DOI: 10.15514/ISPRAS-2019-31(6)-7

Кэширование данных в мультиконтейнерных системах

[1] *Д.А. Грушин, ORCID: 0000-0002-6789-5473 <grushin@ispras.ru>*
[1,2] *Д.О. Лазарев, ORCID: 0000-0002-6253-6447 <denis.lazarev@phystech.edu>*
[1,2] *С.А. Фомин, ORCID: 0000-0002-1151-2189 <fomin@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. Сегодня виртуализация – это ключевая технология облачных вычислений и современных центров обработки данных, обеспечивающая масштабируемость и безопасность, управление глобальной ИТ-инфраструктурой и снижение затрат. Среди методов виртуализации, наиболее популярной стала контейнеризация – изоляция связанных групп процессов, разделяющих общее ядро операционной системы. Эффективность контейнеризации в сравнении с классической аппаратной виртуализацией, проявляется в компактности контейнеров и меньших накладных затратах вычислительных ресурсов – памяти, диска, ЦПУ. Однако в сравнении с классическими архитектурами без изоляции процессов контейнеры могут обходиться дороже, и в любом случае, индустрия ждет дополнительной оптимизации – скорости запуска, экономии памяти и дискового пространства и других ресурсов. В этом может помочь различное кеширование – старейший механизм повышения производительности программ без радикальной модификации алгоритма и оборудования. Однако при этом возникают различные архитектурно-инженерные дилеммы вида «безопасность или эффективность» и здесь мы рассмотрим современные научно-технические подходы к их решению в разных аспектах – ускорение запуска, оптимизация совместного использования, ускорение сборки образов, а также некоторые проблемы безопасности, возникшие из-за агрессивного кэширования в процессорных архитектурах. А в некоторых сценариях использования мультиконтейнерных систем наоборот, скорость и задержки не важны, важно обеспечить максимальную загрузку физических серверов – в этом случае актуальны алгоритмы планирования и размещения контейнеров, и нами приведен обзор теоретических работ на эту тему.

Ключевые слова: контейнеры; кеширование; облачные вычисления

Для цитирования: Грушин Д.А., Лазарев Д.О., Фомин С.А. Кэширование данных в мультиконтейнерных системах. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 125–144. DOI: 10.15514/ISPRAS–2019–31(6)–7

Благодарности: Исследования, результаты которых представлены в этой статье, выполнены при поддержке Российского фонда фундаментальных исследований (проект 17-07-01006).

Data caching in multi-container systems

[1] *D.A. Grushin, ORCID: 0000-0002-6789-5473 <grushin@ispras.ru>*
[1,2] *D.O. Lazarev, ORCID: 0000-0002-6253-6447 <denis.lazarev@phystech.edu>*
[1,2] *S.A. Fomin, ORCID: 0000-0002-1151-2189 <fomin@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*

Abstract. Today, virtualization is a key technology for cloud computing and modern data centers, providing scalability and security, managing the global IT infrastructure and reducing costs. Among the methods of virtualization, the most popular was containerization, that is the isolation of related groups of linux processes that share a common Linux kernel. Containerization is more profitable that classical hardware virtualization because of compactness of containers and lower overhead costs of memory, disk, CPU. However, in comparison with classical architectures without process isolation containers can cost more, and in any case, the industry is waiting for additional optimization – the speed of launch, saving memory and disk space and other resources. Different caching techniques can help in this, because Caching is the oldest mechanism of increasing software productivity without radical modification of algorithms and hardware. However, there are a lot of architectural and engineering tradeoffs. Here we will consider modern scientific and technical approaches to their solution in different aspects – acceleration of launch, optimization of shared usage, acceleration of building container images, as well as some security problems caused by aggressive caching in modern processor architectures. And in some use cases for multi-container systems performance and latency are not important, but we have to ensure the maximum load of physical servers. In these cases, the algorithms of planning and placement of containers are relevant, and we give an overview of theoretical work on this topic.

Keywords: containers; caching; cloud computing

For citation: Grushin D.A., Lazarev D.O., Fomin S.A. Data caching in multi-container systems. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 125-144 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-7

Acknowledgements: The studies, the results of which are presented in this article, were supported by the Russian Foundation for Basic Research (project 17-07-01006).

1. Введение

Кеширование – один из важнейших механизмов повышения производительности, без радикальной модификации алгоритма и оборудования. К сожалению, в реальности не существует компьютеров, реализующих теоретическую модель RAM – Random Access Machine, машину с произвольным доступом к памяти, способную за один условных такт дотянутся до любой ячейки однородной памяти. В реальных вычислениях существует очень строгая иерархия доступа к памяти, от наносекунд к кэшам первого уровня процессора, до секунд доступа к удаленным сетевых хранилищам на шпиндельных жестких дисках.

Как пример, можно привести ориентировочные времена доступа к различным устройствам (иерархия – кэши процессора, память, ssd-диски, сеть, … [1], табл. 1).

Видно, что каждый уровень «попадания в кеш» (не важно, речь идет о диске при кэшировании сети, или про кэш первого уровня при кэшировании второго уровня), может увеличить производительность на порядки. Кроме того, кеширование бывает и вычислительным, т.е. обеспечивается быстрый доступ не только к элементам, хранящимся на более «низком» уровне систем хранения, но и сохраненным результатам вычисления, требующим серьезных временных затрат. Кеширование – старая технология,

термин «cache» появился в 1967 году, но практически этот подход – ровесник «электронно-вычислительных машин».

Табл. 1. Иерархия типов памяти и задержки доступа
Table 1. Hierarchy of memory types and access delays

Операция	Задержка (в наносек)
Доступ к кэшу первого уровня	0.5
Доступ к кэшу второго уровня	7
Доступ к DRAM	100
3D XPoint на основе чтения NVMe SSD	10000
NAND NVMe SSD R/W	20000
NAND SATA SSD R/W	50000
Случайное чтение блоков 4K с SSD	150000
Задержка P2P TCP/IP (физика на физику)	150000
Задержка P2P TCP/IP (ВМ на ВМ)	250000
Последовательное чтение 1MB из памяти	250000
Сетевая задержка внутри ЦОД	500000
Последовательное чтение 1 MB с SSD	1000000
Поиск диска	10000000
Последовательное чтение 1 MB с диска	20000000
Отправка пакетов США → Европа → США	150000000

С другой стороны, контейнеры – недавно появившаяся технология обеспечения дешевой изоляции групп процессов, давшая огромный развитие IT-индустрии, родившая такие практики как DevOps и микросервисная архитектура. Конечно, сама идея некоторой изоляции процессов на уровне операционной системы не нова – «chroot» был введен в 1982 году, но реальная изоляция процессов появилась только в начале нулевых с FreeBSD Jail, эволюционировала к концу нулевых уже к промышленным технологиям LXC, Solaris Containers и OpenVZ/Virtuozzo, но только с появлением Docker в середине десятых годов нашего века стала поистине массовой, настолько, что большинство ITшников при слове «контейнеры» вспоминают только технологии от «докер», что безусловно обидно для первопроходцев контейнеризации.

Контейнеры практически окончили войну между разработчиками и системными администраторами, дав возможность первым практически никак не зависеть от тонкостей целевых платформ, и их особенностей – будь то операционная система и ее дистрибутив, или оборудование с его глюками. Теперь, для каждой выкатки новой версии не требуется многомесячное согласование «отдела разработки» и «отдела администрирования», появилась возможность выкатывать в «production» новые версии ежечасно, более того, появились новые архитектуры оркестрации контейнеров, позволяющие экономить на долгом цикле тестирования и поддерживать в работе несколько версий сервиса, автоматически переключаясь на проверенную версию, в случае проблем с новой. Все это, а также тенденция к глобализации вычислительных ресурсов, обозначаемых сильно заезженным маркетинговым термином «облака», сильно уронило престиж и ценность «бородатых сисадминов», хранителей знаний о взаимодействии зоопарка различного оборудования и операционных систем. Большинству из них пришлось переквалифицироваться в DevOps инженеров, специалистов по сборке контейнеров,

развертывании вычислительной инфраструктуры, и они стали близки с разработчиками как по навыкам – «программирование инфраструктуры как кода» вместо беготни с оборудованием, так и по взаимодействию – DevOps специалистов принято размещать вместе с командой разработки.

