Performance Evaluation Comparison Between Docker and Hypervisor Virtualization

Group 4: Shane Conaboy, Anna DeVries, Mark Tiburu, Li Wang
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA
{conaboy.s, devries.an, tiburu.m, wang.li4}@northeastern.edu

***Abstract*—Hypervisor-based virtualization and container-based virtualization are widely used in high-performance computing applications. This research aimed to compare performance differences of the two virtualization techniques, specially for CPU, network, memory, disk, and I/O performance. This work utilized four benchmarks (IOzone for file system evaluation, LINPACK for testing CPU performance, Netperf for evaluating network bandwidth, and STREAM for measuring memory access) on three different environments (native server, Docker containers, and VMware workstation virtual machine). All experiments were run on the same server, which also spun up the containers and the virtual machine. The server and virtual machine ran CentOS 7, and the containers were built from source for each application. Each benchmark ran ten iterations on each platform, and the results were averaged across the runs. As expected, the virtual machine performed worst for LINPACK, Netperf, STREAM and write-only IOzone benchmarks. Further, the containers gave near native performance for the LINPACK and STREAM benchmarks; however, the native server greatly outperformed the containers during the Netperf benchmark. The read-only IOzone performance yielded unexpected behavior, showing that the virtual machine outperformed both the native server and the containers.**

***Index Terms*—container; hypervisor; benchmarking; virtualization.**

I. Introduction

Virtualization technologies promise flexibility, scalability and security. Virtualization allows multiple users or applications to coexist on a single hardware platform while each program appears to have full access to the machine. One application for such technologies is in the area of high performance computing (HPC) where multiple users share access to a single high-performance server. The applications being run on the server may have conflicting configurations, such as requiring different operating systems. Virtualization can provide the solution to such a problem via its isolation capabilities. Performance is also another important concept in HPC and must be considered when selecting the virtualization technology since some types of virtualization may require more overhead than others.

Two types of virtualization technology are hypervisor-based and container-based virtualization. Hypervisor-based virtualization typically involves running multiple virtualized systems with independent operating systems on a single physical machine. Container-based virtualization provides isolation to groups of processes but does not use independent operating systems for each container. Container technology is more lightweight with less overhead and can offer near-native performance, while hypervisor virtualization can be more flexible and provide a fully isolated, and thus more secure, environment.

This project conducts a performance evaluation comparison between hypervisor-based and container-based virtualization. In particular, it evaluates the CPU performance, memory access speed, disk access speed, network performance, and I/O speed between the two virtualization technologies. The project utilizes four benchmarks (IOzone, LINPACK, Netperf, and STREAM) and three testing platforms (native CentOS 7, docker containers, and VMware Workstation virtual machine).

The remainder of the paper is structured as follows: Section II provides a background of the virtualization technologies utilized in this project; Section III describes the four different benchmarks; Section IV discusses the methodology; Section V provides the experimental results and analysis; Section VI gives concluding remarks; and Section VII recommends possible future work.

II. Background

Virtualization technology utilizes hardware resources to simulate one or more environments. There are five major types of virtualization: data virtualization, desktop virtualization, server virtualization, operating system virtualization, and network functions virtualization. This project focuses on operating system level (OS-level) virtualization which occurs at the kernel. As illustrated in Figure 1, OS-level virtualization utilizes a single physical host machine to run multiple isolated user space instances. This isolation technique reduces hardware costs and increases system security [1].

The next two subsections serve to describe two possible implementations of OS-level virtualization - hypervisor-based and container-based virtualization - and the third subsection explains one form of container-based virtualization (Docker).

A. Hypervisor Virtualization

A hypervisor is software that creates and runs guest virtual machines (VMs) independent of the host's hardware. The hypervisor either works with the host's operating system or directly manages the sharing of hardware resources to support the multiple guest VMs [2].

Fig. 1. OS-Level Virtualization [1].

Two main types of hypervisors are "Type 1" or bare-metal and "Type 2" or hosted. A Type 1 hypervisor runs directly on the host system's hardware as a lightweight operating system; whereas, the Type 2 hypervisor runs like a computer program with a software layer on the operating system [2].

Type 1 hypervisors are the most common. This type requires virtualization software to be directly installed onto the host's hardware in place of the operating system. Type 1 hypervisors provide complete isolation between applications and system resources, making this approach extremely secure. This hypervisor type is often found in data centers for enterprise companies [2]. Software examples include VMware ESX and ESXi, Microsoft Hyper-V, Citrix XenServer, and Oracle VM [3].

