Accelerating Big Data Applications Using Lightweight Virtualization Framework on Enterprise Cloud

Janki Bhimani, Zhengyu Yang, Miriam Leeser, and Ningfang Mi
Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
bhimani@ece.neu.edu, yangzy1988@coe.neu.edu, mel@coe.neu.edu, and ningfang@ece.neu.edu

***Abstract*—Hypervisor-based virtualization technology has been successfully used to deploy high-performance and scalable infrastructure for Hadoop, and now Spark applications. Container-based virtualization techniques are becoming an important option, which is increasingly used due to their lightweight operation and better scaling when compared to Virtual Machines (VM). With containerization techniques such as Docker becoming mature and promising better performance, we can use Docker to speed-up big data applications. However, as applications have different behaviors and resource requirements, before replacing traditional hypervisor-based virtual machines with Docker, it is important to analyze and compare performance of applications running in the cloud with VMs and Docker containers. VM provides distributed resource management for different virtual machines running with their own allocated resources, while Docker relies on shared pool of resources among all containers. Here, we investigate the performance of different Apache Spark applications using both Virtual Machines (VM) and Docker containers. While others have looked at Docker's performance, this is the first study that compares these different virtualization frameworks for a big data enterprise cloud environment using Apache Spark. In addition to makespan and execution time, we also analyze different resource utilization (CPU, disk, memory, etc.) by Spark applications. Our results show that Spark using Docker can obtain speed-up of over 10 times when compared to using VM. However, we observe that this may not apply to all applications due to different workload patterns and different resource management schemes performed by virtual machines and containers. Our work can guide application developers, system administrators and researchers to better design and deploy big data applications on their platforms to improve the overall performance.**

Keywords*—*Virtual Machine (VM), Container, Docker, Apache Spark, Big Data, Cloud Computing, Resource Management, Task Assignment, Workload Evaluation & Estimation

I. Introduction

Big data enterprise cloud environments, such as MapReduce [1] and its implementations like Hadoop [2] and Spark [3], provide a productive high level programming interface for large scale data processing and analytics. MapReduce is a software framework that allows a cluster of computers to process a large set of data in parallel. MapReduce consists of two main steps, "map" and "reduce", where the "map" step dispatches pieces of the huge data set to individual computers in the cluster for processing and "reduce" combines the intermediate data from "map" and derives the final output. Hadoop, a widely adopted cloud computing framework in industry, has been criticized in recent years for its inefficiency of handling iterative and interactive applications. The biggest drawback of Hadoop is that it introduces large amounts of disk I/O operations for storing intermediate data between map and reduce phases. To overcome this, the new framework, Spark, strives to cache all intermediate data (*Resilient Distributed Datasets (RDDs)*) into memory instead of disk. RDDs can be stored in memory between queries without requiring replication and disk access. Each RDD remembers how it was built from other datasets (by transformations like map, join or groupBy) to rebuild itself when required. RDDs allow Spark to outperform Hadoop models by up to 100x in multi-pass analytics.

The most widely used implementation of Spark is by setting up a master-worker framework on hypervisor-based virtualizations (such as Xen, VMware and KVM) using virtual machines (VMs). Recently, the world of virtualization platforms has seen a dynamic shift with the introduction and stable release of containerization technologies such as Docker [4], [5]. Containers and virtual machines have similar resource isolation and allocation benefits but different architectural approach and resource management. Containers are more portable and efficient compared to bare metal and virtual machines [6], [7].

While characterizing and improving performance for multiple instances on a bare metal or virtual machine is not new [8], [9], "Dockerized" big data applications running in cloud frameworks like Spark deserve special attention. There are challenges when deploying a distributed application such as a Spark application on containers, as the operation of Spark which includes handling in-memory RDDs, shared variables, etc. is unaware of the underlying virtualization.

Containers and virtual machines are two different technologies. Each has its own advantages and working patterns. The Docker containers are lightweight when compared to VMs as each container does not have to operate separate OS (operating system) [10]. Containers perform shared resource management but VMs perform distributed resource management. The distributed resource management in VM guarantees stability and security because each virtual machine runs with its specific assigned set of resources, while shared resource management in Docker enables more flexible sharing of resources which may increase overall resource utilization. It is observed that containerization techniques like Docker, LxCs etc. have promising performance for many different applications [11]–[13]. Thus, it is important to compare performance of widely used Big Data processing framework *Spark* on Docker with that of traditional virtual machines.