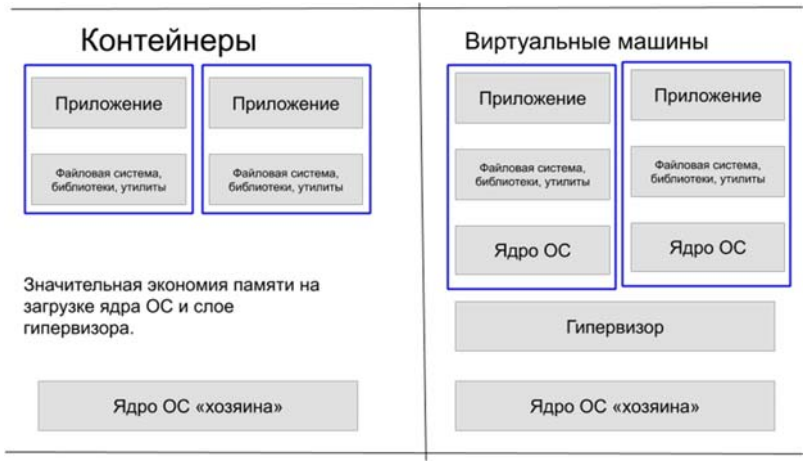

Рис. 1. Архитектурная выгода контейнеров в сравнении с классической виртуализацией
Fig. 1. The architectural benefits of containers versus classic virtualization

Даже если не используются микросервисные архитектуры и используется классический подход «виртуальная машина с сервисами», зачастую предоставляемая провайдерами виртуальная машина – это тоже контейнер (LXC или OpenVZ), ведь таким образом достигается серьезная экономии памяти (все контейнеры используют только одно ядро ОС), и достигается их большая плотность размещения на физических серверах (рис. 1).

Но разумеется, весь этот тренд на изоляцию и дублирование процессов, обеспеченный контейнеризацией находится в некоторой оппозиции к эффективности использования памяти и ресурсов, примерно в такой же, в которой «конфликтуют» архитектурные аспекты «эффективности» и «безопасности». Соответственно, возникают инженерные и научные проблемы, как решить эти технологические дилеммы, как сохранить эффективность при использовании контейнеров, какого рода кеширование, и в каком случае, может тут помочь.

Далее мы приведем несколько разделов с обзором различных аспектов темы «контейнеры и кеши», рассмотрев современные научно-практические подходы к их решению в разных аспектах – ускорение запуска, оптимизация совместного использования, ускорение сборки образов, а также некоторые проблемам безопасности, возникшим из-за агрессивного кэширования в процессорных архитектурах.

В некоторых сценариях использования мультиконтейнерных систем наоборот, скорость и задержки не важны, важно обеспечить максимальную загрузку физических серверов – в этом случае актуальны алгоритмы планирования и размещения контейнеров – и мы привели обзор некоторых теоретических алгоритмов, еще не реализованных в системах виртуализации или контейнерной оркестрации.

2. Межконтейнерное совместное использование файлов

Несмотря на то, что контейнеры сильно выигрывают в плотности размещения по сравнению с виртуальными машинами, есть множество областей, где сложились практики обеспечивающие еще большую плотность сервисов. Одна из таких областей – так называемый shared-хостинг, т.е. хостинг, в котором все сайты живут внутри одной операционной системы (Linux), пользуясь общим сервисом СУБД (обычно MySQL) под разными аккаунтами, общим веб-сервером (обычно связка Apache+PHP), так называемым классическим веб-стеком LAMP, под которым до сих пор работают большинство популярных CMS, Content Management Systems, систем управления контентом – таких как Wordpress, Drupal, Joomla, Bitrix, Mediawiki …, т.е. все те же фреймворки, на которых до сих пор держится большинство сайтов – от блогов и сайтов визиток, до порталов сообществ и интернет магазинов (рис. 2).

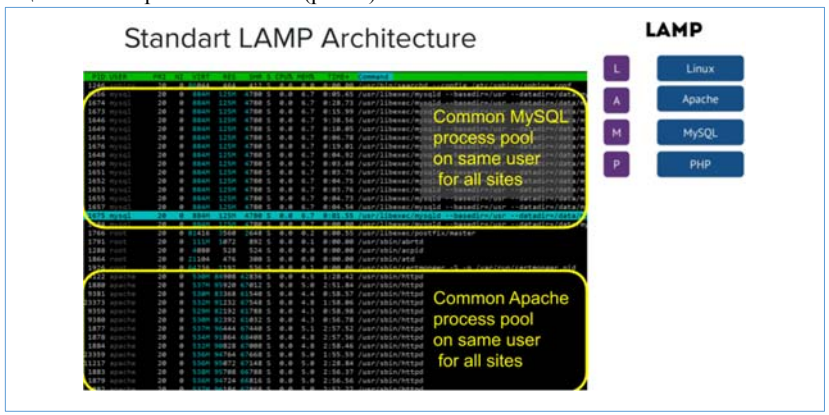

Рис. 2. Архитектура LAMP – Linux, Apache, MySQL, PHP
Fig. 2. LAMP architecture – Linux, Apache, MySQL, PHP

Плотность размещения сайтов внутри при разделяемом хостинге (*шаред* хостинге, до сих пор нет устоявшегося русского перевода для термина *shared hosting*) может быть на порядок больше, чем при хостинге отдельных сайтов внутри виртуальных машин и даже контейнеров, и по сути, ограничена только совестью провайдеров. Обычно, провайдеры пользуются тем, что такие дешевые услуги покупают владельцы слабо посещаемых сайтов – непопулярные блоги, сайты-визитки малоизвестных компаний, хобби-проекты, и часто им удается размещать до десятков тысяч сайтов внутри одного физического сервера.

Однако эта дешевизна имеет и оборотную сторону. В виду отсутствия настоящей изоляции процессов, один сайт получивший внезапную нагрузку, «выедает» все вычислительные ресурсы физического сервера (процессор, память, дисковые IOPSы, сеть) и «кладет», т.е. вызывает отказ в обслуживании у всех остальных сайтов на том же сервере. Причем эта нагрузка может быть не обязательно в результате DDOSa, возможно просто внезапный всплеск посещаемости, когда ссылка на сайт попадает на популярных ресурс («reddit-эффект» или «хабра-эффект», по названию известных коллективных блогов, по ссылке с которых могут пойти одновременно десятки тысяч пользователей). Контейнеризация, т.е. изоляция в отдельных контейнерах хотя бы процессов вебсервера могла бы решить эту проблему, т.к. для отдельного контейнера можно поставить ограничения по потреблению CPU и памяти.