On the contrary, Type 2 hypervisors run on top of the existing host's operating system. This allows users to run different operating systems on a single host system; however, this structure creates a much higher latency than its Type 1 counterpart. Type 2 hypervisors are often used by end users or for software testing [2]. Software examples include VMware Workstation/Fusion/Player, VMware Server, Microsoft Virtual PC, Oracle VM VirtualBox and Red Hat Enterprise Virtualization [3]. Figure 2 illustrates these two types of hypervisors.

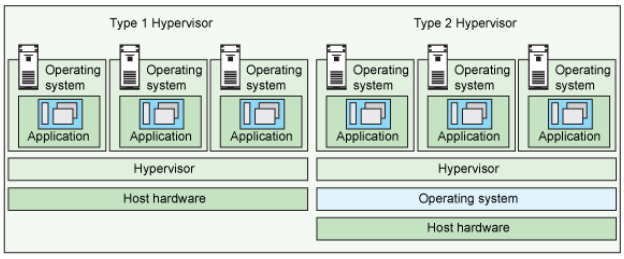

Fig. 2. Hypervisor Virtualization [3].

Several benefits of hypervisors include speed, efficiency, flexibility and portability. First, hypervisors allow VMs to be created almost instantly. While bare metal servers may take hours to build and configure custom hardware, hypervisors can be destroyed and created within minutes due to their ability to provision resources as needed [2].

Second, since physical resources are provisioned as necessary, hypervisors are more efficient (both cost and energy wise) when running several VMs as compared to running multiple underutilized physical machines for the same tasks [2].

Finally, hypervisors allow their VMs (containing different operating systems and applications) to be transferred among different hardware types, increasing their flexibility and portability over natives. Hypervisors allow different OS's to run no matter the underlying hardware type [2].

B. Containers

While hypervisor virtualization is utilized to create entire guest machines, including a full operating system (OS), containers package individual applications and their dependencies. Containers run on top of the existing host OS as isolated processes in user space. Since a full operating system is not included in containers, they are often much smaller than VMs and start up faster [5]. Figure 3 illustrates the structural differences between VMs and containers.

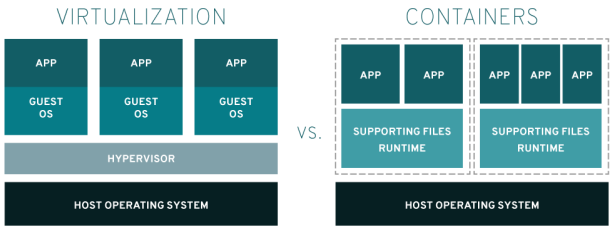

Fig. 3. Virtualization vs. Containers [6].

Containers are implemented via Linux *namespaces* and *control groups*. A *namespace* wraps a system resource in an abstraction that makes it appear to processes within that namespace that they have their own isolated instance of that resource [8]. Such resources could be the file system root directory, process IDs, mount points, and network resources. *Namespaces* create isolation between individual containers as well as the host system. Processes within the container appear, from their perspective, to run on a separate system.

Control groups (*cgroups*) allow allocation of system resources, such as CPU time, memory, or network bandwidth, among groups of processes [7]. Much like how a hypervisor can restrict system resources available to the virtual machines, *cgroups* allow containers to restrict resources available to the packaged application and associated processes.

Since containers exist at the application layer, they have several benefits as compared to VMs. First, containers are smaller (often tens of megabytes in size [5]) because they do not require a full copy of an OS to be packaged. And since containers create processes that utilize the already-running host OS, they are often easier to launch and shut down as compared to VMs that need to boot up and shut down the entire guest OS. Containers can be transported across systems as well, however the systems must share the same OS kernel. In this aspect, VMs are more flexible since they can be run independent of the underlying host configuration.

C. Docker

Docker is a technology built on top of traditional Linux container concepts. It provides a framework for the creation,

running, and distribution of containers. Docker also provides many pre-built container images that can be used standalone or included in a custom containerized application.

III. Benchmarks

Benchmarks are computer programs that assess performance in computer systems. The following subsections detail the benchmarks utilized in this research project.

A. IOzone

IOzone is a file system benchmark that generates and measures various file operations. This work focuses on read and write accesses. The write access test measures system performance on writing a new file. This includes writing data, storing the file, and remembering where the data is stored on the storage media (metadata). The read access test measures system performance to find and read an existing file [15].

B. LINPACK

The LINPACK benchmark reflects the overall performance of the system's central processing unit (CPU). In this program, the system must solve a dense system of linear equations utilizing double precision floating point operations [4].