The main contributions of this paper are:

- Building and comparing the architecture of Spark cluster on hypervisor-based virtualization and con-

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

tainerized virtualization.

- Makespan analysis of Spark applications operating on multiple virtual machines and Docker containers.
- Resource management and utilization study with respect to behavior of applications on different virtualization frameworks.

In this paper, we study what type of virtualization frameworks will be best for operating Spark applications depending upon different resource usages. We observe that although most of the previous work claims that applications perform better on Docker when compared to VM, this may not be generalized for all distributed applications running on the big data enterprise cloud framework of Apache Spark. To the best of our knowledge, our paper is the first paper to compare the performance difference of Spark between VM and Docker.

The rest of this paper is organized as follows. In Sec. II, we describe the related work. In Sec. III and IV, we explain the architecture and current implementation of our framework. Sec. V shows our experimental results and implications. Finally, we conclude our current work and introduce future work in Sec. VI.

II. Related Work

With the growing amount of data, the big data processing framework like Apache Spark is widely used. Apache Spark [14] is a general cluster compute engine for scalable data processing. One of the widely accepted methods of using Apache Spark to process enterprise cloud workloads is by using hypervisor based virtualization framework of Virtual Machines (VMs). The virtual machine technology has been developed for several decades, bringing forth several software solutions (such as Xen, VMware and KVM) and incorporating support for several data processing frameworks such as Hadoop, Spark etc. The main benefit of virtualization include hardware independence, availability, isolation and security. However, the virtual machines incur some performance overhead [15], [16].

Recently, new virtualization technology of containerization (such as Docker [17], OpenVZ [18] and LxC [19]) has been developed which claims to provide the same independence and multi-tenancy with negligible performance overhead compared to VMs. The container-based virtualization implementations offer a lightweight virtualization layer, which promises a near-native performance. Literature [5] reveals that the Docker also performs well for traditional database applications running with high speed NVMe SSDs as storage. Using database processing framework such as Hadoop, Spark etc. in containerized framework rather than hypervisor based VMs may be beneficial. VM deduplication, migration techniques are developed in [20]–[22]. Performance modeling for cloud computing frameworks is studied in [23]–[26]. Recent work [27]–[37] further focus on how to utilize Flash-based SSDs in cloud computing platforms to accelerate their overall performance. Studies [9], [38]–[40] investigate resource allocation and optimization problems.

However, none of the existing work have studied and compared the operation of Apache Spark using Docker and VMs. Some studies investigated performance of Hadoop on virtual machine and Docker, but they did not analyze deeply of the difference of VM and Docker [41], [42]. Some other work like DCSpark [13] describes how to run Spark applications on Docker without conflicting configurations and library dependencies in one physical cluster. When compared to Apache Hadoop, Spark improves the system performance by storing as much as possible intermediate results (e.g., RDDs) into the memory instead of spinning disks (e.g., HDDs). Prior work [43], shows that Spark is about 5x faster than Hadoop, for most commonly used applications like Word Count, K-means, and Pagerank, so in this work we explore the performance of Spark applications on Docker when compared to virtual machines.

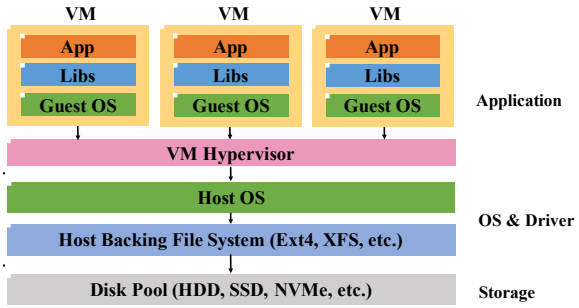

Fig. 1. System architecture of virtual machine hypervisor.

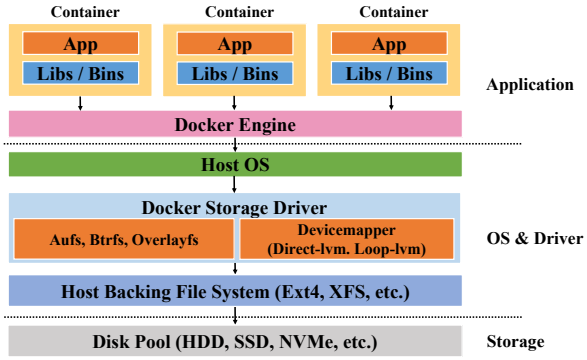

Fig. 2. System architecture of Docker container.