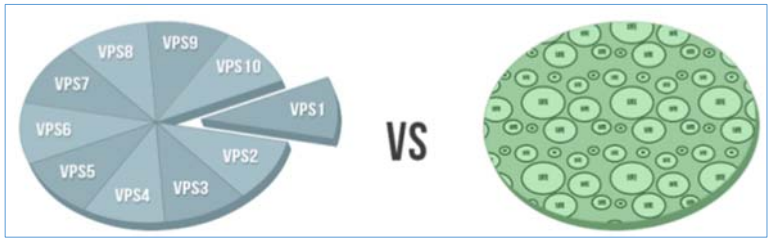

Рис. 3 Преимущество разделяемого хостинга – в плотности
Fig. 3 The advantage of shared hosting is density

Да, контейнеры были бы лучше, если бы их удалось сделать более экономичными, не дублирующими ни общие файлы библиотек и системных процессов (glibc, apache-nginx-php-python-mysql-postgres), ни развесистые файлы прикладных фреймворков на PHP и других языках, которые могут занимать сотни мегабайтов дискового пространства, и, что более вредно, одни и те же файлы в разных контейнерах, занимают свое собственное место в страничном кэше файловой системы в оперативной памяти (рис. 3)! Аналогичная проблема возникает и вне шаредхостинга, в микросервисной архитектуре, когда приложение разбито на десятки микросервисов, каждый из которых тянет с собой как минимум ограниченную версию дистрибутива и легкий веб-сервер (обычно nginx), а максимум… не ограничен, и многогигабайтные образы контейнеров – вполне не редкость.

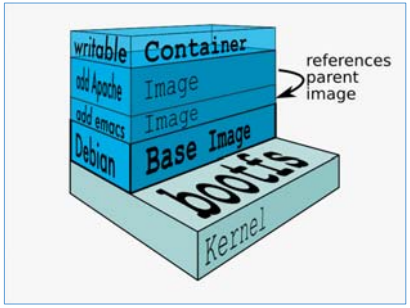

Рис. 4. Слои overlayfs в docker-контейнерах
Fig. 4. Layers of overlayfs in docker containers

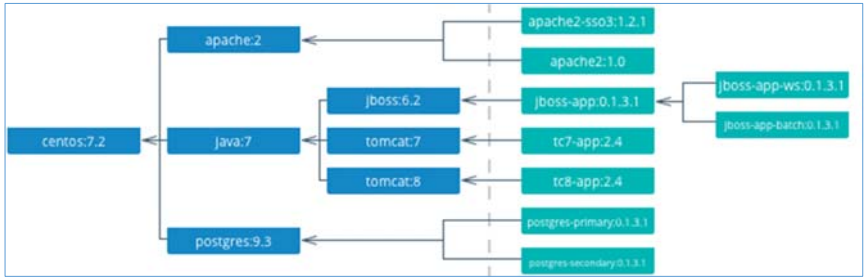

Рис. 5. Иерархия и наследование docker-образов
Fig. 5. Hierarchy and inheritance of docker images

С проблемой экономии дискового пространства удалось справиться технологии Docker-образов, использующих с одной стороны, технологию «слоистой» файловой системы

overlay-fs (рис. 4), когда можно добавлять свои «дельта-слои», добавляющие (или удаляющие) какие-то файлы к уже известным и неизменным образам, с другой – выстраивая эти образы в иерархию зависимостей (рис. 5), плюс инфраструктура построения и распространения таких образов (docker-реестр).

Однако эта технология, overlayfs, не подходит для классического шаредхостинга, где не очень продвинутый пользователь может только «положить и поправить» PHP-файлы (несмотря на то, что в сотнях соседних контейнеров они уже есть, и есть общий докер-образ, включающих нужный PHP-фреймворк), либо где такие общие файлы образуются множеством других образов в обход технологии докер-образов. Или докер вовсе не используется, а используется OpenVZ, LVX или Virtuozzo. Ну и в целом, нельзя заранее выстроить целостную иерархию файловых слоев, каждый контейнер управляется непредсказуемо и независимо, но в целом, очень много одинаковых файлов и очень хотелось бы что-то с этим сделать.

В таком случае, можно использовать технологию PFCache, VZFs, или аналогичную, позволяющую дедуплицировать операции над файлами. Идея этого подхода состоит в следующем. Файловые системы для процессов внутри контейнера создавать не напрямую на блочных устройствах хоста, а на специальном промежуточном блочном устройстве (ПБУ).

Это ПБУ можно создать (рис. 6)

- на обычных дисках хоста с помощью Union FS;
- в файлах хостовой системы, с помощью Loop device или Ploop, ZFS ZVol или подтомов BTRFS;
- LVM-разделов на блочном устройстве хоста;
- на сетевом кластере, с помощью, например, Ceph RBD.

В любом случае, у нас возникает потенциальная возможность контроля за системными файловыми вызовами внутри контейнера.

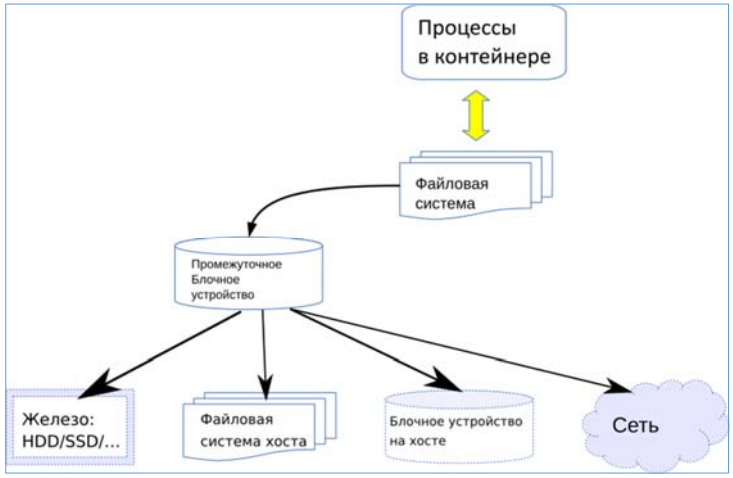

Рис. 6. Промежуточное блочное устройство для перехвата обращений к файлам
Fig. 6. Intermediate block device for intercepting file access

Далее заводится дисковый кеш на отдельном ПБУ, выделенном под эту задачу (на быстром устройстве, например, SSD NMVE). Этим ПБУ будут пользоваться все

контейнеры/VPS внутри узла – физического или виртуального сервера. Затем, для процессов в контейнерах используется файловая система Ext4, где каждый файл, кроме имени, может нести дополнительные атрибуты в метаинформации (xattr) (рис. 7).

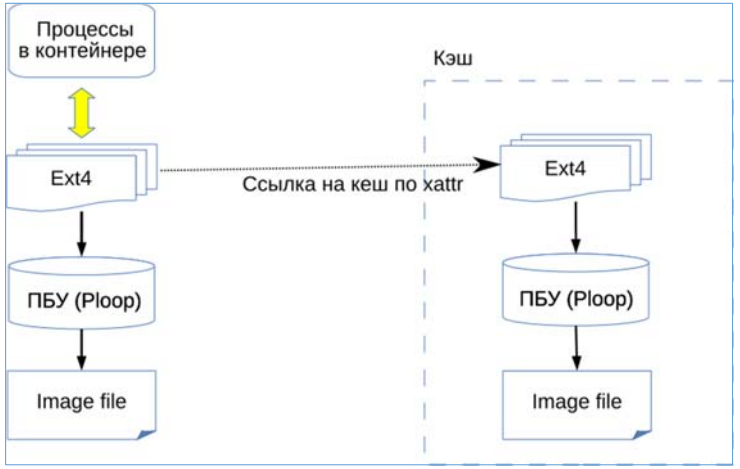

Рис. 7. Кэширование обращений к файлам с использованием файловых атрибутов ext4
Fig. 7. Caching file access using ext4 file attributes

В процессе эксплуатации происходит следующее.

1. На уровне контейнера, специальный пользовательский сервис прописывает SHA1 хеши файлов в их мета-атрибут *xattr*.
2. На уровне ядра:
   a. нужно периодически собирать статистику чтения файлов из ядра, анализирует ее, и добавлять файлы в кэш, если их использование частое;
   b. при чтении файла проверяется, содержит ли он указанный хеш в расширенных атрибутах xattr; если содержит – открывается «общий» файл, вместо файла контейнера;
   c. при записи в файл хеш инвалидируется; таким образом, при последующем открытии будет открываться уже непосредственно файл контейнера, а не его кеш.

