580K Big Project Report

gVisor deeply analyze and capability test

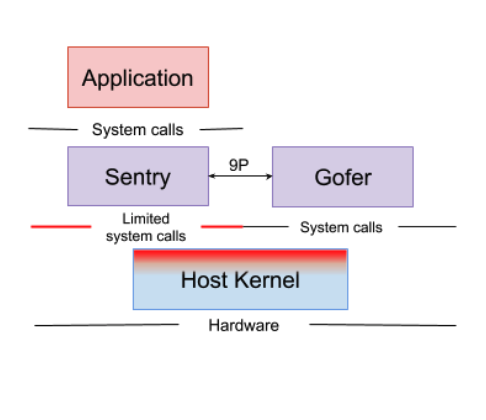
Yu Yang Tuoyan Fan

Abstract

Gvisor is a very safe new container. To test it actual performance, we tested it from many aspects and compared it with other technologies. We tested its network, file read and write, memory efficiency, system-call overheads. And compared with ordinary Docker containers without gVisor on GCP, local OS, and Kata containers (a hypervisor-based approach) on GCP. We found a lot of interesting things and learned a lot.

Background

gVisor provides a virtualized environment in order to sandbox untrusted containers. The system interfaces normally implemented by the host kernel are moved into a distinct, per-sandbox user space kernel in order to minimize the risk of an exploit. gVisor does not introduce large fixed overheads however, and still retains a process-like model with respect to resource utilization.



Simply to say, It’s a:

1. User-space kernel.
2. A runtime container to sandbox untrusted containers.
3. gVisor implements the OCI standard, used by Docker. So, Docker users can switch between the default engine, runc and the gVisor runtime engine, runsc.

The 4 main points of gVisor:

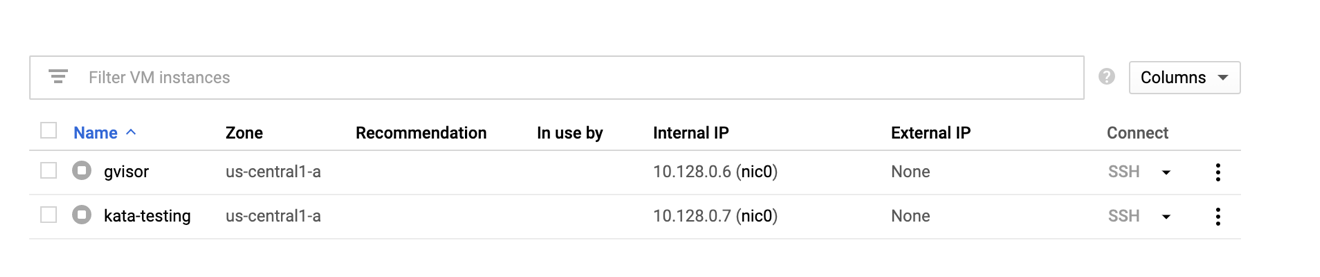
1. The application runs on the sentry, and It just uses 55 unique calls to implement 211 system calls of Linux System. So, it can avoid a bunch of attacks.
2. The I/O calls do not go to the host. The sentry can serve these I/O calls.
3. The network stack is implemented in the Sentry Go. It communicates via a socket with a virtual network device.
4. The sentry implements mmap process-level calls by itself.

Environment setup

Google Cloud Compute machine, which having two CPU, CPU - 2,4G of RAM

20G standard-Disk we use Ubuntu 16.04.6 LTS and Docker.

Kata container VM, have 1 CPU and 20F standard-Disk with Ubuntu 10.04.6 LTS and KVM.



Scope of Report

Compare the performance with other different virtualization techniques:

1. Some real application workloads
2. Network
3. File I/O
4. CPU

Research Methodology

Nowadays, we mainly have three approaches to achieve virtualization.

1. Machine-level virtualization, such as KVM and Xen,

2. Rule-based execution, such as seccomp, SELinux and AppArmor

3. gVisor intercepts application system calls and acts as the guest kernel, without the need for translation through virtualized hardware.

We are prepared to adopt the way of vertical comparison and horizontal comparison, from the CPU, the File I/O, Network and some workloads and make an overall assessment of gVisor performance from its architecture analysis the advantages and disadvantages.

Horizontal comparison: compare the performance of gVisor and kata container.

Vertical comparison: the performance of Docker and gVisor.