III. Virtualization Frameworks

In this section, we study the two different virtualization frameworks, i.e., virtual machine (VM) hypervisor and Docker container. VM hypervisor, which is also called Virtual Machine Hypervisor (VMH) and Virtual Machine Monitor (VMM), has a long history since the 1960s and is widely used even before the cloud computing era. It is an important technique for resource-provisioning, multi-tenancy, and system constitution in IaaS.As shown in Fig. 1, VM Hypervisor (such as Xen, KVM, VMware, etc.) is the virtual platform software that implements multiple guest OSes in a single system server. Specifically, the monitor lies between one or more operating systems and the hardware and gives the illusion to each running OS that it controls the machine. A hypervisor is operated as middleware between the VM and OS. Each VM has its own guest OS so that applications can directly call APIs (e.g., `libs`) to transparently run on each VM.

In contrast, Docker provides application virtualization using a containerized environment, see Fig. 2. A Docker image is an inert, immutable file, from which containers are started.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE I: Difference between VM hypervisor and Docker container.

Specs	Virtual Machine	Container
Products	VMware, Xen, KVM, etc.	Dockers, rkt, etc.
Guest OS	Included	Not included
virtualization Controller	VM hypervisor	Docker engine
Resource Management	Distributed	Shared
Machine Emulation	Complete (isolated OS kernel)	Partial (shared OS kernel)
Spatial Size (Bytes)	Large	Small
Launch Time	Long	Short
Migration	May require image conversion	Easy

Application installation can only be performed inside Docker containers. In order to maintain lightweight characteristics, it is advisable to keep the installation stack within the container as small as possible for better performance. The data management of containers is superintended by Docker storage drivers [44] (e.g., `OverlayFS`, `AUFS`, `Btrfs`, etc.). We use the `AUFS` file system as Docker storage driver for our Docker containers. Generally, `XFS` and `Ext4` are most commonly used host backing file systems. Here, we use `Ext4` for host backing file system of both virtual machine and Docker.

A. Common Components

Both VM and Docker are virtualization approaches and have three layers: application, OS & driver, and storage. Each container works in its own separate workspace in terms of file system and database. The application layer has multiple instances of different applications and workloads operating in multiple virtual machines or containers (see application layer in Figures 1 and 2). Thus, I/Os are generated by the application layer. The OS and Driver layer are comprised of host OS and host backing file system such as `XFS` or `Ext4`. The lowest layer of virtual machine hypervisor and Docker container is comprised of storage media where all data is persisted. The storage media can consist of hard drives (HDDs), solid state drives (SSDs) or even hybrid drives consisting of different types of storage devices.

B. Major Differences

Next, we discuss the major differences between the two virtualization technologies. Table I summarizes these differences.

1) Guest OS: The major difference between VM and Docker is that the latter does not need to maintain a guest OS inside each container. This makes containers "lighter" and also lowers the overhead of managing device drivers in each instance. Containers can thus enable faster start up with better performance when compared to VMs. Furthermore, containers share the host's kernel and thus have less isolation.

2) Virtualization Controller: To manage multiple virtualization instances requires a Virtualization controller to decide the instantiation, termination and inspect low-level information like network ports and IP addresses of all instances. The virtual machine has VM hypervisor as a controller, while Docker consists of Docker engine. This virtualization controller runs on the host and needs to have `sudo` access to all system resources.

3) Resource Management: The virtual machine hypervisor has distributed resource management among different virtual machines, where each VM is assigned the maximum limit of resources it can use. There also exist many smart techniques to distribute resources in an optimal way among different virtual machines [45], [46], but they lack run time flexibility. The Docker container relies on `cgroups` to assign, allocate and manage resources like CPU, Memory, Block I/O, Network, etc. Unlike VM, containerized virtualization performs shared resources management among different active and inactive containers. Resources like memory and page cache are shared among all containers. No prior resource allocations in terms of processing unit and memory are done and all the containers compete for these shared resources at run time. Thus, flexible resource sharing and high compatibility are possible in containerized virtualization.

The shared resource management in Docker can also ensure better resource utilization when there exist some inactive instances. Under hypervisor-based virtualization, the inactive instances may occupy the resources allocated to them. But, the inactive instances would consume negligible resources in containerized virtualization. Thus, containerization allows active instances to use resources that are unused by inactive instances. We investigate the impacts of this distributed and shared resource management in our experiments in Section V.

