Will Docker Container Affect Latency

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 uses different Docker configurations to compare their affect to latency performance. We conclude that the CPU quota configuration might lead to a long tail latency. The Docker bridge will lead to a fixed amount of latency degradation instead of a percentage fallen. Using AUFS will 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 important technologies around the world. It significantly shorten the process of packing, shipping and running any application[23]. 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 virtual machines (VMs), which use hardware-level virtualization, Docker containers employ system-level virtualization which share the same kernel with the host machine[28]. Many researches[16, 7] have proved that containers have a better performance in most cases than virtual machines. Due to these performance reasons, many companies are trying to move their services from virtual machines to containers [10]. However, Docker 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. Their request results should be returned very soon otherwise users might explain. Also, these services are dynamic. Datacenters are processing huge amount of data based on the user information and input and returning results 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[14]. In these cases, not only the throughput is of crucial importance to provide services to as many users as possible concurrently, median latency should also be taken into account to provide users with best interactivity. Each additional time cost in one of the service backend layers would increase the overall latency. If all of these separated services are added another layer like virtual machines, a millisecond latency might be amplified to several thousand milliseconds, thus greatly influence the overall performance of the service.

Tail latency is another problem one should care about[3]. In a Map-Reduce work[4], the request words might be processed by hundreds of machines. Each map machine would have to give their results to a central reduce machine. Thus, the reduce machine has to wait all the works to be done before it can finally move to the next step. In this case, once a simple map work is done over one second and all other works are done within one microsecond, the overall time consumption of this work would be limited to the slowest one. Assume that a task has one hundred sub tasks, each task with a 99% probability to finish in 1 microsecond and 1% probability to finish over 1 second. Then the overall performance of this job is 36.7% probability to finish within one second, which is a rather bad performance. We might draw a conclusion that the one-in-one-thousand case becomes a common case.

Virtual machines might be the source reason of these web services, since the additional layer might bring about significant performance cost [13]. The occurrence of Docker thus provide another choice for these customers. Many cloud center service suppliers like Amazon EC2 [1] and Microsoft Azure [2] provide container services in recent years. To simplify the deployment of applications, small companies are considering to use Docker cloud. Thus it is very important for them to know the trade off between the convenience and latency performance degradation of using Docker to deploy latency-sensitive applications.

The following sections will be organized as follows: Chapter 2 will introduce some background information about Docker technologies. Chapter 3 will carry out experiments and give analysis about their affection to service latency. We will talk about related works in Chapter 4 and give the final conclusion in Chapter 5.

2. BACKGROUND

2.1 Namespace

The mechanism of Linux Namespaces [34] is a way of isolating resources. System resources like process id (PID), inter-process communication (IPC), network, etc. are no longer global, but belongs to a particular namespace. The resource in each namespace are transparent to other namespaces. To create a new namespace, we only need to specific the corresponding flag when calling clone function. LXC [11] and Docker libcontainer [29] use this feature to realize resource isolation. Processes in different containers belongs to different containers. They are transparent to each other and will not interfere with each other.

2.2 CGroups

CGroups is the abbreviation of Control Groups [21]. It is a mechanism provided by Linux kernel that can limit, record and isolate the resources used by process groups. It was introduced to Linux Kernel 2.6.24 in 2007. CGroups can let you define groups for processes in the system and allocate resources to them, including CPU time, system memory, network bandwidth of a combination of them. You can monitor these CGroups, refuse the access of resources and even dynamically allocate resources in running groups.

2.3 Linux Bridge

Bridge mode is the default network setting of Docker, which make use of the Linux bridge feature [32]. When using this mode, each container will be allocated a network namespace and separate IP. When we start Docker server, it will create a virtual network bridge called docker0 on the host machine. All Docker containers created on this machine will be connected to the virtual network bridge. Virtual network bridge works like a physical switch, thus all containers on the host machine are connected in a two-layer network through a switch. In this network, each container should have a IP address. Docker will choose a private IP different from the host IP and sub net defined in RFC1918 and allocate it to docker0, and each container will choose an unused IP from this sub net.

