Latency Impact of Docker Containers: A Closer Look

ABSTRACT

Traditionally, many web services are held on virtual machines (VMs) provided by cloud computing suppliers. Since VMs bring about dramatic performance degradation compared to bare metal, the quality of service (QoS) is affected. Among all the QoS features, service latency is of crucial importance. With the prevalence of Docker, containers, also called "lightweight VM", offer another choice to deploy web applications on the cloud. This paper takes the first to thoroughly analyze the impact of different Docker configurations on service latency. We conclude that the CPU quota configuration might lead to a long tail latency. Docker bridge could lead to a fixed amount of latency degradation instead of a percentage fallen. Using AUFS could bring about extra latency when opening a file or traversing the file system, and have no effect on writing data to a file.

Keywords

Docker; container; latency; linux bridge; AUFS

1. INTRODUCTION

Began from an open-source advanced container engine of dotCloud, a Platform-as-a-Service (PaaS) supplier, Docker is becoming one of the most promising virtualization platform. It significantly shortens the process of packing, shipping and running applications[23]. By packing all the dependencies of the application into several image layers, you can carry the package around and run it with simple commands on almost every laptop, personal computer, and even cloud center as long as running a Linux operating system.

Unlike traditional VMs, which use hardware-level virtualization, Docker containers employ system-level virtualization and share the same kernel with the host machine[27]. Many researches[8, 16] have proved that containers have a better performance in most cases than VMs. Due to these performance reasons, many companies are trying to move services from VMs to containers[11]. However, containers do add additional layers compared to bare-metal hardware, which leads to certain degree of performance degradation.

Many modern web services like Google and Facebook are interactive. Responses should be returned very soon otherwise users might complain. Also, these services are dynamic. Datacenters process huge amount of data based on the user input and response in very limited time. For example, it requires thousands of Memcached machines to do a simple request through Facebook servers[24] and tens of thousands of index servers to do a Bing search[15]. In these cases, not only the throughput is of crucial importance to serve as many clients as possible concurrently, latency should also be taken into account to provide users with the best interactivity. Each additional time cost in one of the service backend layers would increase the overall latency. If all of these separated services require software virtualization layer, a millisecond's latency might be amplified to several thousand milliseconds, thus greatly influencing the overall performance of the service.

Tail latency is another issue to care about[4]. In MapReduce task[5], a program is processed by hundreds of machines. The result of each map machine is passed to a central reduce machine, so the reduce one has to wait all map ones before moving on. In this case, a map work done over one minute encumbers the whole system even if others are finished in several seconds. Assume that a task has one hundred sub tasks and each sub task has a 99% probability to finish in one microsecond while 1% to finish over 1 second. Then the overall performance of this job is 63.3% probability to finish over one second, which is a rather bad performance. The one-in-one-thousand situation becomes a common case.

The additional layer of VM brings about significant performance lost[14] and is part of the source reason of long latency. The occurrence of Docker thus provide another choice. Many cloud service suppliers like Amazon EC2[2] and Microsoft Azure[3] provide container services in recent years. To simplify the deployment of applications, small companies are considering to use these Docker cloud. It is very important for them to know the trade off between convenience and latency degradation when using Docker to deploy latency-sensitive applications.

The following sections are organized as follows: Section 2 introduces some background information about Docker technologies. Section 3 carries out experiments and gives analysis about their affects to service latency. We discuss related works in Section 4 and give a final conclusion in Section 5.

2. BACKGROUND & MOTIVATION

2.1 Namespace & CGroups

LXC[12] and Docker libcontainer[28] use Linux Namespaces[33] together with Control Groups (CGroups)[21] to realize resource isolation and limitation. Resources like CPU, memory and process id (PID) are no longer global, but belongs to a particular namespace. Processes outside a namespace are transparent to ones inside the namespace. Inside processes also has no access to outside resources, thus providing certain level of security.

2.2 Linux Bridge

Bridge mode is the default network setting of Docker, which makes use of the Linux bridge feature[31]. When using this mode, each container is allocated a network namespace and separate IP. Once the Docker daemon starts, it creates a virtual network bridge named docker0 on the host machine. All Docker containers created on this machine will be connected to docker0. Virtual network bridge works like a physical switch, thus all containers on the host machine are connected in a two-layer network through the switch. Docker chooses a private IP different from the host IP and allocate it to docker0. It also selects a sub net defined in RFC1918. Each container on the host machine is assigned an unused IP from this sub net pool.

2.3 AUFS

Another Union File System (AUFS) is a kind of Union

File System[25]. Frankly speaking, it is a file system that supports to mount different directories to a single virtual filesystem. Further deep inside, AUFS supports to set the readonly, readwrite, whiteout-able authority of every member category, just like Git Branch. Also, the layer concept in AUFS supports to logically and incrementally modify the readonly branch without affecting the readonly part. Generally speaking, there are two uses of Union FS. On one hand, it can mount multiple disks to a single directory without the help of LVM[10] or RAID[9]. On the other hand, it enables the cooperation of a readonly branch and a writeable branch. Docker uses AUFS to build container images.

2.4 Motivation

The above mechanisms allows Docker to provide users with great convenience and simplicity. Before packing services into Docker containers, programmers should be aware of the potential latency slowdown of these technologies. Additional program logic might be included to perform resource isolation and limitation. Network and file I/O access also involve passing through extra layers. However, most related works only focus on the throughput instead of latency. We take the first to fill this critical void.

3. LATENCY CHARACTERIZATION

In this section, we first describe our experimental methodology and then evaluate various Docker configurations including CPU, network and file system.

3.1 Experimental Methodology

Since we focus on the latency of real-time services, we incorporate a client-server model which tests the round trip latency for several operations. We conduct our experiment on two HP MicroServer nodes (Intel Xeon E3-1220L processor, 2.3GHz), each with 4GB installed RAM. We employ Apache Thrift[26] to let client side use RPC calls to call the server side and server then return the result. Python is the experiment language. For each call, we measure the latency based on its start time and end time.

In all cases, server is running in a Docker container. To narrow down the experiment interference, we first let CPU #3 (totally 4 CPUs, 0 - 3) excluded from the CPU auto scheduling mechanism, which means that only our container can run on this CPU and all other applications have no access to it. This is implemented using the CPU affinity mechanism[20] and we add '**isolcpus=3**' linux kernel boot option when starting the server host machine. We also disable all interrupts to happen on CPU #3, thus making sure no additional context switch[17] would happen. Each time we run the server container, we have to use the '**–cpuset-cpus="3"**' to force our container run on the specific CPU. '**–cpuset-cpus**' argument is very similar to '**taskset -c**' command since they can both assign a task on a dedicated CPU core. The difference is that '**–cpuset-cpus**' can only be applied to the whole container while '**taskset -c**' can be applied to any process.

3.2 CPU Configurations

In this experiment, we let the client run natively on a machine and the server run in a Docker container on the other machine. Server container uses option '**–net=host**' to expose all host's ports to the container. The client directly calls the server, without extra information like parameters sent or return values received. In each experiment, client continuously sends 1,000,000 requests to the server and then notes down the round trip latency.

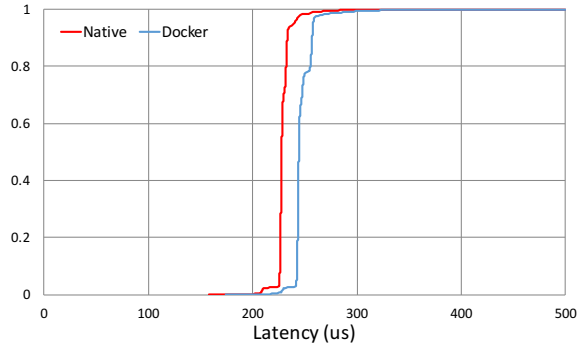


Figure 1: The CDF of latency using bare metal and Docker container

Table 1: Latency measurements of bare metal and Docker container

	mean(us)	median(us)	p99(us)
Bare metal	240.7	241.0	278.0
Docker container	255.2	255.0	295.0

3.2.1 *Baseline: Native Platform*

In our baseline case, we run the server process natively. To make use of CPU affinity, we use '**taskset -c 3**' to let our process run on the target CPU #3. The experiment is repeated for 10 times. Each time 1,000,000 requests are transmitted between client and server. The CDF[13] result is shown as the red line in Figure 1. The mean, median and 99^{th}-percentile position of the measurements are listed in Table 1. Most of the latencies are between 200 and 300 microseconds, and the average and median measurements are about 240 microseconds. However, there are still 1% latencies beyond 278 us and these long latencies would be very common in the real production world. This phenomenon might be caused by the interference of background processes, non-FIFO scheduling, multicore scheduling[18], and interference from other virtual machines or containers in the cloud environment[35].

3.2.2 *Case 1: Using CPU Shares*

We run the server process with '**–cpuset-cpus="3"**' setting to realize CPU affinity and the '**–cpu-shares=1024**' as a default setting in CFS scheduler. Assume that two containers have different shares and are running on the same CPU core and '**–cpu-shares=1024**' is set. Container A has a share of 1,024 and container B has a share of 512, if both containers are CPU-intensive, which means they take almost all the time to do CPU calculation. The CPU time used by container A and container B would be at a ratio of 1024 : 512, which is 2 : 1. We run the test for 10 times. Each time 1,000,000 requests are transmitted between client and server. The CDF result is shown as the blue line in Figure 1, and the mean, median and 99^{th}-percentile position of the measurements are also listed in Table 1.

From the above two test cases, we observe that when using Docker, the CPU latency almost shares the same CDF curve as using bare metal, except a little bias showing an additional fixed amount cost for CPU. Comparing both from

the 99^{th}-percentile column in Table 1 and the CDF curves in 1, Docker container does not have a significant impact on the tail latency performance when using CPU shares. Just like mentioned in the report of IBM, Docker containers do have impact on CPU performance. However, the degradation is very low, 4% in IBM's report about throughput and about 6% about the mean, median, and 99^{th}-percentile performance in our research. We conclude that when running CPU-intensive applications in a Docker container, the performance effect would be very small. Unlike VMs, which use hardware-level virtualization technology, Docker container's instructions do not need to be emulated by VMM. However, Docker containers share the same Linux kernel and use the same instructions as the host machine. An x86 instruction needs to be translated to several instructions to run on an ARM CPU using VMs. With the help of equation 1[22], we can do a rough calculation of the virtualization slowdown, where f_p stands for the fraction of privileged instructions executed by a VM and N_e stands for the average number of instructions required by the VMM to emulate a privileged instruction. The reason why Docker containers bring about a slightly slowdown is because when performing CPU isolation and limitation, the kernel needs to first check the namespace of the running process, thus the additional instructions would cause the extra latency.

$$S_v = f_p \times N_e + (1 - f_p) \quad (1)$$

3.2.3 Case 2: Using CPU Quota

Apart from '–**cpu-shares**', there exist other parameters which limit the resource usage of CPU. '–**cpu-period**' means the period for the processes in the Docker container to be scheduled on the CPU. It is often used together with '–**cpu-quota**' parameter. The unit of these two parameters are both microseconds. When these parameters are set, it means that processes in the container can use no more than '–**cpu-quota**' time during each '–**cpu-period**' time duration. To test whether these two parameters would have the same side effect as '–**cpu-shares**' when only one container is assigned to a CPU core, it can take all the cycles of that CPU, we carry out the following experiment:

We first fix '–**cpu-period=10000**' and vary the value of '–**cpu-quota**' to see the relationship between these two parameters. We choose values 1,000, 1,500, 2,000, 2,500, 3,000, 4,000, 5,000, 7,000, 10,000 for '–**cpu-quota**' and observe the results. These parameters are chosen because the minimum value of both these parameters are 1,000, and the magnitude gap is also set small. For each test, it is performed for 1,000,000 requests. We measure the mean, median, and 99^{th}-percentile of the latencies. We also take the number of requests whose latency is greater than 1,000 us, the minimum time slice, into consideration. These results are shown in Table 2.

From Table 2, we observe that latency increases incredibly when CPU quota only counts for a small ratio of the total CPU period. From 1,000 to 4,000, all mean, median and 99^{th}-percentile are decreasing and so is the number of test cases whose latency is greater than 1,000 us. Figure 2 also shows the relationship between '–**cpu-quota**' and the number of requests whose latency is greater than 1,000 us. From this picture, we can see that the number first drops fast and then slowly as quota increases. When quota reaches over 3,000 us, latency suddenly drops fast and finally goes to about zero at 4,000.

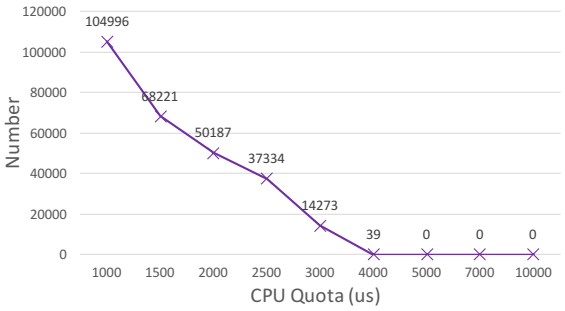


Figure 2: Number of requests' latency beyond 1 ms

Unlike using CPU share, once only a single process is using the CPU, it can take all the CPU resources, using CPU period together with CPU quota options has a force cut off when the CPU usage is over the limited number. Since the client is calling the server continuously, once a request has finished, another request will immediately follows. If the server process uses up its quota during one period, it is sure to give up the CPU and wait until the next period comes. This can cause a very long tail latency in real time services, which is shown as a sudden rise in the 99^{th}-percentile. Once the service is CPU-intensive or being visited quickly, it will add unwilling latency to the service, thus reducing the overall performance.

To prove the above theory, we first compute the last column in Table 2. Assume we need in total time t_{cpu} to do all the computation, which means the total time the process is running on CPU. CPU quota is t_q, and CPU period is t_p. Total number of requests blocked by the CPU options n is computed as follows: $n = t_{cpu}/t_q$. Thus, total time t_{total} needed to compute all the requests is: $t_{total} = n \times t_p$. So once t_{cpu} is determined, we can see that $n \times t_p = t_{cpu}$ is also determined. In our experiment, we assume the CPU time cost for each request is $t_{request}$, and the total number of requests is r. So we see that $t_{cpu} = t_{request} \times r$ is determined, and $n \times t_q$ must be also determined, which is shown as the last column in our experiment. We can observe that for the case 1000, 1500, 2000, 2500, the products are around 100,000,000, which satisfy our formula. However, when t_q comes to over 3000, the product falls incredibly. This phenomenon occurs because at this time, t_q is greater than the overall CPU used. We use *htop* command to measure CPU usage and the CPU use rate of that CPU is around 30%. This is the reason why it suddenly falls at the 3000 point, which has $3000/10000 = 30\%$, and then quickly goes to 0. We also see from Figure 2 that when it comes to over 4000, the mean, median and 99^{th}-percentile are almost not affected, which means that when the CPU usage of the application is less than the ratio of quota to period, it has low impact on the latency performance.

3.3 Network Isolation

In this experiment, server sends or receives various length data to or from the client. We choose message sizes 1KB, 10KB, 30KB, 50KB, 70KB and100KB. All message sizes are chosen from *SPECWeb2009*[29] as the standard web message sizes. We test 1 million requests for each experiment.

Sever is hosted in a Docker container on one machine and client is running natively on another machine. Both server

Table 2: Measurements of latency using '–cpu-period' and '–cpu-quota'

quota(us)	mean(us)	median(us)	p99(us)	# > 1,000us	# × quota
1000	1035.4	307.0	7556.0	104996	104996000
1500	579.5	258.0	5816.0	68221	102331500
2000	489.0	281.0	4765.0	50187	100374000
2500	361.7	260.0	3232.0	37334	93335000
3000	291.9	259.0	1741.0	14273	42819000
4000	258.0	253.0	331.0	39	156000
5000	263.2	255.0	341.0	0	0
7000	256.7	250.0	341.0	0	0
10000	262.6	255.0	333.0	0	0

and client are assigned to a dedicated CPU to reduce performance interference. We compare two Docker configurations, the first one is to use '–**net=host**', which means the container directly use the host network port and the network isolation mechanism is not working. The second one is to use '**-p portA:portB**', which means container uses Linux bridge to communicate to the outside world. The **portA** used by the container is mapped to **portB** of the host machine, and it's working similar to the NAT mechanism[32].

3.3.1 Case 1: Server Receives Data

In this case, client sends data to server using RPC calls and pass data through string parameters. We compare the mean, median and 99^{th}-percentile result between using '–**net=host**' and '**-p portA:portB**'. Results are shown in Figures 3(a-c).

3.3.2 Case 2: Server Sends Data

This time, client calls server with a single parameter indicating size and server returns the corresponding size string. Results similar to Section 3.3.1 are shown in Figures 3(d-f).

3.3.3 Analysis

As is shown in both figures, most of latencies using '**-p portA:portB**' are no more than 110% of latencies using '–**net=host**' in each size group. In fact, when file size becomes much larger, the results are very close to each other, and the slowdown percentage is nearly zero. Some of the comparison groups even witness a phenomenon where Docker bridge outperforms Docker host. This is because many other factors could slightly change the latency, including CPU scheduling, preempting, network contention, etc. They are outside the scope of this study.

However, according to IBM, there is a 80% slowdown using Docker bridge[8]. So why our experiment draws a totally different result from IBM's research? There are several reasons. First, from our observation of the experiment, the degradation is 'one-time'. It only adds a fixed latency to each trip. In our experiment, when the size of message is small, the impact of this fixed latency becomes large – nearly 10%. However, when message size is much larger, the impact of this latency is negligible, and the random vibration takes the dominance of the difference.

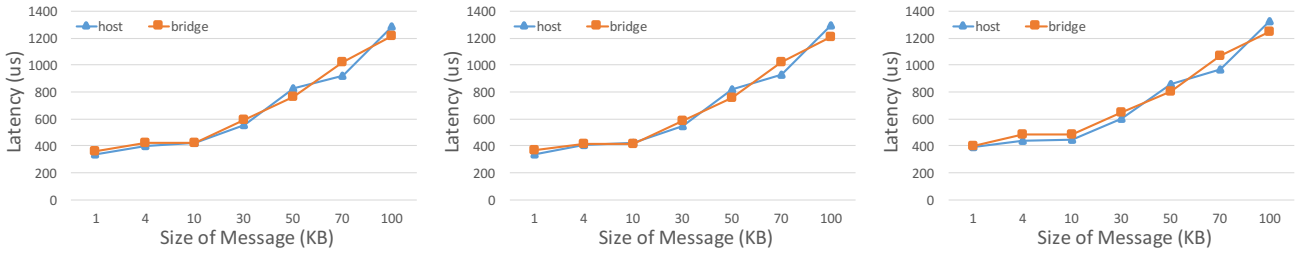
In IBM's report, it uses docker on both server and client side, which means a single round trip message goes through bridges four times, each side two. While in our experiment, it only happens at server side and totally two times. IBM's report uses 100B and 200B message size, while messages used in our experiment are much larger. Also, we use python and Thrift, which add much extra cost to the CPU and network, so the baseline might be much larger than in IBM's report. The size 0 situation is over 200 us as shown in the CPU experiment section. While all latencies in IBM's report are less than 100us. So we can not simply describe the impact of Docker bridge on the network performance as a percentage slowdown. It is a fixed-length degradation.

To understand the fixed-length degradation, we dive into the source code of Linux bridge. Taking the receive stage as an example, when a message has arrived at the NIC, it is uploaded to the upper layer in the network stack. It first checks whether this message is used by more than one application or it is to be multicasted. If not, it does no process including large amount of data copy. If so, the message is copied several times and then sent to each receiver, thus costing huge amount of extra latency. However, apparently in our experiment and IBM's experiment, the message is only used by a single application, without multicasting. Also, the message transmission is done by transferring a pointer to the message. So these extra functions will only cause a fixed extra time. This is why it affect so much in IBM's report while in ours slightly.

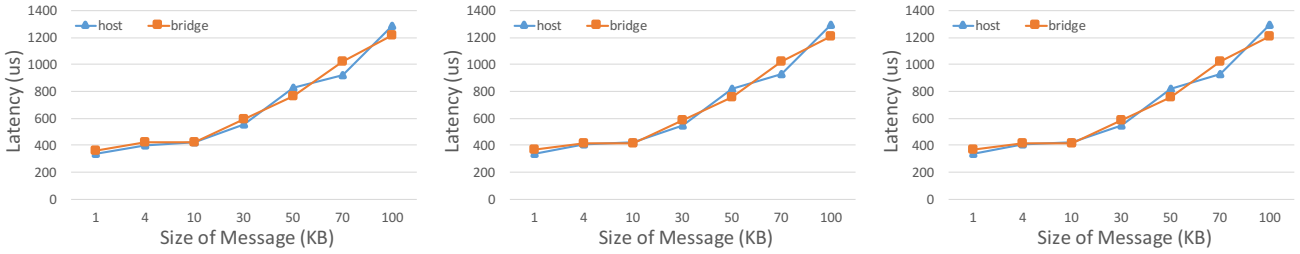
3.4 File Operations Using AUFS

Many applications like SQL server and Hadoop involve disk I/O operations. Disk operation latency of Docker container is thus another issue we should care about. In this experiment, only one application is hosted in a Docker container. The application continuously writes content of different sizes to a file. The length written each time are 1KB, 4KB, 10KB, 30KB, 50KB, 70kB, and 100KB. We compare two situations, the first one is to directly write messages inside the container, while the second is to write messages to a file outside and mounted to the container using '**-v**' option. To make sure that the messages are written to the disk immediately instead of staying in the cache area, each time we write a single message, we will use the '**flush()**' function to force the system to flush messages onto the disk. In each experiment group, the application continuously write 10,000 times and we repeat this operation for 10 times. We only discuss about the write case instead of the read one to avoid the pre-read situation where blocks of files are read in advance. We note down the mean, median and 99^{th}-percentile measurements. The results are shown in Figure 4.

From Figure 4, we can observe that there is almost no impact on performance using AUFS when writing files. This result is the same as the one drawn from IBM's disk I/O report. This phenomenon is explained as follows. When a file is open, due to the copy-on-write feature of AUFS, a new file is created to hide the original file and a pointer to this file is returned to the application. The following operations

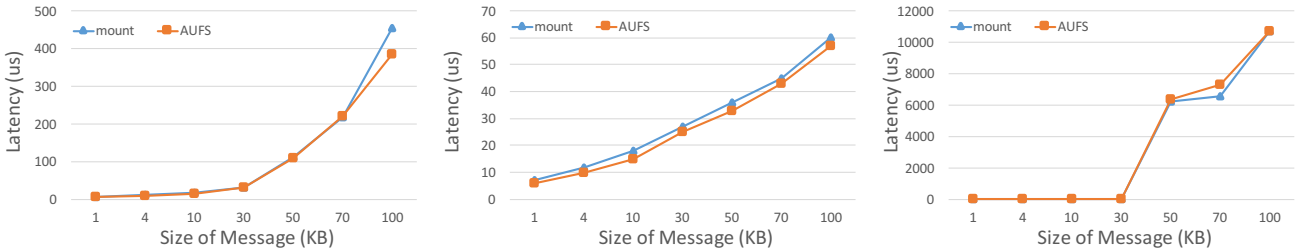


(a) Receive: Mean (b) Receive: Median (c) Receive: 99^{th}-percentile



(d) Send: Mean (e) Send: Median (f) Send: 99^{th}-percentile

Figure 3: Mean, median, and 99^{th}-percentile latency when server receive and send data.



(a) Mean (b) Median (c) 99^{th}-percentile

Figure 4: Mean, median, and 99^{th}-percentile file write latency using mount and AUFS.

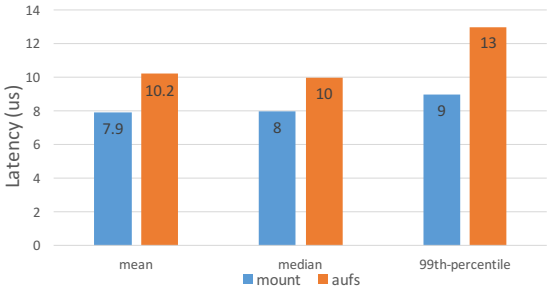


Figure 5: Latency Comparison between mount and AUFS when opening a file

are just like the normal operations on a disk file, and there is no fix-length latency degradation as in the network case.

However, there are cases reported that operating on files involves a significant performance degradation. Considering the feature of AUFS, we guess that the latency lies in the operation of opening a file. To confirm our guess, we carry out another group of experiment where the application continuously open and close a file 1,000,000 times. We note down the time required to open the file and the comparison is shown in Figure 5.

From Figure 5, we can find that there is a huge difference between using mount and AUFS concerning the open time of a file. This is because when using AUFS, since a Docker container is implemented combining several layers together, it must perform several additional functions to decide which layer the file is in. Sometimes it even has to copy another version of an existing file to perform the copy-on-write operation. These costs are huge compared to the time to open a file on the bare metal. Also, when first time writing to an existing file in the container, the larger the original file is, the longer the operation latency will be, which is caused by the cost of copy-on-write feature.

Building an image over another is a very convenient feature of Docker. You can simply add some files or run '**apt get**' command to install applications to the new image. All modifications are done in a new copy-on-write layer based on the original image layers. If a new file share the same name as an existing file, the original file will be hidden and only the newly added file can be seen by user.

Just like open operations which take extra time due to locating the file in multiple layers, programs like '**ls**' which involve traversing the directory have to go through unneeded hidden files. To prove this, we build images from the official **ubuntu** image. Each time a new image is built, we add linux kernel source code directory (53.9MB) to the **/home** directory of the previous image. We build 32 such images, each has a new linux kernel source code directory located in **/home** hiding the original directory. In each directory, we

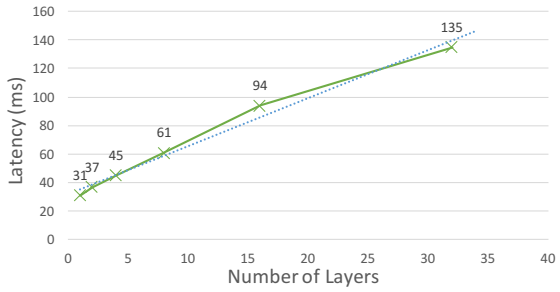


Figure 6: ls -R latency measurement

run the '**time ls -R /home > > /dev/null**' command to note down the time consumption. Each experiment is done 100 times and we choose the mean measurement of the results. The outputs are shown in Figure 6. It is easy to observe from the figure that with the increment of the number of layers, latency is increasing too. Also, the line is almost linear except for some vibration. This might because the **ls -R** command will go through each layer of the **/home** directory. The reason why intercept is not zero is because the additional cost of output messages to **/dev/null** and there exists branch cut in each scan of the hidden layer to avoid too much additional time cost.

4. RELATED WORKS

Several institutions and researchers have published related performance evaluation work on Docker. Most of them focus on throughput, while a few are concerned with latency. Kavita[1] finds that the overall density of containers on a machine is highly dependent on the most demanded resource. He also concludes that VMs have significantly higher overheads than containers in I/O and memory. Eder[7] did a very simple work using kernel bypass[19]. He concluded that applications running in a container does not have a obvious impact on its network latency performance. He uses *OpenOnload* together with *netperf* to realize the bypass. The results are shown and compared with their average, mean and 99^{th}-percentile round trip latency. From that report, he concludes that there are almost no performance degradation using Docker container. However, this test might only suitable for private Docker cloud but not public Docker cloud. This is because kernel bypass requires direct access to NIC, which has potential security problems in public cloud since one can modify the content of other containers as long as they will. However, in a private Docker cloud, kernel bypass can be a very good choice. Conventionally, once a packet is sent, it has to go through user space, kernel space and finally arrive at the NIC. With kernel bypass, the packet can directly sent from user space to NIC, which saves some time.

Recently, researchers from IBM[8] show that Docker container has a significant impact on overall network performance. The report uses *nuttcp* to measure throughput and also *netperf* to gauge latency. The report shows that there is a over 80% degradation on network round trip latency and also consumes more CPU cycles transferring a single byte using Docker container than native. So why there exists such a big difference between Eder's work and IBM's report? The key point lies in the fact that one uses kernel bypass while the other doesn't. Using kernel bypass in a Docker container leads to shorter latency than Docker bridge or Docker host and even faster than bare metal without kernel bypass. From the public cloud perspective, since it is not allowed for customers to use kernel bypass due to security reasons[6], IBM's work is more valuable in this case. However, IBM's report only uses a single group of comparison. It doesn't incorporate more comparison groups to further develop the relationship between round trip latency and other variables like the size of each packet transmitted.

5. CONCLUSION AND DISCUSSIONS

Docker is not so real-time-latency-friendly. Although –**cpu-shares** is a good choice as it doesn't cause a huge tail latency, in public cloud, many applications share a single CPU, using –**cpu-shares** can lead to a single application consumes more CPU than it is allowed to use when other applications are not running on this CPU. This is good for the consumers, but not fair since a consumer enjoys more resources than he has paid. Also, when service providers continuously allocate and de-allocate containers on a CPU, applications running on that core will go through performance vibrations. While using –**cpu-quota**, it is suitable for those CPU intensive works like many scientific workflow and HPC applications[34], but not those real time interactive services due to the possible long tail latency.

Docker assumes that users traditionally run applications on a desktop version of Linux, and thus enable the CFS while disable the real time one. There are two way to let the servers have this additional real-time feature. The first one is to add options for Docker to support real-time scheduling. The other is more difficult but brings a lot good – implement another scheduling that takes both throughput and real time performance into consideration, and we won't need to reboot the machine to change the setting of CPU scheduler.

Many applications involve small size of messages, like Redis Cache[36] and NoSQL databases[30]. When using Linux bridge to transmit message, it takes much more additional time due to short message size and high throughput, thus reducing the overall performance. But for applications like web gallery, since a photo is usually more than 1MB, the fix-length latency slowdown is overwhelmed by the system vibration, thus being less important in these cases. When using Linux bridge together with small size messages, it's not so latency-friendly since the extra time consumption is relatively large. We hope that a new method of transmitting information between Docker containers on different machines can be implemented to reduce this overall latency.

Some applications include many file operations. If the application simply open several big files one time and continuously write and read them, then we can neglect the disk I/O overhead since operating an open file using AUFS is just like natively. However, if the application involves huge number of file open operations, it can cause a huge extra latency (over 20%). So applications running in Docker should not incorporate too many open operations and sometimes the design of the application has to be changed to suit Docker, thus bringing the programmers a lot of trouble.

Another thing we should bare in mind when using Docker is to reduce overall layers. Operations like traversing the file system involves going through hidden files in the low layers of AUFS, and we don't need these useless additional cost. One way to avoid this is to merge the different layers together and delete unnecessary files, either at container start time or image build time. However, additional time would be needed to merge the layers at these two stages.

6. REFERENCES

[1] K. Agarwal, B. Jain, and D. E. Porter. Containing the hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015.

[2] E. Amazon. Amazon elastic compute cloud (amazon ec2). *Amazon Elastic Compute Cloud*, 2010.

[3] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob. Overview of microsoft azure services. In *Microsoft Azure*. 2015.

[4] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[6] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE Int. Conf. on*, 2014.

[7] J. Eder. Accelerating red hat enterprise linux 7-based linux containers with solarflare openonload. technical report, red hat enterprise, 2015, 2009.

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE Int. Symp. On*, 2015.

[9] G. A. Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*, volume 368. 1992.

[10] M. Hasenstein. The logical volume manager (lvm). *White paper*, 2001.

[11] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *Advanced Information Networking and Applications (AINA), 2012 ieee 26th Int. Conf. on*, 2012.

[12] M. Helsley. Lxc: Linux container tools. *IBM devloperWorks Technical Library*, 2009.

[13] T. Hopper. Cumulative distribution function. 2014.

[14] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, 2011.

[15] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4), 2013.

[16] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *CLOUD COMPUTING 2015*, 2015.

[17] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of the 2007 workshop on Experimental computer science*, 2007.

[18] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symp. on Cloud Computing*, 2014.

[19] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conf.*, 2006.

[20] R. Love. Kernel korner: Cpu affinity. *Linux Journal*, 2003(111), 2003.

[21] P. Menage, R. Seth, P. Jackson, and C. Lameter. Linux control groups, 2007.

[22] D. A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conf.*, 2005.

[23] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *the 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2013.

[25] J.-S. Pendry, U. Sequent, and M. K. McKusick. Union mounts in 4.4 bsd-lite. *AUUGN*, 1997.

[26] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[27] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, 2007.

[28] F. Špaček, R. Sohlich, and T. Dulík. Docker as platform for assignments evaluation. *Procedia Engineering*, 100, 2015.

[29] E. SPECweb2009. Commerce workload, 2009.

[30] C. Strauch and W. Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[31] H.-M. Tseng, H.-L. Lee, J.-W. Hu, T.-L. Liu, J.-G. Chang, and W.-C. Huang. Network virtualization with cloud virtual switch. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th Int. Conf. on*, 2011.

[32] G. Tsirtsis. Network address translation-protocol translation (nat-pt). *Network*, 2000.

[33] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. 2003.

[34] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro Int. Conf. on*, 2013.

[35] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *the 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2013.

[36] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.