В результате, держа в страничном кэше в основном «общие файлы» для всех контейнеров, получается значительная экономия как самого кэша и, следовательно, оперативной памяти, а также существенной экономии на дисковых операциях, ведь вместо чтения десяти файлов с диска, читается один, который сразу идет в страничный кэш.

Более детально, для технологии Pfcache с конкретными командами администрирования, этот процесс рассмотрен в статье [2], для технологии VZFS в [3], но аналогичный подход вполне может быть реализован и самостоятельно.

3. Кеширование старта контейнера

Контейнеры, автоматически запускаемые системой оркестрации, такой как Kubernetes, дали «путевку в жизнь» не только микросервисной архитектуре, но и такому модному тренду как «serverless», когда разработчики практически ничего не знают о целевой платформе, и реализуют только отдельные, максимально атомарные функции на удобных для них языках программирования. Всю остальную работу – запуск этих функций в изолированных контейнерах, балансировка, и самое главное – автоматическое

масштабирование – делает за них система оркестрации. Т.е. больше нет необходимости в редких специалистах по «highload»-архитектуре, снобах-архитекторах, умеющих рисовать диаграммы балансировки, или слишком дорогих разработчиках, изобретающих свои велосипеды по динамическому масштабированию числа сервисов при росте нагрузке. Все это может взять на себя система оркестрации, и скорее всего, это будет самый популярный за последние годы, проект Kubernetes.

Но есть еще один момент на переднем крае «технологических проблем» – проблема задержки при холодном старте. Действительно, при внезапном росте нагрузки (приход новых пользователей, неожиданный всплеск), система оркестрации автоматически поднимет дополнительные контейнеры для обслуживания… но сам старт этих контейнеров – дело не быстрое. Тут и старт самого контейнера (что, например, в docker, далеко не мгновенно), и самое главное – старт собственно сервисов внутри контейнера. Не важно, написаны они на «медленном» python, или «быстрых» NodeJS или Java, старт их может занимать сотни миллисекунд, а то и секунды/минуты. Если для обычного сайта, и даже интернет-магазина это наверное не является критической проблемой, то есть множество сервисов, обычно имеющих в своих названиях «real time», требующих реакцию в считанные миллисекунды (обычно порядка сотни мс). Это могут быть разные разновидности роботизированных торговых агентов, причем не обязательно для понятных «человеческих» бирж, где торгуют валютами и ценными бумагами, а например, real time bidding (RTB) для рекламных систем. В тот момент, когда пользователь открывает какую-нибудь страницу в интернете, пока еще идет загрузка, во множестве рекламных систем RTB-систем происходит соревновательная торговля ботов-агентов, за то, какую рекламу показать этому пользователю. Ну и разумеется, быстрая реакция будет нужна и для растущего рынка IoT устройств, особенно медицинских.

Тут как раз очень востребовано упомянутое в «введении» вычислительное кеширование. Как бы сделать так, чтобы запускаемые контейнеры уже оживали в рабочем состоянии, а не мучительно читали конфигурационные файлы и собирали в памяти нужные библиотеки и кеши?

Хорошие новости в том, что для этой задачи сейчас удается применить интересный проект CRIU – *Checkpoint/Restore In Userspace*. Рожденный русскими разработчиками Virtuozzo в 2014 (см. [4]), сейчас он стал уже стабильным и надежным международным продуктом с открытыми исходными кодами, которыми пользуются в своих решениях такие гиганты индустрии, как IBM/RedHat и Google. Изначально этот проект был создан для решения проблемы «живой» миграции контейнера, без остановки, с одного физического узла на другой. Подобное решение для виртуальных машин уже было, и несмотря на сложность живой миграции виртуальных машин, стоит признать, что оно алгоритмически и системно проще – работает на уровне гипервизора, перемещает постепенно блоки диска и оперативной памяти, пока процесс не сойдется и виртуальная машина не оживет на другом сервере. В случае с контейнерами, надо помнить что контейнер – просто группа процессов и ресурсов (сокеты, блокировки и т.п.) внутри одной ОС, их очень непросто даже собрать вместе с клубком их зависимостей (там все сложно, см., например, свежие теоретические модели и алгоритмы [5]), не говоря уже о перемещении на другой сервер, и выполнять большую часть работы надо не в гипервизоре, в пользовательском пространстве, наравне с перемещаемыми процессами.

CRIU безусловно является «швейцарским ножом» для целого ряда разнородных задач (есть даже использование для «паузы в играх»), но именно возможность «заморозить» и «восстановить» группу процессов, оказалась той самой «серебряной пулей» для задачи быстрого «холодного старта», которую и стали пытаться использовать практически только год назад, добиваясь отличных результатов.

Так, в докладе [6] приведены приведенные в табл. 2 цифры обычного времени старта против восстановления через CRUI для тестовых приложений на разных современных фреймворках – видно, что стабильно получается выигрыш практически на порядок, сводя задержку старта к уровню сетевой между географически удаленными датацентрами.

Табл. 1: Выигрыш в скорости запуска контейнера при использовании CRIU
Table 2: Gain in container launch speed when using CRIU

Время до старта (мс)	python 3	nodejs 6	java 8
Обычный старт	919	743	566
Восстановление	78	91	89

Более того, аналогичный подход можно применять не только для того чтобы запускать миниатюрные микросервисы, но и чтобы реплицировать готовые базы или кэширующие сервисы (т.к. решать проблему «прогрева кэшей» для веб-сервисов).

Также в [6] (кстати, это исследование от IBM и Red Hat) приводится эксперимент с наполнением кэширующих NoSQL баз данных 100K короткими записями в сравнении с миграцией уже подготовленного контейнера (табл. 3).

Табл. 2: Выигрыш в скорости репликации хранилища при использовании CRIU
Tab. 3: Gain in storage replication speed when using CRIU

	memcached	redis
Наполнение	1.238	6.254
Миграция	0.806	1.671

Прямо сейчас еще нет проверенных, стабильных решений, реализующих этот подход, есть попытки использовать его в стартапе SwiftyCloud[1] и в проекте gVisor[2], но эта технология уже в плане развития Kubernetes, можно ожидать, что появится в ближайшем году.

Отдельный вопрос при запуске контейнеров системой оркестрации на распределенном ЦОД – обеспечение быстрой доступности самих образов на физических нодах. При быстром интерконнекте (10GB Ethernet, или Infiniband, с низкими задержками) эта проблема может быть решена использованием кластерной файловой системой, CEPH/GlusterFS/ZFS, или использованием специализированных сетевых хранилищ. Но если быстрый интерконнект отсутствует, или невозможен, например, в случае географически распределенных ЦОД, возникает задача предварительной планировки размещения образов контейнеров на всех физических узлах. Да, можно заранее реплицировать реестр всех контейнеров на все узлы, но это может привести и к существенному расходу дискового пространства и сетевой ширины канала, поэтому сейчас есть исследования, как размещать на узлах только часть докер-образов, основываясь на статистике использования или различных алгоритмах, таких как многомерная упаковка контейнеров [7].

4. Кэширование, контейнеры и безопасность

Наверное самой большой иллюстрацией инженерного конфликта между «эффективностью» и «безопасностью», а именно, между «кешированием» и «изоляцией», явилась открытая в 2018 году серия уязвимостей, известная теперь под «брендами» MELTDOWN и SPECTRE. Смешные картинки-логотипы (рис. 8), должны всем донести разницу – если «Meltdown» – «утечка» памяти ядра в юзерспейс, вещь неприятная, но которую уже придумали, как починить, то «Spectre» – страшное привидение, проникающее через любые закрытые двери («изолирующие песочницы», проверки в совершенно корректных программах).