4) Distribution: VM hypervisor distributes VM images, each including guest OS and individual library and application stack. This type of distribution results in better security and independence but larger overhead of start up and application performance. Meanwhile, Docker containers are lightweight without guest OS and are preferred to have one or least possible number of applications in each container. The Docker daemon can only run one storage driver, and all containers created by that daemon instance use the same storage driver. Storage drivers operate with the copy-on-write technique [47], which provides advantages for read intensive applications and further reduces I/O overhead. For applications that generate heavy write workloads, it is advisable to maintain data persistence. Docker volume is a mechanism to automatically provide data persistence for containers. The biggest benefit of this feature is that I/O operations through this path are independent of the choice of the storage driver, and should be able to operate at the I/O capabilities of the host.

IV. Data Processing Engine: Apache Spark

Apache Spark [14] is a general cluster compute engine for scalable data processing, which generalizes two-stage MapReduce to support arbitrary directed acyclic graphs (DAGs) of

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

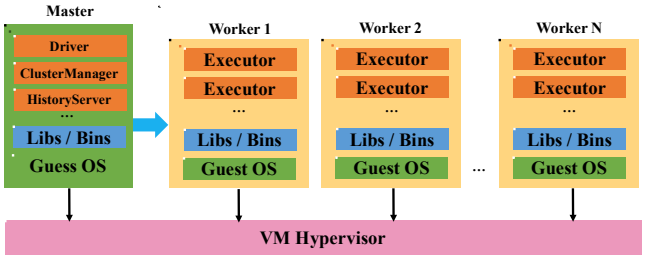

Fig. 3. Framework of Spark on VM.

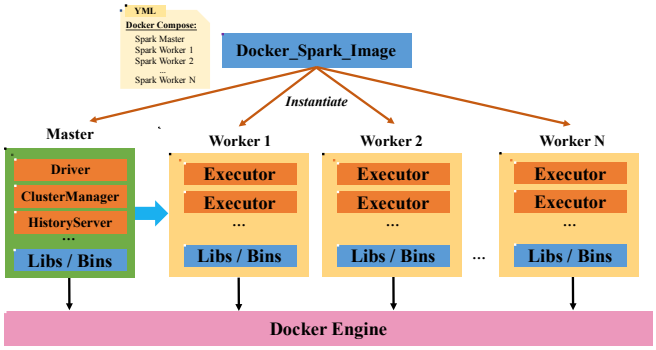

Fig. 4. Framework of Spark on Docker.

tasks and fast data sharing between operations. Most importantly, Spark improves the system performance by storing intermediate results (e.g., RDDs) into the memory instead of spinning disks. We choose Spark to compare its operations on different virtualization frameworks because there are several improvements in the development of Spark from Hadoop [2]:
1. Spark provides an easy-to-use memory abstraction implemented as resilient distributed datasets (RDDs) which avoids a significant portion of slow disk I/Os that occur in Hadoop.
2. Spark supports DAG scheduling. In contrast to the simple programming model of map phase and reduce phase in Hadoop, a Spark job usually consists of multiple stages, which makes resource provisioning in Spark more difficult than that in Hadoop.
3. Spark supports a wider range of applications (like graphs processing and machine-learning libraries), enabling usage of one platform to meet different data analytics needs.

A. Spark Architecture

Spark has one master node and N worker nodes. The master node can be used to create RDDs, accumulators and broadcast variables. It contains a set of modules that work together to schedule and submit tasks from each application. For example, the `driver` module consists of `RDDGraph`, `DAGScheduler`, `TaskScheduler` and `SchedulerBackend`. The `ClusterManager` module creates RDDs and performs a series of transformations to achieve its final result. These transformations of RDDs are translated into a DAG and submitted to the scheduler to be executed on a set of worker nodes.

There are multiple work nodes in Spark and each of which is a `JVM` that run multiple executors. These executors run tasks scheduled by the driver, store computation results in memory, and conduct on-disk or off-heap interaction with the storage systems. Each worker node consists of one or more executors depending on number of cores used by a worker. Each executor can also run multiple parallel tasks.

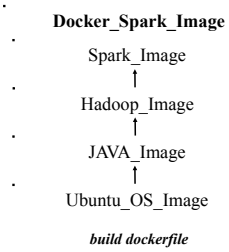

Fig. 5. Method to build `Docker_Spark_Image`.

B. Implementations of Spark on VM and Docker

We build Apache Spark on VM and Docker to investigate the performance difference of both implementations. Figures 3 and 4 show the architecture of Spark on VM and Docker. For both implementations, one VM or Docker container is used as the master node, and all others are worker nodes. Also, each node has its own IP address and port.

