virtio-gpu-pci \
  -display default,show-cursor=on \
  -device qemu-xhci \
  -device usb-kbd \
  -device usb-tablet \
  -device intel-hda \
  -device hda-duplex \
  -drive
file=ubuntu-latest.raw,format=raw,if=virtio,cache=writethrough
\
  -cdrom ubuntu-22.04.1-live-server-arm64.iso
```

6. Follow the on screen process to install ubuntu
7. We can directly boot from the drive file. We do not have to boot from the iso from now on. The command to boot is:

```
qemu-system-aarch64 \
  -monitor stdio \
  -M virt,highmem=off \
  -accel hvf \
  -cpu host \
  -smp 4 \
  -m 512 \
  -device virtio-mouse-pci \
  -bios QEMU_EFI.fd \
  -device virtio-gpu-pci \
  -display default,show-cursor=on \
  -device qemu-xhci \
  -device usb-kbd \
  -device usb-tablet \
  -device intel-hda \
  -device hda-duplex \
  -drive
file=ubuntu-latest.raw,format=raw,if=virtio,cache=writethrough
```

Docker Container Steps:

1. Created a directory called ubuntu-image using command

```
mkdir ubuntu-image
```

2. Created a Dockerfile using vim directly from the terminal and built an image from the file using the following steps.

- a. Run command to create a Dockerfile and open in vim editor:

```
touch Dockerfile
```

```
Vim Dockerfile
```

- b. Added the following to the dockerfile to build the image, install sysbench and run scripts. Added a trail command to keep the container alive after running scripts.

```
FROM scratch
```

```
ADD ubuntu-focal-oci-arm64-root.tar.gz /
```

```
RUN apt-get update
```

```
RUN apt-get install -y sysbench
```

```
RUN apt-get install -y linux-tools-common  
linux-tools-generic
```

```
COPY experiment1.sh /root/experiment1.sh
```

```
COPY experiment2.sh /root/experiment2.sh
```

```
CMD /bin/bash /root/experiment1.sh && /bin/bash  
/root/experiment2.sh && tail -f /dev/null
```

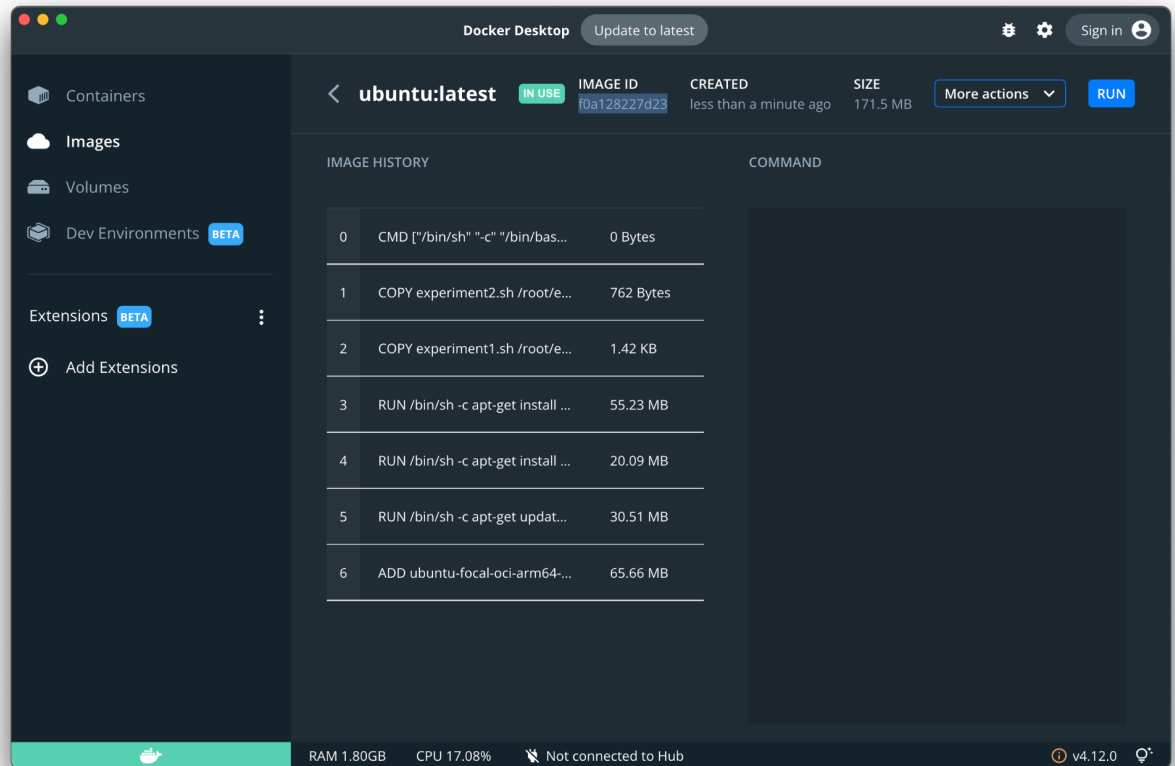
- c. Ran the following command to build image with tags:

