Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра автоматизации систем вычислительных комплексов

Михеев Павел Алексеевич

# Исследование эффективности использования физических ресурсов легковесными контейнерами в облачных системах.

**Научный руководитель:**
к.ф.-м.н.
В.А. Антоненко

Москва, 2018

# Аннотация

Контейнер - изолированная группа процессов в операционной системе, имеющая доступ к ограниченному количеству ресурсов системы. Данная изоляция достигается за счет механизмов ядра ОС, таким образом, контейнеры используют это ядро, не тратя ресурсы на поддержку гостевого ядра.

Контейнеры обладают небольшим временем запуска, большой пропускной способностью, и показатель производительности вычислений в них ближе к показателю вычислений на физичеких ресурсах. Однако не существует систем, позволяющих исследовать производительность и деградацию производительности контейнеров при масштабировании и сравнить ее с производительностью и деградацией для виртуальных машин.

В данной работе предлагается использовать существующие методики измерения производительности виртуальных машин и применить их к легковесным контейнерам.

Целью работы является разработка системы измерения производительности и ее деградации у виртуальных машин и контейнеров, запущенных на физическом сервере.

Данная система позволит сравнить численно производительность контейнеров и виртуальных машин.

# Оглавление

# Введение

Облачные вычисления в современной информационной среде занимают все больше и больше пространства. Развитие открытого программного обеспечения по управлению облаками увеличивает количество тех, кто использует облачные системы как для использования внутри компании, так и для предоставления услуг клиентам. Одним из немногих сдерживающих факторов такого роста вовлеченности использования облачных вычислений является цена физического оборудования.

Традиционно облачные вычисления используют виртуальные машины как сущности, которые так или иначе предоставляются пользователю. Так, в случае облака, работающего по модели Infrastructure-as-a-Service (IaaS), пользователю предоставляется полный доступ к заказанной виртуальной машине, которую он настраивает под собственные требования. В облаках типа Platform-as-a-Service предоставляется доступ к интерфейсам некоторых приложений, запущенных в виртуальной машине. В облаках Software-as-a-Service пользователь запрашивает услугу, обработка которой происходит с помощью виртуальных машин, доступа к которым у пользователя нет.

Количество виртуальных машин, которое может быть запущено в облаке, напрямую зависит от количества физических ресурсов. Под количеством физических ресурсов в данной работе будет пониматься совокупное (по всем серверам) количество ядер процессоров, совокупное количество оперативной памяти и совокупный объем жестких дисков системы.

Пусть X - количество доступных физических ресурсов в облаке, а $\chi_i$ - количество ресурсов, требующихся для $i$-ой виртуальной машины. Тогда $n$ - максимальное число виртуальных машин с такими требованиями, которое может быть запущено на данных ресурсах, если $\sum\limits_{i=1}^{n} \chi_i = X$.

Однако может возникать ситуация, когда количество запрашиваемых

виртуальных машин превосходит максимальное число $n$, определенное выше. При этом у владельца облака нет возможности или необходимости докупить оборудование, так как, к примеру, такая ситуация возникает редко. Важный момент, что ресурсы виртуальной машины могут быть использованы ее процессами не на 100% процентов. Тогда, если предоставить доступ к этим же физическим ресурсам или их части той виртуальной машине, номер которой превосходит $n$, то обе эти виртуальные машины смогут осуществлять свои функции. В таком случае $\sum\limits_{i=1}^{n} \chi_i > X$, но при этом все виртуальные машины размещены.

Введем коэффициент $\theta = \frac{\sum\limits_{i} \chi_i}{X}$ - коэффициент перекрытия (overlap), являющийся отношением суммы ресурсов, требующихся виртуальным машинам системы, к физическим ресурсам системы.

Разумеется, при увеличении коэффициента перекрытия производительность процессов в виртуальных машинах падает, так как при использовании одних физических ресурсов (в первую очередь, ядер процессора) разными виртуальными машинами исполнение машин одновременно будет невозможно. Из-за этого будет наблюдаться деградация производительности виртуальных машин при увеличении коэффициента перекрытия.

Еще одной быстро занявшей рынок технологией стала легковесная виртуализация. В этой технологии виртуализации гостевая операционная система отсутствует, а все процессы запускаются в рамках ядра хостовой операционной системы. Изоляция подобных процессов друг от друга и ограничение доступных им ресурсов достигается за счет специальных механизмов ядра. Такие процессы и их потомки с наложенными на них ограничениями называются контейнерами. Так же как и процессы в виртуальной машине изолированы от процессов другой виртуальной машины, процессы в контейнере изолированы от процессов других контейнеров. Отличие заключается в том, что системные вызовы в вирту-

альной машине идут в операционную систему машины, тогда как системные вызовы контейнеров идут напрямую к ядру операционной системы, в которой данный контейнер запущен.

Отсутствие необходимости поддерживать гостевую операционную систему обладает как достоинствами, так и недостатками. Подробнее о них будет сказано ниже.

Контейнеры могут быть использованы в облачных системах так же, как и виртуальные машины. Они потребляют некоторое количество ресурсов, ресурсы могут разделять между несколькими контейнерами, а при увеличении коэффициента перекрытия будет наблюдаться деградация производительности.

