

SHANGHAI JIAO TONG UNIVERSITY

**学士学位论文**

BACHELOR’S THESIS



论文题目： Latency Impact of Docker Containers: A Closer Look

学生姓名: 章佐铭

学生学号: 5120309626

专 业: 计算机科学与技术

指导教师: 李超

学院(系): 电子信息与电气工程学院

**二甲醚清洁燃料均质压燃燃烧数值模拟研究**

摘要

均质充量压缩着火（HCCI）燃烧，作为一种能有效实现高效低污染的燃烧方式，能够使发动机同时保持较高的燃油经济性和动力性能，而且能有效降低发动机的NOx和碳烟排放。此外HCCI燃烧的一个显著特点是燃料的着火时刻和燃烧过程主要受化学动力学控制，基于这个特点，发动机结构参数和工况的改变将显著地影响着HCCI发动机的着火和燃烧过程。本文以新型发动机代用燃料二甲醚（DME）为例，对HCCI发动机燃用DME的着火和燃烧过程进行了研究。研究采用由美国Lawrence Livermore国家实验室提出的DME详细化学动力学反应机理及其开发的HCT化学动力学程序，且DME的详细氧化机理包括399个基元反应，涉及79个组分。为考虑壁面传热的影响，在HCT程序中增加了壁面传热子模型。采用该方法研究了压缩比、燃空当量比、进气充量加热、发动机转速、EGR和燃料添加剂等因素对HCCI着火和燃烧的影响。结果表明，DME的HCCI燃烧过程有明显的低温反应放热和高温反应放热两阶段；增大压缩比、燃空当量比、提高进气充量温度、添加H2O2、H2、CO使着火提前；提高发动机转速、采用冷却EGR、添加CH4、CH3OH使着火滞后。

关键词：均质充量压缩着火，化学动力学，数值模拟，二甲醚，EGR

**OPTICAL PROPERTIES OF COMPOSITE MATERIALS MADE FROM HYDROGEL AND BUTTERFLY WING SCALES**

**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.

**Key words:** Biomedical Sensor, Lepidoptera scales, Nature photonics, Optical sensor/indicator, Electric field sensitive, pH condition sensitive, Interpenetrating polymer network

**Contents**

1. Chapter One Introduction 1
2. Chapter Two Background & Motivation 4

2.1 Linux Bridge 4

2.2 AUFS 5

1 Chapter I ---------------------------------------------------------------------------------------1

* 1. Section------------------------------------------------------------------------1

1.1.1 Subsection-------------------------------------------------------------------------1

…

References ----------------------------------------------------------------------------------------2

# 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 ([26], Merkel D., 2014: 2.). 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 virtual machines, which use hardware-level virtualization, Docker containers employ system-level virtualization and share the same kernel with the host machine ([30], Soltesz S., 2007: 275.). Many researches 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 virtual machines to containers ([13], He S., 2012: 15.). However, containers do add additional layers compared to bare-metal hardware, which leads to certain degree of performance degradation.

Docker was born to replace virtual machines to some extent. Nowadays, the widely known Infrastructure-as-a-Service (IaaS) platforms like Amazon EC2 uses virtual machines to run applications like cache and database. Most of these applications not only focus on throughput, but also favor real-time low latency. However, most related work of Docker focus mostly on containers’ influence on throughput instead of the latency degradation. Since Docker provides many choices of resource isolation, in this paper, we will do research on how these parameters will affect the latency performance of real time applications.

## Motivation

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. Data centers 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 ([27], Nishtala R., 2013: 385.) and tens of thousands of index servers to do a Bing search ([17], Jalaparti V., 2013: 219.). 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 ([6], Dean J., 2013: 74.). In Map-Reduce task ([7], Dean J., 2008: 107.), 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 CTO of GigaSpaces claimed a list of interesting phenomenon. He pointed out that latency is a serious matter that can lead to huge profit lost in many companies. Every 100ms of latency would cost Amazon 1% of lost in sales. Also, every extra of 0.5 seconds wasted on generating a search page can drop Google's network traffic 20%. Moreover, if a broker's electronic trading platform can not catch up others’ and gets 5 milliseconds behind the competition, they would lose $4 million in revenues per millisecond. Even these latencies seem relatively small, people hate waiting. They feel repulsed by these less interactive services, quickly click away and finally do other things like turning to the opponents' services. People are talking about how to scaling up the capacity of their services, but they sometimes neglect the importance of building low-latency ones. Service suppliers should try their best to decrease service latency, increase interactivity, and finally lower the customer defection rate ([4], Colgate M., 1996: 23.).