In particular, LINPACK utilizes a random matrix $\mathbf{A}$ of size N and a vector $\mathbf{B}$ as $\mathbf{A} * \mathbf{X} = \mathbf{B}$. The matrix is real, general and dense. The matrix elements are pseudo-randomly chosen between (-1.0, 1.0). This algorithm consists of a collection of Fortran subroutines and solves numeric linear algebra with a decompositional approach as [4]:

1) Lower Upper (LU) factorization of $\mathbf{A}$
2) The LU factorization is used to solve the linear system $\mathbf{A} * \mathbf{X} = \mathbf{B}$

The results are carried out to 64-bit precision and performance is measured in terms of Megaflops (millions of floating point operations per second):

$$mflops = operations/(cpu * 1000000) \quad (1)$$

There are three distinguishable zones seen with changing values of N in the LINPACK benchmark. These zones result from changing CPU behavior. First, the rising zone occurs when the processor in under-challenged. Second, the flat zone occurs when the processor is performing at top efficiency. Third, the decay zone occurs when cache memory is challenged and the system must access main memory.

This project utilizes N = 1000 for the LINPACK benchmarks. This is the traditional value chosen for N and only requires the CPU to access the cache, rather than accessing main memory, because it keeps the CPU in the "rising zone". This ensures that only the CPU performance will be tested, rather than memory access.

C. Netperf

Netperf is a networking benchmark [10]. This benchmark focuses on data transfer and request/response performance utilizing either TCP or UDP over IPv4 and IPv6. Netperf also measures throughput by utilizing bulk data between the server and client.

D. STREAM

The STREAM benchmark measures sustainable memory bandwidth and corresponding computation rate for simple vector kernels. The benchmark is designed to work with data sets much larger than the cache size of the system and the code is structured such that data re-use is not possible [9]. Table I lists the kernel operations run by STREAM.

TABLE I
STREAM Kernel Operations

Name	Kernel
COPY	`a(i) = b(i)`
SCALE	`a(i) = q * b(i)`
SUM	`a(i) = b(i) + c(i)`
TRIAD	`a(i) = b(i) + q * c(i)`

IV. Methodology

The experiments were performed on three platforms: a native environment, docker containers, and a hypervisor virtual machine. All benchmarks on both the native and hypervisor VM platforms were compiled from their source codes. On the docker platform, four different container images for the various benchmarks were created from the ground up as discussed below. The native server had the following configuration: Intel(R) Xeon(R) CPU E5-2430 with 6 physical cores, 70.6 GB RAM, 1 GB/s Ethernet, and CentOS 7.

The following outlines the methodology for each individual benchmark.

i. *IOzone*: This project utilized the source code from IOzone's main site [15]. IOzone was configured as follows: time resolution = 0.000001 seconds, processor cache size = 1024 kB, processor cache line size = 32 bytes, file stride size = 17*record_size, and record size = 64 kB. The message sizes tested were 64 kB, 512 kB, 4K kB, and 32K kB. The set of IOzone tests were run twice: once with virtual machine write caching enabled, and once with it disabled. The benchmark was downloaded with *wget* and installed utilizing *rpm* for each environment. The docker container was built from scratch, performing the same download and install as previously mentioned. IOzone ran 10 iterations at each message size utilizing the following parameters: *./iozone -RaI -r 64 -i 0 -i 1*.

ii. *LINPACK*: This project obtained LINPACK's source code from Florida State University's Department of Scientific Computing [12]. The docker image was created and built from scratch, copying the source code from the native environment. The source code was compiled utilizing GNU C compiler version 4.8.5 with the level 3 compiler optimization and the binary was run 10 times on each environment. LINPACK was configured with N = 1000.

iii. *Netperf*: Netperf source code was obtained from Hewlett Packard's GitHub Repository and compiled without any compiler optimization switches [16]. For the VM and native platforms, a server was started by *netserver* and a client transmitted data with the *netperf -a DATASIZE*

command to the server. For docker, a image was created by scratch, copying source code from the native server and compiling the benchmark with GNU C compiler version 4.8.5. On this environment, the server was started by *docker run -dt –net=host –name servername inmagename* and a client transmitted data to the server with *docker run -it –net=host imagename netperf -a DATASIZE –l 10 -i 10*. Each experiment ran 10 iterations and the following message sizes were tested: 64 kB, 128 kB, 512 kB, 1K kB, 2K kB, 4K kB, 8K kB, 16K kB, and 32K kB.

iv. *STREAM*: This project obtained STREAM from the University of Virginia's Department of Computer Science [13]. The source was compiled utilizing GNU C compiler version 9.3.0 with level 1 optimizations. The docker image was created and built from scratch similar to LINPACK. Each platform ran 10 iterations of the benchmark.