В данной работе предлагается исследовать и сравнить производительность виртуальных машин и контейнеров, а так же деградацию производительности машин и контейнеров при увеличении коэффициента перекрытия. Для этого была разработана система, способная запускать виртуальные машины или контейнеры и запускать в них приложения, позволяющие измерять производительность, а так же собирать результаты работы этих приложений.

Основной гипотезой данной работы является следующее: контейнеры обладают более высокой производительностью по сравнению с виртуальными машины, деградация производительности виртуальных машин при увеличении коэффициента перекрытия происходит быстрее, чем у контейнеров.

# Глава 1.

# Постановка задачи

## 1.1. Цель работы

Разработать и реализовать систему, позволяющую сравнить производительность виртуальных машин и контейнеров, а так же деградацию производительности виртуальных машин и контейнеров при увеличении коэффициента перекрытия.

С помощью разработанной системы провести эксперименты, позволяющие проверить следующую гипотезу: **контейнеры обладают более высокой производительностью по сравнению с виртуальными машины, деградация производительности виртуальных машин при увеличении коэффициента перекрытия происходит быстрее, чем у контейнеров.**

## 1.2. План решения задачи

1. Сравнить различные технологии виртуализации.

2. Составить обзор существующих решений, с помощью которого выделить методики оценки производительности виртуальных машин.

3. Выбрать из предыдущего обзора методику оценки производительно-

сти или разработать свою, которую возможно применить для оценки производительности контейнеров.

4. Разработать систему, реализующую методику оценки производительности из предыдущего пункта, позволяющий численно оценить производительность и ее деградацию при увеличении коэффициента перекрытия в случае виртуальных машин и контейнеров.

5. С помощью разработанной системы провести эксперименты, позволяющие проверить следующую гипотезу: **контейнеры обладают более высокой производительностью по сравнению с виртуальными машины, деградация производительности виртуальных машин при увеличении коэффициента перекрытия происходит быстрее, чем у контейнеров.**

# Глава 2.

# Технологии виртуализации

В данном разделе под хостовой операционной системой будет пониматься система, в которой могут быть запущены гостевые операционные системы. Гостевые операцинные системы - это те системы, которые видят лишь свое изолированное окружение, и которые не могут быть осведомлены о наличии других гостевых систем, кроме как через сеть. Под виртуальной сущностью будет пониматься тот процесс в хостовой операцинной системе, который исполняет вычисления гостевой операционной системы.

Гипервизор - специализированное программное обеспечение, которое занимается управлением гостевыми операционными системами: запуском, остановкой, наблюдением и выделением ресурсов. Гипервызиоры бывают двух типов:

1. Нативные гипервизоры, которые запускаются напрямую на оборудовании хоста, контролируют это оборудования и осуществляют наблюдение за гостевыми операционными системами.

2. Гипервизоры, которые запускаются поверх операционной системы хоста и осуществляют мониторинг гостевой операционной системы.

Виртуализация - такой подход к организации вычислений, при котором каждая виртуализированная сущность изолирована от других, при-

чем ей может быть доступна лишь часть общих ресурсов. В данном разделе будет расмматриваться виртуализация центрального процессора, то есть каким образом возможно исполнять гостевую операционную систему изолированно на центральном процессоре хостовой системы.

Существует четыре основных вида виртуализации: полная вирутализация, паравиртуализация, аппаратная виртуализация и легковесная виртуализация. Далее будут рассмотрены каждый из этих видов, а так же их достоинства, недостатки и применимость в облачных системах.

## 2.1. Полная виртуализация

Данный вид виртуализации является исторически первым. Основная его особенность заключается в том, что гостевая операционная системы полностью отделяется от управления инфраструктурой хоста. Гостевая ОС не требует никаких изменений, и не осведомлена, что запущена в виртуальном окружении.

Как следует из названия, данный вид виртуализации позволяет запустить любую гостевую операционную систему в любой хостовой. Единственное требование - это наличие динамического транслятора из машинного языка архитектуры, с которой работает гостевая операционная система, в машиный язык хостовой архитектуры. В основе работы данного вида виртуализации лежит принцип динамической трансляции [1].

Преимуществом данного вида виртуализации является гипотетическая возможность запускать любую гостевую операционну систему.

При этом основное преимущесво оборачивается и основным недостатком. При современном разнообразии вычислиельной техники невозможно иметь трансляторы с любого машинного языка в любой. Но даже при наличии транслятора возможно, что скорость исполнения гостевого кода будет гораздо медленнее, чем в случае исполнения на настоящей ар-

хитектуры. Дополнительно к этим недостаткам добавляются сложность реализации динамической трансляции, связанной, к примеру, с неразличимостью команд и данных.

При этом существует ряд гипервизоров, поддерживающих полную виртуализацию. Примерами могут быть VMware ESXi, Microsoft Virtual Server [1]. Стоит отметить, что в силу закрытости данного програмнного обеспечения, их исследование в данной работе проивзведено не будет.

## 2.2. Паравиртуализация

Проблемой динамической трансляции является исполнение привилегированных инструкций гостевой операцинной системы. При обработке данных инструкций работа гостевой операцинной системы ухудшается.

Паравиртуализация пытается исправить данный недостаток. Это подход подразумевает модификацию гостевой операционной системы таким образом, чтобы исполнение привилегированных инструкций было оформлено как системный вызов в хостовую операционную систему. При этом при использовании паравиртуализации производительность гостевой операцинной системы выше, чем при использовании полной виртуализации [3].