Data Collection

**CPU and IOSTAT analyzes**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Native System CPU Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 28.3091 | 10000 | 28.3503 | 2.76 | 2.83 | 11.53 | 2.95 | 10000 | 28.3503 |
| Test2 | 28.3873 | 10000 | 28.3830 | 2.76 | 2.84 | 3.53 | 2.95 | 10000 | 28.3830 |
| Test3 | 28.5094 | 10000 | 28.5050 | 2.76 | 2.85 | 3.43 | 2.99 | 10000 | 28.5050 |
| Avg | 28.4019 | 10000 | 28.4128 | 2.76 | 2.84 | 3.48 | 2.96 | 10000 | 28.4128 |

|  |  |  |
| --- | --- | --- |
| Native system CPU IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.06 | 0.02 |
| Test2 | 50.02 | 0.02 |
| Test3 | 50.07 | 0.01 |
| Average | 50.05 | 0.017 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker CPU Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 27.4831 | 10000 | 27.4800 | 2.66 | 2.75 | 3.41 | 2.87 | 10000 | 27.4800 |
| Test2 | 27.4316 | 10000 | 27.4281 | 2.66 | 2.74 | 3.40 | 2.87 | 10000 | 27.4281 |
| Test3 | 27.4750 | 10000 | 27.4716 | 2.66 | 2.75 | 4.56 | 2.87 | 10000 | 27.4716 |
| Avg | 27.4632 | 10000 | 27.4599 | 2.66 | 2.746 | 3,79 | 2.87 | 10000 | 27.4599 |

|  |  |  |
| --- | --- | --- |
| Docker CPU IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.04 | 0.06 |
| Test2 | 50.05 | 0.02 |
| Test3 | 49.49 | 0.02 |
| Average | 49.86 | 0.033 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker with gVisor CPU Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 27.2477 | 10000 | 27.2331 | 2.67 | 2.72 | 60.4 | 2.77 | 10000 | 27.2331 |
| Test2 | 27.1667 | 10000 | 27.1629 | 2.66 | 2.72 | 4.42 | 2.77 | 10000 | 27.1629 |
| Test3 | 27.2119 | 10000 | 27.2076 | 2.66 | 2.72 | 3.70 | 2.78 | 10000 | 27.4716 |
| Avg | 27.2087 | 10000 | 27.2012 | 2.66 | 2.72 | 4.06 | 2.77 | 10000 | 27.2012 |

|  |  |  |
| --- | --- | --- |
| Docker with gVisor CPU IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.39 | 0.02 |
| Test2 | 50.36 | 0.02 |
| Test3 | 50.37 | 0.05 |
| Average | 50.373 | 0.03 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Kata-Container CPU Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 27.5429 | 10000 | 27.5233 | 2.67 | 2.75 | 16.86 | 2.81 | 10000 | 27.5233 |
| Test2 | 27.6095 | 10000 | 27.5978 | 2.69 | 2.76 | 35.68 | 2.82 | 10000 | 27.5978 |
| Test3 | 27.5421 | 10000 | 27.5246 | 2.66 | 2.75 | 18.72 | 2.81 | 10000 | 27.5246 |
| Avg | 27.5648 | 10000 | 27.5485 | 2.66 | 2.67 | 17.79 | 2.81 | 10000 | 27.5485 |

|  |  |  |
| --- | --- | --- |
| Kata-Container CPU IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.06 | 0.08 |
| Test2 | 50.07 | 0.04 |
| Test3 | 50.03 | 0.10 |
| Average | 50.05 | 0.07 |

**FILEIO and IOSTAT analyzes**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Native system FILEIO test | | | | | | | | | |
|  | Total time  (s) | Total  events  number | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx.  95  Percentile | Events | Execution  time |
| Test1 | 13.0788 | 10002 | 0.0796 | 0.00 | 0.01 | 0.83 | 0.01 | 625.125 | 0.005 |
| Test2 | 13.1148 | 10001 | 0.0811 | 0.00 | 0.01 | 1.27 | 0.01 | 625.062 | 0.0051 |
| Test3 | 13.1354 | 10002 | 0.2425 | 0.00 | 0.02 | 41.4 | 0.01 | 625.125 | 0.0152 |
| Avg | 13.1097 | 10001.6 | 0.1344 | 0.00 | 0.01 | 14.5 | 0.01 | 625.104 | 0.0083 |

|  |  |  |  |
| --- | --- | --- | --- |
| Native system IOSTAT | | | |
|  | Throughput(MB/s) | Latency(ms) | Utilization(%) |
| Test1 | 11.949 | 11.44 | 92.60 |
| Test2 | 11.915 | 12.0075 | 92.92 |
| Test3 | 11,898 | 11.5075 | 93.44 |
| Average | 11.920 | 11.6517 | 92.98 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker FILEIO test | | | | | | | | | |
|  | Total time  (s) | Total  events  number | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx.  95  Percentile | Events | Execution  time |
| Test1 | 13.1534 | 10001 | 0.5490 | 0.00 | 0.05 | 29.3 | 0.01 | 625.062 | 0.0343 |
| Test2 | 13.1490 | 10001 | 0.2466 | 0.00 | 0.02 | 24.9 | 0.01 | 625.062 | 0.0154 |
| Test3 | 13.1483 | 10000 | 0.6921 | 0.00 | 0.07 | 38.1 | 0.01 | 625.000 | 0.0433 |
| Avg | 13.1502 | 10000.6 | 0.4959 | 0.00 | 0.05 | 30.8 | 0.01 | 625.042 | 0.0310 |