V. Results and Analysis

This section includes four subsections. These cover the results and analysis of each individual benchmark on all three environments.

A. IOzone Results

IOzone measures file I/O performance in terms of speed (kB/second). Table II provides the mean write speeds, mean read speeds, and standard deviations across all the platforms. These results are also illustrated in Figures 5 and 6.

Figure 5 compares the read performance across all platforms. As shown, the VM outperforms both the Native and Docker environments. This difference owes to spatial locality. The virtual machine image is split into six different files on the host system's physical memory. Therefore, any files written in the VM will also be stored in one of these six files. As the benchmark reads files from memory, the OS updates and caches what it predicates will be the next needed piece of memory. One method of prediction is utilizing spatial locality which caches memory located physically nearby recently accessed memory. The concentrated, smaller files of the VM give a greater probability that the cached memory was accurately predicted, leading to higher read performance seen on the VM. For the other file sizes, the native environment largely outperformed the containers and VM at 64 kB, but docker slightly outperformed native for larger file sizes. Thus, this elaborates why the VM is several kBs behind native. Flash Read Cache can accelerate virtual machine performance by utilizing host-resident flash device as cache [19]. This supports write-through or read caching, with data read being satisfied from the cache this speeds up throughput if present.

Since virtual machine disk images are stored as normal files on the host file system, the host OS can utilize caching techniques as with any other file. One such technique that hypervisors can implement is called *write caching*, which buffers writes made by the guest OS, allowing the writes to proceed asynchronously. The write operation can then be reported as having completed before the data has actually be written to disk [18]. This can be combated by disabling the write caching on VMware, providing a more accurate write performance. Figure 4 illustrates the different write performances of the two configurations. The VM with caching disabled was then utilized to compare the VM against the other platforms, as seen in Figure 6.

For the other file sizes, the native environment largely outperformed the container and VM at 64 kB, but docker slightly outperformed both native and VM for larger file sizes during write. VM data writes are written to the backing storage and traversing to the backing storage to store data and back to virtual host does incurs performance hit usually when the cache size is too small to accommodate the entire data set at ago, hence the reason why the performance degradation of the VM with write. The data alignment could influence how it is written to disk, the alignment of the data on the native hardware was oriented well for writing the 64kB file hence the better performance, yet this alignment notion was not replicated to the rest of the data set. All the three platforms performance were not optimal during write, perhaps, there were other heavy workloads running during this studies which might have influence our findings greatly.

TABLE II
IOzone Performance

Native				
Size (kB)	**Write (kB/s)**	**Read (kB/s)**	**Write Std**	**Read Std**
64	168,571.10	15,639.60	14,874.11	16,643.84
512	8,176.50	39,962.80	319.99	7,207.52
4K	7,507.90	103,574.80	76.14	137.29
32K	7,422.50	145,115.30	126.44	10,303.86
Virtual Machine				
Size (kB)	**Write (kB/s)**	**Read (kB/s)**	**Write Std**	**Read Std**
64	53,299.60	144,423.00	8,050.44	17,618.73
512	7,426.60	142,261.50	615.76	17,191.57
4K	7,442.20	245,527.30	140.39	41,186.84
32K	7,351.30	288,621.90	72.01	51,129.22
Docker Containers				
Size (kB)	**Write (kB/s)**	**Read (kB/s)**	**Write Std**	**Read Std**
64	15,613.10	22,067.20	5,792.73	6,493.61
512	13,450.70	32,622.40	2,443.80	8,563.10
4K	10,027.70	29,233.70	2,371.20	13,295.82
32K	10,525.10	26,257.70	2,880.80	5,929.18

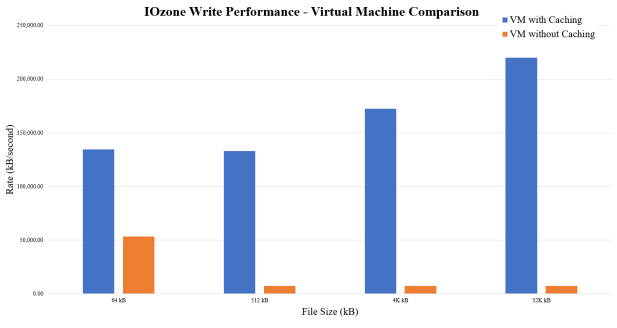

Fig. 4. Virtual Machine Caching Comparison.