```
docker build -t ubuntu .
```

3. Ran container with following command

```
docker run -d --name ubuntu_container -p 2222:22 -m 512m  
--cpus=4 ubuntu
```

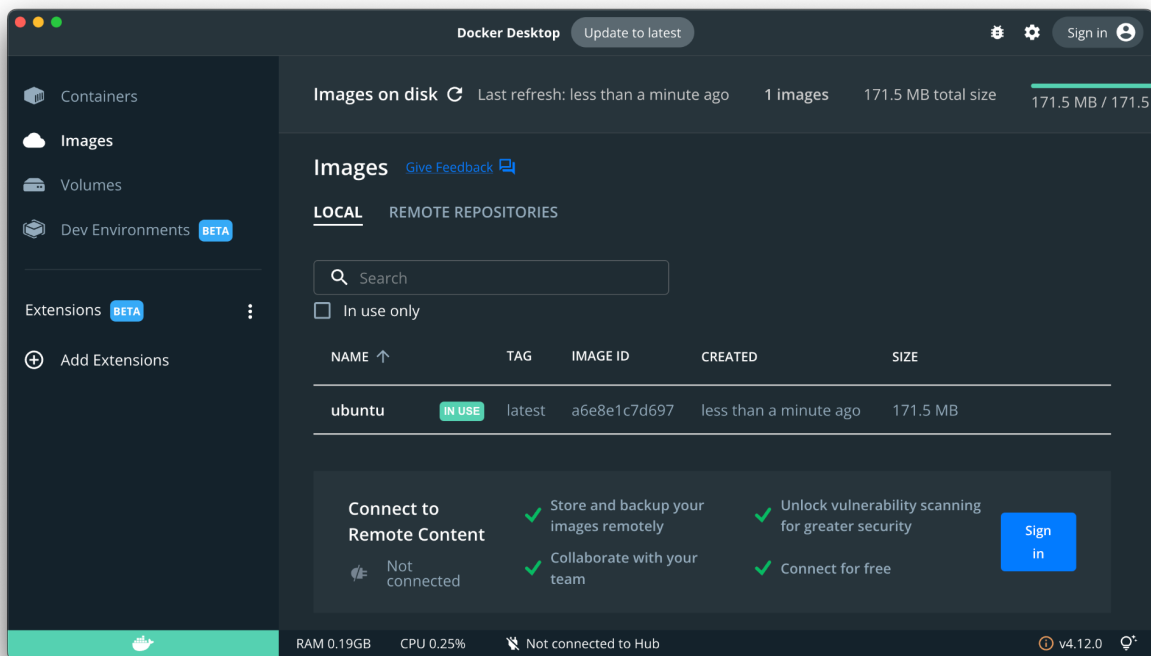