2.4 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 [9] or RAID [8]. On the other hand, it enables the cooperation of a readonly branch and a writeable branch. Live CD uses this way to allow user do some write operations without modifying the OS image. Docker uses AUFS to build container images.

3. LATENCY EXPERIMENTS

In this section, we will first describe the overall method of carrying out the experiment. We will do experiments on various Docker configurations including CPU, network and file system.

3.1 Experiment Method

Since we focus on the real time services' latency, we incorporate a client-server model which tests the round trip latency for several operations. To make our experiment more like the real world services, we employs Apache Thrift[27] to let client side use RPC calls to call the server side and return the result. We use Python as our experiment language. For each call, we measure the latency between the time just before the call happened and just after the call ended. We might choose CDF[12], mean, median, 99^{th}-percentile and some other related measurements to dig into the interrelationship of these results.

3.2 Containerizing and Resource Limitation

In all cases, the server side 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. You can even run '**taskset -c**' in a container to let a process in container run on a certain CPU core.

3.3 CPU Configurations

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

3.3.1 Base Case

First, we will choose a base case as a comparison group to observe the effect produced by Docker container. We will run the server process natively to provide a reference. To make use of CPU affinity, we use '**taskset -c 3**' to let our process run on the target CPU #3 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 red line in Figure 3.3.1, and the mean, median and 99^{th}-percentile position of the measurements are also shown in Table 3.3.1. Most of the latencies are between 200 and 300 microseconds, and the average and median measurement are about 240 microseconds. However, there are still 1% latencies beyond 278 and these long latencies would be very com-

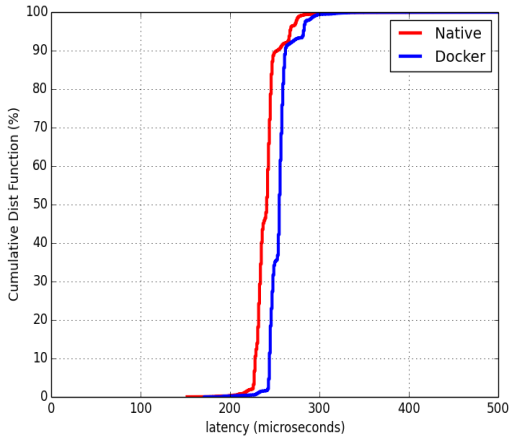


Figure 1: The CDF of latency using bare metal and Docker container

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

mon 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[36].

3.3.2 *Case 1*

To test whether Docker would have an impact on the latency performance, we run the server process in a Docker container, with '**–cpuset-cpus="3"**' set to realize CPU affinity and the '**–cpu-shares=1024**' as a default value set in CFS scheduler. When '**–cpu-shares=1024**' is set, assume that two containers have different shares and are running on the same CPU core. Container A has a share of 1024 and container B has a share of 512, if both containers are CPU-intensive, which means that they take almost all the time to do CPU calculation. The CPU time used by container A and container B would be 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 3.3.1, and the mean, median and 99^{th}-percentile position of the measurements are also shown in Table 3.3.1.

From the results drawn from the above two test cases, we might 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 3.3.1 and the CDF curves in 3.3.1, Docker container does not have a significant impact on the tail latency performance. Just like mentioned in the report of IBM, Docker containers do have several degradation concerning CPU performance. However, the performance 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 might conclude that when running a CPU-intensive application in a Docker container, the performance effect would be very small. Unlike virtual machines, 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 might need to be translated to several instructions to run on an ARM CPU. 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, the kernel needs to first check the namespace of the running process, thus the additional instructions would cause the extra latency.

$$S_v = f_p \times N_e + (1 - f_p) \tag{1}$$

3.3.3 *Case 2*

Apart from '**–cpu-shares**', there exist other parameters to 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 the '**–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 a same side effect as '**–cpu-shares**' that 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. The reason why we choose these parameters is because the minimum value of both these parameters are 1,000, and if their difference is two large, the measurements might be affected and hard to reach the conclusion. For each test, it is performed for 1,000,000 requests. We measure mean, median, 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. The results are shown in Table 2.