Despite the increasing need of virtualization technologies to decrease latency, Docker doesn't seem to be focusing on this part. In fact, although Docker provides us with a simple way to deploy applications, the technologies it employs are not so latency-friendly. Like what has been mentioned in IBM's technical report ([10], Felter W., 2015: 171.), Docker containers take the Linux bridge as the method of network isolation. However, it shows that Docker containers even perform worse in transmission throughput and also have a longer network latency compared to KVM ([18], Kivity A. 2007: 225.). Actually, all technologies used by Docker are not new ones. Most of them have already existed since the year of 2007, and the concept of container also occurs at that time ([30], Soltesz S., 2007: 275.). Docker container is just a combination of these simple technologies. With the concept of Docker images and the emergence of Docker Hub, Docker quickly win the eyes of system deployers. However, since most of these technologies are provided by old versions of Linux Kernel and they focus on resource isolation instead of latency, it will take Docker a long time to find ways to replace those inefficient technologies and thus decreasing latency lost.

Unlike Google or Facebook, which has dedicated data centers for their services, most small companies cannot afford the cost of hardware and the following maintenance. They can only deploy services on cloud centers like Amazon EC2 ([3], Shankar S., 2009.) and Microsoft Azure ([5], Copeland M., 2015:27.). As we have mentioned above, these cloud centers use virtual machines to provide hardware virtualization and have a significant performance cost compared to bare metal. The occurrence of Docker thus providing another choice for these customers. Many cloud center service suppliers provide container services in recent years. To simplify the deployment of applications, these small companies are considering to use Docker cloud. Since the additional layer of virtual machine brings about significant performance lost ([16], Huber N., 2011:563.) and is part of the source reason of long latency, 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.

Previous researches mainly focus on the throughput of CPU, memory and I/O. Some of them talks about memory footprint and the latency brought about by Docker network bridge and methods to shorten this latency. However, these methods are not suitable for public cloud. This paper is intended to solve the problem from the customers’ perspective. Although customers can not the change the services provide by cloud service suppliers, they have the choice to choose their start up configurations and the policies to build their services. We focus on the effect of Docker containers on latency with respect to various configurations. We analyze the effect of Docker container configurations to web service situation. This analysis provides customers with the potential latency cost of Docker containers and helps them to build services with the awareness of these possible degradation.

## Related works

The appearance of Docker is in the year of 2012. However, the history of Linux containers is more than just several years. In the year of 2001, as an initial implementation of “virtual private servers”, Linux-VServer project came into existence. However, it has never been merged to the mainstream Linux operating systems. There are other Linux containers like OpenVZ, which is mainly used to host web applications, that also doesn't share a position in the mainstream Linux. Finally, in the year of 2007, as many features including namespaces and chroot are added to Linux kernel, Linux Containers (LXC) was finally added to the mainstream Linux and becomes the most widely used containers since then.

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. Researchers from IBM ([10], Felter W., 2015: 171.) use KVM as a representative hypervisor and Docker as a representative container and compare the performance of bare metal, virtual machine and container. They use various workloads to stress CPU, memory, and I/O resources. They have found that containers overwhelm virtual machines in almost every case concerning throughput. After these workloads, the research also shows experiments on some real world applications including MySQL and Redis Cache. Both these real world applications exhibit a better performance for Docker containers than virtual machines. The report also reveals that the startup time of KVM is 50x slower than Docker containers. Kavita ([2], Agarwal K., 2015: 8.) tries to increase the number of containers on a host machine with the same kind of workload. He finds that the overall density of containers on a machine is highly dependent on the most demanded resource. He also concludes that virtual machines have significantly higher overheads than containers concerning memory footprint. He uses Kernel Same Page Merging (KSM), a memory de-duplication technology, and finds a 60 times opportunities to lower the memory cost of a virtual machine compared to containers. Canonical does a similar work as Kavita comparing LXD and KVM. All these virtual machines are running Ubuntu 14.04 operating system. Experiment measurements reveals that on a host machine containers have 14.5x higher density than virtual machines. The density bound is mainly caused by memory limitation. Also, the work shows a 57% reduction in network latency than virtual machines. But it doesn't show that the LXD is using Linux bridge technology.