Основным же недостатком данного вида виртуализации является необходимость модификации ядра гостевой системы. Это уменьшает количество операцинных систем, которые могут быть виртуализированны с помощью данного подхода.

Наиболее популярные гипервизоры, поддерживающие паравиртуализацию: Xen [4] и VMware [1]. Однако в данной работе они не будут рассматриваться. Это связанно с тем, что технология Xen практичски не поддерживается сообществом, и, как следствие, не исользуется в облачных системах. Продукты же VMware являются закрытыми, что так же

не позволяет их использовать.

## 2.3. Аппаратная виртуализация

Подход аппаратной реализации подразумевает совпадение машинного языка гсотевой системы и хостовой. При этом архитектура процессора хостовой операцинной системы должна поддерживать аппартную виртуализацию, то есть исполнять код гостевой системы так, будто бы это код хостовой. Привилегированные вызовы автоматически отслеживаются гипервизором, и при обнаружении исполняются на процессоре напрямую [1].

Такой подход позволяет решить проблему необходимости модификации ядра для повышения производительности гостевой системы, а так же использование полной виртуализации, хоть и требует специфической архитектуры процессора хостовой системы. Однако данная технология есть почти во всех современных процессорах, что позволяет использовать данный вид виртуализации в облачных системах.

Аппаратная виртуализация поддерживается множеством гипервизоров. Основные представители - это VirtualBox [5], KVM [6], VMware ESXi [2]. В данной работе будет рассматривать гипервизор KVM, который является модулем ядра Linux.

## 2.4. Общие достоинства и недостатки

Такой подход имеет ряд достоинств и недостатков. С одной стороны, поддержка гостевого ядра расширяет набор возможных операцинных систем, которые можно запустить. С другой стороны, поддержка гостевого ядра требует ресурсов. Так, гостевая операционная система будет потреблять часть выделенной оперативной памяти, часть выделенного


Рис. 2.1. Схема аппаратной виртуализации

дискового пространства и часть ресурсов процессора. При запуске виртуальной машины будет тратиться время на запуск ядра ОС. Оперативная память, выделяемая гипервизором под виртуальную машину, чаще выделяется непрерывным участком, что может приводить к внутренней фрагментации памяти.

## 2.5. Виртуализация на уровне операционной системы

Другим подходом является виртуализация на уровне ОС.

Идея этого подхода такова: пусть операционная система способна выделять своим процессам ( или группе процессов) и их потомкам некоторое подмножество своих ресурсов. Пусть так же ОС способна выделять для каждого такого подмножества ресурсов свои множества идентификаторов процессов. Тогда для каждого подмножества ресурсов возможно запускать процессы, которые в общем случае могут быть запущены только в единичном экземпляре ( в первую очередь, это процесс init ядра Linux). Таким образом можно изолировать подгруппы физических ресурсов, имея возможность запустить на каждой подгруппе свой экземпляр операционной системы. Однако стоит отметить, что ядро этих ОС общее, то есть то, которая использовала хостовая ОС.

Другим названием для этого подхода служит "контейнеризация", "легковесная виртуализация", или "контейнерная виртуализация" а каждую

подгруппу ресурсов со своим множеством процессов называют "контейнеры".

Изначально этот подход был реализован в ядре Linux. Модуль control groups ( cgroups ) ядра позволял выделять подгруппы ресурсов, а модуль namespaces - выделять свои иерархии идентификаторов процессов. На базе этого подхода реализовано несколько видов контейнеров - в первую очередь Linux Containers( далее LXC ).

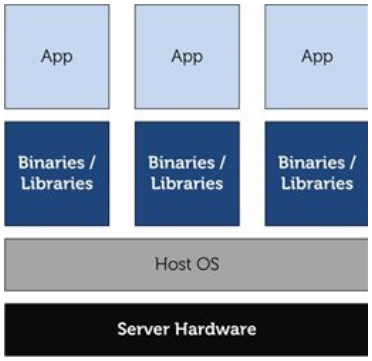


Рис. 2.2. Схема виртуализации на уровне ОС
labelконтейнеры

Выделим необходимые множества ресурсов ОС, требующих выделения подгрупп:

- Точка монтирования: разные контейнеры должны монтироваться в разные точки файловой системы хостовой ОС. Иначе действия контейнеров над файлами не будут синхронизированы.

- Network namespace: это необходимо для обеспечения полной изоляции на уровне сокетов, IP адресов и портов от соседа. Таким образом каждый контейнер имеет строго зафиксированный отдельный IP адрес, а также пространство сокетов, портов и роутинг-таблицу.

- IPC namespace: изолирует семафоры, очереди, мьютексы, shm память для IPC и IPC Sys V в таком виде, что каждый контейнер видит только свои IPC ресурсы и ничьи более.

- PID namespace: изолирует идентификаторы процессов в разных контейнерах друг от друга и от ОС, в которой запущен этот контейнер.
- UTS: позволяет выдавать уникальные hostname для каждого контейнера так, чтобы они не совпадали друг с другом и именем ОС, в которой контейнеры запущены.

Так же между контейнерами надо разделять следующие физические ресурсы: процессор, память и жесткий диск. Для реализации разграничения использования ресурсов используются следующие cgroups: cpu, memory и blkio. Например, с помощью cpu можно ограничить число ядер, которые используются в этой cgroup. Стоит отметить, что при контейнерной виртуализации для каждого контейнера используется отдельный набор пространств имен (namespace ) и отдельный набор cgroups, то есть разделение их между контейнерами не используется.

Типичными представителями контейнеров легковесной виртуализации в ОС Linux является Linux Containers [7] и OpenVZ [8]. Однако, в отличии от LXC, OpenVZ использует свой набор утилит для контейнеризации, пусть общая идея такая же, как описано выше. Чтобы работать с OpenVZ, необходима специальная настройка ядра под эти утилиты [8]. В то время как LXC работает в рамках стандартного ядра Linux.

Важной особенностью является тот факт, что все эти хосты работают внутри одной ОС, то есть все контейнеры, запущенные в рамках одной ОС, используют ядро ОС как свое ядро. Таким образом, нет возможности изменить ядро ОС для одного контейнера, не затронув остальные.

## 2.6. Контейнеризация приложений

Рассмотрим ситуацию, в которой пользователь хочет запустить новое приложение на своей машине. Однако скачивание, установка и настройка системы для работы с этим приложением занимает много времени и

усилий, а некоторые дополнительные пакеты имеют несовместимости с существующими на машине пользователя. Возникает необходимость создания изолированного окружения для приложения, которое содержит все необходимые зависимости и настройки, при этом это окружение недоступно приложениям извне.

Контейнеризация приложения - это упаковка приложения и всех его зависимостей в легковесный контейнер и последующая настройка этого контейнера. Примером может послужить перенос ftp-сервера с машины на машину. После установки в LXC-контейнер самого сервера, необходимо настроить проброс 21 порта контейнера на 21 порт машины пользователя. Таким образом, когда пользователь скачает такой LXC-контейнер и запустит его, то сразу получит работающий FTP-сервер, слушающий 21 порт.

Следующим шагом служит автоматизация работы с LXC-контейнером, то есть необходимые действия по упаковке и запуску осуществляет не пользователь, а программа.

Docker [9] - средство для контейнеризации приложения со всеми его зависимостями, обеспечивающее API высокого уровня к LXC-контейнерам. Приложение упаковывается в Docker-контейнер, файловая система которого может переноситься с машины на машину.

Docker позволяет модифицировать Docker-контейнеры, добавляя в них новые приложения. При этом создается новый контейнер.

Последнее свойство нуждается в пояснении своей реализации. Если в LXC-контейнере используется обычная файловая система, то файловая система Docker-контейнера образована "слоями" из файловых систем - то есть в Docker-контейнере добавление файла в контейнер осуществляется путем записи нового слоя, который содержит файл, поверх старых слоев. В момент запуска Docker-контейнера старые слои интерпретируются как единая файловая система с помощью средства union-filesystem

( разработка команды создателей Docker), причем эта единая система доступна только для чтения. Все изменения записываются поверх нее в новый слой ( стоит отметить, что в документации Docker нет пояснений по поводу реализации этой системы). Такой подход позволяет использовать чужие Docker-контейнеры как основу для новых.

Таким образом, Docker - удобная система для управления приложениями. Так же как и LXC-контейнеры, Docker-контейнеры, запущенные в рамках одной ОС, обладают общим ядром. Это приводит к тем же ограничением на запуск приложения в контейнере, что и для LXC-контейнеров.

За управление Docker-контейнерами отвечает Docker-daemon, процесс, работающий в хостовой ОС. Он запускает необходимые контейнеры ( то есть либо собирает новый LXC-контейнер, либо запускает копию уже существующего). Docker-daemon выдает IP-адрес контейнеру из диапазона, задаваемого IP-адресом и маской сети сетевого интерфейса, который указан в настройках Docker.

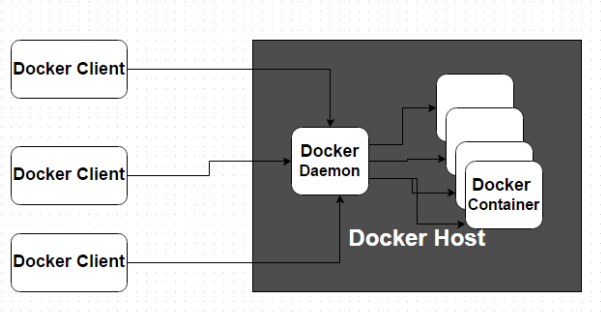


Рис. 2.3. Docker-демон

Сборка контейнера может производиться автоматически по Dockerfile - специальному файлу, в котором описана последовательность действий, необходимая для формирования нужного контейнера. Пользователь может сам написать подобный файл, а затем тот, кто получит такой файл, может сам собрать по нему нужный контейнер. При запуске Docker-контейнера возможно указать ограничения на размер доступной этому

контейнеру оперативной памяти и число ядер процессора, на которых можно исполнять приложения этого контейнера.

Стоит отметить, что Docker не единственное средство для контейнеризации приложения. Существует так же Rocket для Linux [10] и Drawbridge для Windows [11]. Однако наиболее развитым и активно поддерживаемым со стороны общественности является Docker.

Начиная с 2016 года, был разработан runC - описание поведения системы, производящей управление контейнерами. Хоть официально runC не является стандартом, но позволяет унифицировать описание работы контейнера, абстрагируясь от конкретных сущностей, и этому описанию стараются следовать все разработчики контейнеров. Однако родоначальником runC является Docker, то есть Docker работает по классической схеме runC [12]. Таким образом де-факто Docker становится стандартом контейнерной виртуализации.

Начиная с 2016 года, разработчики Windows и Docker анонсировали внедрение Docker в ядро Windows. Достигнуто это будет за счет того, что была добавлена возможность запуска Ubuntu в Windows за счет транслирования системных вызовов ядра Linux в системные вызовы ядра Windows. Аналогичные разработки ведутся и под MacOS.

## 2.7. Выводы

Наиболее часто используемыми технологиями в сфере облачных вычислений являются легковесная и аппаратная виртуализации. Первая технология позволяет управлять контейнерами, а вторая - виртуальными машинами.

В рамках данной работы будут рассматриваться гипервизор KVM для управления виртуальными машинами и система Docker для управления контейнерами приложений.

В KVM существует понятие виртуального процессора (vCPU). Эта абстракция описывает ядро, доступное виртуальной машине. Обычно виртуальной машине выделяются не все количество ядер хостовой системы,а лишь часть. При это по умолчанию в разные моменты времени виртуальная машина может как виртуальные ядра использовать разные ядра хостовой системы. Это контролирует гипервизор. В KVM присутсвует возможность закрепить конкретные ядра за конкретной машиной. [13]

В Docker система выделения ресурсов процессора контейнерам ближе к системы выделения ресурсов для процессов. При это отсутсвует возможность ограничить число доступных контейнеру ядер, возможно лишь четко заификсировать, какими ядрами пользоваться контейнеру. [13]

В KVM выделение оперативной памяти вирутальной машине происходит непрерывным блоком. То есть, даже если машина пользуется не всей доступной памятью, нет возможности передать неиспользуемую память другой машине. [13]

Контейнеры обладают более гибкой системой управления оперативной памятью. Возможно задать лишь верхнюю границу выделяемой контейнеру памяти. При этом неиспользованная память может быть передана другому контейнеру. [13]

Файловая система виртуальной машины является непрерывным файлом в хостовой операционной системе. Контейнеры же монтируют свою файловую систему в файловую систему хоста.

Контейнеры потребляют меньше ресурсов, чем виртуальные машины, так как нет необходимости поддерживать ядро гостевой ОС, и при этом позволяют запускать изолированные приложения. Запуск контейнеров происходит быстрее, чем запуск виртуальной машины. [14].

# Глава 3.

# Обзор существующих решений

В данном разделе будут рассмотрены существующие работы по исследованию производительности виртуальных машин и контейнеров. Будут выделены методики исследования.

## 3.1. Производительность виртуальной машины

Несмотря на то, что виртуальные машины,управляемые одним гипервизором, формально изолированы друг от друга, они используют одно и тоже физическое оборудование. При этом каждая виртуальная машина обладает своим планировщиком ресурсов, который никак не связан с планировщиками других машин, но при этом пытается управлять одним с ними физическим оборудованием. В случае паравиртуализации, при вызове нескольких системных вызовов от разных вирутальных машин в случае операций ввода-вывода обработка этих вызовов гипервизором так же может быть нетривиальной, а потому работа виртуальной машины будет отлична от работы той же системы на оборудовании без

виртуализации.

В работе [15] исследовалось влияние двух Xen-виртуальных машин друг на друга. В них запускались специализированные приложения, измеряющие производительность вычислений на процессоре, скорость обращений в оперативную память, производительность работы с жестким диском. Каждой машине было доступно по половине ресурсов хостовой системы, то есть они не конкурировали за ресурсы напрямую. При этом показатели производительности приложений были ниже, чем у аналогичной хостовой системы с таким же, как у машин, количеством ресурсов.

В работе [16] рассматривалась уже аппартная виртуализация. В качестве гипервизора использовался KVM. Авторы исследовали производительность одной виртуальной машины с помощью приложений, написанных на Matlab. Эти приложения так же измеряли производительность вычислений на процессоре, скорость обращений в оперативную память, производительность работы с жестким диском.

После чего, с помощью тех же приложений, проводилось исследование деградации производительности при использовании машинами одного и того же ядра, то есть в случае, когда обе виртуальные машины использовали один и тот же L1 и L2 кэши. Производительность была снижена из-за того, что гипервизор запускал вируальные машины на ядре не параллельно, а последовательно.

В третьем эксперименте проводились исследования, при которым виртуальные машины работали на разных ядрах одного процессора. В данной конфигурации машины обладали разными L1-кэшами, но одним L2-кэшем. Производительность была выше, чем во втором эксперименте, но ниже, чем в первом.

Во всех экспериментах в случае нескольких машин в каждой из них запускались одни и те же приложения.

Можно заключить, что использование специализированных бенчмар-

ков является подходом к измерению производительности. Результат работы таких приложений на единственной запущенной машине принимается за базовую производительность. После чего происходит запуск дополнительных машин, и во всех запускаются бенчмарки. Размещение дополнительных машин может производиться по двум разным стратегиям: размещение виртуальных машин максимально изолированно друг от друга или с частичным пересечением по доступным ресурсам.