|  |  |  |  |
| --- | --- | --- | --- |
| Docker FILEIO IOSTAT | | | |
|  | Throughput | Latency | utilization |
| Test1 | 11.88 | 13.88 | 92.08 |
| Test2 | 11.884 | 11.065 | 92.80 |
| Test3 | 11.884 | 11.12 | 93.48 |
| Average | 11.883 | 12.02 | 92.78 |

* Max random IO

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker with gVisor FILEIO test | | | | | | | | | |
|  | Total time  (s) | Total  events  number | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx.  95  Percentile | Events | Execution  time |
| Test1 | 12.9845 | 10013 | 11.9987 | 0.03 | 1.20 | 103.51 | 5.81 | 625.812 | 0.7499 |
| Test2 | 13.0237 | 10016 | 12.1777 | 0.03 | 1.22 | 109.35 | 6.17 | 626 | 0.7611 |
| Test3 | 13.0370 | 10014 | 12.2580 | 0.03 | 1.22 | 92.89 | 7.10 | 625.8750 | 0.7661 |
| Avg | 13.0528 | 10014 | 12.1448 | 0.03 | 1.213 | 101.91 | 6.36 | 625.042 | 0.7573 |

|  |  |  |  |
| --- | --- | --- | --- |
| Docker with gVisor FILEIO IOSTAT | | | |
|  | Throughput | Latency | utilization |
| Test1 | 12.049 | 3.63 | 76.73 |
| Test2 | 12.017 | 3.81 | 77.00 |
| Test3 | 12.002 | 3.65 | 73.96 |
| Average | 12.022 | 3.69 | 76.89 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Kata-Container FILEIO test | | | | | | | | | |
|  | Total time  (s) | Total  events  number | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx.  95  Percentile | Events | Execution  time |
| Test1 | 13.5110 | 10000 | 46.0010 | 0.38 | 4.60 | 42.49 | 15.20 | 625 | 2.8751 |
| Test2 | 13.5148 | 10003 | 43.9913 | 0.3 | 4.40 | 68.59 | 14.40 | 625.1875 | 2.7495 |
| Test3 | 13.5106 | 10000 | 45.7135 | 0.36 | 4.57 | 68.11 | 15.21 | 625 | 2.8571 |
| Avg | 13.5254 | 10001 | 45.2353 | 0.34 | 4.52 | 68.31 | 14.93 | 625.042 | 2.8272 |

|  |  |  |  |
| --- | --- | --- | --- |
| Kata-Container FILEIO IOSTAT | | | |
|  | Throughput | Latency | utilization |
| Test1 | 11.565 | 0.80 | 10.335 |
| Test2 | 11.565 | 0.83 | 10.265 |
| Test3 | 11.565 | 0.76 | 10.103 |
| Average | 11.565 | 0.796 | 10.234 |

**Memory Test**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Native System Memory Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 1.1330 | 10240 | 1.1316 | 0.11 | 0.11 | 0.28 | 0.12 | 10240 | 1.1316 |
| Test2 | 1.1223 | 10240 | 1.1209 | 0.11 | 0.11 | 0.28 | 0.12 | 10240 | 1.1209 |
| Test3 | 1.1249 | 10240 | 1.1234 | 0.11 | 0.11 | 0.31 | 0.12 | 10240 | 1.1234 |
| Avg | 1.1267 | 10240 | 1.1253 | 0.11 | 0.11 | 0.29 | 0.12 | 10240 | 1.1253 |

|  |  |  |
| --- | --- | --- |
| Docker IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 28.75 | 0.01 |
| Test2 | 50.39 | 0.01 |
| Test3 | 48.24 | 0.01 |
| Average | 49.31 | 0.01 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker Memory Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 1.1354 | 10240 | 1.1339 | 0.11 | 0.11 | 0.28 | 0.12 | 10240 | 1.1339 |
| Test2 | 1.1300 | 10240 | 1.1285 | 0.11 | 0.11 | 0.39 | 0.12 | 10240 | 1.1285 |
| Test3 | 1.1321 | 10240 | 1.1306 | 0.11 | 0.11 | 0.26 | 0.12 | 10240 | 1.1306 |
| Avg | 1.1325 | 10240 | 1.1310 | 0.11 | 0.11 | 0.24 | 0.12 | 10240 | 1.1310 |