Eder ([9], Eder J., 2015.) did a very simple work using kernel bypass ([22], Liu J., 2006: 29.). He concluded that applications running in a container does not have an 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 99th-percentile round trip latency. From that report, he concludes that there are almost no performance degradation using Docker containers. However, this test is only suitable for private Docker cloud rather than public Docker cloud. This is because kernel bypass requires direct access to *Network Interface Card* (NIC), which has potential security problems in public cloud since one can modify the content of other containers as long as they want. 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 be directly sent from user space to NIC, which saves some time.

On the other hand, IBM’s report ([10], Felter W., 2015: 171.) shows 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 over 80% degradation on network round trip latency and also consumes more CPU cycles transferring a single byte using Docker containers than natively. 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 ([8], Dua R., 2014: 610.), 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.

There are also works showing that containers don’t have a significant performance lost concerning network performance compared to bare metal. Xavier ([37], Xavier M. G., 2013: 233.) uses Xen as an representative of virtual machines and compare its performance to various kinds of containers including LXC, OpenVZ, and VServer. He presses these technologies with various kinds of well-known benchmarks and draws to the conclusion that containers outperform virtual machines in every high performance cases. However, different from other researches, his work doesn’t show a high performance lost in I/O cases. This is because these old-type containers don't employ technologies including AUFS or Linux bridge. In our work, we find that these two technologies are key reasons to the latency lost in Docker containers.

## Contributions

This thesis makes the following contributions:

First and foremost, we investigate the latency slowdown caused by CPU configurations. We explore and compare two kinds of configurations, the first one is CPU shares and the second one is CPU quota. We find that CPU shares almost has no impact on the performance lost while it cannot limit the CPU usage of a container when only one container is running on a CPU. On the other hand, CPU quota can successfully limit the CPU usage of a container, but it has the potential to lead to a rather long tail latency.

Secondly, we build a research platform to evaluate the network latency performance of Docker containers. The platform employs a client-server architecture. The server is hosted in a Docker container and we measure the round trip latency of a client request. We choose two situations, the first is server sending data and the second is server receiving data. We compare using Docker host and Linux bridge configurations. We draw the conclusion that containers do have some impact on the performance lost using Linux bridge compared to directly using the host machine’s port. However, this performance is not as exaggerated as described in IBM’s report that Docker bridge causes 80% performance lost compared to bare metal. In fact, this is more like a fix-length performance degradation. The smaller the transmitted message is, the more relatively significant the performance slowdown is.

Thirdly, we analyze the latency impact of Docker using AUFS to do file operations. We find that Docker containers do not have impact on performance when writing an existing file. However, when it comes to operations related to the file system instead of a single file, situation changes. Operations like opening a file would lead to extra latency due to locating the file in multiple AUFS layers and the extra cost of creating the copy-on-write layer. When listing a directory which has many hidden files in low layers, the hidden files will also be scanned instead of just the superficial ones. The total scanning time is linear to the sum of the number of hidden and superficial files.

## Organization of This Paper

The following sections are organized as follows: Section 2 introduces some background information about Docker related 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.

# Background

## Docker

Docker, an open-source advanced container implemented by dotCloud, is making a huge impact on the filed of cloud computing. Docker wraps the whole runtime environment into the unit of Docker containers to divide and schedule resources. It is a platform designed for developers and system administrators to build, ship and release distributed applications. It is also a cross-platform, portable and easy of use container solution. Docker is implemented in Go language and its source code is hosted on Github. Docker provides developers a fast and automatic way to deploy applications. It incorporates many operating system-level Linux kernel technologies like namespaces and control groups to provide resource isolation, resource limitation and security.

### Container and virtual machine

Many people are familiar with virtual machines. In our everyday life, we might run Ubuntu Linux on Windows PC using VMware or play games in a Windows virtual machine using Parallel Desktop on Mac OS. These are all hardware-level virtual machines. They use software to simulate the instructions that are used by the operating system run in the virtual machine as if they are just operating on the bare metal. Hardware-level virtual machines like VMWare and Parallel Desktop are more used on personal PCs, while Xen and KVM are more used on servers and public/private clouds.