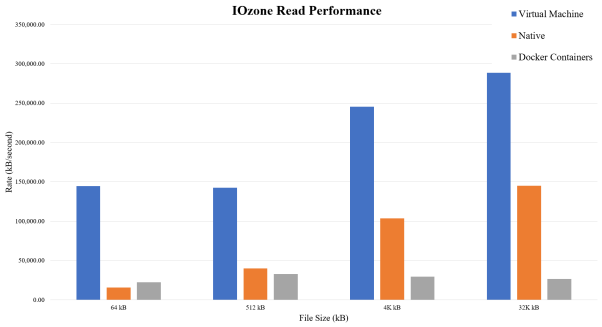

Fig. 5. IOzone Read Performance.

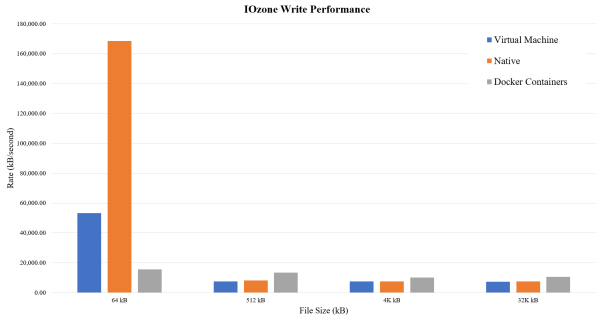

Fig. 6. IOzone Write Performance.

B. LINPACK Results

As mentioned above, LINPACK measures CPU performance in computer systems by the rate of execution in floating point operations per second. Table III provides the sample mean and root-mean-square-error of the LINPACK performance results across the three different platforms, and Figure 7 illustrates the mean performance across the platforms. The native environment achieved the highest mean performance at 2,123.7 MFLOPS with the docker container performing near native at 2,118.7 MFLOPS. The virtual machine performed worst at 1,981.2 MFLOPS.

TABLE III
LINPACK RATE OF EXECUTION COMPARISON

Platform	Mean (MFLOPS)	RMS
Native	2,123.695	44.311
Docker Containers	2,118.670	74.840
Virtual Machine	1,981.232	74.685

The native Linux and docker containers performed nearly identical, with a difference of approximately 5 MFLOPS. These platforms performed 7% better than the VM. The performance similarity between docker and the native environment owes to the little OS involvement for the LINPACK execution. Since docker containers are implemented through *namespaces* and *control groups* as mentioned in the Background Section, rather than through a hypervisor as a standalone full operating system, there is little overhead because containers run on the application layer with direct access to hardware. This allows

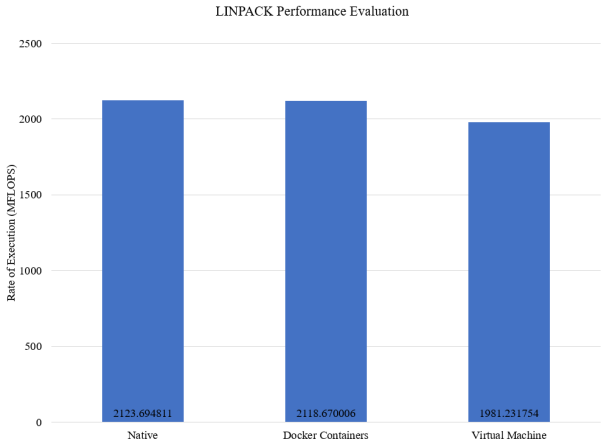

Fig. 7. LINPACK Mean Rate of Execution.

docker to have near native performance when running CPU heavy workloads.

VMs, on the other hand, rely on communication between the hypervisor and host OS to utilize hardware such as a CPU. A VM running LINPACK would first need to create processes that are sent and queued in the hypervisor, then the hypervisor would pass these processes to the host OS. Finally, the host operating system schedules the processes to physical cores according to task priority. These VM processes, however, are not the only instructions competing for computation time. The host also has native processes such as those necessary to maintain the operating system. This scheduling priority and overhead of submitting processes to the hardware incurs time and takes a performance hit for virtual machines especially in CPU heavy workloads.

C. Netperf Results

Table IV represents the mean throughput obtained from Netperf. Higher throughput refers to better performance than lower throughput. Figure 8 illustrates the mean throughput at each message size. From the data, the native environment

TABLE IV
NETPERF THROUGHPUT PERFORMANCE COMPARISON

Message Size	VM	Native	Docker
64	1001.64	11538.08	8235.91
128	951.91	11553.39	8705.89
256	966.76	11236.39	7928.42
512	1034.84	11219.85	8627.47
1K	966.90	11272.47	8234.17
2K	977.80	11529.47	8167.53
4K	967.57	11392.40	8182.21
8K	1011.33	11270.97	9098.99
16K	983.17	11230.40	8507.94
32K	984.23	11299.52	8175.12

shows the highest throughput and the VM shows the lowest. Specifically, the native environment transmits data from client to server 0.91x faster than the VM and 0.29x faster than the docker containers.

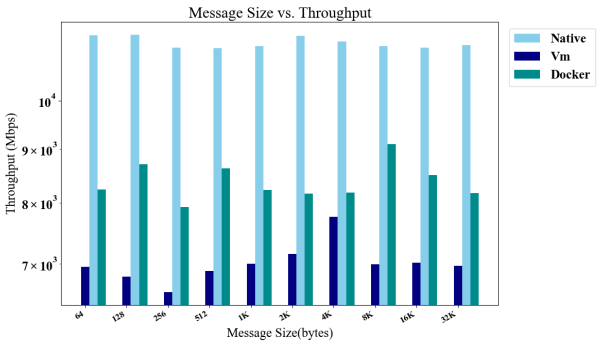

Fig. 8. Netperf performance comparison of VM, Native and Docker.

The native performance owes to the hardwired Ethernet. The machine utilizes Ethernet cable CAT6, which in certain cases can transmit data of up to 10 GB/s.

Docker also performs significantly better than the VM, but not as well as the native environment. This result signifies that the extra layers of networking on Docker incurs additional performance overhead. As could be seen in 9, Dockers add containers on the host to a veth pair and connects it to a bridge and the bridge to the network via NAT. These extra layers contribute to the performance overhead.

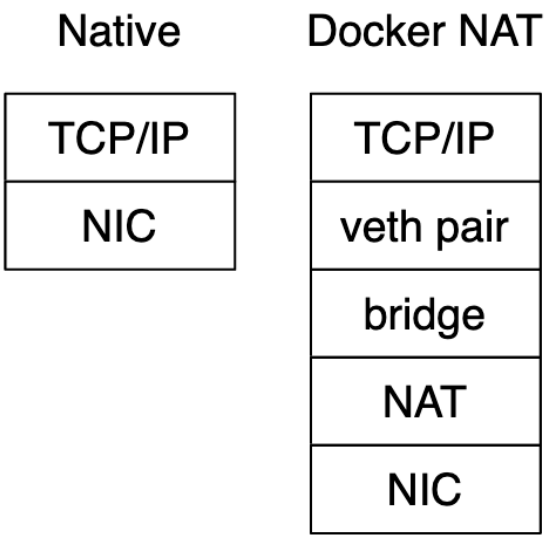

Fig. 9. Network configurations [20].

However, the VM performs worst compared to the other platforms. This performance results from the virtual machine's architecture. A VM emulates the kernel and host devices of an actual machine (e.g. network card, required the most for this benchmark). This causes lower performance results because the VM is spending a large amount of time abstracting an entire system, requiring it to send all processes through the hypervisor to the host OS for scheduling.

D. STREAM Results

Table V shows the mean and standard deviation memory speeds for the STREAM benchmark experiments. Figure 11 illustrates the mean speeds of each environment. Docker performed nearly as well as native, while the results on the VM were approximately 16% slower. Both docker and the VM had greater variation in the results as compared to native performance, with the VM having the greater variation of the two.

TABLE V
STREAM PERFORMANCE RESULTS (MB/S)

	Native		Docker		VM	
Operation	**Mean**	**Std**	**Mean**	**Std**	**Mean**	**Std**
Copy	11658	416	11604	590	9800	1320
Scale	11322	377	11274	584	9498	1278
Add	12750	473	12711	712	10656	1557
Triad	12631	389	12563	626	10502	1448

The poorer performance of virtual machines is due to virtual memory translation. When processes access memory, the operating system performs a translation from the virtual to physical address. Since VMs are virtualized, the guest OS physical address must also be translated to a host physical address via the hypervisor. This extra translation layer, depicted in Figure 10, incurs extra overhead for any memory accesses and is reflected in the performance results.