For the VM setup, we have a number of virtual machines running on a physical server via VM Hypervisor. Each VM has its own guest OS as well as its own separate Spark data processing workspace to manage executor files and database as shown in Figure 3.

The Spark implementation on Docker comprises of multiple simultaneously operating containers. As shown in Figure 4, we do not need to maintain a guest OS in each container. The container approach uses `Bridge Docker()` to connect all containers. We build our `Docker_Spark_Image` with a `Dockerfile`, which will be used to instantiate all containers of Spark on the Docker framework.

Figure 5 illustrates the internal layers of the Spark on Docker image. We first instantiate `Ubuntu` from its image available on Docker Hub [48]. On top of the `Ubuntu` OS layer, we load and install `Java`, `Hadoop` (for HDFS), and `Spark` and then commit it as `Docker_Spark_Image`. Finally, we compose our Spark cluster using this `Docker_Spark_Image`, a *.yml* file containing master and worker environment details like ports, DNS, cores, volume directory, etc., and a *.conf* file that lists details like max retries, event log directory, etc.

V. Evaluation

A. Testbed and Platform Configurations

Table II summarizes the configuration of our testbed. We deploy one master and eight worker nodes for Spark on both VM and Docker frameworks as shown in Figures 3 and 4. We use open source measurement tools (`dstat` [49], `iostat` [50], `blktrace` [51]) to measure performance metrics such as total execution time of each application (makespan), CPU utilization, memory utilization, disk I/O rate, and network traffic.

B. Benchmarks

We consider different representative benchmarks in our experiments in order to help Spark developers and users evaluate a range of applications in a standard development scenario. We implement a set of algorithms that are commonly used in big data processing (see Table III). We categorize these algorithms into three types: machine learning, graph computation and SQL queries. Among them, most machine

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE II: Testbed configuration.

Component	Specs
Server	PowerEdge R630 Server
Processor	Intel(R) Xeon(R) CPU E5-2660 v4
Processor Speed	2.00GHz
Processor Cores	56 Cores
Processor Cache Size	35M
Memory Capacity	64GB RDIMM
Memory Data Rate	2400 MT/s
Operating System	Ubuntu 14.04 LTS
Docker Version	17.03
VM Hypervisor	VMware Workstation 12.5
HDD Speed	7.2K RPM SATA 6Gbps
HDD Capacity	1 TB

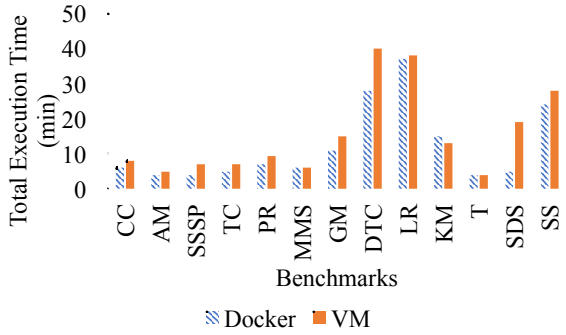

Fig. 6. Comparing total execution time of Spark applications running on hypervisor based VM and containerized Docker.

learning algorithms and some graph computation algorithms are iterative and their execution time can thus be determined by the number of iterations. We also conduct sensitivity analysis of these applications under different number of iterations.

TABLE III: Benchmark Details.

Application Type	Benchmark
Machine Learning	K-Means(KM) MinMaxScaler(MMS) GaussianMixture(GM) Logistic Regression (LR) DecisionTreeClassification(DTC) Tokenizer(T) Alternating Least Squares(ALS)
Graph Computation	PageRank(PR) ConnectedComponents(CC) TriangleCounting(TC) AggregateMessages(AM) Single-Source Shortest Paths (SSSP)
SQL Queries	SQLDataSource(SDS) SparkSQL(SS)

C. Study of Execution Time

We first focus on studying the total execution time when Spark applications are operated using either hypervisor virtualization (VM) or containerized virtualization (Docker). We deploy Spark applications on VM and Docker to ensure that both implementations are performed with the same versions of Ubuntu, Spark, Hadoop, etc., and the same resource allocation settings like number of cores, memory capacity, etc. in Spark configurations. Figure 6 shows the execution time of Spark applications running on hypervisor-based virtualization and containerized Docker. We see that Spark applications on Docker containers perform mostly better than on VMs. This is mainly because of (1) faster startup time and 2) faster read operation of Spark applications on Docker containers. A containerized Spark application usually starts in a couple of seconds, but the same application on virtual machines needs a couple of minutes to start up. Secondly, compared to VM, read operations of Spark applications can be executed faster on Docker containers because Docker maintains its own storage driver and performs copy-on-write (COW). Meanwhile, we observe that some applications like *MMS* and *T* receive similar performance on both Docker and VM. More interestingly, the K-Means (KM) algorithm has lower execution time on VM than on Docker. We show results of some more experiments using K-Means and investigate its behavior later in this Section.