On the other side is the container-based virtualization, which is also called operating-system-level virtualization. OS-level virtualization’s coming into people’s eyesight is because of the *chroot* mechanism introduced in Unix-like operating systems. Chroot was traditionally used to run multi services in a multi-user environment and leave one not affecting each other’s running applications on the same machine. Began from chroot, a lot of prototype containers were implemented like the famous Linux Container (LXC), FreeBSD jail and OpenVZ. Although Docker became famous in recent years, container is not a new concept since the appearance of container technologies mentioned above can be traced back to the year of 2007. It is the mature of cloud environment, the spring of web applications, and also the completeness of Docker environment that makes Docker so popular. Unlike hardware-level virtual machines, OS-level virtual machines use the host operating system, but each container only has access to its own files. A container contains all the files it needs to run a program, it has its own libraries, **/boot** directory, **/usr** directory, **/home** directory and so on. The whole running container can even have only a single file as long as the binary program you want to run has no dependency. Also, programs running inside a container cannot see processes outside the container, including those running directly on the native host and also applications running in other containers. This is implemented using the Linux namespace mechanism. Containers also use chroot mechanism to limit resources like CPU usage, memory usage and I/O usage.

IBM’s report ([10], Felter W., 2015: 171.) shows that in almost all cases (except for network latency), Docker containers performs better than virtual machines and is very competitive to native. This is because virtual machines have to use software to simulate hardware. It can cost several instructions to simulate a single instruction using software, thus dramatically slowdown the performance. While on the other hand, container processes are just running using the host operating systems, except for the fact that they are isolated and resource limited. Containers also exhibits a much shorter start time than virtual machines, while most virtual machines use tens of seconds or even several minutes to start, it only takes a container several seconds or even less than a second. It also takes much less memory footprint to run processes than virtual machines because containers do not need those extra files to start a whole operating system.

Although containers do bring us about better overall performance than virtual machines, there also exist a lot of problems to be solved. The most important part is security. As I have mentioned above, containers use the host operating system, which means that any security threats in the host operating system can be made use of to attack the host and other containers running on the same machine. The host operating system can see everything running inside a container, and the content is not private for users. Once a container process successfully takes the administration of the host operating system, it can operate all other container processes at its will.

Another current limitation for containers is that most of them can only run on Linux. Since running a process in container requires the host operating system to be Linux, running Linux processes on Windows systems may not be practical. Also, since containers do not have the authority to modify its host operating system, it is not possible to load kernel modules dynamically, thus limiting its ability.

### Resource Isolation Using Namespace

The mechanism of Linux Namespaces is a way of isolating resources. System resources like PID, 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 and Docker libcontainer 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.

Traditionally, many resources are organized globally in Linux and many other Unix-like systems. Although it allows to allocate some authorities to certain IDs and the root UID 0 user is allowed to do almost everything, other user IDs will be limited. For example, user UID n is not able to kill the processes of UID m (m ≠ n) if m is not a descendant of n, but they are allowed to see each other. Sometimes, we might want processes and users not be able to see each other, here Linux namespace comes.

Using KVM cannot always allocate resources properly. Each user needs a separate kernel and a full suit of user applications. Linux namespace allows to run a single kernel on a physical machine and all resources are abstracted by namespace. This allows us to put a set of processes into containers and each containers are isolated from each other. Some certain shares are also allowed among containers to lower the isolation of them.

Take the PID namespace as an example. Suppose globally there are 30 processes running on the machine, processes 11 to 20 belongs to container A and processes 21 to 30 belongs to container B. Here, container A sees processes 11 to 20 as processes 1 to 10, while container B sees processes 21 to 30 as processes 1 to 10. The root user can see all processes 1 to 30 as their global PIDs. A can not see processes in B and it is not able to see processes 1 to 10 owned by the root user, so cannot B. Thus the PID isolation is implemented.

There are two ways to create new namespaces. When using fork or clone system calls to create new processes, there are certain options to choose to share the same namespace with parent or create a new namespace. We can also use *unshare* system call to separate namespace from one’s parent process.

### Resource Limitation Using CGroups

CGroups is the abbreviation of Control Groups. 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 and a combination of them. You can monitor these CGroups, refuse the access of resources and even dynamically allocate resources in running groups.