[1] https://github.com/swiftycloud
[2] https://gvisor.dev

Рис. 8: Логотипы уязвимостей Meltdown и Spectre
Fig. 8: Vulnerability logos for Meltdown and Specter

Для тех, кто не в курсе, напомним основные идеи этих уязвимостей. Гонка последних десятилетий за производительностью процессоров, чтобы соответствовать маркетинговым ожиданиям так называемого «закона Мура», привела не только к росту параллелизма и ядер, но и хитрым техникам реализации микроархитектуры процессоров, основанных на аппаратных кешах и спекулятивных вычислениях. Вспомните таблицу иерархии задержек доступа к памяти из введения – доступ к оперативной памяти по сравнению с кешами процессора в сотни раз дольше, и при любой операции, где операнд не попадает в кеш, возникает «голодание по данным» – процессор бессмысленно простаивает в ожидании данных, и часто в операциях ветвления в зависимости от значения в памяти, выгодно, не дожидаясь выяснения результата в условии, выполнить «упреждающие вычисления» (*instruction speculation*, *speculative execution*) наиболее вероятной ветки, параллельно дождавшись «приехавшего условия», ну и если «не повезло», то отбросить вычисленные результаты, восстановив регистры (до оперативной памяти результаты все равно не успеют доехать, это еще дольше, чем читать RAM). Кроме ветвлений так ускоряют и некоторые циклы, и в целом, это, кроме многоядерности, и было основным методом роста производительности процессоров основных производителей популярных архитектур (Intel, AMD, ARM, Power), с тех пор, как из-за инженерных ограничений практически остановилась миниатюризация и рост частоты. К сожалению, это открыло возможность к еще одной «атаке по побочным каналам» (*side-channel attack*), используя в качестве такого канала время чтения данных.

Если кратко, то заставив процессор упреждающе выполнить операцию с косвенной адресацией по атакуемому участку памяти, разумеется, мы не получаем ни значение этой ячейки напрямую, ни то, куда она указывает – это даже не нужно атакующему, и процессор, выполнив упреждающе эту операцию, поймет, что сделал это зря, спокойно сбросит регистры, но значение атакуемой ячейки осядет в одном из кэшей процессора, в подсистеме TLB, Translation Lookaside Buffers, который не сбрасывается вообще никогда. Затем, можно устроить перебор выборок значений из собственной памяти, с замером времени выполнения, и статистически достоверно «поймать» тот момент, когда выборка будет быстрее – и тем самым понять, какое значение осталось в кеше TLB. В целом, это комбинация достаточно известных подходов «исследование состояния аппаратуры методом временного прозвона» и «получение данных, осевших в кэше» примененный конкретно к архитектуре процессоров. Атаки Spectre (их две, первого и второго рода) еще более хитрые, там происходит обход проверки границ (*bounds check bypass*) и манипуляция целевым кешем адресов ветвлений (*branch target injection*). За подробными деталями этих атак отошлем к [8, 9], а также десяткам популярных статей и презентаций, которых легко найти по «брендам этих уязвимостей», заметим, что многим специалистам по ИБ эти уязвимости были известна, и скорее всего, они активно эксплуатировалась и раньше.

Эти уязвимости безусловно были опасны во многих сценариях и вовсе не связанных с виртуализацией и контейнерами, везде, куда каким-нибудь образом, можно было

доставить атакующий код. Так, например, Javascript мог атаковать память браузера и вытащить логины-пароли. Но если с Javascript в браузере удалось быстро разобраться – загрубление таймеров доступных для JS-кода, модификация JIT компилятора с использованием хитрых программных конструкций *retpoline*, сохраняющих большую часть выгод упреждающих вычислений, но сбивающих выполнение этих атак, то с областью виртуализации, особенно со стороны облачных провайдеров, предоставляющих публичные услуги, или крупных частных корпоративных центров данных с виртуальными контурами доступа, и автоматической миграцией виртуальных контейнеров между физическими серверами, все хуже, причем именно для контейнерной виртуализации. С помощью Meltdown и Spectre пока не удалось преодолеть барьер между гостевой системой и гипервизором при физической виртуализации, а вот попасть из контейнера в адресное пространство ядра – да, удалось.

Да, производители оборудования, операционных систем, и систем виртуализации уже выпустили множество патчей на всех уровнях:

1. обновление микрокода для процессоров (особенно Intel и AMD);
2. обновление ядра ОС Linux и Windows – патчи KPTI (*Kernel Page Table Isolation,* изоляция таблицы страниц ядра), KAISER (*Kernel Address Isolation to have Side-channels Efficiently Removed* – изоляция пространства адресов ядра во избежание атак по стороннему каналу);
3. обновление компиляторов, использующихся для сборки критических бинарников – ядра ОС, гипервизоров, для реализации в них конструкций *retpoline* (новорожденное слово из «return» и «trampoline») [10]

Но, во-первых, каждое из таких исправлений может уронить производительность на 10-20%, что сильно снижает эффективность контейнерной виртуализации по сравнению с аппаратной. Во-вторых, специалисты признают, что все это полумеры, рассчитанные на затруднение эксплуатации уже реализованных по этим уязвимостям эксплоитов, и надежно избавится от этих проблем можно будет только сменив оборудование, или хотя бы процессоры на самые современные. Основные выводы тут такие – сейчас, до смены поколения серверов, лучше избегать контейнерной виртуализации при предоставлении публичных услуг хостинга, когда недоверенный атакующий код злоумышленника может внезапно оказаться среди контейнеров других клиентов, если этого нельзя избежать – то обязательно поставить все необходимые патчи на всех уровнях, проверив нагрузочными тестами, что это не вызовет критического падения производительности в тех задачах, для которых применяются контейнеры.

5. Кеширование при сборке образов контейнеров

В отличие от контейнеров LXC и OpenVZ, контейнеры Docker обрели огромную популярность среди разработчиков, именно благодаря эффективной реализации «шаблонов», «образов» контейнеров, которые, благодаря «слоистой структуре» overlayfs и иерархическим ссылкам обеспечивают дедупликацию данных, и экономию диска и памяти. С другой стороны – инфраструктура так называемых докер-реестров, публичных и приватных, решает проблему эффективного совместного использования этих образов и доставку и выкатку их как на тестовые среды, так и на «боевые» системы. В результате докер-образы стали практически самым универсальным и популярным методом упаковки приложения, вытесняя линукс-пакеты, которых надо было раньше мучительно собирать под каждый целевой дистрибутив, и разные другие инсталляционные форматы – ведь теперь, приложение в виде docker-контейнера можно запустить не только на Linux, но даже и на Windows-системах. В результате расцвели практики непрерывной интеграции, с сборкой и тестированием приложения после каждого минимального изменения.

Но тут же возникли проблемы максимального ускорения этого процесса – если раньше разработчики выпадали из «потока разработки», оправдываясь, что оно «компилируется», то сейчас, простую «компиляцию» заменила сборка новых докер-образов (установка зависимостей, размещение артефактов, быстрое автоматическое тестирование). Даже если простая сборка образа в чистом окружении системы сборки будет занимать, скажем, минут 10, то при росте команды и числа коммитов в день ожидание готового результата будет расти нелинейно, даже в пуассоновской модели теории массового обслуживания [11] (рис. 9).

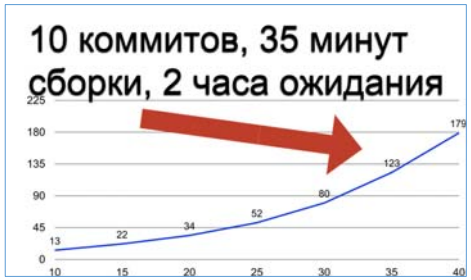

Рис.9: Нелинейный рост времени ожидания сборки при росте числа коммитов
Fig. 9: Non-linear increase in build waiting time with an increase in the number of commits

Решения есть – нужно обеспечить максимальную атомарность всех изменений в докер-образах, чтобы каждое изменение изменяло только один, небольшой «слой» докер-образа, ссылаясь на уже неизменные существующие в докер-реестре образы. Это можно сделать на «человеческом уровне», обязав разработчиков и докер инженеров, максимально атомизировать собираемые докер-образы при сборке, путем специальных соглашений в формировании Dockerfile (см. [11]). Но как любое решение, зависящее от людей («стандарты кодирования», «архитектурные политики» и т.п.), это трудно соблюдать, и в индустрии есть желание это автоматизировать.