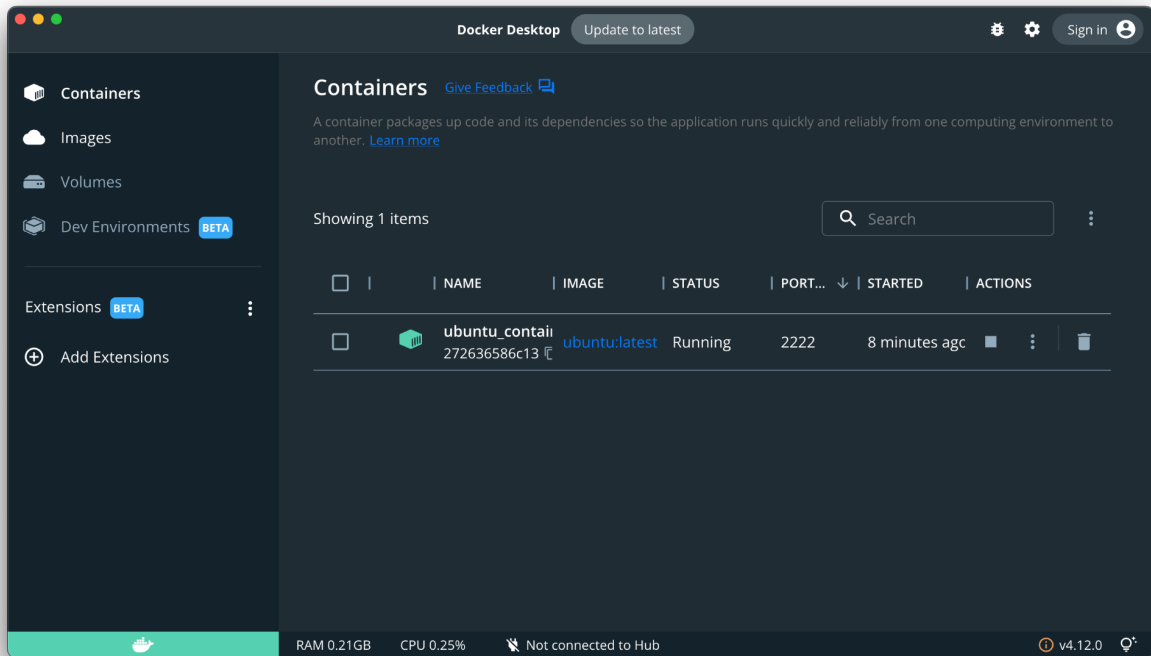
4. Additionally, docker desktop dashboard was used to visualize and manage containers. Also used the logs page and terminal in the dashboard to read benchmark results.