CGroups mainly provide the following functions:

**Resource limitation:** Limit the resource usage, like memory usage upper bound and cache limitation of file system.

**Prioritization:** Control the priorities of processes when dealing with resources.

**Accounting:** Monitoring the processes, mainly used to account.

**Control:** Suspend, resume or run processes.

There are many concepts in CGroups. *Task* refers to a process in the system. *Control group* is a group of processes divided by a certain standard. The basic unit of CGroups are implemented by control groups. A process can either be added to a control group, or can be moved from one control group to another. The processes in a control group can use resources allocated to this group. They are also under the limitation of the resources of this certain group. Control groups can be organized as a *hierarchical* format, i.e., a control group tree. The son node of a control group inherits and is limited by its parent node’s attributes. There are also *subsystem*s in CGroups, and one subsystem is a resource controller. For example, the CPU subsystem is used to control CPU allocation.

When creating new control groups, there exists a lot of limitations. Each time a new hierarchy is created, all tasks in this subsystem is the default control group of that hierarchy, i.e., root group. A subsystem can be at most attached to one hierarchy and one hierarchy can be attached many subsystems. A task can be the members of many control groups, but none of these groups can be in the same hierarchy. A newly forked task will automatically become its parent’s control group member. It can then be moved to different control groups on demand. As is shown on Figure.

## Related Linux Technologies

### Linux Bridge & Veth Pair

*Bridge mode* is the default network setting of Docker. When using this mode, each container will be allocated a network namespace and separate IP. When we start Docker daemon, 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. For example, Docker will choose the 172.17.0.0/16 subnet and it will allocate 172.17.42.1/16 to docker0 bridge. You can use *ifconfig* to monitor docker0 since its working as a virtual network interface card. Assume that the IP address of host machine is 10.10.101.105/24, the topology of a single machine environment is shown in Figure.

To create the above network settings, Docker will first create a pair of virtual network called *veth pair*. Veth always occurs in pairs. They form into a data tunnel and the information comes into one end will go out to the other end. So veth endpoints are usually used to connect two network devices. Docker names one of the veth pair as *eth0* and put it inside the newly created container. The other end is put on the host machine, named like veth23f6. This network device will also be added to docker0 network bridge. It can simply check by using the *brctl show* command. Docker will then choose an IP from the subnet and allocate it to the newly created container.

In bridge mode, all containers connecting to the same network bridge can communicate with each other. Containers can also communicate with the outside world. This is implemented by modifying the *Iptable*. Iptable will help do the transmission that all packets sent to eth0 of Docker container will be sent out to docker0 first and then the outside world. All packets sent to docker0 will first decide which container it belongs to and finally sent to the corresponding eth0 of the Docker container.

### Scheduler

The *scheduler* class was introduced in Linux version 2.2. It has implemented real-time, non-preemptive and non-real-time task scheduling policies. In Linux version 2.4, a relatively simple scheduler was implemented and it’s running in O(N) time complexity. Earlier Linux 2.6 scheduler is called O(1) scheduler. It aims to solve the problem of O(N) scheduler’s having to iterate all the task queue to decide the next task and it is more efficient. O(1) scheduler is easy to expand and more iterative. However, the implementation of O(1) scheduler is very heavy and need huge amounts of code. It is hard to understand and thus difficult to manage.

In Linux Kernel version 2.6.21, the scheduler implemented by Kolivas, called *Complete Fair Scheduler* (CFS) was incorporated. Its main idea is to provide the fairness in term of providing CPU time to different tasks. When CPU time allocated to a certain task loses balance, it should be allocated enough time to be scheduled on the CPU.

To realize fairness, CFS maintain a time quantity in a place called *virtual runtime*. The less virtual runtime is, the less time a task has been allowed to run on CPU, which means that it needs more time to be scheduler to CPU. CFS also include the sleep fairness concept to make sure that those not running tasks (e.g. waiting I/O) will be allocated a certain amount of CPU time when they finally need.

However, different from former Linux schedulers, CFS is not maintaining tasks in a running queue. Instead, it maintains a *red-black tree* (RB-tree) based on time priority. RB-tree has many interesting and useful attributes. First, it is self-balance, no route on the tree is longer than twice of other trees’. Secondly, it runs at a O(log n) speed (n is the number of tree nodes), which means that you can insert and delete tasks fast.

