

Universität Hamburg
Technische Universität Hamburg Harburg

Institut für Technische Bildung und Hochschuldidaktik
Studiengang Lehramt an Beruflichen Schulen
Berufliche Fachrichtung Elektrotechnik

Bericht: Projekt „Parallelrechnerevaluation“

Container für HPC

Gutachter

Erstgutachter/in: Dr. Michael Kuhn

Zweitgutachter/in: -

Abgabedatum: 05.09.2019

Verfasser:

Name: Juri Bunge

Adresse: Hammer Weg 36, 20537 Hamburg

Email: juri.bunge@web.de

Lehramt an beruflichen Schulen

Fachrichtung: Elektrotechnik/Informationstechnik

Unterrichtsfach: Berufliche Informatik

Matrikelnummer: 7071837

Inhalt

| | |
|--|----|
| Lesehinweise | 1 |
| Motivation | 1 |
| Einleitung | 1 |
| Begriffsklärung..... | 2 |
| Versuchsaufbau | 3 |
| Auswahl der Komponenten..... | 3 |
| Auswahl der Hardware | 3 |
| Auswahl des Hypervisors | 3 |
| Auswahl der Container Engine..... | 3 |
| Auswahl der Betriebssysteme..... | 4 |
| Einrichten der Testumgebung | 5 |
| Installation von Docker und Einrichten der Container | 5 |
| Installation von VirtualBox und Einrichten der VM | 5 |
| Benchmark | 6 |
| Sysbench..... | 6 |
| Geekbench | 6 |
| Read/Write Test mit dd..... | 7 |
| Bewerten und Einordnen der Benchmark Ergebnisse..... | 7 |
| CPU Benchmark | 7 |
| RAM Benchmark..... | 8 |
| File IO Benchmark | 9 |
| Geekbench | 9 |
| Sequenzielles Lesen/Schreiben mit dd | 11 |
| Fazit der Benchmarks | 11 |
| Vorteile von Containern | 11 |
| Nachteile von Containern..... | 15 |
| Image Baumstruktur | 16 |
| Automatisches, geplantes und manuelles Hochfahren | 16 |
| Homogene Umgebung..... | 17 |
| Atomare Aufgabenverteilung | 17 |
| Anwendungsmöglichkeiten von Containern | 17 |
| Datenbanken | 17 |
| Monitoring | 17 |
| CMS..... | 17 |

| | |
|-----------------------------|----|
| Backup..... | 17 |
| Basis Image..... | 17 |
| Container für HPC..... | 18 |
| Ausblick | 19 |
| Fazit..... | 20 |
| Anhang | 22 |
| Abbildungsverzeichnis | 23 |
| Literaturverzeichnis..... | 25 |

Lesehinweise

Motivation

Im Rahmen meiner Ausbildung zum Fachinformatiker für Systemintegration bin ich bereits vor einigen Jahren auf Docker und die Containervirtualisierung aufmerksam geworden. In meinem Betrieb wurden zu diesem Zeitpunkt weiterhin Virtuelle Maschinen (VM) verwendet, da die Technik in den Augen der Geschäftsleitung noch nicht ausgereift war. Außerdem war es zu der Zeit schwer abschätzbar, in welchem Umfang sich die Containervirtualisierung durchsetzen wird. Ein zuverlässiger Blick in die Zukunft als „Early Adopter“ ist in der IT-Welt üblicherweise nicht einfach.

Im Zuge meines Studiums bin ich dann erneut auf Docker und Container gestoßen und mir wurde bewusst, dass ich mich früher oder später mit dem Thema auseinandersetzen sollte und möchte. Als angehende Lehrkraft ist es wichtig neuen Technologien gegenüber aufgeschlossen zu sein und nach dem Studium mit neuem Wissen an die Schulen zu kommen.

Ein weiterer wichtiger Themenblock der IT ist bei mir bisher beruflich bedingt weitestgehend in den Hintergrund getreten. Linux-Systeme habe ich weder zur Zeit meiner Berufstätigkeit noch privat verwendet. Mir ist jedoch bewusst, dass ich meine Kenntnisse auch hier weiter ausbauen sollte.

Im Projekt „Parallelrechnerevaluation“ aus dem Wahlpflichtbereich des Informatikteils meines Studiums hat sich nun die Möglichkeit angeboten beide Themen zu verbinden, da die Containervirtualisierung typischerweise mit Linux in Verbindung steht.

Einleitung

„Containervirtualisierung (auch: Betriebssystemvirtualisierung) ist eine Methode, um mehrere Instanzen eines Betriebssystems (als sog. „Gäste“) isoliert voneinander auf einem Hostsystem zu betreiben.“ (Wikipedia 2019).

Dem gegenüber steht die Virtualisierung mittels eines Hypervisors. Container stehen auf den ersten Blick in Konkurrenz zu VMs und daher habe ich mich entschieden, einen Vergleich von VM und Container durchzuführen.

Es gibt allgemein zwei Typen von Virtualisierung:

Typ-1 Virtualisierung (Bare-Metal-Virtualisierung):

Ein Hypervisor wird direkt auf der Hardware installiert und verwaltet die VM. VMware ESXi, Microsoft HyperV und Citrix XenServer sind typische Beispiele.

Typ-2 Virtualisierung (Desktop-Virtualisierung):

Das System setzt ein Betriebssystem als Basis voraus auf dem der Hypervisor als Anwendung läuft. VMware Workstation und Oracle Virtualbox sind typische Beispiele.

Container stehen zu beiden Typen in Konkurrenz, da mithilfe der Container Technologie Aufgaben aus beiden Bereichen gelöst werden können. Um den Aufwand in einem angemessenen Rahmen zu halten, wurde im Test lediglich eine Typ-2 Virtualisierung verwendet.

Das Projekt lässt sich grob in drei Phasen einteilen:

1. Recherche

Ich habe Vorwissen im Bereich Virtualisierung und Systemintegration, Grundlagenwissen für Container und Linux musste ich mir neu aneignen.

2. Benchmark

Performance ist ein wichtiger Bestandteil der Virtualisierung und wenn Container VMs Konkurrenz machen wollen, müssen sie mindestens ähnlich performant sein. Ein Vergleich mithilfe eines Benchmarks gab mir außerdem eine Aufgabe mit angemessenem Schwierigkeitsgrad, um mein neues Wissen anzuwenden.

3. Einordnen der Ergebnisse

Meine eigenen Daten habe ich zunächst losgelöst vom Internet aufgrund eigener Erfahrungen ausgewertet. Danach habe ich andere Ergebnisse recherchiert und Überlegungen angestellt, wie Vorteile von Container genutzt werden können.

In diesem Bericht werde ich zunächst ein paar Begriffe aus der Containervirtualisierung klären und meinen Versuchsaufbau beschreiben. Danach werde ich die Auswahl der Komponenten aufgrund meiner Recherche begründen. Anschließend beschreibe ich kurz die Einrichtung der Container Engine und des Hypervisors. Danach werde ich die Ergebnisse auswerten und einordnen. Um Vor- und Nachteile von Container zu erläutern, werde ich zum Teil auf die Struktur von Containern eingehen.

Danach hatte ich zunächst geplant mit ansatzweise ausgearbeiteten Konzepten Beispiele für Anwendungsmöglichkeiten von Containervirtualisierung zu geben. Ich habe mich jedoch dafür entschieden, allgemeine Anforderungen an Containerkonzepte von der Anwendung zu trennen, da sich das Prinzip in der Anwendung wiederholen würde. Zum Schluss gebe ich ein Ausblick und ziehe ein Fazit.

Begriffsklärung

Image: Images sind die Basis von Containern. Sie bestehen aus Datei-System-Ebenen, die auf übergeordneten oder Basis-Images aufbauen. Images sind statisch und ändern sich nicht. Es können mehrere Container von einem Image gestartet werden.

Dockerfile: Ein Dockerfile ist ein Text-Dokument, welches alle Befehle beinhaltet, die normalerweise manuell ausgeführt werden, um ein Docker Image zu starten.

Container: Ein Container ist die Instanz eines Images zur Laufzeit.

Versuchsaufbau

Auf einem Client wird ein Linux-Betriebssystem installiert, welches die Grundlage meines Systems wird. Darauf wird ein Hypervisor als Applikation installiert, indem eine Virtuelle Maschine erstellt wird. Außerdem wird eine Container-Engine auf dem Basis-System installiert, welches für die Containervirtualisierung zuständig ist.

Die Benchmarks werden in drei verschiedenen Umgebungen durchgeführt:

1. Container
2. Virtuelle Maschine
3. Host

Container und Virtuelle Maschine werden verglichen und der Host stellt einen Referenzwert dar.

Auswahl der Komponenten

Auswahl der Hardware

Die Hauptanforderungen an die Hardware waren einfacher Zugang und ausreichend Leistung. Ein Laptop ist sehr handlich und für eine Typ-2 Virtualisierung gut geeignet. Das gewählte Gerät hat realistische Spezifikationen für einen Studentenlaptop und verfügt über genügend Leistung, um eine VM zu hosten. Es wird damit meinen Anforderungen gerecht.