5. The resultant image ID is: f0a128227d23. The image history is:

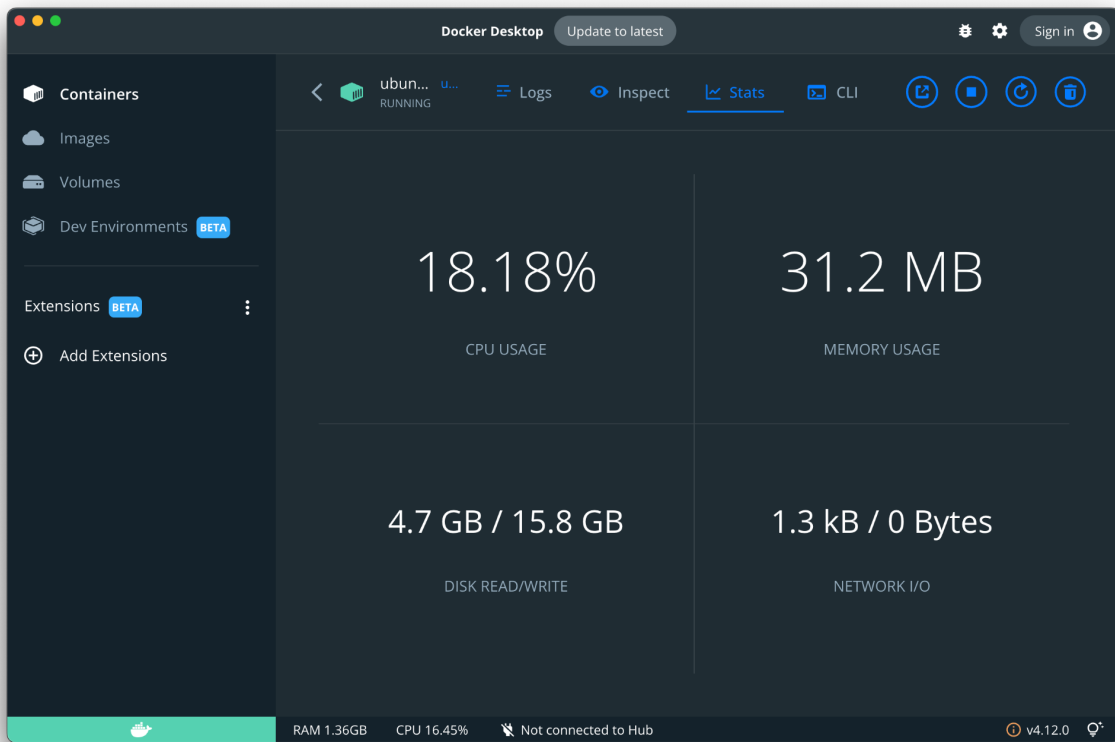


Proof of Experiment:

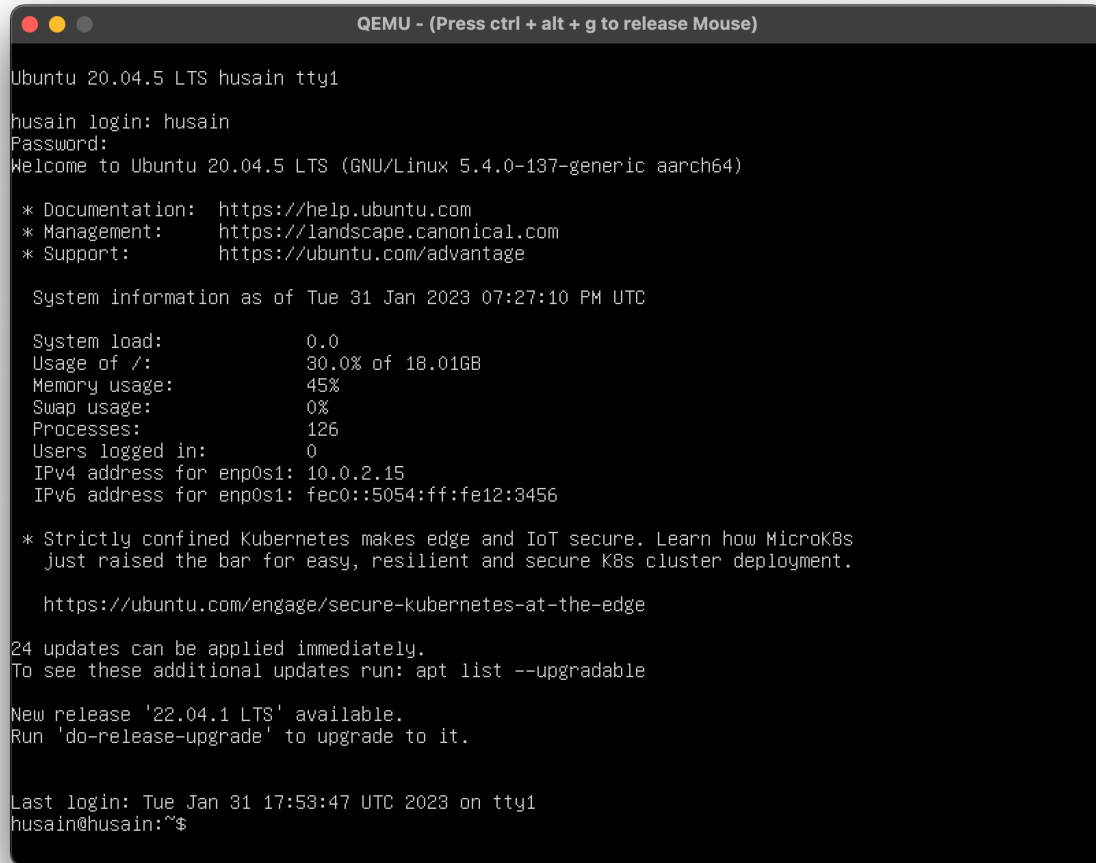
Images attached to show proof of experiment

Docker Proof:





QEMU Proof:



```
QEMU - (Press ctrl + alt + g to release Mouse)

Ubuntu 20.04.5 LTS husain tty1

husain login: husain
Password:
Welcome to Ubuntu 20.04.5 LTS (GNU/Linux 5.4.0-137-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue 31 Jan 2023 07:27:10 PM UTC

System load:          0.0
Usage of /:           30.0% of 18.01GB
Memory usage:         45%
Swap usage:           0%
Processes:            126
Users logged in:      0
IPv4 address for enp0s1: 10.0.2.15
IPv6 address for enp0s1: fec0::5054:ff:fe12:3456

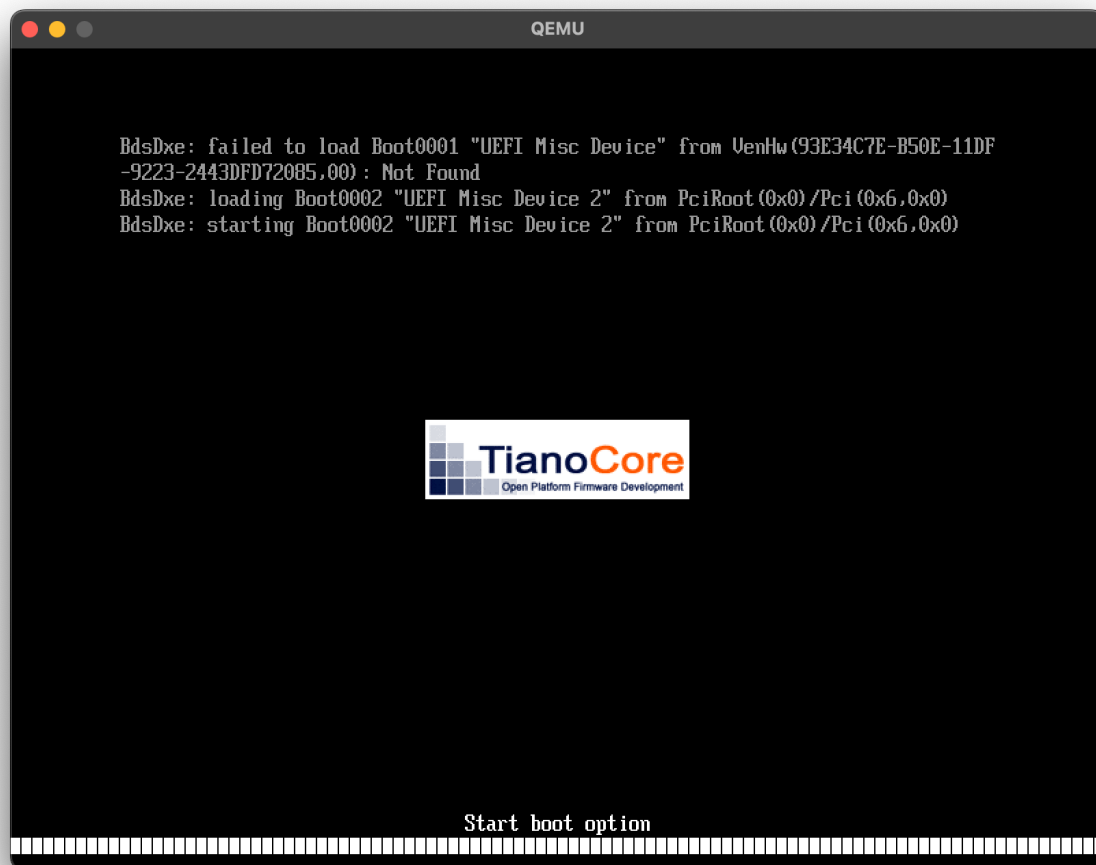
 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

24 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

New release '22.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Tue Jan 31 17:53:47 UTC 2023 on tty1
husain@husain:~$
```



Measurements How-to :

Experiment 1 CPU:

- Sysbench cpu test was used to test the CPU
- For the three different scenarios, the values for cpu-max-prime and num-threads were changed. The three scenarios for these parameters were:
 - a. cpu-max-prime=20,000 num-threads=4
 - b. cpu-max-prime=30,000 num-threads=4
 - c. cpu-max-prime=20,000 num-threads=3
- The reason to choose these parameters is that this allows us to compare difference in max-prime value while keeping threads the same and also let's us compare difference in threads while keeping max-prime the same.
- For docker, the Dockerfile runs the scripts automatically and nothing else has to be done. The top performance reports were read by accessing log files from the terminal.

- For QEMU, the script had to be made executable by running `chmod +x` and then running it.

Experiment 2 Fileio:

- Sysbench filio test was used to benchmark file operations
- We compare the 5 different fileio modes: seqwr, seqrewr, seqrd, rndrd, rndwr, rndrw. These 5 modes are either sequential or random in nature.
- The script creates 128 test files of 16MiB each making the total size of 2 GiB.
- The script then uses iostat to monitor performance.
- The script then deletes the test files after each run to avoid caching which could affect results.

Shell Scripts:

Script 1: CPU Test