The execution time of some iterative applications might be determined by application-related variables such as number of iterations. Thus, we further investigate the performance of these Spark applications (e.g., PageRank, Logistic Regression (LR) and K-Means) across different setups for application-related variables. Figure 7 shows the total execution times of these three applications as a function of number of iterations for PageRank and LR and number of clusters for K-Means. First, we observe that even with varying the number of iterations, Docker always performs better than VM for PageRank, see Fig. 7(a). For a small number of iterations, the speed-up of Docker when compared to VM is around 2x. As the number of iterations increases, we get up to 10x speed-up for Docker. This is because for each iteration, PageRank in Spark has a high reuse factor of two particular RDDs, which are persisted by Docker storage driver and thus can be quickly retrieved from main memory. We also notice that the execution time of LR is almost the same on Docker and VM across different number of iterations, see Fig. 7(b).

However from Fig. 7(c), we see that VM performs slightly better than Docker for K-Means. Here we vary the number of clusters (K) instead of iterations in our experiments. We notice that among all applications we consider, K-Means is the most "shuffle-intensive", especially when the number of clusters is relatively high. Specifically, higher value of K means that it needs to categorize the input data into more number of clusters. This further increases the shuffle selectivity of K-Means because for each data point the number of labeling options increases and the distance of each data point from all cluster centroids needs to be fetched every iteration. Moreover, unlike those in-memory operations (e.g., map, reduce, join, etc.), the shuffle operation is more expensive in Spark, since it involves cross-executor broadcastings and longer time due to corresponding disk I/Os, data serializations, and network traffics. We analyzed these I/Os and found that the number of "writes" is larger while performing shuffle. The Docker file system (i.e., AUFS) performs copy-on-write (COW) for every write operation. During shuffle, many COW operations are triggered in Docker which may lead to a throttling stall of operating threads. This dramatically reduces the benefits brought by Docker, and slows down the performance compared to VM. Thus, we conclude that it is advisable to use VM rather than Docker for shuffle intensive applications in Spark. We also verified this observation by experimenting with some other shuffle intensive applications such as Bigram, TeraSort. The results have the similar trends as those from K-Means. We omit the results due to lack of space.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

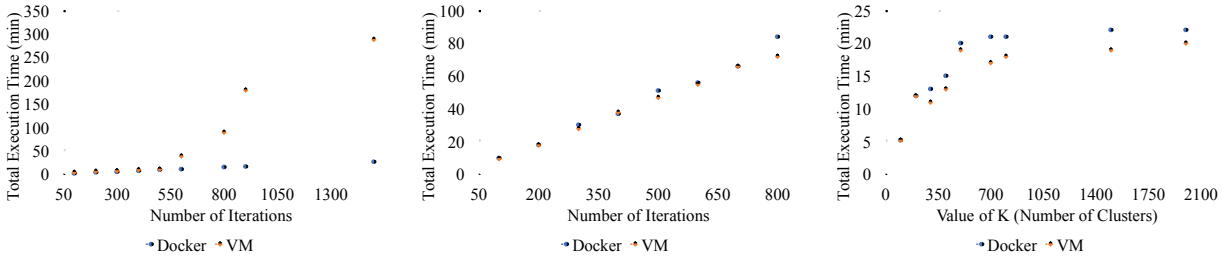

Fig. 7. Total execution time with increasing number of iterations for (a) PageRank, (b) Logistic regression, and with increasing number of clusters for (c) K-Means.

