Evaluation of Docker Containers Based on Hardware Utilization

Preeth E N*, Fr. Jaison Paul Mulerickal†, Biju Paul‡ and Yedhu Sastri‡
Rajagiri School of Engineering and Technology
Kerala, India 682039
Email: preeth.pen@gmail.com

***Abstract*—Docker is an open platform for developers and system administrators to build, ship, and run distributed applications using Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows. The main advantage is that, Docker can get code tested and deployed into production as fast as possible. Different applications can be run over Docker containers with language independency. In this paper the performance of these Docker containers are evaluated based on their system performance. That is based on system resource utilization. Different benchmarking tools are used for this. Performance based on file system is evaluated using Bonnie++. Other system resources such as CPU utilization, memory utilization etc. are evaluated based on the benchmarking code (using *psutil*) developed using python. Detail results obtained from all these tests are also included in this paper. The results include CPU utilization, memory utilization, CPU count, CPU times, Disk partition, network I/O counter etc.**

***Index Terms*—Docker, Containers, Resource utilization, Benchmark;**

I. INTRODUCTION

Docker is an open platform for developing, shipping, and running various applications in a faster way. Docker enables the applications to run separately from the host infrastructure and treat the infrastructure like a managed application. Docker also helps to ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code, Docker does this by combining a lightweight container virtualization platform with workflows and tooling that help to manage and deploy applications[1]. Docker uses resource isolation features of the Linux kernel such as cgroups (Control group is a Linux kernel feature that limits, accounts for and isolates the resource usage) and kernel namespaces(variable or identifier) to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting virtual machines. Docker provides a way to run almost any application securely isolated in a container. The isolation and security provided by Docker allow to run many containers simultaneously on a single host. Multiple containers can share the same kernel, but each container can be constrained to only use a defined amount of resources available in the host machine. Building on top of facilities provided by the Linux kernel, a Docker container, as opposed to a traditional virtual machine, does not require or include a separate operating system, instead, it relies on the kernel's functionality and uses resource isolation (CPU, memory, block I/O, network, etc), and separate namespaces to completely isolate the application's view of the operating system. Docker also implements a high-level API that provide lightweight containers that run processes in isolation. By using these isolated containers, resources can be services restricted, and processes provisioned to have a private view of the operating system with their own process ID space, file system structure, and network interfaces. The lightweight nature of containers, which run without the extra load of a hypervisor, means maximum utilisation of the hardware [1]. Since Docker is a new technology there is limitation of references.

II. VIRTUAL MACHINE VS DOCKER

Virtual machines are used extensively in cloud computing. In particular, the cloud services like Infrastructure as a Service (IaaS) largely uses virtual machines. Widely used cloud platforms such as Amazon EC2 make VMs available to customers and also run many services. Many Platform as a Servive (PaaS) and Software as a Service (SaaS) providers are built on IaaS with all their workloads running inside VMs[2]. Since virtually all cloud workloads are currently running in VMs, VM performance is a crucial component of overall cloud performance. Container-based virtualization presents an interesting alternative to virtual machines in the cloud [3]. Container based virtualization such as Docker can be used instead of virtual machines for faster functionalities as it is much faster to start a container and shut downing the container. The table below will summarize the difference between virtual machines and Docker.

Figure 1 shows the architectural difference in Virtual machines and Docker. In Virtual machine hypervisor is placed above the host OS, which co-ordinates the guest systems. Guest operating systems are placed over the hypervisor. All these complexities are avoided in the case of Docker, which is a containerization technology. Different containers are placed directly over the host operating system and a Docker daemon to manage the containers. Figure 1[1] shows the architectural difference between Docker and Virtual machine.

Table I
VIRTUAL MACHINE VS DOCKER

Virtual Machine	Docker
Virtual machines runs on a virtual hardware and guest OS will be loaded in its own memory.	Guests share same OS, which is the Host OS, which is loaded in the physical memory.
Communication between guests are through network devices, may be software.	Communication between guests is through pipes, sockets, bridges etc..
Security depends on the hypervisor.	Lacks security measures.
More overhead due to its complexity.	Less overhead as it is light weight containers.
Sharing of libraries and files are not possible.	Sharing of file possible (Ex: using SCP command in LINUX).
Takes time in booting.	Faster booting.
Uses more memory as it has to store the complete OS for each guests.	Less memory usage, as it shares the Host OS.

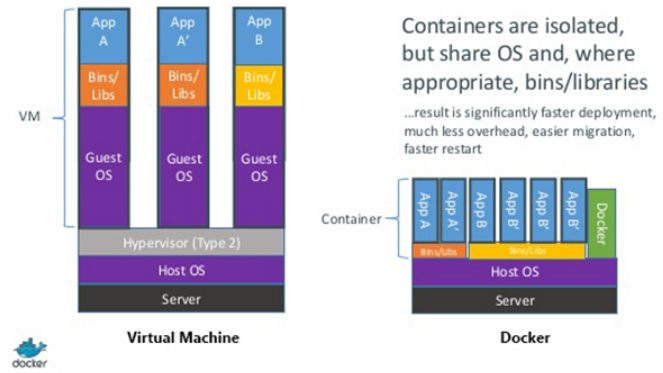

Figure 1. Docker vs virtual machine[1]

III. DOCKER ARCHITECTURE

Docker uses a client-server architecture, the client containers talks to the Docker daemon which is in the host machine. Docker daemon does the heavy lifting of building, running, and distributing Docker containers. Both the Docker daemon and the Docker container may be placed in a single machine or can also be placed in a remote machine. Docker daemon and the containers communicate through a bridge, docker0. Figure 2 shows the Docker architecture. Docker contains three components: Docker images, Docker registries, Docker containers.