Так, появляются проекты, такие как Kaniko Cache[3], которые автоматически кэшируют в докер-реестре все промежуточные слои, вне зависимости от того, как разработчики оформляли докер-файлы. Да, это приводит к дополнительным затратам дискового пространства, но существенно ускоряет скорость сборки.

6. Исследования оптимального размещения мультиконтейнерных конфигураций на физических серверах

Кроме рассмотренных ранее сценариев использования, где требовалось экономить время запуска или сборки, часто возникают и альтернативные требования – когда скорость запуска не важна, а требуется максимальная энергоэффективность оборудования. Опуская возможности инженерной оптимизации датацентров – эффективные технологии охлаждения и электропитания, на программном уровне, для максимальной энергоэффективности требуется обеспечить максимальную загрузку имеющихся физических серверов, чтобы исключить использование в холостую. Еще недавно [12] ситуация была совершенно неудовлетворительной; условно говоря, средний сервер был загружен всего лишь на 10%, при этом его совокупное удельное энергопотребление падало не больше чем в два раза, от такового при максимальной нагрузке.

Сейчас, с ростом популярности виртуализации и глобализации вычислительных ресурсов в ЦОД крупных провайдеров, таких как Google и Amazon, ситуация значительно

[3] https://cloud.google.com/cloud-build/docs/kaniko-cache

улучшилась. Впрочем, крупные провайдеры обычно не ставят задачу 100% утилизации каждого сервера, оставляя резерв для низколатентного маневрирования мощностью под внезапные заказы дополнительной мощности от клиентов, но эта задача актуальна для небольших и средних специализированных ЦОД, где задержки адаптации к меняющейся нагрузке не так важны, а важна именно максимальная загрузка. Как пример, можно привести различные системы обработки пакетных заданий – научных расчетов, сборок сложного ПО (линукс-дистрибутивов), рендер-фермы и т.п. Для таких задач еще нет реализованного в ПО виртуализации оптимальных решений, но идут исследования ученых, как из университетов, так и крупных компаний (IBM, Redhat, VMWare), предлагающих различные алгоритмы эффективного планирования.

6.1 Консолидация виртуальных машин на основе решения декомпозированной многоцелевой задачи об упаковке

В статье [13] специалистов IBM рассмотрена задача размещения виртуальных машин (ВМ) на физических серверах, известная как *Virtual Machine Consolidation* или *Virtual Machine Packing Problem*. Решение использует тот факт, что обычно провайдеры виртуальных машин предоставляют виртуальные машины с конечным множеством наборов параметров. Задача сведена к 2-шаговому варианту многомерной задачи упаковки в контейнеры *multi-dimensional bin packing problem*. На первом шаге виртуальные машины объединяются в кластеры методом k-средних. ВМ из одного кластера изменяются так, что все становятся одинаковыми. Для полученных ВМ проводится метод динамического программирования. На втором шаге из полученных наборов виртуальных машин строится их расположение на физических серверах. Существенно уменьшено время работы алгоритма по сравнению с ранее известными алгоритмами, такими как одномерный First Fit и многомерный Bin Packing.

Метод First Fit для одномерной задачи максимально прост: «Очередной объект упаковывается в последний созданный контейнер, в который он помещается. Если объект не помещается ни в один контейнер, то для его упаковки создаётся новый, куда он и размещается». Зато он может работать в режиме онлайн, когда объекты на вход поступают последовательно в некотором неизвестном порядке и при упаковке очередного объекта не известны размеры следующих за ним по порядку объектов. В этом случае алгоритм гарантирует, что асимптотически отношение суммы размеров созданных контейнеров к сумме размеров объектов будет не более, чем *17/10*, как показано в [14]. Однако оффлайновый алгоритм First Fit, на вход которому подаются объекты с размерами, упорядоченными по убыванию, как показано в [15], гарантирует уменьшение асимптотики данной величины до *11/9*. Данный алгоритм известен как *First Fit Decreasing* и в [13] применяется его обобщение на многомерный случай – «Иерархический многомерный алгоритм Bin Packing для виртуальных машин».

Задача размещения виртуальных машин сводится к многомерной задаче упаковке в контейнеры, где измерения соответствуют ресурсам различных типов, которые необходимы для создания ВМ, а именно: CPU, RAM, IOPS.

Задача ставится следующим образом: даны параметры серверов: $C = \{c_1, c_2, c_3, \dots, c_n\}$ такие, что $\forall c_i \in C, c_i \geq 0$ для каждого сервера C. Каждый параметр соответствует количеству ресурса определённого типа на физическом сервере. Также известны параметры ВМ для каждой машины $V: V = \{v_1, v_2, v_3, \dots, v_n\}$.

Требуется расположить ВМ на физических серверах так, чтобы каждая ВМ находилась на сервере и сумма каждого из параметров по ВМ, находящихся на сервере, не превосходила значение этого параметра по серверу. При размещении требуется минимизировать число используемых физических серверов.

Задача в данной постановке сводится к многомерной задаче упаковки объектов в контейнеры *Multi-dimensional Bin Packing*, где V, или ВМ – упаковываемые объекты, а C, или серверы – контейнеры. Поэтому задача далее рассматривается в терминах многомерной задачи упаковки в контейнеры.

Рассмотрим построение шаблонов для оптимального решения.

Фиксируется натуральное число K. Применяется метод k-средних кластеризации к набору объектов или параметров ВМ, в результате применения которого ВМ разобьются на K кластеров, и в каждом кластере будут ВМ с близкими параметрами.

Объекты из каждого кластера преобразуются следующим образом: пусть в кластере Cl находятся объекты $V_1, \dots, V_{last}$ с параметрами $v_1^1, \dots, v_n^1, \dots, v_1^{last}, \dots, v_n^{last}$. Тогда после преобразования, параметр с номером i для каждой новой ВМ будет равен $v_i^{k\ new} = v_i^{new} = \max\limits_{1 \le j \le last} v_i^j$, и все объекты из одного кластера будут одинаковыми.

Далее для этих изменённых объектов методом динамического программирования строятся все возможные наборы объектов, такие, что число объектов из каждого кластера в наборе не превышает общего числа объектов, принадлежащего данному кластеру и такие, что для каждого параметра сумма данных параметров в наборе не превосходит количества параметра данного типа в самом вместительном по параметру данного типа контейнере.

Считая любые 2 объекта из одного кластера идентичными, данные наборы можно построить за $O(n^k)$ операций, где K – число кластеров, а n – число ВМ, которые нужно упаковать.

Аллокация ВМ производится алгоритмом, похожим на First Fit Decreasing. Сначала наборы ВМ упорядочиваются с помощью метрики из оценки МакКарти (short-circuit evaluation), а именно, если у наборов Set_1 и Set_2 сумма параметров по всем ВМ равна $\{c_1^{set1}, \dots, c_n^{set1}\}$ и $\{c_1^{set2}, \dots, c_n^{set2}\}$ соответственно, то $Set_1 > Set_2$ тогда и только тогда, когда $\exists i \in \{1, \dots, n\}$: если $j < i$, то $c_j^{set1} = c_j^{set1}$, а $c_i^{set1} > c_i^{set1}$. При оценке по данной метрике сначала сравниваются наборы по сумме первых параметров всех элементов. Если для одного из наборов данная величина больше, чем для другого набора, то данный набор больше другого. Если же суммы первых параметров равны, рассматриваются вторые параметры и т.д.