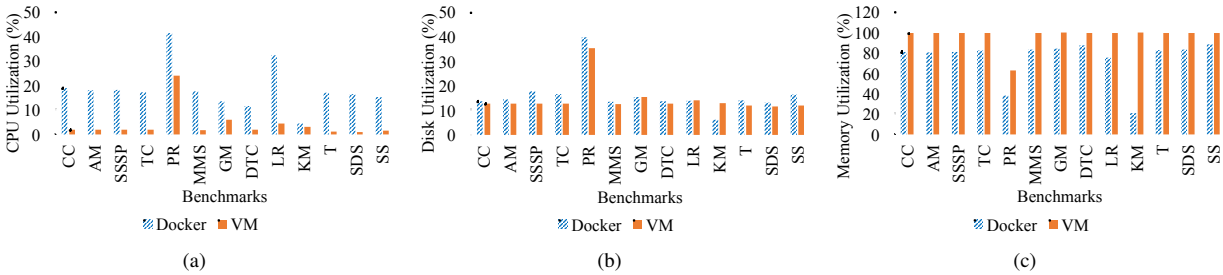

Fig. 8. Average resource utilization while operating different benchmarks; (a) CPU Utilization, (b) Disk Utilization, and (c) Memory Utilization.

D. Study of System Resource Utilization

Finally, we investigate the utilization of different system resources when running various benchmarks on VM or Docker frameworks. Figure 8 present the CPU, disk and memory utilization under different applications. First, we can see that Docker has much higher CPU utilization ratios compared to VM as shown in Figure 8(a). This means that Docker can use CPU resources more efficiently. Secondly, Docker's disk utilization ratios are also slightly higher than those of VMs for most applications (Fig. 8(b)). Interestingly, the K-Means (KM) application is an exception, which has lower disk utilization under Docker than under VM. This further validates our observation (Sec. V-C) regarding the operations of shuffle intensive applications. In contrast, as shown in Figure 8(c), Docker has lower memory utilization ratios compared to VM across all applications. The reason is that Docker bypasses the guest OS so it demands less memory.

VI. Conclusions

In this paper, we investigated the performance of big data processing applications running on Spark with different virtualization frameworks. We built the end-to-end software stack for the real implementation of different virtualization frameworks and evaluated the robustness of this software stack with various applications. We compared the *de facto* hypervisor-based virtualization technique with the recently emerging lightweight containerized virtualization framework (Docker) when deploying Spark. To best of our knowledge, this is the first research work to study the Spark architecture and the operations of Spark applications on different virtualization frameworks. We explored the impacts of different virtualization techniques on Spark applications' execution latency and system resource utilization. Our experimental results show that it is good to use Spark with Docker for map and calculation intensive applications because Docker provides lightweight operation, copy-on-write (COW) and intermediate storage drivers that assist Spark applications to perform better. On the other hand, we observed that it is not advisable to run shuffle intensive Spark applications on a containerized environment. In the future, we plan to develop a hybrid virtualization environment that can offer the options of both hypervised and containerized instances for Spark applications and help determine the best choice automatically based on the nature of applications.

VII. Acknowledgements

We would also like to show our gratitude to Danlin Jia and Han Gao for sharing their hard work and pearls of wisdom with us during the course of this research. This work was partially supported by National Science Foundation Career Award CNS-1452751 and AFOSR grant FA9550-14-1-0160.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[3] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache Spark perspective," *Global Journal of Computer Science and Technology*, vol. 15, no. 1, 2015.

[4] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[5] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[6] 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*. IEEE, 2015, pp. 171–172.

[7] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 610–614.

[8] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 31–38.

[9] H. Gao, Z. Yang, J. Bhimani, T. Wang, J. Wang, B. Sheng, and N. Mi, "AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-Stage Big Data Frameworks," in *26th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2017.

[10] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[11] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[12] T. P. (BlueData), "Lessons learned from running spark on docker."

[13] Z. Lei, H. Du, S. Chen, C. Zhu, and X. Liu, "Dcspark: Virtualizing spark using docker containers," in *Audio, Language and Image Processing (ICALIP), 2016 International Conference on*. IEEE, 2016, pp. 13–18.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[15] 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*. IEEE, 2013, pp. 233–240.

[16] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 409–416.

[17] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[18] "OpenVZ," https://openvz.org.

[19] "Linux Container," https://en.wikipedia.org/wiki/LXC.

[20] J. Roemer, M. Groman, Z. Yang, Y. Wang, C. C. Tan, and N. Mi, "Improving Virtual Machine Migration via Deduplication," in *11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS 2014)*. IEEE, 2014, pp. 702–707.

[21] Z. Yang, M. Ghosh, M. Awasthi, and V. Balakrishnan, "Online Flash Resource Migration, Allocation, Retire and Replacement Manager Based on a Cost of Ownership Model," Patent US15/094 971, US20 170 046 098A1, 2016.

[22] Z. Yang, J. Wang, and D. Evans, "A Duplicate In-memory Shared-intermediate Data Detection and Reuse Module in Spark Framework," Patent US15/404 100, 2017.