Стоит отметить, что в данных работах не исследовалась деградация производительности в случае, когда коэффициент перекрытия превышает 1 (см. Введение).

В качестве дополнительных источников исследований производительности виртуальных машин можно исследовать работы [17] и [18]. Однако приведенные в них математические модели сосредоточены на проблеме миграции виртуальных машин и деградации их проивзводительности в этом случае, что выходит за рамки данной работы.

## 3.2. Производительность виртуальных машин и контейнеров

Существует ряд работ, сравнивающих производительность виртуальных машин и контейнеров.

В работе [19] в качестве гипервизора был выбран KVM, а в качестве системы управления контейнерами - Docker. В качестве приложений, считающих производительность, использовались бенчмарки, а в качестве реального приложения -MySQL. Сравнение производилось между контейнером и виртуальной машиной, которым доступны все ресурсы системы. По результатм работы, можно заключить, что контейнеры превосходят виртуальную машину, однако незначительно.

В работе [20] сранивалась производительность виртуальной машины

под управлением гипервизора Virtual Box и Docker-контейнера. Каждому были доступны все ресурсы системы. В качестве бенчмарков использовались стандартные бенчмарки, входящие в пакет бенчмарков Phoronix-test-suite [21]. По результатам экспериментов производительность контейнера либо значительно превышает производительность виртуальной машины, либо пракически эквивалентна, но так же в большую сторону.

Работа [13] затронула проблему деградации производительности при небольших (1.5 - 2) размерах коэффициента перекрытия. В данной работе авторы сравнивали виртуальные машины под управлением KVM и LXC-контейнеры. В них запускались специально разработанные бенчмарки. По результатам данной работы можно заключить, что контейнеры даже при коэффициенте перекрытия более 1 показывают большую производительность, чем виртуальные машины. Авторы работы связывают это с тем, что виртуальные машины, в отличии от контейнеров, жестко ограничены в своих ресурсах. То есть гипервизор не даст виртуальной машине ресурсов больше, чем ей выделено. Однако система управления контейнерами может такое позволить применительно к контейнерам.

## 3.3. Выводы

По результатам обзора можно сделать следующие выводы:

1. Для измерения производительности можно использовать готовые бенчмарки.

2. Базовые измерения производительности должны производиться в единственной виртуальной сущности.

3. Запуск дополнительных виртуальных сущностей должен произыодиться по двум стратегиям: либо новой сущности доступны все ре-

сурсы, либо только часть.

4. Гипотеза, приведенная в главе 1 может быть подтверждена.

# Глава 4.

# Практическая реализация

## 4.1. Требования

По результатам обзора, проведенном в 3, были сделаны выводы о требованиях к системе измерения и сравнения производительности виртуальных машин и контейнеров, а так же измерение и сравнение процесса деградации проивзводительности виртуальных машин и контейнеров.

Производительность предлагается измерять с помощью бенчмарков. В силу их великого разнообразия, система измерения должна быть способна запустить любой бенчмарк в виртуальном носителе. Стоит отметить, что не существует понятия проивзодительность виртуальной машины или контейнера в отрыве от термина производительность приложения. То, насколько производительна виртуальная сущность, можно судить только по результатам работы приложения.

В качестве виртуальных сущностей, используемых в системе, предлагается использовать виртуальную машину под управлением гипервизора KVM и Docker-контейнеры.

Дополнительно предлагается исследовать производительность Docker-контейнеров, запущенных в виртуальной машине. Данный подход набирает популярность в облачных системах. Гипотеза, приведенная в 1,

предполагает более низкую проивзодительность виртуальных машин по сравнению с контейнерами, а, значит, и контейнеров в виртуальной машине по сравнению с контейнерами в хостовой операционной системе.

Система будет работать с одним сервером со своим набором физических ресурсов. Все виртуальные сущности будут запускаться только на этом сервере.

Пусть X - количество доступных ядер процессора на сервере, а $\chi_i$ - количество ядер, требующихся для $i$-ой виртуальной сущности. Введем коэффициент $\theta = \frac{\sum\limits_i \chi_i}{X}$ - коэффициент перекрытия (overlap), являющийся отношением суммы ядер, требующихся виртуальным сущностям системы, к количеству ядер сервера.

Система должна уметь запускать виртуальные сущности с ограничением на доступные ресурсы и без ограничений. Под ресурсами понимается объем оперативной памяти и количество ядер процессора.

В первом случае предполагается, что все виртуальные сущности обладают одинаковыми требованиями к количеству ресурсов. Тогда при увеличении количества запущенных виртуальных сущностей коэффициент перекрытия растет со скоростью $\upsilon = \frac{\chi}{X}$, где $\chi$ - количество ядер виртуальной сущности, $X$ - общее число ядер сервера.

Во втором случае предполагается, что виртуальной сущности доступны все ресурсы системы. Тогда при увеличении количества запущенных виртуальных сущностей коэффициент перекрытия растет со скоростью $\upsilon = 1.0$.

Система будет исследовать деградацию производительности виртуальной сущности как зависимость производительности $P$ от количества запущенных виртуальных сущностей.

## 4.2. Алгоритм работы реализованной системы

На рисунке 4.1 представлена диаграма классов разработанной системы.