- Lenovo ThinkPad T430
- I5 3320M 2x2.60 GHz
- 8 GB RAM
- 128 GB SSD

Auswahl des Hypervisors

In meiner beruflichen Laufbahn habe ich hauptsächlich mit VMware ESXi Bare Metal virtualisiert. Für Typ-2 Virtualisierung habe ich ebenfalls VMware Workstation Player verwendet. In diesem Projekt wollte einen anderen beliebten Hypervisor ausprobieren - VirtualBox.

Auswahl der Container Engine

Von Containervirtualisierung habe ich zuerst durch Docker erfahren. Nach einer Internetrecherche habe ich herausgefunden, dass dies kein Zufall war, sondern durch Zahlen belegt werden kann. Docker hat im Jahr 2017 99% der Container gestellt. Im letzten Jahr ging die Zahl auf 83% zurück, da andere Container Engines auf den Markt gekommen sind.



Abbildung 1 – Docker Usage Report

Quelle: <https://jaxenter.de/docker-tools-vergleich-39670>

Docker bietet eine eigene umfassende Dokumentation und auch externe Forenbeiträge mit aufkommenden Fragen sind zahlreich vorhanden. Der Vorsprung von Docker gegenüber anderen Container Engines wird hier in meinen Augen sehr deutlich.

Im ersten Moment liegt es nahe, dass Schulen Open Source Software verwenden wollen, um Kosten zu sparen (z.B. Linux). Andererseits ist es als Lehrkraft üblich führende Software zu verwenden (z.B. Windows), da SchülerInnen auf den Markt vorbereitet werden sollen. Wenn der Markt Docker als Container Engine verwendet, wird in einer Berufsschule, wenn überhaupt, auch Docker gelehrt.

Aufgrund der guten Dokumentation und der momentan Marktdominanz habe ich mich für Docker entschieden.

Auswahl der Betriebssysteme

Die Verwendung eines Linux Betriebssystem liegt bei Container Virtualisierung nahe. Außerdem werde ich der Einfachheit halber auf allen Systemen (Container, VM und Host) das gleiche Betriebssystem verwenden. Da ich mit keinem Linux Betriebssystem vertraut bin, habe ich auf Anraten meines Betreuers Fedora verwendet.

Hier eine Abbildung meines Versuchsaufbaus mit allen gewählten Komponenten:

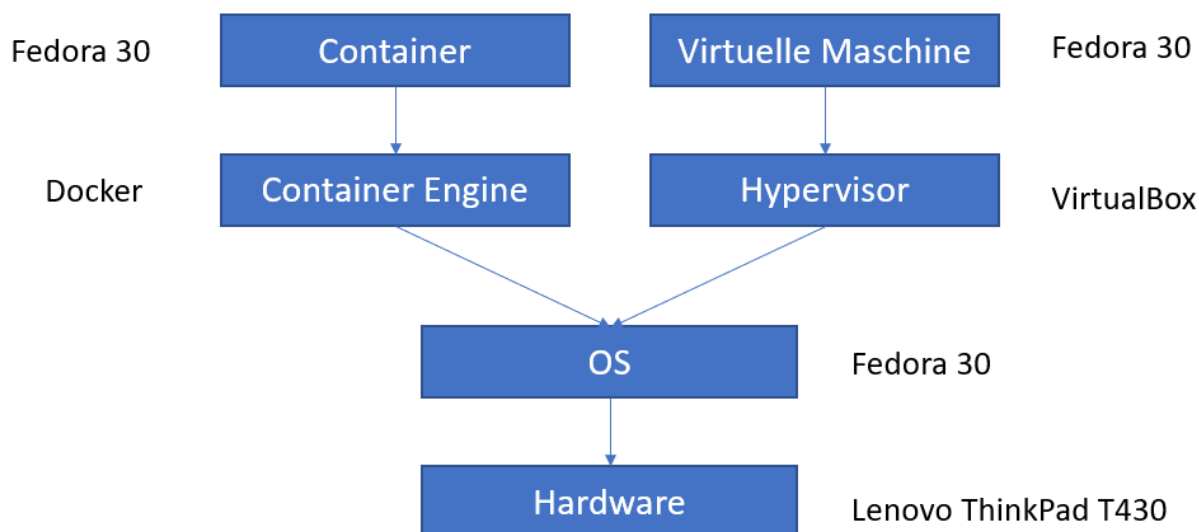


Abbildung 2 - Versuchsaufbau

Einrichten der Testumgebung

Installation von Docker und Einrichten der Container

Bei der Installation von Docker habe ich mich zu großen Teilen an die Anleitung der Homepage <https://docs.docker.com/get-started/> gehalten, sodass die Engine schnell installiert ist und der Daemon beim Hochfahren des Betriebssystems automatisch mitgestartet wird.

Als Grundlage eines Containers steht immer das Image. Im Docker Hub gibt es aktuell (Stand 08.08.2019) 2,5 Millionen Images, aus denen der User mit dem pull-command wählen kann. Es bleibt einem selbst überlassen, inwiefern das Image angepasst wird.

Zunächst habe ich in der Library nach einer Möglichkeit gesucht, mir die Containerverwaltung zu erleichtern und sie anschaulicher zu gestalten. Ich habe Portainer (<https://www.portainer.io/>) installiert und mir einen Überblick der Möglichkeiten von Docker gemacht. Das Image habe ich nicht weiter angepasst, sondern es lediglich heruntergeladen und den Container gestartet (später noch relevant).

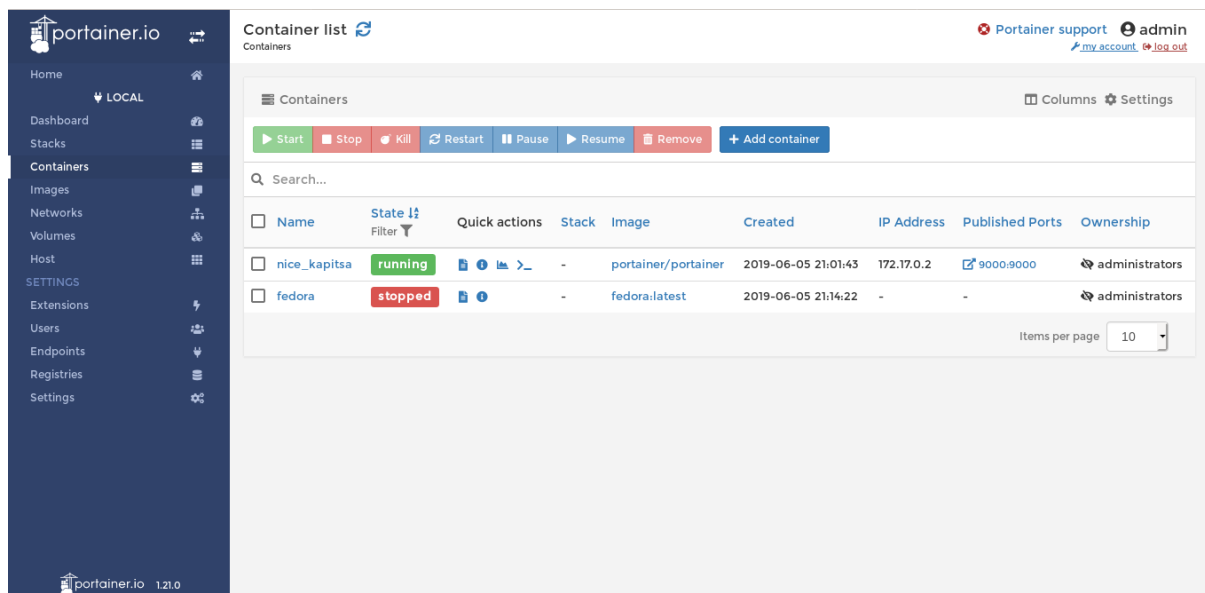


Abbildung 3 - Portainer Übersicht

Danach habe ich ein Fedora Image geladen und meinen zweiten Container für die Benchmarks erstellt. Durch Portainer kann ich mich auf das Terminal des Containers schalten und darauf arbeiten als wäre ich lokal angemeldet. Der andere Weg ist das Bearbeiten der Container durch eine Konfigurationsdatei (Dockerfile vgl. Abb. 4). Der Container startet vom Basis-Image und führt die Befehle des Dockerfiles im Terminal aus.

Installation von VirtualBox und Einrichten der VM

Das Image wird von der VirtualBox Internetseite heruntergeladen, entpackt und der Installer gestartet. Die meisten Konfigurationsschritte werden automatisch durchgeführt. Relevante Anpassungen sind die Zuweisung der Ressourcen (2 Kerne und 4 GB RAM) und das korrekte Erkennen einer SSD-Festplatte. Die VM wird erstellt und das Betriebssystem wird analog zum Host installiert.

Benchmark

Sysbench

Nach einer Internetrecherche bin ich auf den Benchmark **Sysbench** gestoßen. Mit Sysbench kann CPU, RAM und ein IO-Test durchgeführt werden und er ist auf Fedora verfügbar. Die Einstellungen können hier selbst vorgenommen werden. Die Ergebnisse werden in eine .txt-Datei exportiert.

```
#Download base image Fedora
FROM fedora:30

#Sysbench installieren
RUN dnf install sysbench -y

#Ordner anlegen
RUN mkdir /home/$USER/Dokumente/Sysbench
RUN cd /home/$USER/Dokumente/Sysbench

#Sysbench RAM Benchmark
RUN sysbench --num-threads=1 --test=memory --memory-block-size=1M --
memory-total-size=100G run > CO_RAM.txt

#Sysbench CPU Benchmark
RUN sysbench --num-threads=1 --test=cpu --cpu-max-prime=20000 run >
CO_CPU.txt

#Sysbench File-IO Benchmark
RUN sysbench --test=fileio --file-total-size=25G prepare
RUN sysbench --test=fileio --file-total-size=25G --file-test-mode=rndrw
--max-time=300 --max-requests=0 run > CO_File_IO.txt
```

Abbildung 4 – Dockerfile-Ausschnitt des Fedora Containers

RAM: Es werden 100 GB in Blöcken von je 1 MB in den RAM geschrieben.

CPU: Alle Primzahlen zwischen 1 und 20.000 werden gesucht, dabei wird nur ein Kern verwendet.

File-IO: Es werden 1024 Dateien mit einer insgesamten Größe von 25GB erstellt. Dann werden zufällig Dateien ausgelesen und beschrieben. Hierbei wurde die Zeit auf 300 Sekunden beschränkt (max-requests=0 bedeutet unendlich Anfragen möglich).

Geekbench

Nach dem individuellen Benchmark von Sysbench habe ich noch einen weiteren Benchmark mit einem anderen Ansatz durchgeführt. Geekbench bietet einen detaillierten CPU Benchmark, der ohne eigene Einstellungen standardisiert durchgeführt wird und daher auch mit anderen Ergebnissen vergleichbar ist (nicht nur die eigenen). Eine Datei wird heruntergeladen und ausgeführt. Die Ergebnisse werden im Browser auf der Geekbench Webseite (<https://browser.geekbench.com>) präsentiert und können bei Bedarf im eigenen Profil gespeichert werden. Was genau beim Benchmark passiert kann nur anhand der Ergebnisse interpretiert werden. Im Anhang befindet sich ein Ausschnitt eines detaillierten Ergebnisses (s. Anhang 1).

Read/Write Test mit dd

Für einige Anwendungsszenarien auf Servern ist zufälliges Lesen und Schreiben relevant und dies wurde mit Sysbench getestet. Mit dem Linux Tool dd können einfache sequenzielle I/O Performance-Messungen durchgeführt werden. Da hier lediglich drei Mal dieselbe Festplatte getestet wird erwarte ich keine großen Unterschiede.

```
dd if=/dev/zero of=./test.file bs=1M count=10000 oflag=direct
```

```
dd if=./test.file of=/dev/null bs=1M count=10000
```

Eine Datei **test.file** wird erstellt und man erhält eine Schreibgeschwindigkeit. Danach wird der Buffer Cache gelöscht und die Datei anschließend gelesen. Um ein aussagekräftiges Ergebnis zu bekommen, wird dieser sehr kurze Test mehrmals wiederholt, da Cron Jobs oder andere parallellaufende Prozesse im Hintergrund die Performance leicht beeinträchtigen können.

Bewerten und Einordnen der Benchmark Ergebnisse

Ich werde zunächst auf alle Ergebnisse individuell eingehen und am Ende dieses Kapitels ein zusammenfassendes Fazit ziehen. Zu Beginn des Projekts waren nur die geplanten Umgebungen (Host, VM und Container) im Benchmark Test. Nach dem ersten Prüfen der Ergebnisse habe ich es für interessant gehalten, wie ein anderer Hypervisor bei den Tests abschneidet, da die Ergebnisse im FileIO-Test zu stark von meinen Erwartungen abwichen. Ich habe bei meiner Internetrecherche herausgefunden, dass KVM eine gute Performance bieten soll und daher die Tests mit einer VM in KVM durchgeführt.

CPU Benchmark

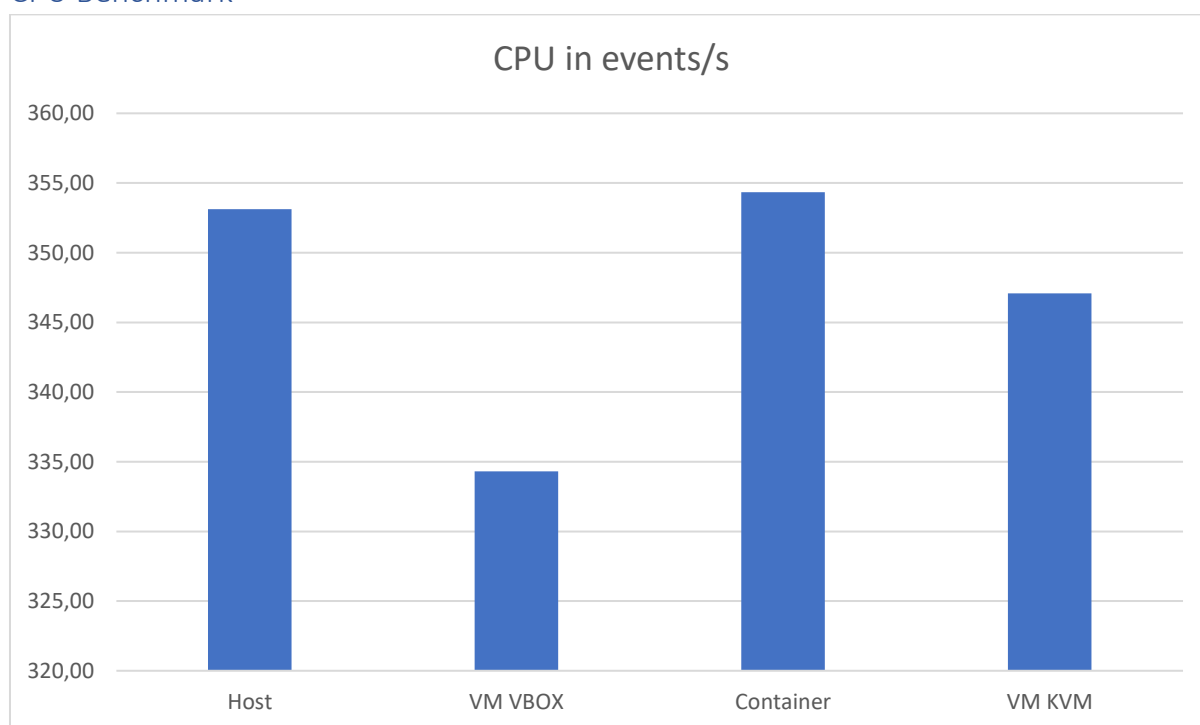


Abbildung 5 - CPU Benchmark Ergebnisse

Die CPU Werte im Sysbench Test sind alle auf einem Niveau (die Skala ist unten abgeschnitten und zeigt die Unterschiede größer als sie sind). Die KVM Umgebung ist kaum hinter Container und Host.

RAM Benchmark

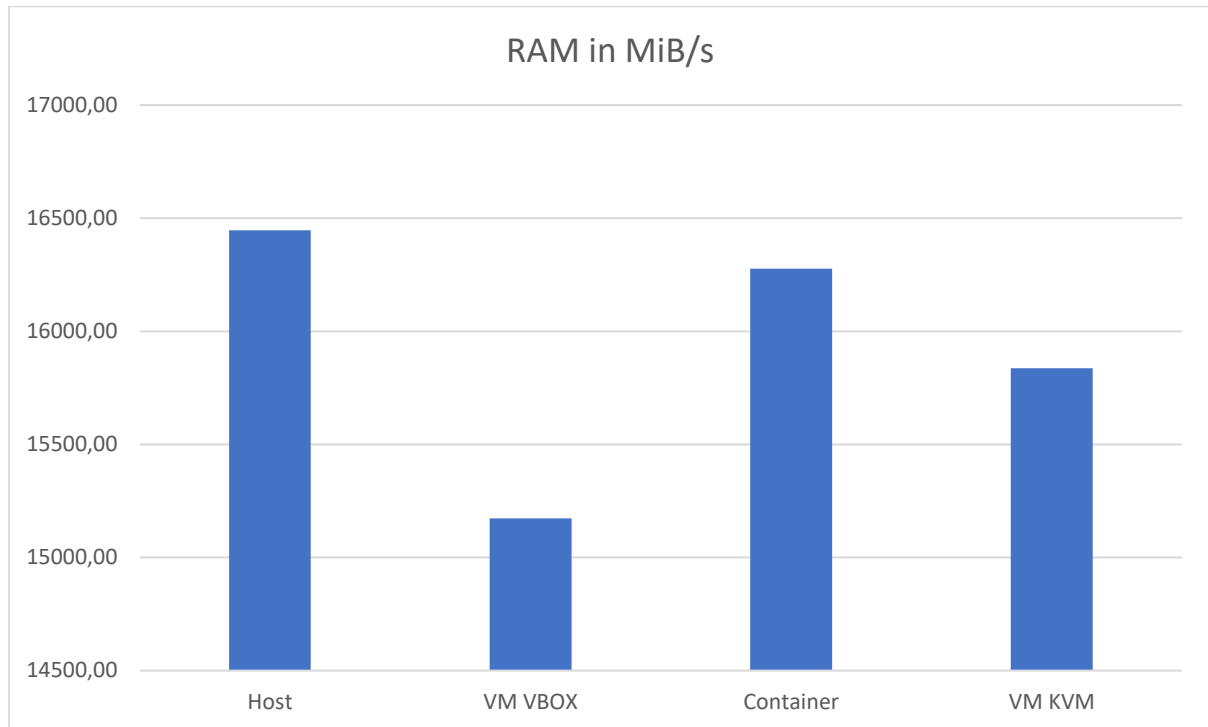


Abbildung 6 - RAM Benchmark Ergebnisse

Auch hier ist die Skala im unteren Wertebereich abgeschnitten. Die KVM-VM hat auch beim RAM mehr Leistung als die VBOX Umgebung. Insgesamt sind die Werte aber ebenfalls auf einem ähnlichen Niveau.

File IO Benchmark

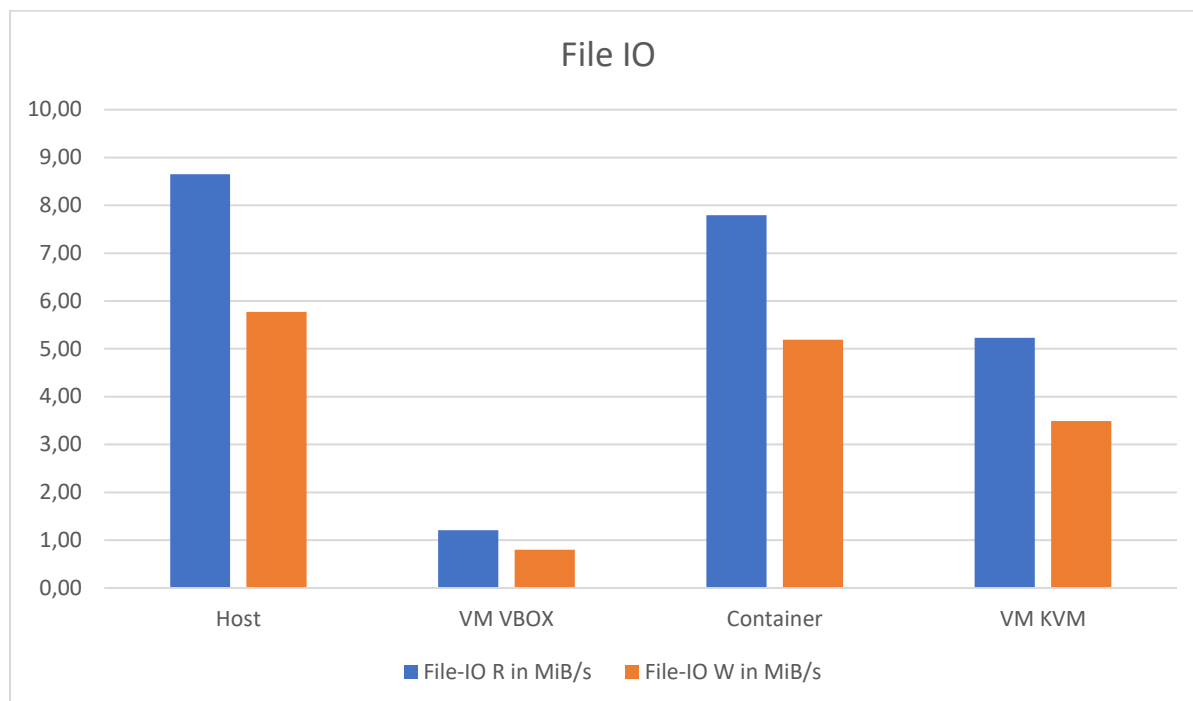


Abbildung 7 - File IO Benchmark Ergebnisse

Auch nach mehreren Testdurchläufen und Überprüfen der Hypervisor Konfiguration haben die Testergebnisse der VM in Virtualbox nicht den Erwartungen entsprochen. Das zufällige Lesen und Schreiben der Dateien bedingt die für eine SSD Festplatte allgemeinen niedrigen Werte, aber die VM fällt in meinen Augen zu stark ab. Diese Werte waren der Hauptgrund, einen weiteren Hypervisor zu testen. Die VM in KVM hat eine deutlich bessere Performance gezeigt auch wenn sie immer noch deutlich hinter Container und Host liegt.

Geekbench

The screenshot shows the Geekbench Browser interface with a table of benchmark results. The table includes columns for ID, Name, Platform, Architecture, Single-core Score, and Multi-core Score.

| # | Name | Platform | Architecture | Single-core Score | Multi-core Score |
|----------|--|----------|--------------|-------------------|------------------|
| 13801433 | QEMU Standard PC (Q35 + ICH9, 2009) Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS) 2594 MHz (2 cores) | Linux 64 | x86_64 | 3438 | 5449 |
| 13756256 | LENOVO 2349SBE Intel Core i5-3320M 3300 MHz (2 cores) | Linux 64 | x86_64 | 3600 | 6794 |
| 13755856 | Container Intel Core i5-3320M 3300 MHz (2 cores) | Linux 64 | x86_64 | 3309 | 6450 |
| 13755577 | innotek GmbH VirtualBox Intel Core i5-3320M 2594 MHz (2 cores) | Linux 64 | x86_64 | 3283 | 5434 |

Abbildung 8 - Geekbench Benchmark Ergebnisse

Auf der Abbildung sind die Endergebnisse aller Benchmarks zu sehen. Von oben nach unten gelesen: VM (VMware Workstation), Host, Container, VM (VirtualBox)

Die Single Core Performance ist in allen Umgebungen ähnlich. Es fällt auf, dass die Multicore Performance bei Host und Container ca. 20% besser ist. Eine mögliche Erklärung dafür wäre,

dass sowohl Container als auch Host den vollen und direkten Zugriff auf die CPU haben, wohingegen den VMs nicht die gesamte CPU bereitgestellt wird.

Sequenzielles Lesen/Schreiben mit dd

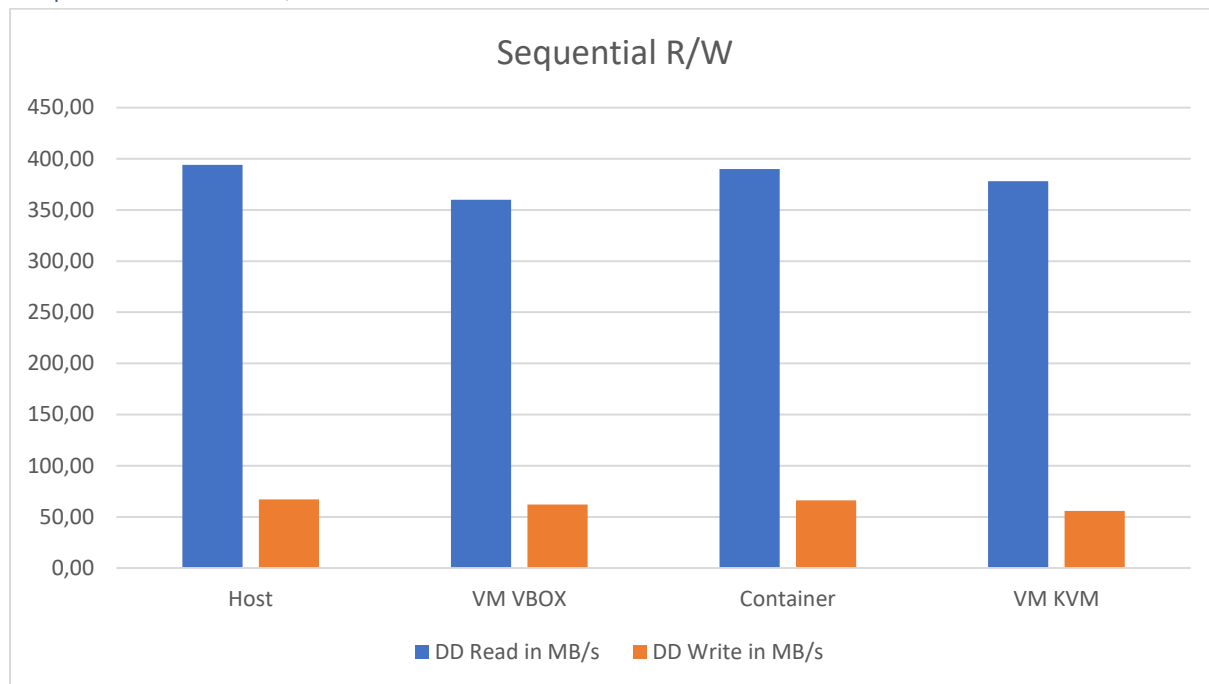


Abbildung 9 - Sequenzieller R/W Benchmark Ergebnisse

Die Ergebnisse in diesem Test sind alle in einem ähnlichen Bereich und passen zu den Werten aus dem Random-R/W-Test. Der Docker Container schneidet kaum besser als die VMs ab.

Fazit der Benchmarks

Das Gesamtbild aller Benchmark Ergebnisse zeigt, dass der Host als Referenzwert entweder gleichwertig mit dem Container oder besser performt hat. Die Virtuelle Maschine lag ca. 5% dahinter. Bis auf die Ausnahmen im File IO Test, was in meiner Internetrecherche nicht bestätigt werden konnte, waren die Unterschiede von VM und Container nicht groß genug, um einen umfassenden Wechsel von VM zu Container zu befürworten.

Diese Ergebnisse decken sich mit den Ergebnissen einiger Webseiten im Internet. KVM soll lediglich eine 2% geringere Performance bieten als ein Bare Metal System (<https://major.io/2014/06/22/performance-benchmarks-kvm-vs-xen/>). Das Maximum an Performance ist durch Virtuelle Maschinen also beinahe erreicht, sodass Container in dieser Hinsicht nicht viel besser sein können.

Klassische Virtualisierung hat sich im Markt bereits etabliert. Container müssen daher eindeutige Vorteile nachweisen können, um Unternehmen einen Grund zu geben, von klassischer Virtualisierung zumindest teilweise auf Containervirtualisierung umzusteigen.

Vorteile von Containern

Um die Vorteile von Containervirtualisierung zu verstehen, muss zunächst der Aufbau geklärt werden. Vor allem die Unterschiede zur klassischen Virtualisierung sind hier interessant.

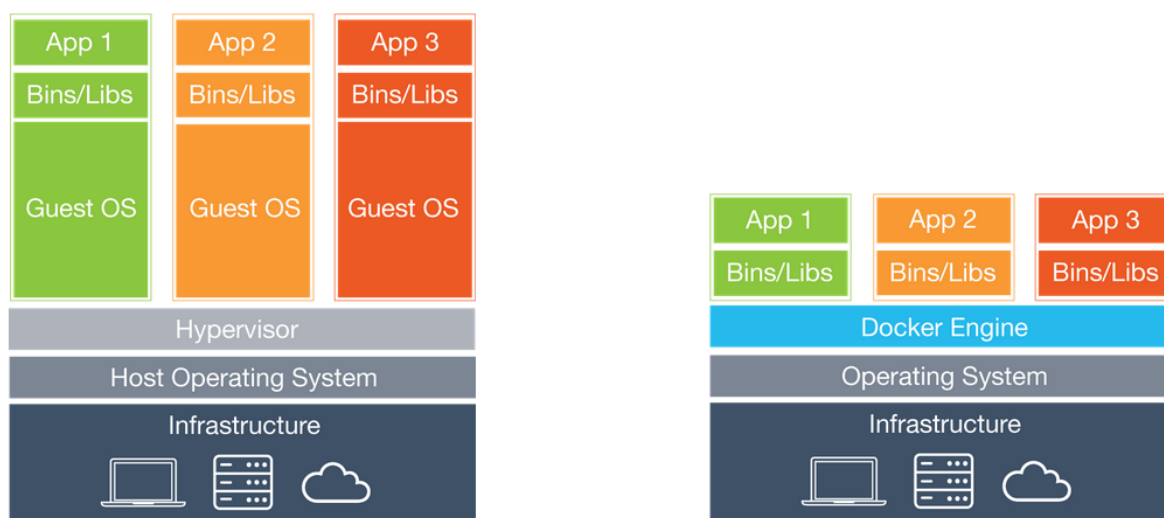


Abbildung 10 - Container vs. VM

Quelle: <https://sysdig.com/blog/2018-docker-usage-report/>

Bei der **klassischen Virtualisierung** (links) ist an unterster Stelle die Infrastruktur. Darauf wird ein Host Operating System installiert, auf dem wiederum ein Hypervisor läuft (bei Bare Metal Virtualisierung fällt das Host OS weg). Der Hypervisor verwaltet die VMs, die jeweils ein eigenes Betriebssystem für sich selbst haben. Die Maschinen laufen unabhängig voneinander und unabhängig vom Host OS (wenn vorhanden). Wenn mehrere Virtuelle Maschinen dasselbe Betriebssystem als Grundlage haben, sind viele Daten redundant vorhanden, da in jeder VM z.B. Fedora Betriebssystemdateien vorhanden sind.

Die **Containervirtualisierung** (rechts) versucht diesen Overhead zu verringern, indem diese Betriebssystemdateien miteinander geteilt werden. Der Aufbau der Container sieht sehr ähnlich aus. Anstatt eines Hypervisors haben wir die (Docker-) Engine, aber es fehlen die Betriebssysteme innerhalb der Container. Grundlage dafür ist die Tatsache, dass auf Dateien mehrfach **lesend** zugegriffen werden kann, ohne dass dabei Probleme entstehen. Mithilfe eines **Union File Systems** kann ein Betriebssystem in lesende und schreibende Bereiche getrennt werden, sodass der lesende Bereich für Container verfügbar ist. Docker verwendet hierfür **AUFS** (advanced multi layered unification filesystem).

Bei Containervirtualisierung wird daher von zwei unterschiedlichen Größen gesprochen:

Size (Größe): Der Teil eines Containers, der vom Image abweicht und daher persistent auf eine Festplatte geschrieben werden muss.

Virtual Size (virtuelle Größe): Der Teil eines Containers, der mit dem Image übereinstimmt und daher lesend von beliebig vielen Containern verwendet werden kann.

Wenn mehrere Container von einem Image erben, summiert sich also lediglich die Größe der Container und nicht die virtuelle Größe. Durch diese Vorteile in der Architektur kann also bei richtiger Anwendung Speicherplatz gespart werden. In meinem Versuchsaufbau habe ich zwei verschiedene Container und eine VM dessen Speicherauslastung ich vergleichen kann.

Virtuelle Maschine: Ohne große Datenmengen in Form von zusätzlichen Anwendungen oder anderen Dateien liegt der Speicherplatz einer VM bei ca. 5-10GB.

Fedora Container: In diesem Container wurden die Benchmarks durchgeführt, sodass einige Programme nachinstalliert werden mussten. Da kein neues Image vom Basis Fedora 30 Image erstellt wurde, weicht der Container dementsprechend davon ab. Er hat eine Größe von 230MB und eine virtuelle Größe von 505MB. Das Fedora Container Image ist deutlich kleiner als das VM Image, da im Container weniger Programme vorinstalliert sind und keine Graphische Oberfläche bereitgestellt wird.

Portainer Container: Dieser Container weicht nicht von seinem Image ab. Das Portainer Image wurde geladen und ist 75,4MB groß. Der Container hat eine Größe von 0 MB und eine virtuelle Größe von 75,4 MB.

Es folgen zwei Graphen, die den benötigten Speicher im Vergleich zur Anzahl der Container/VMs darstellen.

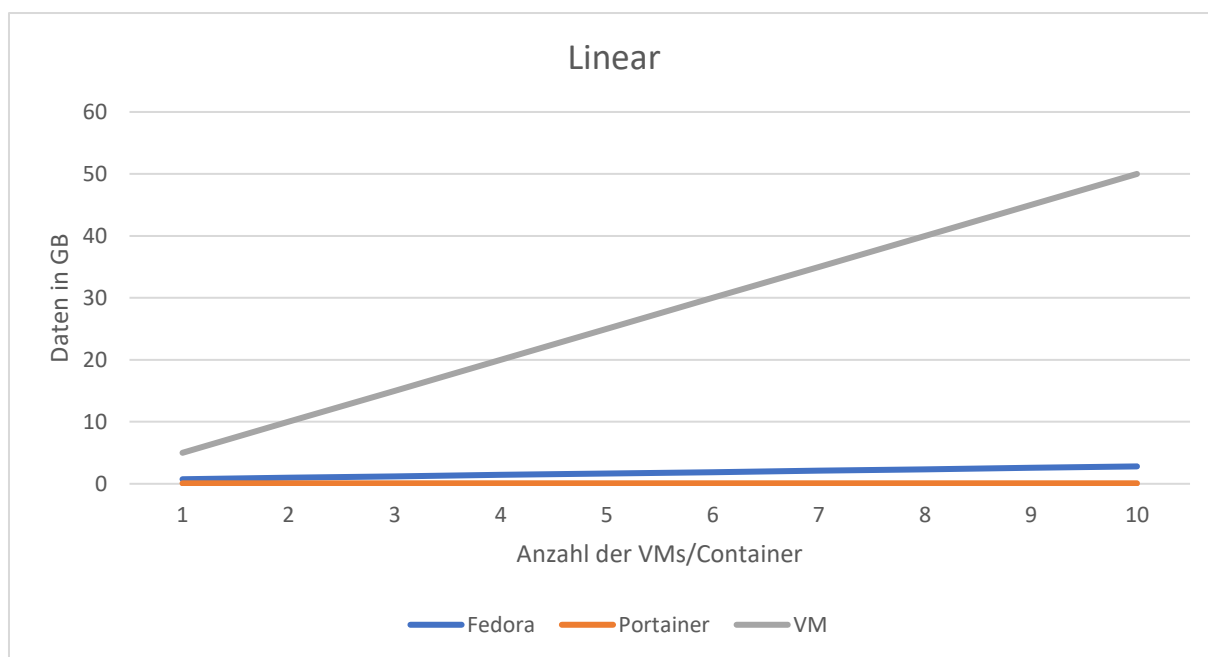


Abbildung 11 - Lineare Speicherauslastung

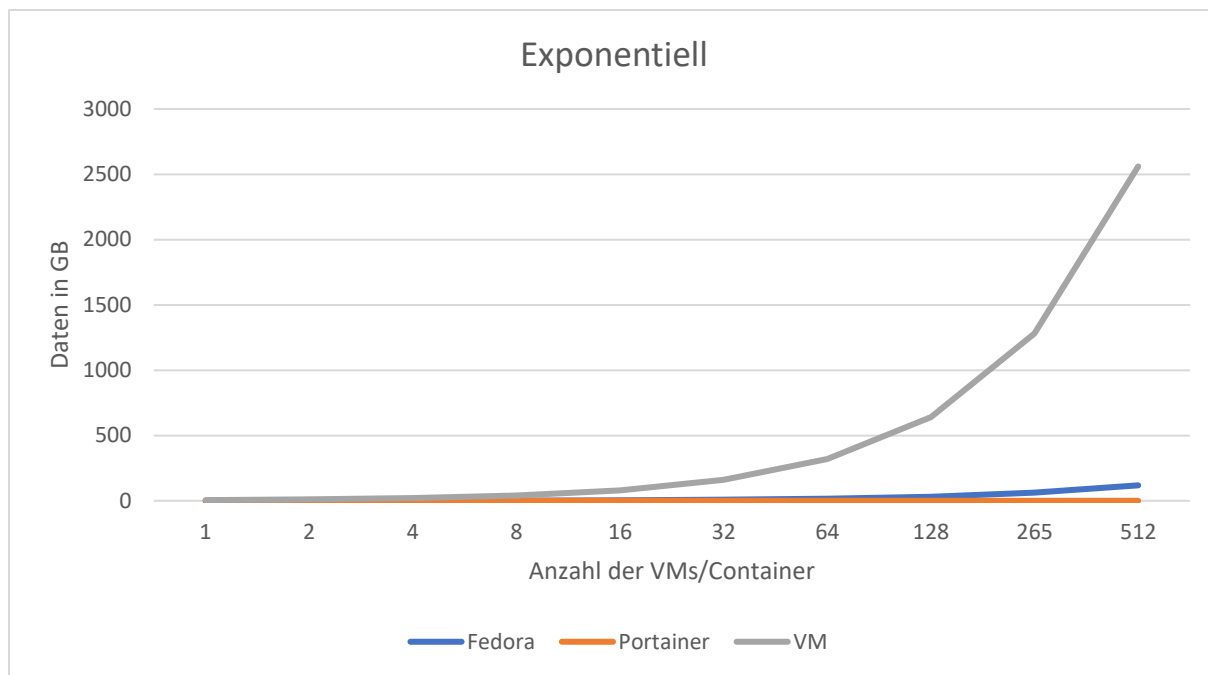


Abbildung 12 - Exponentielle Speicherauslastung

Es ist leicht nachvollziehbar, dass die Container in beiden Fällen deutlich weniger Speicher benötigen, da sie sowohl kleiner als auch effizienter sind. Es lässt sich durch die Vermeidung von redundanten Daten eine signifikante Menge an Speicherplatz einsparen. Dabei sollte bereits erwähnt werden, dass Container und Virtuelle Maschine vermutlich nicht 1:1 in der Anzahl verwendet werden würden. Die Erklärung hierfür hängt mit einem weiteren Vorteil zusammen und folgt im nächsten Absatz.

Container fahren schneller hoch. Je nach Betriebssystem benötigt eine VM bis zu 2 min. Ein Container kann dagegen innerhalb von Sekunden betriebsbereit sein. Dieser Geschwindigkeitsvorteil bietet eine weitere Möglichkeit Ressourcen zu sparen. Aus meiner Praxiserfahrung kann ich sagen, dass die meisten virtuellen Maschinen konstant laufen (24/7 an 365 Tagen im Jahr). Ein Grund dafür ist, dass die meisten VMs mehrere Aufgaben erledigen und daher unregelmäßig aber oft genug benötigt werden. Es lohnt sich nicht, die Maschinen bei Nichtbenutzung herunterzufahren und bei Bedarf anzuschalten. Container können atomare Aufgaben erledigen, da sie, wie zuvor erläutert, keine redundante Betriebssystemdateien anhäufen. Auf eine Virtuelle Maschine können 100 Container kommen, welche die meiste Zeit ausgeschaltet sind, da sie nicht benötigt werden. Durch ein geeignetes Container Konzept können weitere Ressourcen gespart werden.

Ein weiterer Vorteil von Container ist der Einsatz beim Wissenschaftlichen Arbeiten. Ergebnisse können in Form vom Konfigurationsfile (Dockerfile) leichter geteilt und nachvollzogen werden, sodass die Endergebnisse nicht nur in Tabellenform oder Ähnlichem geteilt werden. Diese Reproduzierbarkeit kann auch in anderen Anwendungsgebieten von Vorteil sein.

Im Anschluss gehe ich auf die Nachteile von Containern ein. Danach werde ich allgemeine Anforderungen von Containerkonzepten erläutern, welche die Vorteile von Containern ausnutzen können.

Nachteile von Containern

Beim Arbeiten mit Containern sind mir diese Nachteile bzw. Schwierigkeiten gegenüber einer VM aufgefallen.

1. **Wenig GUI:** Die meisten Containerverwaltungen bieten zunächst keine graphische Oberfläche an. Auch die Container selbst funktionieren oft nur über die Kommandozeile. Genau darin liegt ihr Vorteil. Sie sind dadurch schlanker als die VMs. Graphische Schaltflächen der Menüführung eines GUI zu erkennen ist einfacher als sich an Befehle zu erinnern. Wenn Anwendungen selten benutzt werden, müssen die Befehle nachgeschlagen werden und die eigentlich schnellere Benutzung einer Kommandozeile wird langsam. Einige Container präsentieren über den Browser eine GUI (z.B. Portainer).
2. **Eingeschränkte Nutzung:** Nicht alle Aufgaben, die momentan von VMs gelöst werden, können zum jetzigen Zeitpunkt von Containern übernommen werden. Einige Programme haben kein entsprechendes Image und können daher nicht direkt ersetzt werden. Die Technologie ist sehr neu und Programme, die beispielsweise nur auf alten Windows-Betriebssystemen funktionieren, können nicht in Containern zum Laufen gebracht werden.
3. **Wenig Isolation:** Durch das Teilen von Dateien zwischen den Container können evtl. nicht alle aktuellen Sicherheitsstandards erfüllt werden. Auch wenn von diversen Container Engines behauptet wird, sicherer als z.B. Docker zu sein, haben VMs allgemein eine höhere Isolationssicherheit. Einige Angriffsmöglichkeiten sind:
 - a. **Kernel Exploits:** Ein Angreifer verursacht Kernel Probleme auf dem Container, was sich negativ auf den Host auswirkt, da der Kernel von allen geteilt wird.
 - b. **Denial of Service Attacks:** Ein Container wird angegriffen, benötigt daher mehr Ressourcen und andere Container können ihre Aufgaben nicht erfüllen. Dieser Angriff funktioniert zwar auch bei Virtuellen Maschinen, für gewöhnlich sind den VMs aber zu Beginn bereits Ressourcen zugewiesen. Eine VM kann also nur eine bestimmte Menge an Ressourcen beanspruchen, wohingegen die Standard-Einstellungen von z.B. Docker keine Einschränkungen haben.
 - c. **Container Breakout:** Ein Angreifer bekommt Zugriff auf einen Container. Ohne Usernamespace werden root-Rechte auf dem Container auf den Host übertragen.
 - d. **Poisoned Image:** Das geladene Image ist bereits infiziert. Auch diese Gefahr besteht bei VMs. Meiner Erfahrung nach werden hauptsächlich VM-Images von den bekannten Hersteller-Webseiten (Windows, Linux, etc.) installiert, sodass hier weniger Gefahr besteht. Die Images von Container dagegen sind kleiner und durch das schnelle Teilen können infizierte Images schneller in den Umlauf gelangen.

Allgemeine Anforderungen an ein Containerkonzept

Image Baumstruktur

Bei der Containervirtualisierung wird auf ein Vererbungskonzept gesetzt, wodurch eine Baumstruktur entsteht. An oberster Stelle ist das Basis Image (Wurzel). Würden alle Container direkt von einem Basis Image erben, wären die Unterschiede zum Image sehr groß. Die Container müssen ihre Änderungen auf die Festplatte schreiben und die Container benötigen mehr Speicher. Wenn viele Container dieselben Änderungen haben, z.B. eine zusätzlich benötigte Anwendung, ist es sinnvoll ein neues Image zu erstellen. Das neue Image erbt von der Wurzel und dient diesen Containern als neue Vorlage.

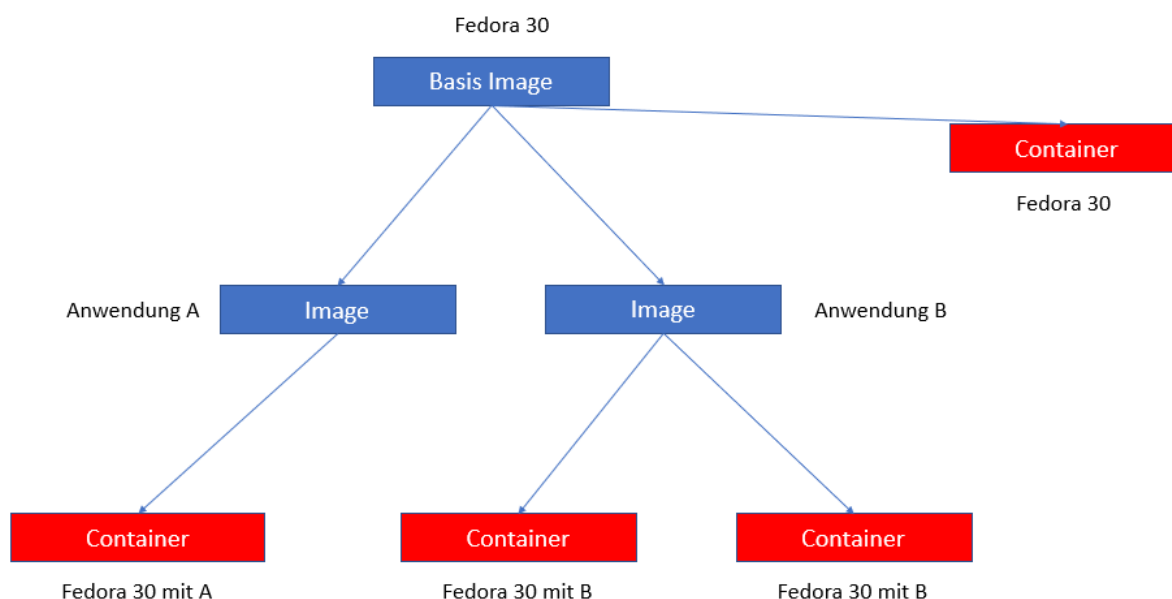


Abbildung 13 - Image Baumstruktur

Automatisches, geplantes und manuelles Hochfahren

Um das schnelle Hochfahren der Container auszunutzen und dadurch Ressourcen zu sparen, gibt es grundsätzlich drei Möglichkeiten:

1. **Automatisches Hochfahren:** Container werden durch einen Trigger hochgefahren. Wenn ein Container benötigt wird, wird er automatisch gestartet. Bsp.: Ein User möchte eine Datei ausdrucken und stellt eine Anfrage an einen Drucker. Der Drucker startet seinen Container, der den Treiber bereitstellt. Nach dem Druck schaltet sich der Container wieder aus.
2. **Geplantes Hochfahren:** Container werden planmäßig benötigt und durch einen Scheduler regelmäßig hochgefahren. Bsp.: Ein Container führt ein tägliches Backup durch. Der Container wird planmäßig gestartet, sichert die Dateien von A nach B und fährt wieder runter.
3. **Manuelles Hochfahren:** Container werden per Hand hochgefahren, wenn sie benötigt werden. Bsp.: Der Portainer Container wird nur gestartet, wenn ich das graphische Interface benutzen möchte und anschließend wieder runtergefahren.

Homogene Umgebung

Es ist wünschenswert eine möglichst homogene Container Umgebung zu schaffen. Weniger Basis Images bedeuten eine kleinere Baumstruktur, was zu einer geringeren Speicherauslastung und weniger Administrationsaufwand führt.

Atomare Aufgabenverteilung

Damit Container die meiste Zeit ausgeschaltet bleiben können, müssen die Aufgaben atomarer verteilt werden als es bei Virtuellen Maschinen der Fall ist. Bsp.: Ein Backup Server als VM hat mehrere Backupjobs, sichert regelmäßig alle Dateien und stellt alle Dateien unregelmäßig wieder her. Die Folge ist, dass er immer hochgefahren ist, da er oft benötigt wird. Dieselbe Aufgabe würden ca. 50 Container übernehmen können, die jeweils nur einige Dateien, Datenbanken oder andere Container sichern. Im Vergleich zur VM sind sie daher seltener eingeschaltet.

Anwendungsmöglichkeiten von Containern

Datenbanken

Zu den beliebtesten Images im Docker Hub zählen die Datenbanken. Oracle DB, redis, mongo, mysql, postgres und mariadb haben über 10 Millionen Downloads.

Monitoring

Viele Anbieter von Monitoring Software bieten ihre Anwendungen im Docker Hub an und sind als „Verified Publisher“ gekennzeichnet.

CMS

Beliebte Content Management Systeme wie Word Press, Joomla und Drupal sind auch als Image verfügbar. Ein CMS benötigt in der Regel viele Ressourcen und daher können hier eine Menge Ressourcen gespart werden.

Backup

Datenbanken und CMS sind Bereiche, die sehr tief in der IT-Infrastruktur eines Unternehmens integriert sind. Die Umstellung von VM auf Container bedeutet daher viel Aufwand und wird vermutlich nur erwägt, wenn eine Umstellung bereits aus anderen Gründen sinnvoll ist.

Ein Backup Konzept lässt sich leichter in ein aktives System integrieren und kann auch parallel zu einer VM Sicherung getestet werden.

Basis Image

Die meisten Linux Distributionen sind auch als Image erhältlich. Anwendungen, die auf VMs mit CentOS, Ubuntu, Debian oder Fedora laufen, können auch in Containern verwendet werden.

Container für HPC

Die Flexibilität von Containern ist auch für den HPC Bereich interessant. Docker verwendet das Linux-Feature cgroups, um Ressourcen Management zu betreiben. Dadurch sollte kein Container dazu in der Lage sein, zu viel CPU, RAM oder Netzwerkressourcen zu beanspruchen. Docker Container benötigen root-Zugriff, was auf Server Clustern ein Sicherheitsrisiko darstellen kann und den Umgang mit einem HPC User als Administrator erschwert. Docker benötigt außerdem eine relativ neue Kernel Version (min. 3.10), was für einige ältere Cluster ein Problem sein könnte.

Docker ist nicht ohne Grund der beliebteste Anbieter von Container Engines. Für viele User ist das Programm sehr gut geeignet. Mit dem Tool Podman kann eine Docker Alternative, die sehr ähnlich zu Docker ist, auf Clustern installiert werden. Für den HPC Bereich würde ich aber vermutlich eine andere Software wählen. Singularity soll eine Container Engine sein, die speziell für den HPC Bereich entwickelt wurde (<http://www.admin-magazine.com/HPC/Articles/Singularity-A-Container-for-HPC>). Das Tool ist sehr kompatibel mit Docker, sodass auch der große Image Hub von Docker verwendet werden kann. Die Default Einstellungen von Singularity sind besser an HPC Umgebungen angepasst und ein Container besteht lediglich aus einer einzigen Datei, die alle Informationen über den Container beinhaltet.

Ein anderer Anwendungsbereich für HPC ist das Einbinden von Grafikkarten. Bedingt durch die Architektur können Container im Gegensatz zu VMs direkt auf die Grafikkarte zugreifen. Mit NVIDIA müssen dafür auf den Containern keine extra Treiber installiert werden, wenn sie den Treiber des Hosts nutzen können. In der NVIDIA GPU Cloud (<https://www.nvidia.com/de-de/gpu-cloud/>) können Entwickler und Forscher auf Container mit GPU-optimierten Software für z.B. KI und maschinelles Lernen zugreifen.

Welches System der nächste Standard für Container Engines im HPC Bereich wird, ist noch nicht entschieden. Shifter (<https://iopscience.iop.org/article/10.1088/1742-6596/898/8/082021>) ist ein weiterer Versuch Container in den HPC Bereich zu bringen und die momentanen Unzulänglichkeiten von z.B. Docker auszubessern. Beide Tools (Singularity und Shifter) sind vermutlich für HPC gut geeignet. Für mehr Aussagekraft sind weitere Tests nötig. Die Effizienz, die von Containern ausgeht, wird aber definitiv im HPC verwendet werden.

Ausblick

Ein Blick in die Zukunft von Technologie ist immer schwierig. Insbesondere in der Informationstechnologie kann in wenigen Jahren viel passieren. Containervirtualisierung wird definitiv einen Platz in der IT haben, da die Vorteile zur klassischen VM für wichtige Themen der kommenden Jahre, wie z.B. Cloud, HPC und Machine Learning sehr relevant sind. Wie groß die Relevanz sein wird, hängt auch davon ab, wie sehr sich VM und Container angleichen werden. Wenn die klassische Virtualisierung einige Vorteile von Containern (z.B. Vermeidung von redundanten OS-Dateien durch ein layered filesystem) ebenfalls umsetzen, können VM und Container vermutlich koexistieren.

Global Player wie Google und Amazon sind bereits in das Container-Geschäft eingestiegen. Sie verwenden Container unter anderem für Machine Learning Projekte. Sowohl Amazon AWS (<https://aws.amazon.com/de/machine-learning/containers/>) als auch Google Kubernetes Engine (<https://cloud.google.com/kubernetes-engine/>) verwenden Docker Images.

Es stellt sich die Frage, ob Docker, Inc. sich auf dem Markt behaupten kann. Andere Unternehmen könnten einen technologischen Fortschritt erreichen und damit die etablierten Container Engines verdrängen. Das momentan typische Aufkaufen kleinerer Unternehmen durch ein größeres ist hier ebenfalls denkbar.

Für Technologie Anbieter ist der Einstieg in Containervirtualisierung zum aktuellen Zeitpunkt vermutlich eine gute Idee. Containerverwaltung und Engine ist nur für einige Unternehmen relevant, da es nur wenige Anbieter benötigt. Das Bereitstellen von Images kann auch für die breite Masse von Entwicklern und anderen IT Unternehmen interessant sein.

User könnten zunächst einige Tests mit Containern durchführen und dann entscheiden, ob ein größerer Umstieg von VM zu Containern sinnvoll ist. Auch wenn nicht alle Aufgaben durch Container gelöst werden, können sie in bestehende oder neu einzurichtende IT Systeme integriert werden. Bei der Entscheidung der Container Engine sollte zunächst ein Blick auf die verfügbaren Images geworfen werden. Im Zweifel hat Docker durch ihren Marktvorsprung die meisten Images und viele Anbieter sind kompatibel zu den Docker Images.

Es ist immer sinnvoll, auf einem aktuellen Stand der Technik zu sein und mit dem Wissen über Containervirtualisierung hat man eine gute Alternative zur klassischen VM.

Fazit

Zu Beginn des Projekts habe ich mir Ziele gesetzt, meine Kenntnisse in folgenden Bereichen zu verbessern:

1. Linux

Die ersten Schritte habe ich mit dem empfohlenen Linux Grundkurs gemacht. Nachdem ich genügend Grundlagen gesammelt hatte, konnte ich mit dem Installieren der Tools beginnen. Die Benchmarks waren eine gute Möglichkeit dem Versuchsaufbau einen Sinn und mir mehr Übung mit der Linux Kommandozeile zu geben. Einige einfache Hürden haben länger gedauert als erwartet aber beim nächsten Kontakt bin ich besser dadurch vorbereitet. Ich bin überaus zufrieden mit den Linux-Kenntnissen, die ich mir aneignen konnte.

2. Allgemeines Wissen über Container

Der Kompetenzzuwachs ist bei den Containern vermutlich am größten, da ich hier keinerlei Vorkenntnisse hatte. Durch Internetrecherche und eigenes Ausprobieren konnte ich viel herausfinden. Die meisten Container Engines funktionieren ähnlich und haben teilweise auch ähnliche Befehle. Auch das Verwenden der Docker Images kann zu einem allgemeinen Wissen über Container gezählt werden. Der Vorsprung von Docker ist so groß, dass es die meisten Images anbietet und somit die Konkurrenz dazu bringt, kompatibel zu den Images zu sein. Ich habe einen guten Einblick in das Arbeiten mit Containern bekommen und könnte mit ein wenig Einlesen die gängigen Container Engines verwenden.

3. Spezielles Wissen über Docker

Ich bin nicht sehr tief in die Konfiguration von Docker eingestiegen, da hier die meisten Änderungen vorbehalten sind. Docker kann für mich in der Zukunft weniger relevant sein und spezielles Wissen über Befehle halten sich ohne weitere Anwendung nicht lange im Gedächtnis. Dennoch habe ich einen guten Überblick bekommen.






4. Unterschiede von Container und VM

Gerade zum Ende des Projekts habe ich den Fokus vom Arbeiten mit Docker und den Containern auf den in meinen Augen wichtigsten und nachhaltigsten Punkt gelenkt – die Abgrenzung zur klassischen Virtualisierung. Hier hatte ich bereits einige Anknüpfungspunkte und auch das meiste Interesse. Ich habe Vor- und Nachteile herausgearbeitet und kann gut einschätzen unter welchen Umständen Containervirtualisierung der klassischen Virtualisierung vorgezogen werden sollte.

Ich habe positive Erfahrungen in Projekten gemacht, in denen Themen nicht nur angeboten und verteilt wurden, sondern die Zeit investiert wurde, gemeinsam ein Projektthema zu erarbeiten. Auch in diesem Projekt hatte ich den Anspruch ein passendes Thema für mich zu finden und in Zusammenarbeit mit der Seminarleitung ist uns das gelungen. Zu Beginn des Projekts war der Titel der Arbeit „Container für HPC“. Es war geplant, einen Weg zu finden, Container auf dem WR-Cluster laufen zu lassen und deren Performance mit VMs zu vergleichen. Die Komplexität und der damit verbundene Aufwand des Projekts haben mich dazu gebracht, den HPC Teil nur theoretisch in einem Kapitel zu betrachten. Am Ende habe ich den Fokus auf die Anwendung von Containern gelegt, da ich der Meinung bin, hier eine fundiertere Meinung zu haben.

Insgesamt bin ich mit meinem Kompetenzzuwachs zufrieden und hoffe, dass ich einige interessante Inhalte präsentieren konnte.

Anhang

| Multi-Core Performance | | |
|-----------------------------|---------------------------|--|
| Multi-Core Score | 6450 | |
| Crypto Score | 5753 | |
| Integer Score | 7316 | |
| Floating Point Score | 6490 | |
| Memory Score | 4616 | |
| AES | 5753 4.33 GB/sec |  |
| LZMA | 7481 11.7 MB/sec |  |
| JPEG | 9115 73.3 Mpixels/sec |  |
| Canny | 8111 112.5 Mpixels/sec |  |
| Lua | 5503 5.66 MB/sec |  |

Anhang 1 - Geekbench Details Container

Abbildungsverzeichnis

| | |
|---|----|
| Abbildung 1 – Docker Usage Report | 3 |
| Die Abbildung zeigt den Docker Usage Report von sysdig. | |
| Quelle: https://sysdig.com/blog/2018-docker-usage-report/ (<i>Abgerufen: 5. August 2019, 13:31 UTC</i>) | |
| Abbildung 2 - Versuchsaufbau..... | 4 |
| Die Abbildung zeigt meinen Versuchsaufbau (eigene Darstellung). | |
| Abbildung 3 - Portainer Übersicht | 5 |
| Die Abbildung zeigt einen Screenshot meines Portainer Containers im Browser. | |
| Quelle: https://www.portainer.io/ | |
| Abbildung 4 – Dockerfile-Ausschnitt des Fedora Containers | 6 |
| Die Abbildung zeigt einen Ausschnitt eines von mir geschriebenen Dockerfiles. | |
| Abbildung 5 - CPU Benchmark Ergebnisse..... | 7 |
| Die Abbildung zeigt die CPU Benchmark Ergebnisse von Sysbench (eigene Darstellung) | |
| Abbildung 6 - RAM Benchmark Ergebnisse | 8 |
| Die Abbildung zeigt die RAM Benchmark Ergebnisse von Sysbench (eigene Darstellung) | |
| Abbildung 7 - File IO Benchmark Ergebnisse | 9 |
| Die Abbildung zeigt die File IO Benchmark Ergebnisse von Sysbench (eigene Darstellung) | |
| Abbildung 8 - Geekbench Benchmark Ergebnisse | 9 |
| <i>Die Abbildung zeigt einen Screenshot von meinem Geekbench Account in der Übersicht.</i> | |
| Quelle: https://browser.geekbench.com/ | |
| Abbildung 9 - Sequenzieller R/W Benchmark Ergebnisse | 11 |
| Die Abbildung zeigt die Sequenziellen R/W Benchmark Ergebnisse von dd (eigene Darstellung) | |
| Abbildung 10 - Container vs. VM..... | 12 |
| Die Abbildung zeigt die Unterschiede in der Architektur von Containervirtualisierung und VM | |

Quelle: <https://jaxenter.de/docker-tools-vergleich-39670> (Abgerufen: 14. August 2019, 14:31 UTC)

Abbildung 11 - Lineare Speicherauslastung 13
Die Abbildung zeigt die Speicherauslastung von Containern und virtuellen Maschinen auf einer linearen Skala (eigene Darstellung).

Abbildung 12 - Exponentielle Speicherauslastung 14
Die Abbildung zeigt die Speicherauslastung von Containern und virtuellen Maschinen auf einer exponentiellen Skala (eigene Darstellung).

Abbildung 13 - Image Baumstruktur 16
Die Abbildung zeigt eine Container Image Baumstruktur mit Fedora 30 als Basis Image (eigene Darstellung).

Literaturverzeichnis

Wikipedia, Die freie Enzyklopädie (2019)

Seite „Containervirtualisierung“. In: Wikipedia, Die freie Enzyklopädie.

Bearbeitungsstand: 7. Mai 2019, 09:26 UTC.

URL: <https://de.wikipedia.org/w/index.php?title=Containervirtualisierung&oldid=188319799>

(Abgerufen: 5. August 2019, 12:31 UTC)