Tasks are saved in the RB-tree according to their time. The more time a task needs to be scheduled on a CPU, the more left side it is on the RB-tree. For the sake of fairness, the scheduler will choose the left corner task on the RB-tree to be scheduled next. Tasks will be then added its real run time. At the same time, since its virtual runtime has been increased, it is moved several steps right on the RB-tree. Thus, all tasks on the RB-tree is chasing each other and they form a dynamic balance on CPU scheduling time.

Aside from CFS, which mainly aims at interactivity for desktop users, there are real time requests for server uses. Linux has implemented two of them. The first one is *SCHED\_FIFO*. It implements a first-in-first-out algorithm. Once a task starts to be executed, it will continue to go on until it gives up CPU at its will, blocked, or preempted by higher priority real-time tasks. When two tasks are of the same priority, they are scheduled according to the first-in-first-out principle. The other algorithm is SCHED\_RR, and it has the concept of time slices. Process with the same priority once uses up its time slice, it will give way to the next task in the queue, and it is then assigned to the tail of the waiting queue.

### Tail Latency

Nowadays, not every service is so simple like a client and server mode. If a certain client sent a request, the server side might just be an interface. The real work is handled by the huge data center in the background. For example, in a Google search operation, the request words might be processed by hundreds of machines doing Map-Reduce work. Each 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.

The concept of tail latency was first proposed by Google. It has brought great attention since then. To observe the tail performance of Docker in this paper, we use another measurement: 99% performance, or the *99th-percentile*. This is used to show the tail latency performance of our work. Since Docker containers are often used in the cloud environment, we add this to our research to solid our research.

# Latency Characterization

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

## Apache Thrift

Thrift was first started from the famous company Facebook ([29], Slee M., 2007: 8.). In the year of 2007, it was submitted to Apache and became an open source project. At that time, Thrift was used to solve the problem in Facebook that various systems needed to transmit huge amount of data while the language environment are different. Thrift can support many kinds of languages like C++, C#, Cocoa, Erlang, Haskell, Java, Perl, PHP, Python, Ruby. Thrift can work as a high performance communication middleware among different languages. It supports the serialization and many kinds of RPC services. Thrift is suitable for building large scale data communication and storage tools. In large systems, the inner data communication has obvious advantage over JSON and XML.

Thrift incorporates a client and server architecture. It has its inner Transport Protocol *TProtocal* and transport standard *TTransports*. Thrift is described as the following structure shown in Figure.

Thrift enables you to choose the transport protocol between client and server. The transport protocols are generally divided into text and binary. The latter one is with better performance. Compared to XML and JSON, its packet size is very small. It also has affinity to high concurrency, large data and multi language environment.

## 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 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.

## Containerizing and resource limitation

There are a lot of container resource limitation parameters:

**cpu-period**: This means the period for the process to schedule the Docker process. It is used together with the **cpu-quota** parameter. The unit of these two parameters are both microseconds. When these parameters are set, it means that the processes can not use longer than **cpu-quota** time during each **cpu-period** time duration. Once the process reaches its time slice, it will be cut off and not able to use the remaining time slice.

**cpu-quota**: As is mentioned above. **cpu-quota** together with **cpu-period** have a limitation that their minimum value should be no less than 1000 us. When **cpu-quota** is larger than **cpu-period**, it means that the container can use more than one CPU core resources.

**cpu-shares**: When this parameter is set, assume that two containers have different shares and running on the same 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 should be a ratio of 1024: 512, which is 2: 1.

**net:** There are four choices for this parameter. **bridge** uses the network namespace mechanism, which means container uses Linux bridge to communicate to the outside world. It has its own IP address different from the host machine. Socket used by the container are mapped to a socket on the host machine. Which is just like NAT mechanism. **host** means the container directly use the host network port and there is no network isolation. **container** let several containers run on a single machine share a same network namespace, which means they have the same IP address and contend for ports. **none** doesn’t mean the container has no network communication with the outside world. It just leaves the user to control the network settings.

**p a: b:** This is used together with **net=bridge** (the default network setting). Which means that the host’s port **a** is mapped to the inside container port **b**, and it’s acting like the NAT. All requests come to host’s port **a** will be sent to container’s port **b**, and all messages sent from container’s port **b** will be transferred to host’s port **a**.