[23] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and gpu computing," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[24] ——, "Design space exploration of gpu accelerated cluster systems for optimal data transfer using pcie bus," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–7.

[25] J. Bhimani, N. Mi, and M. Leeser, "Performance prediction techniques for scalable large data processing in distributed mpi systems," in *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*. IEEE, 2016, pp. 1–2.

[26] J. Bhimani, N. Mi, M. Leeser, and Z. Yang, "FiM: Performance Prediction Model for Parallel Computation in Iterative Data Processing Applications," in *10th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2017.

[27] J. Bhimani, H. Huen, J. Yang, M. Awasthi, V. Balakrishnan, and J. Martineau, "Intelligent Controller for Containerized Applications," 2017, uS Patent App. 15/379327.

[28] J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, and N. Mi, "Improving Flash Resource Utilization at Minimal Management Cost in Virtualized Flash-based Storage Systems," *Cloud Computing, IEEE Transactions on*, no. 99, p. 1, 2015.

[29] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A Fresh Perspective on Total Cost of Ownership Models for Flash Storage in Datacenters," in *2016 IEEE 8th International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[30] Z. Yang, J. Tai, J. Bhimani, J. Wang, N. Mi, and B. Sheng, "GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous IO Workloads," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[31] Z. Yang, J. Wang, D. Evans, and N. Mi, "AutoReplica: Automatic Data Replica Manager in Distributed Caching and Data Processing Systems," in *1st International workshop on Communication, Computing, and Networking in Cyber Physical Systems (CCNCPS)*. IEEE, 2016.

[32] Z. Yang, M. Ghosh, M. Awasthi, and V. Balakrishnan, "Online Flash Resource Allocation Manager Based on TCO Model," Patent US15/092 156, US20 170 046 089A1, 2016.

[33] Z. Yang, S. Hassani, and M. Awasthi, "Memory Device Having a Translation Layer with Multiple Associative Sectors," Patent US 15/093 682, 2015.

[34] Z. Yang and M. Awasthi, "I/O Workload Scheduling Manager for RAID/non-RAID Flash Based Storage Systems for TCO and WAF Optimizations," Patent US15/396 186, 2017.

[35] Z. Yang, J. Wang, and D. Evans, "Adaptive Caching Replacement Manager with Dynamic Updating Granulates and Partitions for Shared Flash-Based Storage System," Patent US15/400 835, 2017.

[36] ——, "Automatic Data Replica Manager in Distributed Caching and Data Processing Systems," Patent US15/408 328, 2017.

[37] J. Wang, Z. Yang, and D. Evans, "Efficient Data Caching Management in Scalable Multi-stage Data Processing Systems," Patent US15/423 384, 2017.

[38] J. Wang, T. Wang, Z. Yang, Y. Mao, N. Mi, and B. Sheng, "SEINA: A Stealthy and Effective Internal Attack in Hadoop Systems," in *International Conference on Computing, Networking and Communications (ICNC 2017)*. IEEE, 2017.

[39] J. Wang, T. Wang, Z. Yang, N. Mi, and S. Bo, "eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[40] T. Wang, J. Wang, N. Nguyen, Z. Yang, N. Mi, and B. Sheng, "EA2S2: An Efficient Application-Aware Storage System for Big Data Processing in Heterogeneous Clusters," in *26th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2017.

[41] C. Gokhan, Z. Karakaya, and A. Yazici, "Systematic mapping study on performance scalability in big data on cloud using vm and container," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2016, pp. 634–641.

[42] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 365–368.

[43] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[44] "Docker Storage Driver," https://docs.Docker.com/engine/userguide/storagedriver.

[45] C. Peng, M. Kim, Z. Zhang, and H. Lei, "Vdn: Virtual machine image distribution network for cloud data centers," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 181–189.

[46] R. Knauerhase, V. Tewari, S. Robinson, M. Bowman, and M. Milenkovic, "Dynamic virtual machine service provider allocation," Jan. 2 2004, uS Patent App. 10/754,098.

[47] "Docker Copy On Write Strategy," docs.docker.com/engine/userguide/storagedriver/imagesandcontainers.

[48] "Docker Hub: Explore Official Repositories," Accessed in 05/2017, https://hub.docker.com/.

[49] "dstat," https://dag.wiee.rs/home-made/dstat.

[50] "iostat," https://linux.die.net/man/1/iostat.

[51] "blktrace," https://linux.die.net/man/8/blktrace.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE