SparkOS: Spark Microservice Provisioning using Unikernels

Jivjot Singh, Bikramdeep Singh and Manpreet Singh

Department of Computing Science, Simon Fraser University, Burnaby, BC

***Abstract*— With growing enthusiasm for container technologies like Docker, a relatively new architectural model - Unikernel [1] is also beginning to gain popularity. The unikernels aim to improve performance by separating functionality at the component level and eliminating unnecessary components. Apache Spark has become popular data processing tool due to its in-memory processing capabilities which provides significant edge over Hadoop. Since Spark is a resource intensive application, we present an approach of combining Spark with unikernels with a goal of achieving performance gain and security over standard spark deployment. The benchmark data supports performance gain.**

***Keywords*-Unikernels; Apache Spark; OSv; Performance Benchmarking**

I. INTRODUCTION

The success of Cloud Computing can be attributed to its underlying technology - Virtualization which is responsible for providing resource isolation and flexibility on a shared cluster. However these advantages come at a cost of an additional layer over an already heavily layered software stack. Despite containing layered architecture most services deployed as applications on Virtual Machines perform single tasks such as an application server or a database. This observation provided an opportunity for compiling a single application into a standalone Virtual Machine in order to improve performance known as 'UniKernels'.

Spark uses distributed memory abstraction to process large volumes of data. The performance of Spark depends upon a number of factors such as the implemented algorithm, size & type of input data and computing capabilities of the cluster nodes on which it is running. However, Spark as any Big data processing tool, is often considered to be a resource heavy application and generally deployed on dedicated cluster independent of other applications, which makes it a perfect candidate for unikernel deployment.

In this paper, we present an approach of provisioning Spark as a microservice on unikernels and evaluate its performance. The objective is to benchmark and quantitatively evaluate the performance advantages of unikernels against typical Spark deployments on Linux.

II. BACKGROUND

The traditional operating systems manage resources and are designed to run multiple isolated applications on single machine. However, it is fairly common nowadays to deploy single applications on a Virtual Machine resulting in inept performance due to necessary programs and services. Unikernels [2] the other hand relies on hypervisor to isolate single application running on a single virtual machine. Library operating systems are used to build unikernels, where only a minimal set of services required by the application are provided to the unikernel. These images can then be directly run on a hypervisor without any need of a guest OS.

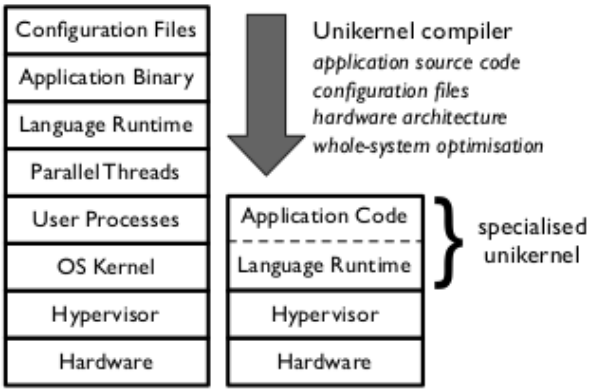

Fig. 1. General OS vs Unikernel

In addition to performance gains, unikernels also claims flexibility and versatility for elastic cloud computing, big data analytics and cross platform environments, which makes them a good fit for provisioning Spark. Since the cost to spawn a unikernel is very less, the Spark jobs can respond to load spikes by spawning new unikernel instances making it elastic.

III. RELATED WORK

Since Unikernels are relatively new to Cloud Computing there is not significant amount of work, both from the development and performance evaluation perspective. However, recent attempts [3] have been made for both micro and macro bench-marking of unikernel applications such as TCP/UDP, DNS and HTTP against traditional Linux deployments.

On the other hand there is notable amount of work done for evaluation and performance benchmarking of Spark. Some examples include *Spark-perf* which is library containing suites of performance tests for Spark, PySpark, Spark Streaming, and MLlib. The tests can be parameterized to test against multiple Spark and Test configurations. There are some other tools such as PREDIcT which aims at

predicting runtime for network intensive iterative algorithms implemented on MapReduce framework.

However, there aren't any significant attempts of combining Spark and Unikernels, so the benchmark data is not available. Therefore the goal here is to port existing benchmarking techniques to Spark-Unikernels and hence evaluate the performance and benchmark the performance.

IV. COMPONENTS

A. OSv

There are several open source projects gaining popularity in the unikernel world. MirageOS, OSv, Rump Kernels and HaLVM are among the most popular with each of them placing emphasis on different aspect of the unikernel approach. For instance OSv and Rump Kernels aim for compatibility with legacy software whereas MirageOS and HaLVM focus on safety and security.

For this project, we have selected OSv [4] which is slim, bare bone unikernel including just the functionality necessary to run Java or POSIX application. OSv takes a lean approach by reusing existing code thereby taking advantage of most of the existing support infrastructure from existing systems.

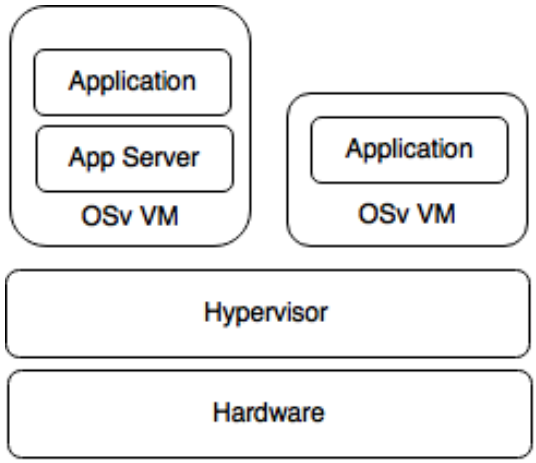

Fig. 2. OSv Stack

OSv [5] unlike traditional operating systems share a single thread space for all the threads. This implies that system calls such as fork() will not be available, however this results into performance gain such as context switching between threads does not incur a TLB flush nor does performing a system call incur context switch overhead. Also OSv offers superior I/O performance, manageability and ease of use. Caches, load balancers, NoSQL and other I/O intensive workloads are ideal targets for OSv.

Also OSv is packed with number of useful tools such as Capstan which is a tool for packaging and running application on OSv for QEMU/KVM platform. Also OSv offers support for running applications on on Amazon Web Services(AWS) and is so popular that the beta program is already over-subscribed.

B. Linux

To establish baseline for performance we selected Linux to evaluate the performance of Spark benchmark tests. We are using Ubuntu Server 14.04 LTS running on top of Xen hypervisor and the virtualization type is HVM. The reason for choosing Linux is that it the most popular choice for production deployments in most cloud environments.

C. Hypervisor

Hypervisor is an essential component while designing any cloud system. The choice of hypervisor greatly affects the performance of the system, therefore great attention must be paid while building a system. There are mainly two types of hypervisors, *Type I* which run directly on the host's hardware to control the hardware and to manage guest operating systems e.g, XenServer and VMWare ESXi. *Type II* which run on a conventional operating system just as other computer programs e.g, QEMU & VirtualBox. We have performed experiments on following hypervisors.

1) Xen: Xen provides an intermediate between full virtualization and containers, it thrives to create an efficient and flexible system without compromising with performance and security. Xen acts as a multiplexer between domains and expose a limited image of hardware rather than emulating it. Xen multiplexer is a thin, bare metal hypervisor and perform little more than multiplexing operations and few checks. However to perform more advanced operations such as domain creation and resource allocation to domains the hypervisor spawns a control domain named domain 0 which has higher privileges than the other domains.

2) KVM: KVM performs the task of converting Linux kernel into a bare metal hypervisor and thus leveraging advanced features of Intel VT-X and AMD-V x86, thereby delivering unprecedented performance. Also KVM includes Linux security features including SELinux to add access controls, multi-level security and enforcement. As a result, organizations are protected from compromised virtual machines which are isolated and cannot be accessed by any other processes.

We are performing most of our tests on XenServer due to its performance and support on AWS cloud. However, we are also performing some tests on the KVM hypervisor to see how is it performing in comparison to XenServer.

D. Spark

Spark has become the most popular tool for big data applications due to its performance. Unlike Hadoop, the Spark processing is done in-memory resulting in higher performance. Due to its popularity in the Big-Data community we selected it as a candidate for our project. We are using current version of Spark i.e 1.6.1. The spark is deployed on the 5 node cluster where we have a single master and four worker nodes.

E. Amazon Web Services (AWS)

There are multiple options for deploying VM images on cloud ranging from open source solutions such as Apache

CloudStack to hosted solutions. We are using AWS for deploying and benchmarking the Spark unikernels mainly due to the following reasons:

- AWS provides agility to setup cluster and quickly change the deployment configurations
- AWS is built on top of XenServer which is our choice for testing unikernels
- OSv promises stability and provides support and documentation for deploying images on AWS
- Application configuration can be scaled quickly and infinitely to test elasticity

V. Overview

Fig3 represents Spark's master/worker architecture taken from Spark's official documentation [6]. According to it, Spark applications runs as independent processes coordinated by Spark-Context object. Spark Context can connect to various cluster managers for allocating resources across applications. Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.

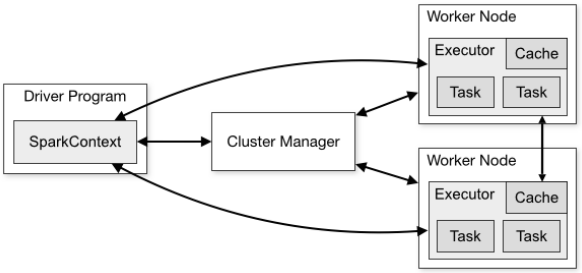

Fig. 3. Spark cluster architecture

Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs).

However, this architecture is not easy to migrate to OSV unikernel. As stated before unikernel does not support new process to spawn(fork), but in case of spark context every new job runs as a separate process on the slave nodes. So in order to mitigate this problem we created as special Java Class, that will spawn a new Thread instead of spawning a new process. We receive the class name from the master and use the Java class loader, which searches the current classpath and return abstract class instance of the class name. We then load the class and run its main function as a separate thread.

Above approach did help us to run a job but anytime system exit was used whole Spark slave used to shutdown, to tackle this we used the Java security manager. We ran the above job thread under different ThreadGroup, and wrote a SecurityManager, which restricted a thread belonging a particular ThreadGroup to use a Sys.exit, and instead exception was generated and only the job thread was killed.

We also encountered an exception in osv, when spark tries to acquire File Channel lock. As this system lock is used to acquire exclusive lock for a file. We deleted this as we are already working in a thread and no other process access is possible.

VI. Evaluation

For evaluating the Spark's performance on unikernels and Linux, we are using *Spark-perf* which is a performance testing framework for Apache Spark built by *Databricks*. The framework consists of suites of performance tests for Spark, PySpark, Spark Streaming and MLib. All the test configurations can be parameterized to test against multiple Spark configurations.

A. Model for Estimating Execution Time

The spark jobs are executed in multiple stages and each stage consists of multiple tasks. The following expression [7] describes the Spark job:

$$Job = Stage_i | 0 \leq i \leq M$$

$$Stage_i = Task_{i,j} | 0 \leq j \leq N$$

Where M represents number of stages in a job and N is the number of tasks in a stage. Now since different stages within the job are run in sequence. So the execution time can be defined by following expression [7]:

$$Jobtime = Startup + \sum_{1}^{m} Stage_s + CleanUp$$

Our model consists of running each test multiple times and measuring the execution time by taking difference between start and end time for all the jobs to finish their tasks. Then calculating median and standard deviation for all the resulting timings.

$$Exectime = \text{median}(\sum_{1}^{numTrials} (\sum_{1}^{numJobs} Job_{time,i} - Job_{startTime}))$$

B. Tests

Following are the tests being performed for Spark Performance benchmarking:

- *Scheduler Throughput Test:* The objective of this test is to measure the schedular//task launching throughput by scheduling a large number of null tasks.
- *Key-Value Operations:* This category of tests include key value MapReduce operations over a large dataset. The set of tests include *Aggregate-By-Key* for Strings, *Aggregate-By-Key* for Integers.

C. Test Configuration Parameters

All the tests are parameterized and the parameter details are as follows:

- *NUM_TRIALS:* This represents the number of times the test should be run
- *INTER_TRIAL_WAIT:* Number of seconds to sleep between trials
- *NUM_TASKS:* Number of tasks to create run in each job
- *NUM_JOBS:* Number of jobs to run
- *RANDOM_SEED:* Seed for random number generator
- *CLOSURE_SIZE:* Task closure size (in bytes)
- *REDUCE_TASKS:* Number of reduce tasks
- *NUM_RECORDS:* Number of input pairs
- *UNIQUE_KEYS:* (approx) number of unique keys
- *KEY_LENGTH:* Lenth of keys in characters
- *UNIQUE_VALUES:* (approx) number of unique values per key
- *VALUE_LENGTH:* Length of values in characters
- *NUM_PARTITIONS:* Number of input partitions
- *PERSISTENCE_TYPE:* Input persistence (memory, disk, hdfs)
- *STORAGE_LOCATION:* Directory used for storage with 'hdfs' persistence type
- *HASH_RECORDS:* Use hashes instead of padded numbers for keys and values
- *WAIT_FOR_EXIT:* JVM will not exit until input is received from stdin

D. Experimental Setup

- *Cluster Configuration:* All the tests were performed on Amazon EC2 cluster containing 5 nodes where one node served as master and the other nodes served as worker nodes. Each node was created using *t2.small* instance type having 1 VCPU, 2GB of memory and Volume Type is 8GB SSD. The availability zone for each node is *us-west-2b*.
- *OS Configuration:* The Operating System used for Linux is Ubuntu Server 14.04 LTS (HVM) and OSv version used is 0.24.
- *Spark Configuration:* We used Spark version 1.6.1 in standalone cluster mode to perform all our experiments.

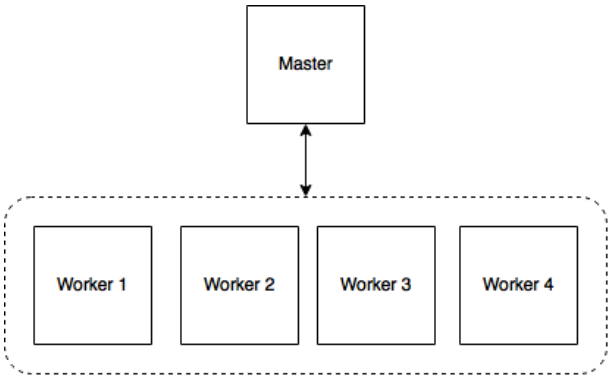

Fig. 4. Cluster Configuration

E. Experiments

We recorded the boot time and time taken by of each Ubuntu and OSv nodes to register on Spark cluster. Following table summarizes the timings:

TABLE I
NODE UP TIME

System	Local(s)	EC2(s)
Ubuntu	10	52
Osv	2	43

The first experiment is Scheduler throughput test. The experiment is performed using *Spark-perf* with the following configuration:

TABLE II
SCHEDULER THROUGHPUT CONFIGURATION

Parameter	Value
NUM_TRIALS	5
INTER_TRIAL_WAIT	3
CLOSURE_SIZE	0
RANDOM_SEED	5

And varying the NUM_TASKS from 1000 to 10000 with a step size of 1000. The results of the experiment for Ubuntu and OSv can be seen in Table III and Table IV respectively.

TABLE III
RESULTS: SCHEDULER THROUGHPUT (UBUNTU)

NUM_TASKS	Mean	Std Dev	Max	Min
1000	1.985	0.193	1.679	2.143
2000	2.616	0.626	2.513	3.889
3000	4.612	1.036	2.428	4.639
4000	3.389	1.312	2.966	5.937
5000	3.766	0.521	3.651	4.809
6000	5.206	0.311	4.652	5.382
7000	5.374	0.222	4.951	5.457
8000	5.873	0.207	5.866	6.308
9000	6.737	0.193	6.398	6.854
10000	7.304	0.206	7.066	7.57

TABLE IV
RESULTS: SCHEDULER THROUGHPUT (OSV)

NUM_TASKS	Mean	Std Dev	Min	Max
1000	6.054	4.326	4.787	14.532
2000	22.686	2.824	20.307	27.121
3000	33.219	6.410	33.011	46.712
4000	50.822	0.735	49.29	50.873
5000	61.358	2.834	60.679	67.001
6000	83.076	6.051	71.085	84.625
7000	98.36	4.739	88.737	99.17
8000	107.547	3.439	103.319	111.743
9000	118.048	0.631	117.466	118.996
1000	133.07	2.659	130.563	137.023

The next experiment is aggregate by Keys (Strings) and

the configuration is as follows:

TABLE V
AGGREGATE BY KEY (STRINGS)

Parameter	Value
NUM_TRIALS	5
INTER_TRIAL_WAIT	3
NUM_PARTITIONS	20
REDUCE_TASKS	20
PERSISTENT_TYPE	memory
KEY_LENGTH	10
VALUE_LENGTH	10
CLOSURE_SIZE	0
RANDOM_SEED	5

And varying the NUM_RECORDS (start 10000000), UNIQUE_KEYS (start 1000) and UNIQUE_VALUES (start 1000) with a step size of 1000. We increased all three values linearly and simultaneously. The results of the experiment can be seen in Table VI and VII.

TABLE VI
RESULTS: AGG BY KEY (STRINGS) UBUNTU

NUM_RECORDS	Mean	Std Dev	Min	Max
10000000	15.606	0.034	15.554	15.636
20000000	36.143	0.182	35.825	36.256
30000000	53.993	0.238	53.762	54.341
40000000	69.699	0.034	69.646,	69.727
50000000	85.642	0.225	85.175	85.662
60000000	103.944	0.207	103.851	104.329
70000000	114.425	0.227	114.136	114.691
80000000	139.536	0.400	138.715	139.588
90000000	155.076	0.592	154.437	155.885
100000000	175.803	2.100	175.681	180.196

TABLE VII
RESULTS: AGG BY KEY (STRINGS) OSV

NUM_RECORDS	Mean	Std Dev	Min	Max
10000000	1.321	0.273	1.405	1.325
20000000	26.606	0.523	26.595	27.709
30000000	50.492	0.199	50.087	50.525
40000000	67.982	1.260	67.873	70.599
50000000	103.223	0.265	102.753	103.375
60000000	121.819	1.669	121.711	125.304
70000000	141.618	0.699	141.454	143.011
80000000	163.35	0.645	162.096	163.556
90000000	187.448	6.285	183.651	198.471
100000000	370.666	1.036	370.101	372.527

VII. RESULTS

Figure 5 & 6 shows the Box plot for timings for Job Scheduler. This graph shows the median time to completion. The Box width represent the std deviation of the observations. We also plot maximum and minimum recorded value for reference.

It can be observed from Figures 5 & 6 that for Job Scheduler, OSv does not provide better performance as

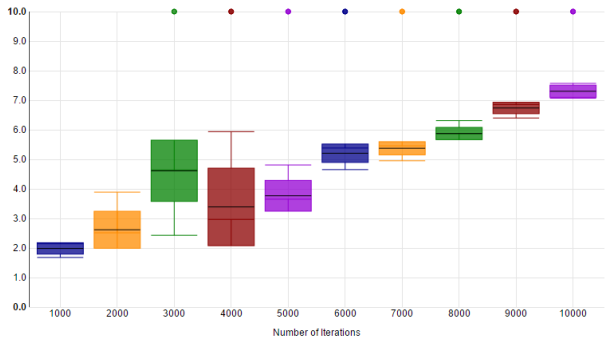

Fig. 5. Job Scheduler Timings for Ubuntu

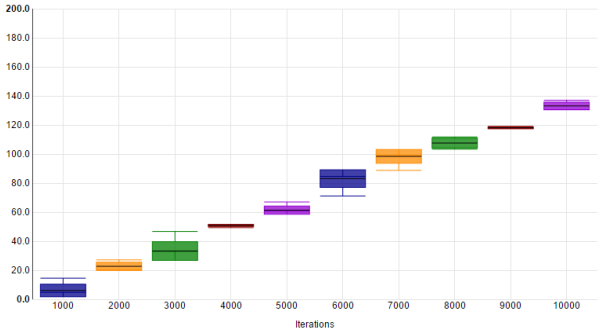

Fig. 6. Job Scheduler Timings for OSv

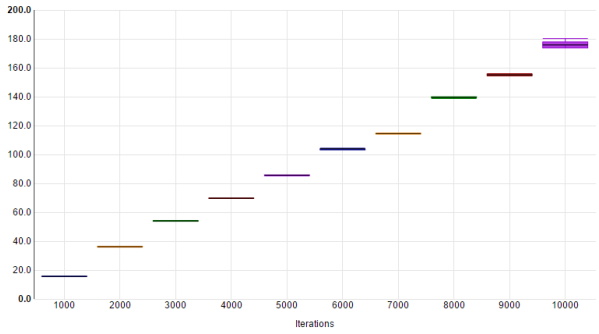

Fig. 7. Agg By Key Timings for Ubuntu

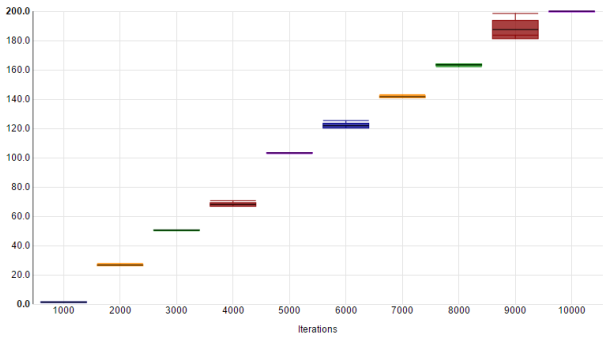

Fig. 8. Agg By Key Timings for OSv

compared to Ubuntu. But in fact the time difference is 10 fold. It is observed that as the number of task increased linearly, the job completion time increased linearly in both Ubuntu and OSv. Another interesting observation is that std deviation in Ubuntu is higher where as OSv gives very consistent result.

Figure 7 & 8 shows the Box plot for timings for Agg By Key Job. The chart description is same as that of Job Scheduler.

In case of Ubuntu the behavior is as expected, that is as the NUM_RECORDS increased linearly the timings also increased the same way. But in case of OSv we observe some interesting behavior, for NUM_RECORDS = 10000000, OSv outperform Ubuntu by a scale of 10, but after load increases we see an exponential jump in its timings and eventually for large inputs it fairs poorly as compared to Ubuntu.

VIII. CONCLUSION

It can be observed from the experimental results that OSv job completion time increases exponentially whereas Ubuntu gives linearly consistent performance. The latency was highly noticeable in the job aggregate-by-key where we move from 1000 keys to 2000 keys. We conjecture that as data and load increases for OSv its Java garbage collector starts to intervene regularly and consumes the CPU cycles from the spark job. For fair comparison we ran the job in sparks default java configuration where Min Heap Size and Max Heap Size were set 1GB. Using this configuration we conclude that OSv is not scalable to larger inputs. But in case of smaller dataset its shows better performance than Ubuntu. So if this problem can be recognized and solved, it has the potential to provide better performance and throughput. As OSv is in beta stage and has many stability issues, it need to be re-evaluated after a stable release is rolled out. Theoretically OSv should outperform Ubuntu, and needs to be tuned before it can be used in production environments. Once this bottleneck is identified and fixed, it can provide an efficient scalable solution where nodes can be added on the fly depending the job size. As image size is very small(approx 300 Mb) it can be scaled up/down very easily, and be less resource intensive on shared cloud resources such as EC2.

FUTURE WORK

Since Unikernels are relatively new there is lot of scope to improve existing functionalities and add new functionalities. Specifically for this project following are some key areas on which we would like to focus in future:

- Investigate reason for high latency in unikernels compared to traditional OS for certain kind of tasks such as scheduler throughput.
- Porting other parts of Apache Spark framework such as Python API and MLib to unikernel.
- Extend scope of benchmarking and evaluation by performing more tests from the Spark-perf suite.

GitHub Repo: Spark Osv Migration

$https://github.com/jivjot/spark/tree/branch-1.6$

REFERENCES

[1] Unikernels: Rise of the Virtual Library Operating System. Anil Madhavapeddy and David J. Scott

[2] The Next Generation Cloud: The Rise of the Unikernel. A Xen Project publication April 2015

[3] A Performance Evaluation of Unikernels. Ian Briggs, Matt Day, Yuankai Guo, Peter Marheine, Eric Eide. School of Computing, University of Utah.

[4] OSv – Optimizing the Operating System for Virtual Machines. Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav HarEl, Don Marti, and Vlad Zolotarov, Cloudius Systems

[5] Performance Evaluation of OSv for Server Applications. Ian Briggs, Matt Day, Eric Eide, Yuankai Guo, Peter Marheine - School of Computing, University of Utah.

[6] Spark Official Documentation for Cluster Overview $http://spark.apache.org/docs/latest/cluster-overview.html$

[7] Performance Prediction for Apache Spark Platform. K. Wang ; Dept. of Comput. Sci. & Eng., Univ. of Connecticut, Storrs, CT, USA ; M. M. H. Khan.