**v a: b:** With this option, we can map the host’s files or directories to container’s file system. So we can bypass the AUFS mechanism and directly access the host files. For example, if we use **-v /home/username:/home**, when we enter the Docker container’s **/home** path, we can see all the files in the host’s **/home/username** directory. This just acts like the traditional Linux **mount** command. We can have access to the files and directories in **/home/username** directory. If we visit files in a container that are not mounted from host, we might open a new file which is copied from the original file and all the operations are done in this new file.

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 ([23], Love R., 2003: 8.) 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 ([20], Li C., 2007: 2) 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.

## 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.

### 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 ([15], Hopper T., 2014.) result is shown as the red line in Figure \ref{fig:cpucdf}. The mean, median and 99th-percentile position of the measurements are listed in Table \ref{tbl:cpubase}. 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 ([21], Li J., 2014: 1), and interference from other virtual machines or containers in the cloud environment ([39], Xu Y. 2013: 329).

### Case 1: Using CPU Shares

We run the server process with the **cpuset-cpus=“3”** setting to realize CPU affinity and the **cpu-shares=1024** as a default setting in CFS scheduler. 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 \ref{fig:cpucdf}, and the mean, median and 99th-percentile position of the measurements are also listed in Table \ref{tbl:cpubase}.

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 99th-percentile column in Table \ref{tbl:cpubase} and the CDF curves in \ref{fig:cpucdf}, 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 99th-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 \ref{eq:vmsv} ([25], Menascé D A., 2005: 407), 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.

### 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 99th-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 \ref{tbl:cpuperiod}.

From Table \ref{tbl:cpuperiod}, 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 99th-percentile are decreasing and so is the number of test cases whose latency is greater than 1,000 us. Figure \ref{fig:cpuquo} 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.

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 \ref{tbl:cpuperiod}. 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 \ref{fig:cpuquo} that when it comes to over 4000, the mean, median and 99th-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.

### CPU Interference

There is no meaning running a single process on a CPU core while at the same time limiting its available CPU resources. In the real world Docker cloud, if the CPU resources allocated to a container is very limited and less than one CPU core, sharing a single core among many containers can not be avoided. Once a latency sensitive container is allocated such configurations of resources, then there is a urgent need to know the interference between the newly added container and the existing latency sensitive one.

In this experiment, server side is held in a container on CPU \#3 of the server host machine. Server uses `\textbf{--net=host}' to eliminate the interference of Linux bridge. Client side is also held on a dedicated CPU on the client machine, while not inside a container. The client continuously ping the server using Apache Thrift, while at the same time no additional data is transmitted between the two processes except for the necessary Apache Thrift overhead. Although the server container is authorized to use all the memory and network resources, since we are focusing on the CPU interference of Docker containers, CPU quota is limited. We fix CPU period 10000 us not changed among all the experiments. On the other hand, CPU quota is varying from 5000 us to 9000 us for the server container. At the same time, another container is running on the same CPU as the server container. This container is continuously running a matrix multiplication process. The matrix multiplication involves two 512 x 512 matrixes. In each iteration, we log down the execution time. The sum of the CPU quotas of the matrix container together with the latency container is equals to the CPU period container. For example, if the CPU quota of latency container is 9000 us, then that of matrix container is $10000 - 9000 = 1000 us$. The reason why we choose the number over 5000 while less than 9000 is because this period makes sure that the latency container have over 30\% of CPU, which is the maximum CPU consumption mentioned in the previous sections, while at the same time the two containers can interfere with each other. We log down the execution time of 100 iterations and the average measurement is shown in Figure. The mean, median and $99^{th}$-percentile measurements of the latency container are shown in Figure.

## 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* ([32], 2009.) 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 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 uses 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 ([35], Tsirtsis G., 2000).