From Table 2, we might observe that the latency is increasing incredibly when CPU quota only counts for a small ratio of the total CPU period. From the quota 1000 to 4000, all mean, median and 99^{th}-percentile are decreasing and the number of test cases whose latency is greater than 1,000 us is also decreasing. Figure 3.3.3 also show 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 go to about zero at 4,000.

We make the following guess about the above phenomenon. 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 will have a force cut

Table 2: Measurements 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

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 has used up its quota during one period, it will sure to give up the CPU and wait until the next period comes. This would 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 our guess, we first compute the last column in Table 2. Assume we need in total time t_{cpu} to do all the computation, which means that the total time the process is running on the CPU. The cpu quota is t_q, and the cpu period is t_p. The total number of requests blocked by the cpu option n is computed as follows: $n = t_{cpu} \div t_q$. 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 may see that $t_{cpu} = t_{request} \times r$ is determined. So $n \times t_q$ must be also determined. Which is shown in our experiment as the last column. We can observe that for the case 1000, 1500, 2000, 2500, the product are around 100,000,000, which satisfy our formula. However, we 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 3.3.3 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.

3.4 Network Isolation

In this experiment, server side will send or receive various length data to or from client side. This is done by client calling the server with various length string parameter while server returns nothing. We choose message sizes 1KB, 10KB, 30KB, 50KB, 70KB, 100KB. These message sizes are chosen as the standard web message sizes from *SPECWeb2009* [30]. For each experiment, there are 1,000,000 requests.

Sever is hosted in a Docker container on one machine and client is running natively on another machine. Both server 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 the first one is to use '**-p portA:portB**', which means container uses Linux bridge to communicate to the outside world. It has its own IP address different from the one own by host machine. The **portA** used by the container is mapped to **portB** of the host machine, and it is working very similar to the *NAT* mechanism [33].