Под носителем в данном разделе будет пониматься та виртуальная сущность, исследование которой производится, то есть либо контейнер, либо виртуальная машина.

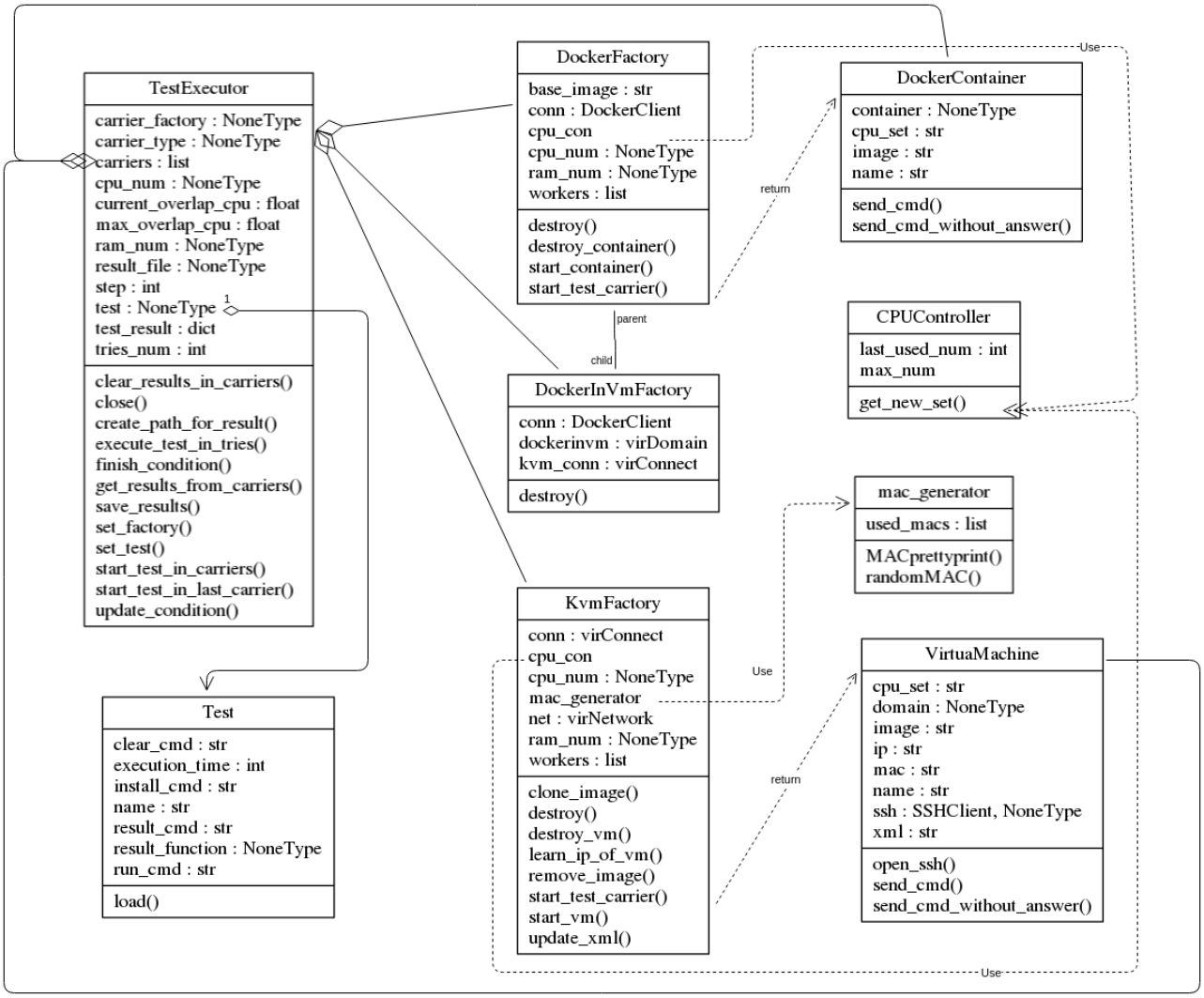


Рис. 4.1. UML-диаграмма классов

Основной алгоритм работы системы можно описать следующим образом. В начале работы создается объект класса TestExecutor, содержащий описание теста, который должен производиться в виртуальном носителе, тип исследуемого виртуального носителя, а так же ограничения на

ресурсы экземпляров сущности.

После этого происходит запуск первого виртуального носителя. В него производится установка необходимого теста, команда по установке содержится в описании теста. Образ, полученный в результате усановки приложения, становится базовым, то есть тем, чьи копии будут использовать будущие экземпляры виртуальной сущности с этим тестом.

После получения объекта виртуальной сущности, являющейся объектом класса DockerContainer или VirtualMachine (в зависимости от конфигурации TestExecutor), во всех уже запущенных экземплярах происходит запуск команды. Чтобы не было ситуации, при которой тест в одном носителе происходит раньше, чем в другом, запуск теста планируется на определенный момент в будущем.

После этого система ожидает завершения теста в последнем запущенном носителе. Это необходимо для корректного получения результатов, которые будут готовы только после завершения теста во всех носителях.

При проведении тестовых запусков было замечено, что не всегда тесты завершаются во всех носителях в момент завершения теста в последнем носителе. Поэтому получение результатов выглядит следующим образом: по всем носителям запускается команда сбора результатов, так же содержащаяся в описании теста. Если результаты отсутсвуют в носителе, то это означает, что тест в данном носителе еще не завршен. За каждый не полученный результат система добавляет время ожидания. В случае, если получены не все результаты, система блокируется на это время ожидания. По истечении времени, происхоит новая попытка получить результаты.

После успешного получения результатов, система к запуску нового экземпляра и повторения шагов, описанных выше.

При запуске каждого экземпляра происходит обновление текущего значения коэффициента перекрытия.

В классе TestExecutor существует поле max_overlap_cpu, В данном поле содержится то значение коэффициента перекрытия, при котором надо остановить систему и удалить текущие запущенные виртуальные сущности.

Классы VirtualMachine и DockerContainer содержат методы send_cmd и send_cmd_without_waiting. С помощью данных методов происходит исполнение переданной как параметр команды в виртуальной сущности.

За запуск новых экземпляров виртуальной сущности отвечают класса KvmFactory, DockerFactory и DockerInVmFactory. Данные классы содержат методы start_test_carrier, которые возвращают объекты типа DockerContainer или VirtualMachine. Так же эти классы содержат поле типа CPUController. Этот класс отвечает за планирование ядер процессора виртуальным сущностям.

Класс Test хранит описание теста в виде, понятном классу TestExecutor. Он создается один раз при запуске системы.

Класс mac_generator является вспомогательным классом для класса KvmFactory. Его функция - генерация уникальных MAC-адресов для виртуальных машин.

## 4.3. Применимость

Реализованная система предназначена в первую очередь для того, чтобы исследовать производительность вируальных машин и контейнеров. Несмотря на то, что в данной работе сравниваются KVM-машины и Docker-контейнеры, система легко может быть доработана для работы с другими гипервизорами и системами управления контейнерами.

В данной системе может быть запущено любое приложение, причем это может быть необязательно специализированный бенчмарк. Поэтому данная система может быть использована в облачных системах для

исследования новых приложений. С помощью нее возможно запустить приложение в разных виртуальных сущностях, изучить его производительность и сделать вывод о наилучшей стратегии запуска данного приложения в облаке, причем с учетом возможного масштабирования и раюты данного приложения рядом с другими приложениями, работающими на том же физическом сервере.

# Глава 5.

# Экспериментальное исследование

# Глава 6.

# Заключение

# Литература


[1] Hyungro Lee. Virtualization Basics: Understanding Techniques and Fundamentals. // School of Informatics and Computing, Indiana University, 2014.

[2] Homepage of VMWare ESXi. http://www.vmware.com/ru/products/esxi-and-esx.html (дата обращения 01.09.2017)

[3] Hasan Fayyad-Kazan, Luc Perneel, Martin Timmerman. Full and Para-Virtualization with Xen: A Performance Comparison. // Journal of Emerging Trends in Computing and Information Sciences, 2013.

[4] Homepage of Xen. https://www.xenproject.org (дата обращения 01.09.2017)

[5] Homepage of VirtualBox. https://www.virtualbox.org (дата обращения 01.09.2017)

[6] Homepage of KVM. https://www.linux-kvm.org (дата обращения 01.09.2017)

[7] Homepage of LXC. URL: https://linuxcontainers.org/ru/ (дата обращения 01.09.2017)

[8] Homepage of OpenVZ. URL: https://openvz.org/ (дата обращения 31.10.2017)

[9] Homepage of Docker. URL: https://docs.docker.com/ (дата обращения 01.09.2017)

[10] Homepage of Rocket. URL: https://coreos.com/rkt/docs/latest/ (дата обращения 31.10.2017)

[11] Homepage of DrawBridge. URL: http://research.microsoft.com/en-us/projects/drawbridge/ (дата обращения 31.10.2017)

[12] Homepage of runC. URL: https://runc.io/ (дата обращения 30.04.2017)

[13] Lucas Chaufournier, Prateek Sharma, Prashant Shenoy. Containers and Virtual Machines at Scale: A Comparative Study. // Middleware '16 Proceedings of the 17th International Middleware Conference, 2016.

[14] Михеев Павел. Разработка и реализация системы масштабирования виртуальных сетевых функции с помощью легковесной виртуализации. // МГУ, 2017.

[15] Younggyun Koh, Rob Knauerhase [и др.]. An Analysis of Performance Interference Effects in Virtual Environments. // Performance Analysis of Systems & Software IEEE International Symposium on, 2007.

[16] George Kousiouris, Tommaso Cucinotta, Theodora Varvarigoua. The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks. // The Journal of Systems and Software, 2011.

[17] Anton Beloglazov, Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers. // Wiley Online Library, 2011.

[18] Haikun Liu, Hai Jin, Cheng-Zhong Xu, Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. // Springer US, 2013.

[19] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. // 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015.

[20] Rabindra K. Barik, Rakesh K. Lenka [и др.]. Performance Analysis of Virtual Machines and Containers in Cloud Computing. // International Conference on Computing, Communication and Automation (ICCCA2016), 2016.

[21] Homepage of Phoronix-test-suite. URL: http://www.phoronix-test-suite.com/ (дата обращения 01.03.2018)