### Case 1: Server Receives Data

**REFERENCE**

1. Interesting
2. Agarwal K, Jain B, Porter D E. Containing the hype[C]//Proceedings of the 6th Asia-Pacific Workshop on Systems. ACM, 2015: 8.
3. Shankar S. Amazon elastic compute cloud[J]. 2009.
4. Colgate M, Stewart K, Kinsella R. Customer defection: a study of the student market in Ireland[J]. International Journal of Bank Marketing, 1996, 14(3): 23-29.
5. Copeland M, Soh J, Puca A, et al. Overview of Microsoft Azure Services[M]//Microsoft Azure. Apress, 2015: 27-69.
6. Dean J, Barroso L A. The tail at scale[J]. Communications of the ACM, 2013, 56(2): 74-80.
7. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.
8. Dua R, Raja A R, Kakadia D. Virtualization vs containerization to support paas[C]//Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014: 610-614.
9. Eder J. Accelerating red hat enterprise linux 7-based linux containers with solarflare openonload[R]. Red Hat Enterprise, 2015.
10. Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers[C]//Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. IEEE, 2015: 171-172.
11. Gibson G A. Redundant disk arrays: Reliable, parallel secondary storage[M]. Cambridge, MA: MIT press, 1992.
12. Hasenstein M. The logical volume manager (LVM)[J]. White paper, 2001.
13. He S, Guo L, Guo Y, et al. Elastic application container: A lightweight approach for cloud resource provisioning[C]. Advanced information networking and applications (aina), 2012 ieee 26th international conference on. IEEE, 2012: 15-22.
14. Helsley M. LXC: Linux container tools[J]. IBM devloperWorks Technical Library, 2009.
15. Hopper T. Cumulative Distribution Function[J]. Month, 2014.
16. Huber N, von Quast M, Hauck M, et al. Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments[C]//CLOSER. 2011: 563-573.
17. Jalaparti V, Bodik P, Kandula S, et al. Speeding up distributed request-response workflows[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 219-230.
18. Kivity A, Kamay Y, Laor D, et al. kvm: the Linux virtual machine monitor[C]//Proceedings of the Linux symposium. 2007, 1: 225-230.
19. Kratzke N. About Microservices, Containers and their Underestimated Impact on Network Performance[J]. CLOUD COMPUTING 2015, 2015: 180.
20. Li C, Ding C, Shen K. Quantifying the cost of context switch[C]//Proceedings of the 2007 workshop on Experimental computer science. ACM, 2007: 2.
21. Li J, Sharma N K, Ports D R K, et al. Tales of the tail: Hardware, os, and application-level sources of tail latency[C]//Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014: 1-14.
22. Liu J, Huang W, Abali B, et al. High Performance VMM-Bypass I/O in Virtual Machines[C]//USENIX Annual Technical Conference, General Track. 2006: 29-42.
23. Love R. Kernel korner: CPU affinity[J]. Linux Journal, 2003, 2003(111): 8.
24. Menage P, Seth R, Jackson P, et al. Linux control groups[J]. 2007.
25. Menascé D A. Virtualization: Concepts, applications, and performance modeling[C]//Int. CMG Conference. 2005: 407-414.
26. Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.
27. Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at facebook[C]//Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). 2013: 385-398.
28. Pendry J S, Sequent U, McKusick M K. Union mounts in 4.4 BSD-lite[J]. AUUGN, 1997: 1.
29. Slee M, Agarwal A, Kwiatkowski M. Thrift: Scalable cross-language services implementation[J]. Facebook White Paper, 2007, 5(8).
30. Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(3): 275-287.
31. Špaček F, Sohlich R, Dulík T. Docker as platform for assignments evaluation[J]. Procedia Engineering, 2015, 100: 1665-1671.
32. SPECweb2009 E. commerce workload, 2009[J].
33. Strauch C, Sites U L S, Kriha W. NoSQL databases[J]. Lecture Notes, Stuttgart Media University, 2011.
34. Tseng H M, Lee H L, Hu J W, et al. Network virtualization with cloud virtual switch[C]//Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, 2011: 998-1003.
35. Tsirtsis G. Network address translation-protocol translation (NAT-PT)[J]. Network, 2000.
36. Wright C, Cowan C, Smalley S, et al. Linux security modules: General security support for the linux kernel[C]//null. IEEE, 2003: 213.
37. Xavier M G, Neves M V, Rossi F D, et al. Performance evaluation of container-based virtualization for high performance computing environments[C]//Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013: 233-240.
38. Xavier M G, Neves M V, Rossi F D, et al. Performance evaluation of container-based virtualization for high performance computing environments[C]//Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013: 233-240.
39. Xu Y, Musgrave Z, Noble B, et al. Bobtail: Avoiding long tails in the cloud[C]//Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). 2013: 329-341.
40. Zawodny J. Redis: Lightweight key/value store that goes the extra mile[J]. Linux Magazine, 2009, 79.

**…**