A. Docker images

Docker images are read-only template which is used to create Docker containers. For example, an image can contain an Ubuntu 12.10 operating system with Apache and other web applications installed. Docker also allows to create new images and also one can use already created images.

B. Docker registries

A Docker registry holds Docker images. It is a public or private store for downloading and uploading Docker images.

C. Docker containers

Docker container is similar to a directory. It holds everything needed for an application to run. Containers are created from Docker images. A container can be run, started, stoped, moved and deleted.

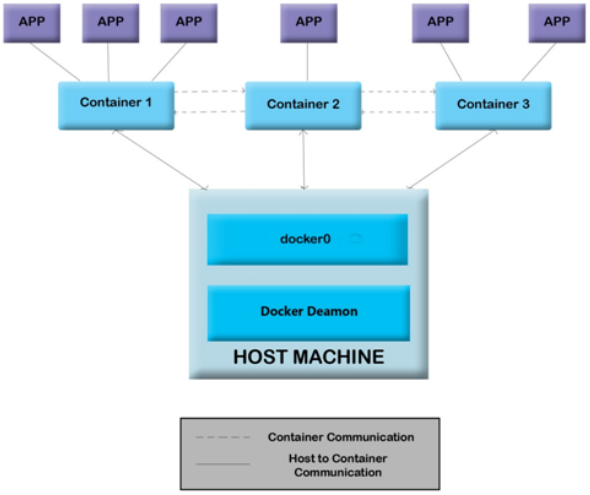

Figure 2. Docker Architecture

IV. PERFORMANCE EVALUATION

This paper was implemented on a LINUX system. Docker was installed directly on a 4 core LINUX system without any hypervisor. Configuration of the host machine include 4Gb RAM, 512Gb hard disk, core i5 processor with LINUX-Ubuntu version 12.04 64 bit OS

A. Bonnie++

Bonnie++[4] is a benchmarking suite in C++ which is aimed at performing several tests of hard drive and file system performance. The output for Bonnie++ on Docker is given below. Bonnie++ consist of many tests for evaluating the I/O performances. It tests file system performances with respect to data read and write speed. A fixed data size of 40Mb was given and tests were performed to measure the reading and writing speed of the container. Bonnie++ was excecuted on a container with container id: 111c5ab491fe, this unique id is used to identify the container. When writing the file by doing 40 million putc() macro invocations, Bonnie recorded an output rate of 507 K per second, that is the writing speed of the container. The operating system used 98

Table II
BONNIE++ ON DOCKER

Container ID	File size	Sequential output		Sequential input	
		output rate	CPU time	input rate	CPU time
111c5ab491fe	40Mb	507Kb/s	98 %	1351Kb/s	97 %

The output of Bonnie++ when executed on a LINUX-Ubuntu 12.04 machine is shown below. This is to show the

performance variation in Docker and a Host OS. There is a difference in the speed, that is the input rate and output rate is greater for Host OS. It is clear that the Host OS have more resourses than a container running over it, which partially uses the resources provided by the Host machine. Hence the Host OS shows slightly better performance.

Table III
BONNIE++ ON HOST OS

User ID	File size	Sequential output		Sequential input	
		output rate	CPU time	input rate	CPU time
user-OptiPlex	40Mb	536Kb/s	98 %	4388Kb/s	96 %

B. psutil

psutil[5] is a cross-platform library for retrieving information onrunning processes and system utilization (CPU, memory, disks, network)in Python. It is mainly used for system monitoring, profiling and limiting process resources and managing running processes. It supports many operating systems like LINUX, Windows, FreeBSD, Sun Solaris etc on both 32 bit and 64 bit architectures.

1) Memory utilisation: Memory utilization returns the memory used by the containers. The memory usage will be given in different fields as shown in the table. Every fields are expressed in bytes

- Total: indicates the total physical memory available in the host machine
- Used: It is the memory that is already being used by previously running processes
- Free: The memory which is not being used and is readily available

Along with the above mentioned fields there are also some platform-specific fields, it varies for different platforms.

- Active: The memory which is currently in use or was used recently. Used in UNIX
- Buffers: The memory used for caching file system metadata.
- Cached: The memory used for caching various things. This is used for improving the performance of the system.

Total memory used was 20.2 percent. The output for testing the memory usage of Docker container is tabulated below

Table IV
MEMORY USAGE OF DOCKER AND HOST OS

Machine	Total	Used	Free	Active	Buffers	Cached
Docker	401862 6560	13984 11264	26202 15296	64681 1184	14143 8976	70796 9024
Host	401862 6560	13953 18784	26233 07776	64727 0400	14145 5360	70448 3328

The table IV shows the memory usage of Host operating system. Memory is similar to in both cases, that is Docker can access the memory available in Host.