```
0ls#!/bin/bash
echo
"*****"
*****"
echo "Running Experiment 1"
echo
"*****"
*****"

echo "Running sysbench CPU test 1"
sysbench --test=cpu --cpu-max-prime=20000 --num-threads=4 run &
PID=$!

echo "Collecting performance data using top while sysbench test is
running"
top -b -d1 -p $PID >> cpu_one.log &
TOP_PID=$!

wait $PID
kill $TOP_PID
echo
"*****"
*****"
```

```

echo "Running sysbench CPU test 2"
sysbench --test=cpu --cpu-max-prime=30000 --num-threads=4 run &
PID=$!

echo "Collecting performance data using top while sysbench test is
running"
top -b -d1 -p $PID >> cpu_two.log &
TOP_PID=$!

wait $PID
kill $TOP_PID
echo
"*****"
*****"

echo "Running sysbench CPU test 3"
sysbench --test=cpu --cpu-max-prime=20000 --num-threads=2 run &
PID=$!

echo "Collecting performance data using top while sysbench test is
running"
top -b -d1 -p $PID >> cpu_three.log &
TOP_PID=$!

wait $PID
kill $TOP_PID
echo
"*****"
*****"

echo "Tests Done"

echo "Tests Done"

```

Script 2: FileIO Test

```

#!/bin/bash
echo
"*****"
*****"

echo "Running Experiment 2"
echo
"*****"
*****"

```

```

for mode in seqwr seqrewr seqrd rndrd rndwr rndrw; do
    echo "Testing $mode mode..."
    sysbench fileio --file-test-mode=$mode prepare

    # Start iostat to monitor disk I/O
    iostat -dxk 1 &

    # Run sysbench fileio test
    sysbench fileio --file-test-mode=$mode run

    # Stop iostat
    kill $!

    sysbench fileio --file-test-mode=$mode cleanup
    echo
    "*****"
    "*****"
done