However, since containers run directly on the host, there is no extra overhead for performing address translation. As seen in the results, this allows container processes to access memory nearly as fast as native processes. The slight performance degradation and higher variance could be a result of overhead associated with docker and control groups. Docker allows limiting the memory available to containers, which requires monitoring the container's memory usage. Although the STREAM benchmark was run without memory limits, Docker can still report statistics on the container's memory usage via the memory control group. Both operations can add some overhead and impact performance.

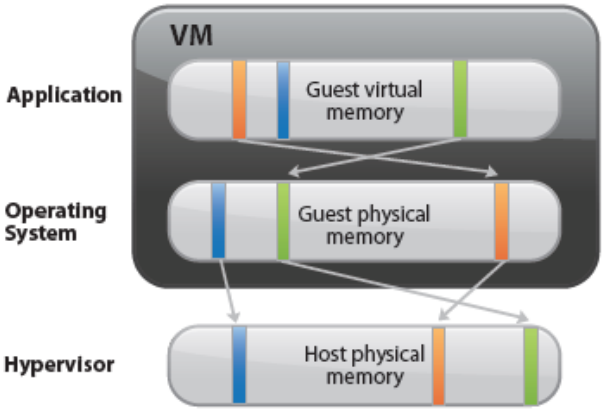

Fig. 10. Virtual Machine Memory Translation Layers [11].

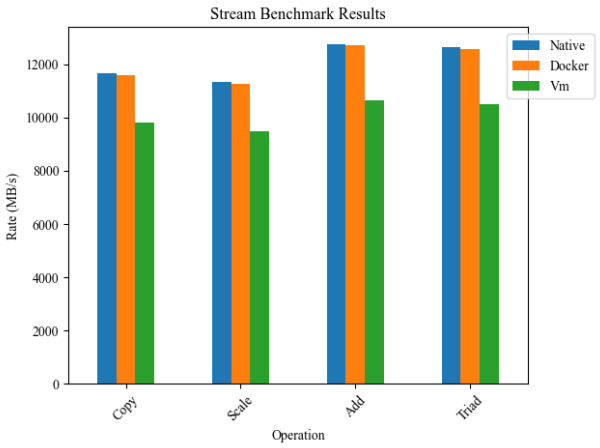

Fig. 11. STREAM benchmark mean memory speeds.

VI. Conclusion

This paper discussed two types of virtualization techniques, type 2 hypervisors and containers, and assessed the performance of each when performing different types of tasks. Hypervisor virtualization allows multiple virtual machines with independent operating systems to run on the host hardware. Similarly, containers allow multiple applications to run on a single system as isolated processes. Four benchmarks programs were run to compare performance: LINPACK for CPU performance, IOzone for disk bandwidth, Netperf for network bandwidth, and STREAM for memory bandwidth. Each benchmark was run on the native machine and within each virtualization environment.

The virtual machine performed worst in the LINPACK, Netperf, and STREAM benchmarks. These results are expected as the hypervisor layer causes overhead when executing these types of tasks. The container implementation performed nearly as well as native for LINPACK and STREAM, which is also expected since containers have direct access to the underlying hardware and do not require any extra virtualization layers. However, containers do have extra network address translation layers, which caused lower performance, as compared to native, in the Netperf benchmark. Somewhat surprisingly, the virtual machine performed best in terms of the read and write bandwidth tests of the IOzone benchmark. The write bandwidth performance is due to a hypervisor technique called *write caching*, which buffers writes from the guest virtual machine and commits them to disk later, allowing the write calls to return much faster in the guest operating system. Disabling write caching caused the write performance to drop to what was measured when running the benchmark natively. The increased read performance for the virtual machine is hypothesized to come from the hypervisor executing a prefetching strategy when performing disk reads in case nearby data is requested from the program in the future. Spatial locality could also be a factor since the virtual machine's hard disk is stored as a regular file on the host operating system.

Overall, both the virtual machine and containers performed comparably to the native machine, even with some performance degradation for certain types of workloads. Neither has a clear advantage over the other if looking strictly at performance metrics. Containers may be more suited to network-based applications as this is where the biggest difference was observed. When deciding which virtualization technology to use, it is important to also consider other factors, such as ease of use, overall size, and security characteristics.

VII. Future Work

This project compared the performance of a type 2 hypervisor versus docker containers for network, disk, CPU and memory intensive workloads. One possible future research topic would be to repeat this experiment with a type 1 hypervisor instead. As mentioned in the Background Section, type 1 hypervisors sit on top of the hardware (removing the extra layer of a host operating system). This may incur less overhead and result in more competitive performance results then its type 2 counterpart.