2) CPU times: CPU times returns the values of the times cpu where used. That is, it represents the time, in seconds, the cpu has spent in different modes. The table below shows different attributes which represents the time. The attributes varies depending on the platform.

Table V
CPU TIMES OF DOCKER AND HOST OS

Machine	User	Nice	System	Idle	Iowait	Irq
Docker	24.27	20.55	18.28	14312.59	89.93	3.55
Host	24.97	20.55	18.71	14456.88	90.08	3.59

Table V shows the CPU times of Docker and Host OS. There is no mentionable difference seen in both the results, which shows Docker gets all the resources available in Host OS.

3) CPU count: CPU count returns the number of logical CPUs in the system. If logical is false, that is no logical CPUs then it returns the number of cores. This paper was implemented on a 4 core machine and hence the output was returned as 4. Host OS also have a 4 cores as it is the hardware Specification of the Host machine. Docker also gets 4 core processor.

4) Disk usage: Disk usage returns the disk usage statistics about the given path, and is expressed in four fields, total, used and free expressed in bytes and the percentage of disk usage.

- Total: Indicates the total amount of space available in the disk in bytes.
- Used: Amount of space used in the disk
- Free: The space available for usage
- Percent: percentage of disk space used

Table VI
DISK USAGE OF DOCKER AND HOST OS

Machine	Total	Used	Free	Percent
Docker	488003694592	6466883584	456723259392	1.3
Host	488003694592	6466891776	456723251200	1.3

Table VI shows the disk usage of Docker container and the Host OS. The result implies that the disk available for the Host OS is almost similar to that of Docker. Hence it is clear that the disk available for Host OS is also available for Docker.

5) Network I/O counter: Network I/O counter returns the system wide network I/O statistics in different attributes:

- Byte sent: Number of bytes sent.
- Byte received: Number of bytes received.
- Packets sent: Number of packets sent.
- Packets received: Number of packets received.
- Error in: Total number of errors while receiving.
- Error out: Total number of errors while sending.
- Dropped in: Total number of incoming packets which were dropped.
- Dropped out: Total number of outgoing packets which were dropped.

The table below shows the output with the above attributes

Table VII shows the network I/O counter of Docker and Host OS. There is a reduced packet transfer in Host OS.

Table VII
NETWORK I/O COUNTER IN DOCKER AND HOST OS

Machine	sent (bytes)	Received (bytes)	Packets sent	Packets re-ceived	Error in	Error out	Drop-ped in	Drop-ped out
Docker	476856	16864785	8573	11970	0	0	0	0
Host	45128	2520642	315	22377	0	0	0	0

6) System users: System user returns the attributes like, name of the user, host name and the creation time as a floating point number expressed in seconds. Here this returns a NILL value, because Docker does not allow multiple users and it also does not support multi-tasking. Hence the number of users shown will always be NILL

7) Boot time: Boot time returns the system boot time expressed in seconds since the epoch, an epoch is an instant in time chosen as the origin of a particular era. This timestamp can be converted to human readable time. Boot time is 1429346140.0, which is equal to 18/04/2015, 03:35:40

8) Process iteration: Process iteration return an iterator yielding a process class instance for all currently running processes on the local machine. Every instance is created only once and then it is cached into an internal table which is updated every time an element is yielded. Cached process instance are checked for identity so that no 2 processes are using same PID, if so the cache instance is updated.

Table VIII
PROCESS ITERATION IN DOCKER

Pid	Name
1	Bash
19	Python
152	python

Table IX shows the process iteration in Host OS and table VIII shows process iteration in Docker. The result shows there are a lot of processes running in the Host OS. There is only 3 processes running on the Docker container, and these process are running in isolation because the processes running on Docker is not shown in the process list of Host OS. Thousands of processes are running in the Host OS in that some are tabulated below.

Table IX
PROCESS ITERATION IN HOST OS

Pid	Name
1	init
2	kthreadd
3	ksoftirqd/0
5	kworker/0:0H
6	kworker/u32:0
7	rcu_sched
8	rcuos/0
9	rcuos/1
10	rcuos/2

V. CONCLUSION

From the evaluations done using Bonnie++ and *psutil* performance tool, it is clear that Docker is performing well, and can be compared to the performance of an OS running on bare-metal. The promise of Docker is that it will work better than a virtualized node, and that has yet to be evaluated as a future work. Also we hope that the pressing security concerns about Docker container system will also be addressed by the Docker community for its bright future.

REFERENCES

[1] Docker (Webpage), https://docs.docker.com/
[2] RajdeepDua, A Reddy Raja and DharmeshKakadia ," Virtualization vs Containerization to support PaaS " ,2014 IEEE International Conference on Cloud Engineering.
[3] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys 07, pages 275287, 2007.
[4] Bonnie++ (Webpage), http://www.coker.com.au/bonnie++/
[5] psutil (Webpage), https://pypi.python.org/pypi/psutil/