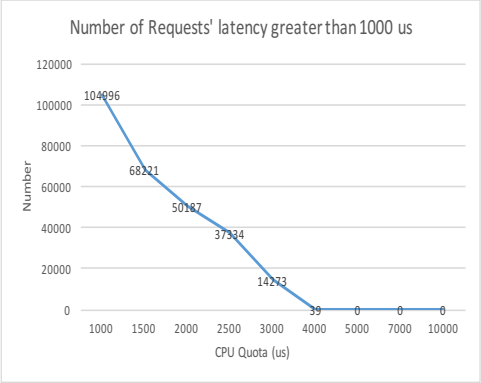


Figure 2: Number of requests' latency greater than 1000 us

3.4.1 Case 1: Server Receive Data

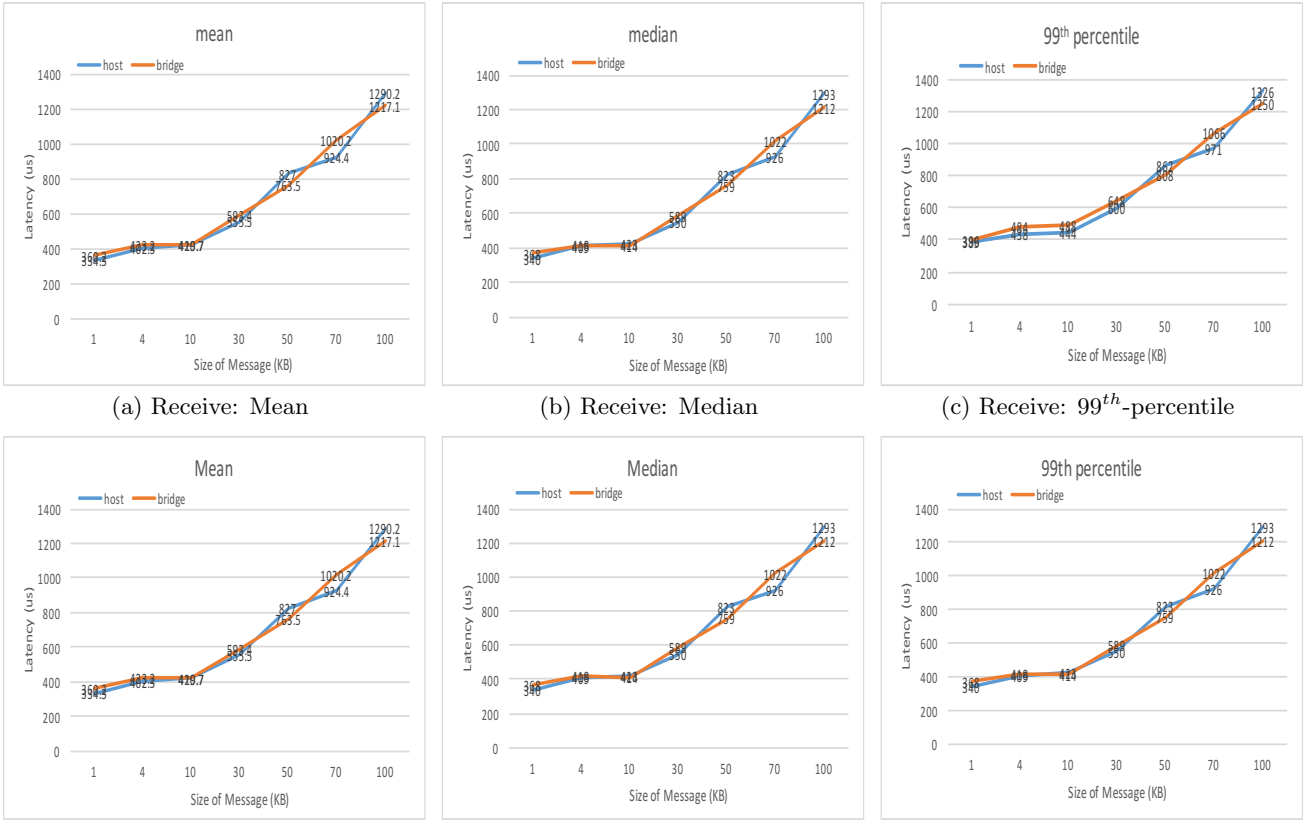
In this case, the client is sending data to the server using RPC calls and pass data through string parameters. We compare the mean, median and 99^{th}-percentile result between using '–**net=host**' and '**-p portA:portB**'. The results are shown in Figure 3(a-c).

3.4.2 Case 2: Server Send Data

In this case, the client is calling server with a single parameter indicating size and server is returning the corresponding size string. We compare the mean, median and 99^{th}-percentile result between using '–**net=host**' and '**-p portA:portB**'. The results are shown in Figure 3(d-f).

3.4.3 Analysis

As are shown in both figures, most of the latency using '**-p portA:portB**' is no more than 110% of the latency using '–**net=host**' according to each size group. In fact, when the file size becomes much larger, the result might be very close to each other, and the slowdown percentage is much



(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.

close to zero. Some of the comparison group even shows a phenomenon where Docker bridge performs better than Docker host. We explain this situation as being caused by the random vibration. This is because many other factors would cause the latency to change slightly, including CPU scheduling, preempting, network contend, and so on. These are not the parts should be talked about in our experiment.

However, IBM's report shows that there is a 80% slowdown using Docker bridge. So why our experiment draws a totally different result from IBM's research? I think there are several reasons to explain this. First, from our observe of the experiment, we might think the digression is a 'one-time degradation'. Which means that it only add a fix amount of latency to each transaction. In our experiment, when the size of the message is small, the impact of this fix amount of latency is very large – nearly 10%. However, when the message size becomes much larger, the impact of this latency is negligible, and the random vibration might be the dominant reason of the difference.

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

So where does this fix-size degradation come from? We dive into the source code of Linux bridge to see where this happens. Let us choose 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 will first check whether this message is used by more than one application or will it be multicasted. If not, it will not do any process including large amount of data copy. If so, the message will be copied several times and then send to each receiver, thus cost huge amount of extra latency. However, apparently in our experiment and IBM?s experiment, the message is only used by a single application, so there is no process of copying the message. Also, the message transmission is done by transferring a pointer to the message. So these extra functions will only cause a fix amount of extra time. This is why it affect so much in IBM?s report while in ours slightly.