Алгоритм аллокации ВМ работает следующим образом.

Выбирается самый большой по метрике МакКарти набор ВМ из оставшихся наборов и сервер с самой большой оценкой оставшихся параметров. Если набор ВМ можно расположить на сервере так, что сумма каждого параметра не превышается, а ВМ из каждого кластера осталось достаточно, чтобы набрать набор, то выбираются ВМ наибольшего размера из оставшихся из каждого кластера и на сервере размещаются данные ВМ. Если этого нельзя сделать, то набор удаляется из рассмотрения.

6.2 Консолидация виртуальных машин с учетом производительности

В работе [16] был предложен алгоритм «PACMan» для задачи размещения виртуальных машин. В отличии от [13], алгоритм строился с учётом того, что в результате уплотнения ВМ происходит размещение нескольких ВМ на одном сервере. Как следствие, происходит конфликт за использование разделяемых ресурсов, таких как CPU или RAM, в результате чего происходит ухудшение качества и скорости работы ВМ. Алгоритм позволяет сохранить ухудшение качества в заранее указанных пределах, при этом добиваясь производительности, близкой к лучшей возможной.

Предполагается возможной оценка числа раз, в которое ухудшается работа каждой ВМ относительно изолированного размещения на отдельном сервере, при размещении набора ВМ на одном сервере для каждой ВМ каждого набора, исполняемого на том сервере.

Для размещаемого набора S ВМ вводится мера удельной стоимости совместного размещения

$$V(S) = \frac{w(S)}{|S|},$$

где за $w(S)$ обозначено использование ресурсов каждой ВМ из набора. Качество работы алгоритма будем оценивать суммой затрат $E(ALG)$, равной сумме стоимостей всех размещенных наборов ВМ:

$$E(ALG) = \sum_{S \in phisical\ server} |S|\ V(S).$$

Все наборы ВМ с допустимым ухудшением работы каждой размещаются в порядке неубывания $V(S)$. Обозначим за $\mathfrak{F}$ множество всех этих наборов.

Алгоритм размещения ВМ работает следующим образом:

1. выбираем набор S из $\mathfrak{F}$ с наименьшей стоимостью $V(S)$;
2. если все ВМ из него можно разместить на сервере, то делаем это;
3. если это сделать нельзя, удаляем набор S из множества наборов.

Пусть на сервере нельзя разместить больше k ВМ. Была доказана следующая:

Теорема. Для всех входных данных, алгоритм имеет сумму затрат $E(ALG) = O(\ln k \cdot E(OPT))$,

где $E(OPT))$ – сумма затрат оптимального алгоритма размещения ВМ. Однако так как максимальное число k ВМ, которые можно разместить на сервере может быть велико, то и расчёт ухудшения для каждого набора ВМ занимает $O(k \cdot n^k \cdot \ln n)$ операций, где n – число размещаемых ВМ и может быть вычислительно сложным. Однако, если, аналогично работе Dow, разбить множества ВМ на l кластеров методом k-средних, то, возможно, за счет уменьшения $E(ALG)$, вычислительная сложность алгоритма может быть уменьшена до $O(l \cdot n^l \cdot \ln n)$ для любого $l \in \mathbb{N}$.

6.3 Оптимальные онлайновые детерминированные алгоритмы и эвристики для динамической консолидации виртуальных машин

В работе [17] был предложен динамический алгоритм уплотнения расположения виртуальных машин в дата-центрах. Задача размещения ВМ представлена в виде задачи *bin packing* упаковки в контейнеры с разными значениями размеров и стоимости контейнеров. Размеры контейнеров здесь – число доступных ЦП узлов дата-центра, а стоимости соответствуют потреблению энергии узла.

Был предложен алгоритм, являющийся обобщением эвристики *Best Fit Decreasing Height* для задачи *bin packing*. Данная эвристика упаковывает объекты, расположенные в порядке убывания их веса в тот контейнер, при упаковке в который остаётся наименьшее число незаполненного пространства. Как показано в [18], данный алгоритм использует не более, чем 11/9·OPT+1 контейнеров, где OPT – это число решений, используемых оптимальным решением задачи.

Алгоритм *Power Aware Best Fit Decreasing* для размещения ВМ работает следующим образом: Сначала все ВМ размещаются по убыванию числа используемых ЦП, затем очередная ВМ размещается в хост, на котором из-за добавления ВМ происходит наименьшее среди всех хостов увеличение энергопотребления.

6.4 Консолидация контейнеров в облачных центрах данных с обеспечением эффективного энергопотребления

В работе [19] предложен фреймворк, уплотняющий расположение контейнеров на виртуальных машинах и сокращающий потребление энергии на хосте.

В последнее время наряду с такими облачным сервисами, как IaaS (инфрастуктура как сервис), PaaS (платформа как сервис) и SaaS (программное обеспечение как сервис), всё шире и шире входит в употребление новый вид сервиса – CaaS, или контейнеры как сервис. Благодаря ему получается уменьшить время запуска системы.

Слой CaaS находится между слоем IaaS, предоставляющим виртуальные вычислительные ресурсы и слоем PaaS, предоставляет приложениям среду выполнения и соединяет эти два слоя вместе.

Наиболее широко используемым инструментом контейнеризации на сегодняшний день является ПО Docker. Однако в [20] показано, что запуск контейнеров Docker-а на ВМ позволяет достичь близкой, а, в некоторых случаях, даже лучшей производительности, чем запуск в “родной” среде, благодаря изолированности ВМ. Подход разворачивания контейнеров внутри ВМ был рассмотрен в работе.

Серверы, элементы сети и системы охлаждения являются основными потребителями энергии в современном дата-центре. В работе с помощью алгоритмов миграции контейнеров уменьшается количество работающих серверов, благодаря чему уменьшается суммарное по серверам потребление энергии.

Однако, уменьшая потребление энергии, следует удовлетворять SLA-соглашениям (Service Level Agreement). В случае развёртывания контейнеров на ВМ, SLA-соглашения нарушаются тогда и только тогда, когда ВМ, на которой запущен контейнер не имеет ЦП, необходимых ему для работы. Таким образом, за SLA сумму по всем ВМ, не получивших требующегося числа ядер, отношений разности чисел требуемых и полученных чисел ЦП к требуемому числу ЦП, или

$$SLA = \sum_{i=1}^{N_s} \sum_{j=1}^{N_{vm}} \sum_{p=1}^{N_v} \frac{CPU_r(vm_{j,i}, t_p) - CPU_a(vm_{j,i}, t_p)}{CPU_r(vm_{j,i}, t_p)}.$$

Здесь N_s – число серверов, N_{vm} – число ВМ, N_v – число нарушений SLA-соглашений, $vm_{j,i}$ – ВМ j на сервере i , t_p – время, в которое произошло нарушение SLA-соглашений с номером p, $CPU_r(vm_{j,i}, t_p)$ – число ЦП, требующихся ВМ j для развертывания контейнеров, расположенной на сервере i в момент времени t_p, и $CPU_a(vm_{j,i}, t_p)$ – число ЦП, выделенных ВМ j в момент времени t_p.

Алгоритмы, отвечающие за миграцию контейнеров, работают в три этапа:

1. определение ситуаций, при которых контейнеры должны мигрировать;
2. определение, какие контейнеры мигрируют;
3. определение, куда контейнеры мигрируют.

Данные этапы в фреймворке реализованы следующим образом.