Another suggestion would be to evaluate the security and isolation characteristics of virtual machines versus containers. Virtual machines have a hypervisor and independent operating system, giving a greater level of isolation compared to containers. Researching the effects of this isolation and security could help inform engineers which solution to choose when implementing systems.

A third suggestion is to compare the effects of compiler optimizations on virtualization. This research would focus on whether the results are affected uniformly with differing optimizations levels across the virtualization techniques and native environments.

Finally, the last suggestion would be to further investigate IOzone. As mentioned above, IOzone results are shewed due to VMs abilities to cache disk information. Future work should look into methods that may more accurately depict VMs I/O disk performance. For instance, a performance evaluation of virtual machines on a shared file system with the host machine may provide more accurate results.

References

[1] "What is virtualization?," *Red Hat*. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-virtualization. [Accessed: Mar. 9, 2021].

[2] "What is a hypervisor?," *VMware*. [Online]. Available: https://www.vmware.com/topics/glossary/content/hypervisor. [Accessed: Mar. 24, 2021].

[3] "What is Hypervisor and what types of hypervisors are there?," *VapourApps*, May 13, 2016. [Online]. Available: https://vapour-apps.com/what-is-hypervisor/. [Accessed: Mar. 24, 2021].

[4] J. Dongarra, *The LINPACK Benchmark: An Explanation*. In: Houstis E.N., Papatheodorou T.S., Polychronopoulos C.D. (eds) Supercomputing. ICS 1987. Lecture Notes in Computer Science, vol 297. Springer, Berlin, Heidelberg, 2005. https://doi.org/10.1007/3-540-18991-2_27

[5] "What is a Container?," *Docker*. [Online]. Available: https://www.docker.com/resources/what-container. [Accessed: Apr. 4, 2021].

[6] "What's a Linux container?," *Red Hat*. [Online]. Available: https://www.redhat.com/en/topics/containers/whats-a-linux-container. [Accessed: Apr. 4, 2021].

[7] Red Hat, ”Resource Management Guide,” [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide. [Accessed: 4 Apr. 2021].
[8] ”namespaces(7) - Linux manual page,” *Michael Kerrisk*. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html. [Accessed: Apr. 4, 2021].
[9] ”STREAM FAQ’s,” *Department of Computer Science, University of Virginia*. [Online]. Available: https://www.cs.virginia.edu/STREAM/ref.html. [Accessed: Apr. 4, 2021].
[10] Netperf, https://arxiv.org/pdf/1704.05592.pdf [Accessed 7 April 2021]
[11] VMware, Inc., ”Understanding Memory Resource Management in VMware ESX Server,” 2009. [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/perf-vsphere-memory_management.pdf. [Accessed 7 April 2021].
[12] ”The LINPACK Benchmark,” *Florida State University Department of Scientific Computing*, July 12, 2019. [Online]. Available: https://people.sc.fsu.edu/ jburkardt/c_src/linpack_bench/linpack_bench.html. [Accessed: Feb. 15, 2021].
[13] ”STREAM,” *Department of Computer Science, University of Virginia*. [Online]. Available: https://www.cs.virginia.edu/stream/. [Accessed: Apr. 10, 2021].
[14] IOzone DockerHub, https://hub.docker.com/r/ashael/iozone. [Accessed: Mar. 15, 2021].
[15] IOzone Source Code, http://www.iozone.org/src/current/iozone-3-491.x86_64.rpm. [Accessed: Mar. 15, 2021].
[16] Netperf Source Code, https://github.com/HewlettPackard/netperf/releases/tag/netperf-2.7.0. [Accessed: Apr. 10, 2021].
[17] A Comprehensive introduction to Docker, Virtual Machines, and Containers, https://www.freecodecamp.org/news/comprehensive-introductory-guide-to-docker-vms-and-containers-4e42a13ee103/. [Accessed: Apr. 10, 2021].
[18] ”Host Input/Output Caching,” *Oracle*. [Online]. Available: https://docs.oracle.com/en/virtualization/virtualbox/6.0/user/iocaching.html. [Accessed: Apr. 17, 2021].
[19] ”About VMware vSphere Flash Read Cache,” *VMware*. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.storage.doc/GUID-07ADB946-2337-4642-B660-34212F237E71.html. [Accessed: Apr. 13, 2021]
[20] ” Network configurations,” https://course.ece.cmu.edu/ ece845/sp18/docs/containers.pdf [Accessed: Apr. 13, 2021]