|  |  |  |
| --- | --- | --- |
| Docker IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.00 | 0.01 |
| Test2 | 28.39 | 0.01 |
| Test3 | 48.24 | 0.01 |
| Average | 49.12 | 0.01 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Docker with gVisor Memory Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 1.1691 | 10240 | 1.1665 | 0.11 | 0.11 | 0.30 | 0.13 | 10240 | 1.1665 |
| Test2 | 1.1470 | 10240 | 1.1447 | 0.11 | 0.11 | 0.30 | 0.13 | 10240 | 1.1447 |
| Test3 | 1.1733 | 10240 | 1.1710 | 0.11 | 0.11 | 0.32 | 0.13 | 10240 | 1.1710 |
| Avg | 1.1631 | 10240 | 1.1607 | 0.11 | 0.11 | 0.306 | 0.13 | 10240 | 1.1607 |

|  |  |  |
| --- | --- | --- |
| Docker with gVisor IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 50.00 | 0.01 |
| Test2 | 28.39 | 0.01 |
| Test3 | 48.24 | 0.01 |
| Average | 49.12 | 0.01 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Kata-container Memory Test | | | | | | | | | |
|  | Total time  (s) | Number of event | Total execution time | Min  (ms) | Avg  (ms) | Max  (ms) | Approx. 95 percentile | events | Execution time |
| Test1 | 1.2208 | 10240 | 1.2067 | 0.11 | 0.12 | 22.53 | 0.15 | 10240 | 1.2067 |
| Test2 | 1.1942 | 10240 | 1.1808 | 0.11 | 0.12 | 14.78 | 0.15 | 10240 | 1.1808 |
| Test3 | 1.1967 | 10240 | 1.1852 | 0.11 | 0.12 | 16.05 | 0.15 | 10240 | 1.1852 |
| Avg | 1.2039 | 10240 | 1.1909 | 0.11 | 0.12 | 17.78 | 0.15 | 10240 | 1.1909 |

|  |  |  |
| --- | --- | --- |
| Kata-Container IOSTAT | | |
|  | User utilization(%) | Kernel utilization(%) |
| Test1 | 48.00 | 1.00 |
| Test2 | 48.24 | 0.5 |
| Test3 | 47.76 | 1.99 |
| Average | 48 | 1.16 |

**Network**

|  |  |  |
| --- | --- | --- |
| Native system Network Iperf | | |
|  | Transfer(GBps) | Bandwidth(Gbps) |
| Test1 | 33.2 | 28.5 |
| Test2 | 33.3 | 28.7 |
| Test3 | 33.3 | 28.6 |
| Average | 33.3 | 28.6 |

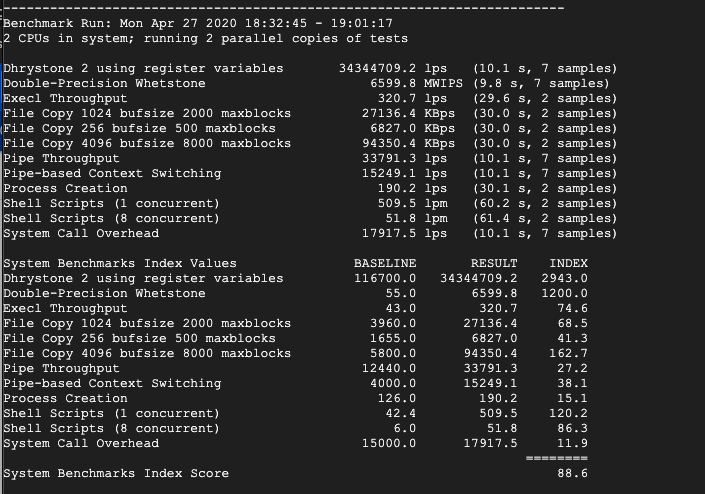
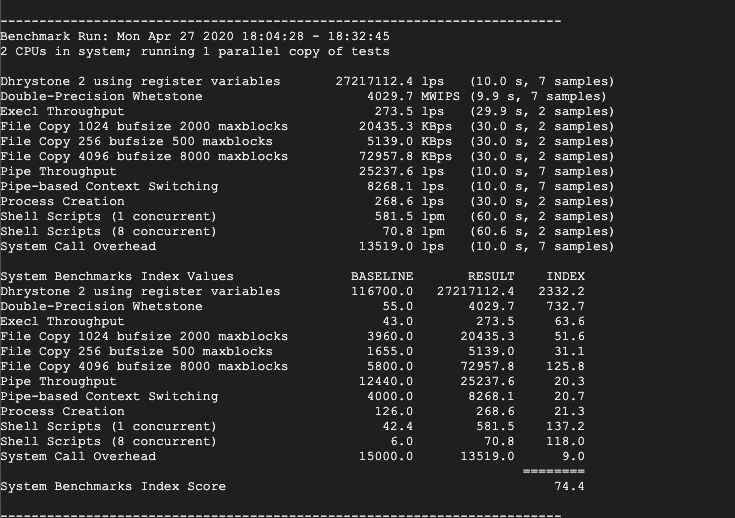
|  |  |  |
| --- | --- | --- |
| Docker Network Iperf | | |
|  | Transfer(GBps) | Bandwidth(Gbps) |
| Test1 | 14.9 | 12.5 |
| Test2 | 14.8 | 12.7 |
| Test3 | 14.8 | 12.8 |
| Average | 14.8 | 12.7 |

|  |  |  |
| --- | --- | --- |
| Docker with gVisor Iperf | | |
|  | Transfer(GBps) | Bandwidth(Gbps) |
| Test1 | 2.03 | 1.74 |
| Test2 | 2.03 | 1.71 |
| Test3 | 2.01 | 1.77 |
| Average | 2.03 | 1.74 |

|  |  |  |
| --- | --- | --- |
| Kata container Network Iperf | | |
|  | Transfer(GBps) | Bandwidth(Gbps) |
| Test1 | 35.6 | 30.8 |
| Test2 | 35.2 | 30.6 |
| Test3 | 36.1 | 30.3 |
| Average | 35.6 | 30.6 |

**UNIX Benchmarks**

For gVisor:

****

For native system:

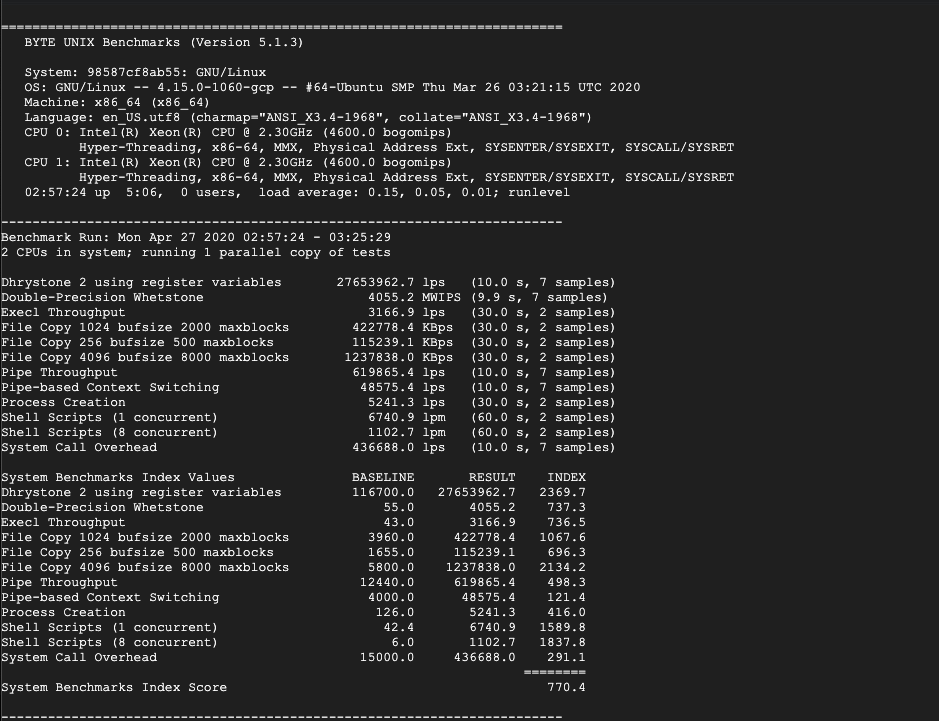
图片包含 游戏机, 文字, 截图

描述已自动生成

图片包含 游戏机, 文字

描述已自动生成

For docker:



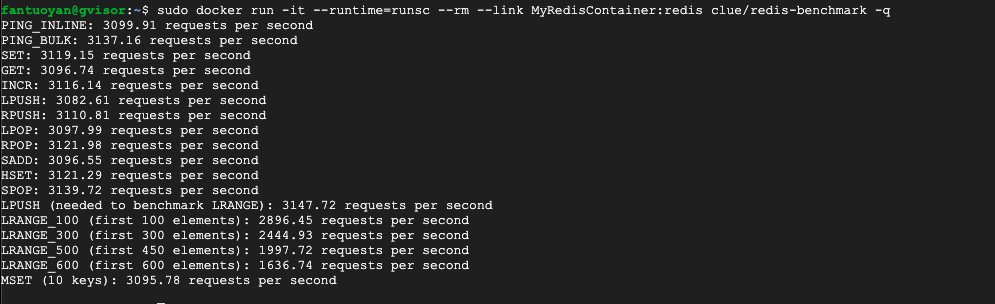
For Kata-container:

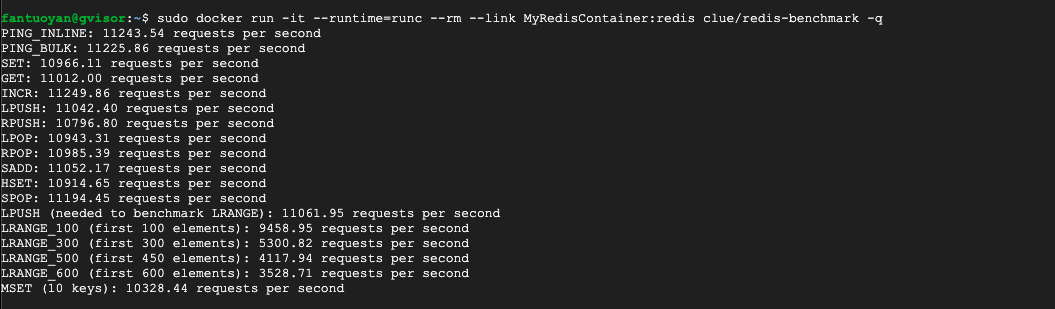
图片包含 游戏机, 文字, 截图

描述已自动生成

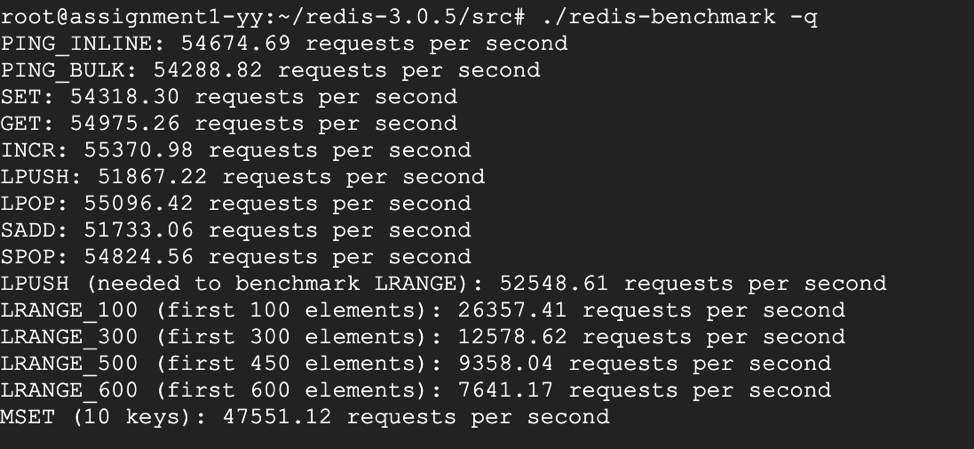
**Redis Benchmark test**

For gvisor:

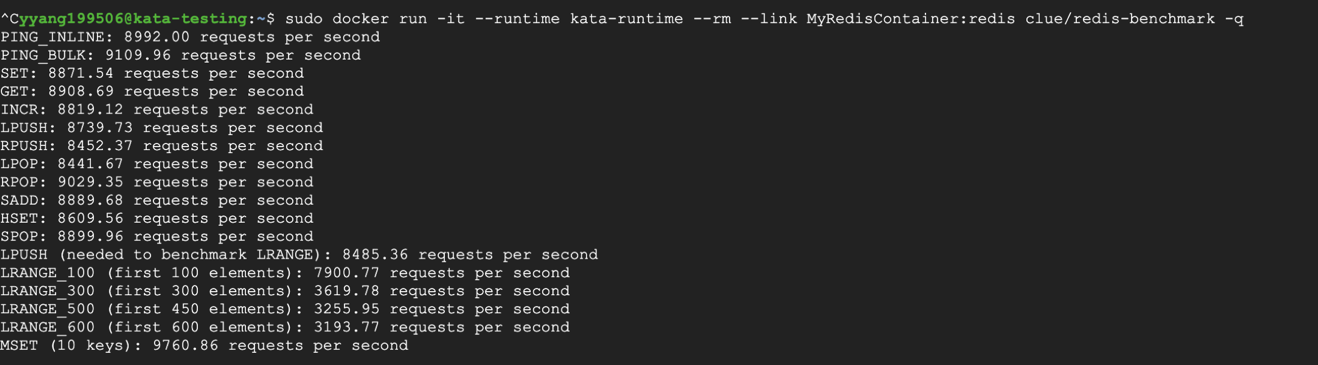
****

For docker runc:****

For native:

****

For kata :



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | PING\_LINE | PING\_BULK | SET | GET | INCR | LPUSH | RPUSH | LPOP | RPOP |
| DOCKER | 11243 | 11225 | 10966 | 11012 | 11249 | 11042 | 10796 | 10943 | 10985 |
| GVISOR | 3099 | 3137 | 3119 | 3096 | 3116 | 3082 | 3110 | 3097 | 3121 |
| KATA | 8992 | 9109 | 8871 | 8908 | 8819 | 8739 | 8452 | 8441 | 9029 |
|  | SADD | HEST | SPOP | LPUSH | LRANGE100 | LRANGE300 | LRANGE500 | LRANGE600 | MSET |
| DOCKER | 11052 | 10914 | 11194 | 11061 | 9458 | 5300 | 4117 | 3528 | 10328 |
| GVISOR | 3096 | 3121 | 3139 | 3147 | 2896 | 2444 | 1997 | 1636 | 3095 |
| KATA | 8889 | 8609 | 8899 | 8485 | 7900 | 3619 | 3255 | 3193 | 9760 |

Ab nginx

手机屏幕的截图

描述已自动生成

**Some Related key commands:**

The runtime options of docker:

1. runsc,
2. runc,
3. kata-runtime

The images:

1. csminpp/ubuntu-sysbench,
2. networkstatic/iperf,
3. MyRedisContainer:redis clue/redis-benchmark

The test mode of sysbench:

1. Fileio
2. Cpu
3. memory

sudo docker pull csminpp/ubuntu-sysbenchd

sudo docker run --runtime=runsc -it csminpp/ubuntu-sysbench /bin/sh

sudo docker run --runtime=runc -it csminpp/ubuntu-sysbench /bin/sh

sudo docker run --runtime=runc -it --rm networkstatic/iperf3 -c 172.17.0.1

sysbench --test=memory --memory-block-size=1M --memory-total-size=10G run

sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw prepare

sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw run

sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw cleanup

sysbench --test=cpu --cpu-max-prime=20000 run

sudo docker run -it --runtime kata-runtime csminpp/ubuntu-sysbench /bin/sh

sudo docker run -it --runtime=kata-runtime --rm --link MyRedisContainer:redis clue/redis-benchmark -q

**Result and arch analyze**

**In CPU performance:**

we can see that their performance has no big difference, Gvisor has no runtime cost imposed for CPU operations. Because it does not perform emulation or otherwise interfere with the raw execution of CPU instructions by the application. In the other word CPU overhead are very limited for kata and gVisor and docker.

**In File I/O:**

By vertically comparing gVisor and kata, the gap between the two is not very obvious. So gVisor does not lose throughput.

But latency, we can see that gVisor has a significantly lower latency than docker and native system. This is related to its structure because it handles I / O using gofer. So, latency decreases.

Utilization in the statistical time. All processing I / O time divided by the total statistical time reflects the busyness of the device to a certain extent. It can be seen from the data that its utilization rate is slightly reduced, which means that the busy level is slightly lower, which is related to its security structure. For benchmarks that are bound by raw disk I/O and a mix of compute, file system operations are less of an issue.

**In memory performance:**

The Sentry provides an additional layer of indirection, and it requires memory in order to store state associated with the application. This memory generally consists of a fixed component, plus an amount that varies with the usage of operating system resources (e.g. how many sockets or files are opened).

**Network:**

It can be seen that the bandwidth and transmission speed of the gvisor is extremely slow, which is almost ten times slower in our test results, and is not suitable for running applications with higher network transmission requirements.

The reason is it use the user space network stack.

For high-performance networking applications, you may choose to disable the user space network stack and instead use the host network stack, including the loopback. Note that this mode decreases the isolation to the host. It means that it will increase the risk of host kernel to be attacked

**Redis benchmark:**

In this part, we can see that the network of gVisor performance is very bad compare to docker and kata. This is in line with the performance of the network bandwidth and transmission element we did before. GVisor sacrifices a lot of network performance to improve security.

**Arch analyze:**

Another Sandbox idea is provided by gVisor, which is very lightweight with minimal additional memory consumption, but also provides a level of isolation from the VM solution. The ptrace-based gVisor introduced in this share has poor performance of system calls and poor application compatibility. GVisor can integrate well with Docker, but integration with Kubernetes is still in the experimental stage. When integrating with Docker, gVisor follows the OCI (Open Containers Initiative) standard, so it can be executed as a Runtime for Docker.

How to work?

The gVisor, running as a non-privileged user, intercepts the application's system calls, completely isolating the application from the kernel. Instead of simply acting on the system calls made by the application directly to the kernel, gVisor implements most of the system calls, allowing the application to indirectly access the system resources by simulating the system calls. The gVisor makes system calls to the operating system when the system calls themselves are impersonated, and these system calls are filtered by using Seccomp.

How does the gVisor intercept the application's system calls?

There are two modes of operation for gVisor, and this share introduces only the gVisor based on Ptrace.