3.5 File Operations Using AUFS

Many applications like SQL server and Hadoop might involve disk I/O operations. Disk operation latency of Docker container is thus another issue we should care about. In this experiment, we will not employ a client-server structure. Instead, only one application is hosted in a Docker container. The application will continuously write content of different sizes to a file. The length written each time are 1KB, 4KB, 10KB, 30KB, 50KB, 70kB, 100KB. We will compare two situations, the first one is to directly write messages inside the container, while the second is to write message to a file outside and mounted to the container using '**-v**' option. To make sure that the message 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 message flushed onto the disk. In each experiment group, the application will continuously write 10,000 times and we repeat this operation for 10 times. We will only discuss about the write case instead of the read case 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 might observe that there is almost no impact on performance using AUFS. This result is the same as the one drawn from IBM's disk I/O report. This phenomenon may be 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 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 might involve 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 3.5.

From Figure 3.5, we might find 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 using 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 might be 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.

3.6 Increasing AUFS Layers

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.

We have observed in the last section that there is no performance slowdown when operating on an opened file. However, the open operation would take extra time due to locating the file in multiple layers and the cost of creating a new copy-on-write layer. Since it might require extra time to locate files in all layers, we guess that when traversing a directory that has many layers, it might go through almost every hidden files in each layer instead of just the visible files.

To prove our guess, we build images from the official **ubuntu** image. Each time we build a new image, 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 run the '**time ls -R /home > > /dev/null**' command

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

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

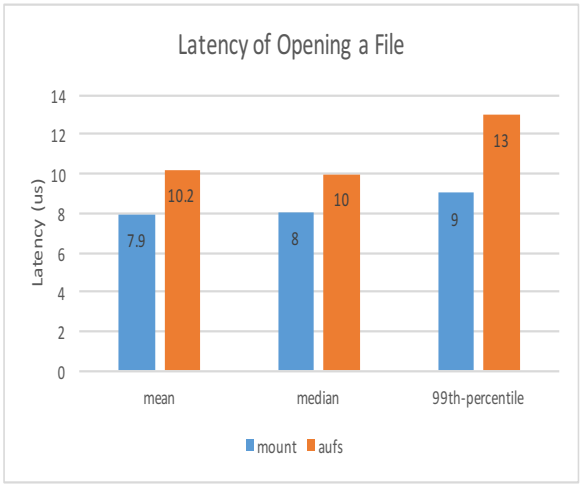


Figure 5: Latency Comparison between mount and AUFS when opening a file

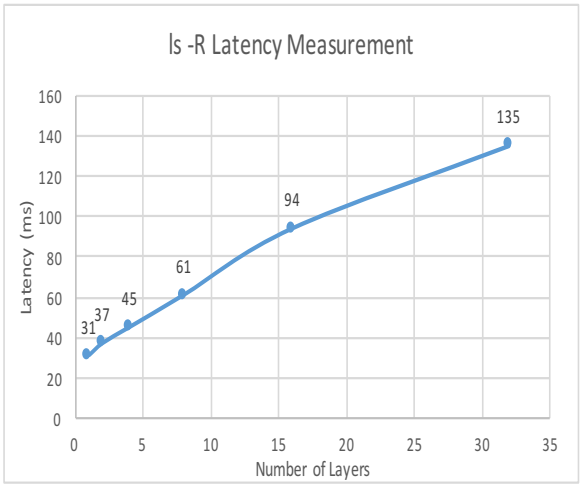


Figure 6: ls -R latency measurement

to note down the time consumption. Each experiment is done 100 times and we choose the mean measurement of the results. The output are shown in Figure 3.6. 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 are branch cut in each scan of the hidden layer to avoid too much additional time consumption.