echo "All tests completed"

```

Performance Tools:

1. top:
 - a. top was used to collect data for the cpu test.
 - b. The results were stored in log files for each scenario and then accessed from the terminal after script execution.
 - c. Note: results obtained in QEMU were finer (more frequent timestamps) then for Docker
2. iostat:
 - a. This was used for the fileio test.
 - b. Results were printed directly to the console after each mode was tested.

Performance Data:

Experiment 1 CPU:

Here, we test the cpu and vary parameters for three different situations

N = cpu max prime number

T = number of threads

Docker results:

		Avg	Min	Max	Std. Deviation
N = 20000, T = 4	Time	10.0007	10.0003	10.0012	0.0000254845
	Events	147435.5	145895	149056	2145.15
	Events/sec	14744.4	14583.94	14904.86	261.854.
	CPU% User	100	100	100	0
	CPU% Kernel	0	0	0	0
N = 30000, T = 4	Time	10.00045	10.0004	10.0005	0.00005
	Events	86361.5	86371	87340	484.5
	Events/sec	8684.52	8635.59	8735.55	47.46
	CPU% User	99.03	79.3	100	3.836
	CPU% Kernel	0.6	0	1.3	0.572
N = 20000, T = 2	Time	10.00035	10.0003	10.0004	0.00005
	Events	81671	81494	81840	1677
	Events/sec	8166.27	8148.59	8180.95	27.68
	CPU% User	97.8	82.76	100	5.993
	CPU% Kernel	1.6	0	2.3	0.384

QEMU results:

		Avg	Min	Max	Std. Deviation
N = 20000, T = 4	Time	10.00025	10.0002	10.0003	0.00005
	Events	159393.5	158628	159159	327.5
	Events/sec	15937.93	15861.33	15914.53	27.2
	CPU% User	99.08	79	100	4.45
	CPU% Kernel	0.75	0	2.2	0.8929

N = 30000, T = 4	Time	10.0003	10.0002	10.0004	0.0001
	Events	91091	91086	91096	7.07
	Events/sec	9108.295	9107.69	9108.90	0.8556
	CPU% User	99.10	79.7	100	3.725
	CPU% Kernel	1.1	0	2.5	0.9876
N = 20000, T = 2	Time	10.00025	10.0002	10.0003	0.00005
	Events	83143	83197	83097	65.05382
	Events/sec	8314.025	8319.10	8308.95	7.17713
	CPU% User	99.12	81.3	100	3.5272
	CPU% Kernel	0.9634	0	2.4	0.75372

Analysis:

- Performance on QEMU was better on QEMU than docker for all three sets. This was expected as QEMU delivers near native performance.
- For both the cases, performance decreased when cpu-max-prime value increased as there is a noticeable drop in events per second, indicating an inverse relationship between CPU performance and cpu-max-prime.
- Halving the number of threads also resulted in almost half of the original performance as is evident by comparing case 1 and 2 for each virtualization technology. This indicates there is an almost linear relationship between threads and performance.
- As there is a direct relationship between threads and performance, and an inverse relationship between cpu-max-prime and performance, the best Events/sec was achieved for the first case for both Virtualization technologies.

Experiment 2:

Here, we test the fileio using sysbench.

The different modes tested are: seqwr, seqrewr, seqrd, rndrd, rndwr, rndrw

Docker Data:

		Avg	Min	Max	Std
seqwr	Read MiB/s	0	0	0	0
	Write MiB/s	110.44	106	115.23	4.24
	Latency	0.65	0.6	0.8	0.05
seqrewr	Read MiB/s	0	0	0	0
	Write MiB/s	130.66	120.52	141.08	10.57
	Latency	0.055	0.05	0.07	0.005
seqrd	Read MiB/s	137.30	137	137.82	0.15
	Write MiB/s	0	0	0	0
	Latency	0.13	0.11	0.14	0.02
rndrd	Read MiB/s	102.34	86.67	119	15.67
	Write MiB/s	0	0	0	0
	Latency	0.155	0.13	0.18	0.022
rndrw	Read MiB/s	44.98	44.96	45	0.50
	Write MiB/s	20.995	29.97	30	0.007
	Latency	0.08	0.07	0.09	0.02
rndwr	Read MiB/s	0	0	0	0
	Write MiB/s	96.005	89.9	102.11	7.976975
	Latency	0.075	0.07	0.08	0.005

QEMU Data:

		Avg	Min	Max	Std
seqwr	Read MiB/s	0	0	0	0
	Write MiB/s	612.645	592.32	632.97	20.325
	Latency	0.01	0.01	0.01	0

seqrewr	Read MiB/s	0	0	0	0
	Write MiB/s	733.825	673.37	797.93	59.8
	Latency	0.01	0.01	0.01	0
seqrd	Read MiB/s	1001.76	956.29	1011.72	12.26
	Write MiB/s	0	0	0	0
	Latency	0.02	0.02	0.02	0
rndrd	Read MiB/s	142.33	140	145	2.52
	Write MiB/s	0	0	0	0
	Latency	0.11	0.11	0.11	0
rndrw	Read MiB/s	100.08	99.17	100.99	0.91
	Write MiB/s	66.72	66.1	66.74	0.61
	Latency	0.03	0.04	0.04	0
rndwr	Read MiB/s	0	0	0	0
	Write MiB/s	257.94	256.43	256.8	2.308
	Latency	0.04	0.03	0.03	0

Analysis:

- As was the case with CPU benchmarks, file io performance is also better on QEMU. Unlike the CPU benchmark, the performance here is significantly higher for QEMU compared to Docker (Read speeds for seqrd are almost 7x higher on QEMU).
- Latency is also significantly lower for QEMU in each mode compared to docker (latency is almost 5.5x higher in docker for the seqrewr mode).
- Another interesting observation to note is that latency for QEMU was consistent every time for a given mode which was not the case in docker. For example, the latency across all 5 executions for the seqwr mode was 0.01 on QEMU.
- Read and write speeds for sequential modes were higher than random modes for both virtualization technologies. The highest read speeds was for seqrd while highest write speeds were for seqrewr modes for both docker and QEMU.