To understand how the gVisor intercepts system calls, you need to first understand Ptrace: Ptrace is a system call interface provided by Linux that allows you to establish a relationship between Tracer and Tracee between two processes. Tracer can control Tracee, for example, when Tracee receives the signal and actively enters the stopped state, then Tracer can choose whether to do some operations on Tracee (such as setting the register context of Tracee or the contents in memory, etc.), after the operation, Tracer can choose whether to let Tracee continue to execute.

In addition to receiving a signal, Tracee can also be told by Tracer that it is in the stopped state when it is about to enter a system call or when it is about to leave a system call. Specifically, Ptrace can use PTRACE\_SYSEMU to control Tracee to stop when the system is about to be redeployed. The gVisor also intercepts the application's system calls through this command.

Make the application Tracee:

When the gVisor starts with the Docker Runtime, you can see a similar interprocess relationship: docker-containerd-shim is the container's launcher; Sentry, the gVisor program that intercepts system calls to the mock kernel, is also Tracer. Stubs can be ignored for the moment, and the subprocess of the Stub is the application we want to put in the Sandbox. Sentry creates the stub, which then creates the application process, and Sentry attaches to the stub and the application through Ptrace attach. When the application is about to make a system call, it actively stops, which is where sentry intercepts and simulates the system call.

This is similar to using the GDB debugger C/C++ program, where when a target program is given to GDB to debug from the command line, the program runs as a child process and GDB ACTS as a Tracer attach to the application to control the application. Like GDB, sentry starts the application in a similar way, except that sentry starts a stub and then lets the stub start the application as its child process.

After the application is initially attached to the sentry, the sentry is responsible for starting the application.

In the operating system startup scenario, the binaries of the application are loaded by the operating system, such as allocating virtual memory space to hold code segments in the binaries, data segments, Shared libraries, or stack space to initialize the application. In the gVisor startup application scenario, a similar process is done by sentry:

An important role of the Stub process is to create the application as the initial template for the application. In fact, the stub, as a child of sentry, will actively unmmap almost all the memory regions in the virtual memory address space (see all the memory regions in the virtual memory address space of a process by viewing /proc/${pid}/maps) and even code and data segments. Only two memory regions are reserved.

Sentry initial set up the application of the RIP (register) instruction execution of initial value for the application of binary read application entry in the address, the address is generally in the application of virtual address space of a lower position, and through the PTRACE\_SYSEMU instructs the application starts, until meet the following two events into the stop state:

A system call is about to be made

Received a signal from the kernel or process

How to excuate process

When an application automatically enters the stopped state before it enters a system call, sentry reads the application's system call number and system call input to attempt to simulate the system call. Take, for example, the read sys\_read of a file, which finds the specified file, opens and reads the contents of the file, and writes memory to the virtual memory address specified by the application system call parameter. When Sentry receives a system call from this, it sends the file read request through the 9p protocol to the aforementioned gofer process (a socket pair transfer 9p protocol is established between Sentry and gofer). The gofer process performs the actual file read and returns the read to Sentry via the 9p protocol. Sentry writes the contents of the file read to the application's virtual memory (if the address has no corresponding virtual memory address segment, it is allocated and copied), then sentry writes the actual simulated results of the system call to the application's registers, and lets the application proceed. The restored application gets the result of the system call, so the system call is satisfied when the application is confused as to whether the actual system call was performed directly by the operating system or simulated by sentry.

"Application execution" in the description of file read system call processing actually also describes the application file system access control, in fact in the "application startup" in order to omit the description of the root file system mount, in mount the root file system simulation also involves by 9 p protocol access to file system. File writes are handled very similarly.

In addition to file reading and writing, there are many other system calls, such as Shared memory or other IPC, locking, creating threads or processes, sending signals, socket, execv, epoll, eventfd, pid namespace, etc., and gVisor are all simulated.

For example socket:

Sentry implements the basic TCP/IP protocol stack in user mode. Before starting the application, gVisor will start a temporary start process, which will enter the network namespace created by docker. The socket that creates AF\_PACKET is bound to the veth device to take over the network traffic on the device (also removing the IP from the device) and pass the socket to the sentry process. Later, when sentry intercepts the socket system call of the application, the network packets are actually sent from the veth device through the socket. Network packets received from the veth device are delivered to the socket layer of the application after passing through the sentry network protocol stack.

Summary:

All in all, Gvisor as a lightweight virtual technology, its overall performance is very good. For example, it has achieved defense in depth through sentry and the gofer file system, which plays a very good role in protecting the host kernel. But at the same time it also exposed some problems are network problems. It used the space network stack to protect the kernel. But it also lost part of the bandwidth and transmission rate. But overall, gVisor is a very good virtualization technology.