4. RELATED WORKS

Several institutions and researchers have published related performance tests about Docker. Most of them are about throughput, while a few are concerned with latency. Eder [6] 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 used *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 concluded 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 might cause secure problems in public cloud since one can modify the content of other containers as long as they will. However, in private docker cloud, kernel bypass can be a very good choice. Conventionally, once a packet is sent, it has to gone 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 might save some time.

The famous IBM report [7] infers 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% digression 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 might lies in the fact that one uses kernel bypass

while the other doesn't. Using kernel bypass in a Docker container might lead to shorter latency than Docker bridge or Docker host without latency 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 [5], IBM's work might be 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

Docker might not be so real time latency-friendly. When using the CPU settings, although **−cpu-shares** might be a good choice since it will not cause a huge tail latency. In public cloud, you have to use a CPU to share multiple applications, using CPU shares will let 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 resource than he has paid. Also, when service providers continuously allocate and deallocate containers on that CPU, applications running on that CPU will gone through a large performance vibration. While using **−cpu-quota**, it is a good choice for those CPU intensive works like many scientific workflow and HPC applications [35], but not those real time works due to the possible long tail latency.

Since the Linux scheduler has two kinds of scheduling algorithms, one is real time [26] and the other is CFS [15]. 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 might be to add options for Docker to support to real-time scheduling. The other might be more difficult but will bring a lot good – implement another scheduling that takes both throughput and real time performance into consideration. And we will not need to reboot the machine to change the setting of CPU scheduler.

Many applications involve small size of messages, like Redis Cache [37] and NoSQL databases [31] . When using Docker bridge to transmit message, it will take much more additional time due to their 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-size latency slowdown might be overwhelmed by the system vibration, thus being less important in these cases. When using Linux bridge together with small size messages, it might not be so friendly since the extra consumption is relatively large. We hope that a new method of transmitting information between Docker containers on different machines implemented to reduce this overall latency.

Some applications include many file operations. If the application simply open several big files one time and continuously operate on them, then we can neglect the disk I/O overhead since operating an open file using AUFS is just like operating an open file natively. However, if the application involves huge number of file open operations, it might 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 might have 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. Some operations like traversing the file system might involve going through hidden files in the low layers of the AUFS. However, we may not need these additional cost since they are totally useless. 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. Then no needless files would be found when traversing the file system.

6. REFERENCES

[1] E. Amazon. Amazon elastic compute cloud (amazon ec2). *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[2] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob. Overview of microsoft azure services. In *Microsoft Azure*, pages 27–69. 2015.

[3] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

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

[7] 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 International Symposium On*, pages 171–172. IEEE, 2015.

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

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

[10] 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 international conference on*, pages 15–22. IEEE, 2012.

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

[12] T. Hopper. Cumulative distribution function. *Month*, 2014.

[13] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.

[14] 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):219–230, 2013.

[15] M. T. Jones. Inside the linux 2.6 completely fair scheduler. *IBM Developer Works Technical Report*,

2009, 2009.

[16] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *CLOUD COMPUTING 2015*, page 180, 2015.

[17] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 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 Symposium on Cloud Computing*, pages 1–14. ACM, 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 Conference, General Track*, pages 29–42, 2006.

[20] R. Love. Kernel korner: Cpu affinity. *Linux Journal*, 2003(111):8, 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 Conference*, pages 407–414, 2005.

[23] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[25] J.-S. Pendry, U. Sequent, and M. K. McKusick. Union mounts in 4.4 bsd-lite. *AUUGN*, page 1, 1997.

[26] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

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

[28] 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, pages 275–287. ACM, 2007.

[29] F. Špaček, R. Sohlich, and T. Dulík. Docker as platform for assignments evaluation. *Procedia Engineering*, 100:1665–1671, 2015.

[30] E. SPECweb2009. commerce workload, 2009.

[31] C. Strauch, U.-L. S. Sites, and W. Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[32] 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 International Conference on*, pages 998–1003. IEEE, 2011.

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

[34] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. page 213. IEEE, 2003.

[35] 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 International Conference on*, pages 233–240. IEEE, 2013.

[36] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, 2013.

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