1. Для каждого хоста статически заданы пороги недогрузки UL (*under-load*) и перегрузки OL (*over-load*), определяемые специальными алгоритмами. Если хост перегружен или недогружен, происходит миграция контейнеров
2. Рассмотрены алгоритмы Mcor, выбирающий контейнеры с наиболее коррелированной нагрузкой с сервером, на котором контейнеры развёрнуты, и алгоритм MU, выбирающий контейнеры, которые используют больше всего ЦП.
3. Были рассмотрен три алгоритма для выбора хоста, работающие по принципам алгоритмов для классической задачи *Bin Packing*: *First Fit Host Select*, выбирающий последний созданный подходящий хост для размещения контейнера; *Random HS*, выбирающий случайный подходящий хост и *Least Full HS*, выбирающий хост с наименьшим числом использованных ЦП из имеющихся.

При увеличении порога UL и уменьшении порога OL, число миграций контейнеров увеличивается, и, следовательно, создаётся большее число ВМ и увеличивается использование ресурсов. С другой стороны, при увеличении OL и уменьшении UL, риск нарушения SLA-соглашений увеличивается. Оптимальные значения OL и UL для алгоритма Mcor выбора мигрирующих контейнеров и алгоритма First Fit HS выбора хоста назначения миграции равны OL=0.8, UL=0.7 соответственно. По сравнению с другими методами выбора мигрирующего контейнера и назначения миграции и значениями OL, UL, достигается экономия ресурсов на 7.4% при средних нарушениях SLA меньше 5%

7. Заключение

В данной работе рассмотрено множество аспектов технологий кэширования в мультиконтейнерных системах облачных вычислений.

Рассмотрены практические технологии, позволяющие улучшить производительность мультиконтейнерных систем за счет различного кэширования — ускорение скорости старта за счет кэширования образов и состояний контейнеров, улучшение экономичности за счет разделения файлов, кэширование слоев образов при сборке и т.п. При этом рассмотрены и проблемы безопасности, которые приносят в контейнерную виртуализацию технологии аппаратного кэширования данных в процессорах.

Отдельно рассмотрены теоретические подходы улучшения производительности мультиконтейнерных систем виртуализации за счет различных алгоритмов оптимального планирования.

Список литературы / References

[1]. Latency numbers every programmer should know. URL https://gist.github.com/hellerbarde/2843375, accessed 15.11.2019.

[2]. Увеличение плотности контейнеров на ноде с помощью технологии PFCACHE. Блог компании Rusonyx / Increase container density on a node using PFCACHE technology. Rusonyx company blog. URL https://habr.com/ru/company/rusonyx/blog/444696/, accessed 15.11.2019.

[3]. Virtuozzo User's Guide. URL https://docs.virtuozzo.com/pdf/virtuozzo_7_users_guide.pdf, accessed 15.12.2019.

[4]. Павел Емельянов. CRIU – как маленький open-source проект меняет жизнь большой компании. Материалы 11-й международная конференция Linux Vacation/Eastern Europe, 2015, http://0x1.tv/20150627H / Pavel Emelyanov. CRIU – how a small open-source project changes the life of a large company. In Proc. of the 11th International Linux Vacation / Eastern Europe Conference, 2015, http://0x1.tv/20150627H (in Russian).

[5]. Николай Ефанов. Единая теория восстановления деревьев процессов Linux – огибая подводные камни checkpoint-restore. Материалы 15-й Центрально-восточноевропейской конференция в России по разработке программного обеспечения, 2019, http://0x1.tv/20191115BN / Nikolay Efanov. The unified theory of Linux process tree restoration - enveloping the pitfalls of checkpoint-restore. In Proc. of the 15th Central & Eastern European Software Engineering Conference in Russia, 2019, http://0x1.tv/20191115BN (in Russian).

[6]. Mike Rapoport, Adrian Reber. To Kill or to Checkpoint – That is the Question. In Proc. of the Open Source Summit + Embedded Linux Conference & OpenIoT Summit Europe, 2018. https://static.sched.com/hosted_files/osseu18/2a/kill-or-checkpoint.pdf.

[7]. Grushin D.A., Kuzyurin N.N. On Effective Scheduling in Computing Clusters. Programming and Computer Software, vol. 45, issue 7, 2019, pp. 398–404. doi:10.1134/S0361768819070077.

[8]. Moritz Lipp, Michael Schwarz et al. Meltdown: Reading Kernel Memory from User Space. In Proc. of the 27th USENIX Security Symposium, 2018, pp. 973-990.

[9]. Paul Kocher, Jann Horn et al. Spectre Attacks: Exploiting Speculative Execution. In Proc. of the IEEE Symposium on Security and Privacy, 2019, pp. 1-19.
[10]. Paul Turner. Retpoline: a software construct for preventing branch-target-injection. URL https://support.google.com/faqs/answer/7625886, accessed 15.12.2019.
[11]. Николай Пасынков. Слои Docker для ускорения сборки проекта. Материалы 15-й Центрально-восточноевропейской конференция в России по разработке программного обеспечения, 2019, http://0x1.tv/20191114DJ / Nikolay Pasynkov. Docker layers to speed up project builds. In Proc. of the 15th Central & Eastern European Software Engineering Conference in Russia, 2019, http://0x1.tv/20191114DJ (in Russian).
[12]. Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431. U.S. Department of Energy, Office of Scientific and Technical Information. URL https://www.osti.gov/servlets/purl/929723, accessed 15.12.2019.
[13]. Dow Eli M. Decomposed Multi-Objective Bin-Packing for Virtual Machine Consolidation. PeerJ Computer Science, vol. 2, 2016, article e47.
[14]. Xia Binzhou, Zhiyi Tan. Tighter Bounds of the First Fit Algorithm for the Bin-Packing Problem. Discrete Applied Mathematics, vol. 158, issue 15, 2010, pp. 1668-1675.
[15]. György Dósa. The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is $FFD(I) \leq 11/9OPT(I) + 6/9$. Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. Lecture Notes in Computer Science, vol. 4614, 2007, pp. 1-11/
[16]. Roytman Alan, Aman Kansal et al. PACMan: Performance aware virtual machine consolidation. In Proc. of the 10th International Conference on Autonomic Computing, 2013, pp. 83-94.
[17]. Beloglazov Anton, and Rajkumar Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. Concurrency and Computation: Practice and Experience, vol. 24, issue 3, 2012, pp. 1393-1550.
[18]. A Simple Proof of the Inequality FFD (L) $\leq$ 11/9 OPT (L) 1, $\forall$L for the FFD Bin-Packing Algorithm. Acta Mathematicae Applicatae Sinica, vol. 7, issue 4, 1991, pp, 321–331.
[19]. Piraghaj, Sareh Fotuhi, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers. In Proc. of the IEEE International Conference on Data Science and Data Intensive Systems, 2015, pp. 368 - 375.
[20]. Ali Qasim. Scaling web 2.0 applications using docker containers on vsphere 6.0. VMware VROOM! Blog. URL https://blogs.vmware.com/performance/2015/04/scaling-web-2-0-applications-using-docker-containers-vsphere-6-0.html, accessed 15.11.2019.

Информация об авторах / Information about authors

Денис Олегович ЛАЗАРЕВ – стажер-исследователь ИСП РАН, аспирант МФТИ. Научные интересы: задачи упаковки в полосы и контейнеры, случайные графы и дискретная оптимизация.

Denis Olegovich LAZAREV – a research trainee at ISP RAS and a postgraduate student of MIPT. Research interests: bin and strip packing problems, random graphs and discrete optimisation.

Дмитрий Андреевич ГРУШИН является научным сотрудником ИСП РАН. Его научные интересы включают оптимизацию размещения задач в распределенных вычислительных системах, виртуализацию, контейнеризацию, облачные технологии.

Dmitry Andreevich GRUSHIN is a researcher at ISP RAS. His research interests include scheduling optimization in distributed computing systems, virtualization and containerization.

Станислав Александрович ФОМИН – программист ИСП РАН, преподаватель МФТИ. Научные интересы: теория сложности, дискретная оптимизация.

Stanislav Aleksandrovich FOMIN – ISPRAS programmer, lecturer at MIPT. Research interests: theory of complexity and discrete optimisation.