Содержание

1. Введение

В настоящее время живая миграция – это технология, широко использующаяся при администрировании распределенных систем, дата-центров и кластеров. По своей сути живая миграция представляет собой перенос виртуальной машины (ВМ) с одного физического сервера на другой без прекращения работы виртуальной машины и остановки работы сервисов. Ценность этой технологии состоит в том, что с её помощью можно решать множество задач: балансировка нагрузки между серверами, подготовка к запланированному ремонту или установке ПО, управление энергопотреблением и т.д. На данный момент живая миграция поддерживается в таких известных продуктах, как Xen, Hyper-V Server, OpenVZ, VMware ESX, KVM и многих других. В каждом продукте разработчики реализовали разные алгоритмы и оптимизации, которые будут описаны ниже. В связи с разнообразием существующих методов встает вопрос о сравнении различных реализаций живой миграции.

Существует три основных параметра, по которым можно судить об эффективности реализации живой миграции:

1. Общее время миграции виртуальной машины с одного сервера на другой (total migration time);
2. Время, которое виртуальная машина недоступна во время миграции (downtime);
3. Количество данных, переданных по сети для осуществления миграции (network traffic).

Кроме этого существует несколько побочных метрик, таких, как доступность ВМ во время миграции; влияние миграции на производительность приложений, исполняющихся как внутри самой ВМ, так и на хосте; влияние миграции на нагрузку сети и другие.

В данной работе будет проведен анализ существующих алгоритмов живой миграции и предложена оптимизация алгоритма живой миграции в QEMU.

2. Алгоритмы живой миграции

Выделяют три основных алгоритма живой миграции:

1. Pre-copy
2. Post-copy
3. Trace and replay

Первый метод (Рис. 1) является основным и наиболее распространенным. Он заключается в том, что без остановки работы ВМ используемая ею память итерационно пересылается на целевой сервер. При этом в первой итерации пересылается вся память, а в последующих пересылаются только те страницы памяти, которые были изменены во время предыдущей итерации (эта стадия и называется pre-copy). Данный процесс продолжается до тех пор, пока количество страниц, оставшихся для пересылки, не достигнет определенного значения, либо пока количество страниц, изменяемых за одну итерацию, не перестанет сокращаться. После этого ВМ останавливается, на целевой сервер переносятся оставшиеся страницы и состояние ВМ (стадия stop-and-copy). По окончанию этой стадии работа ВМ возобновляется на целевом сервере.

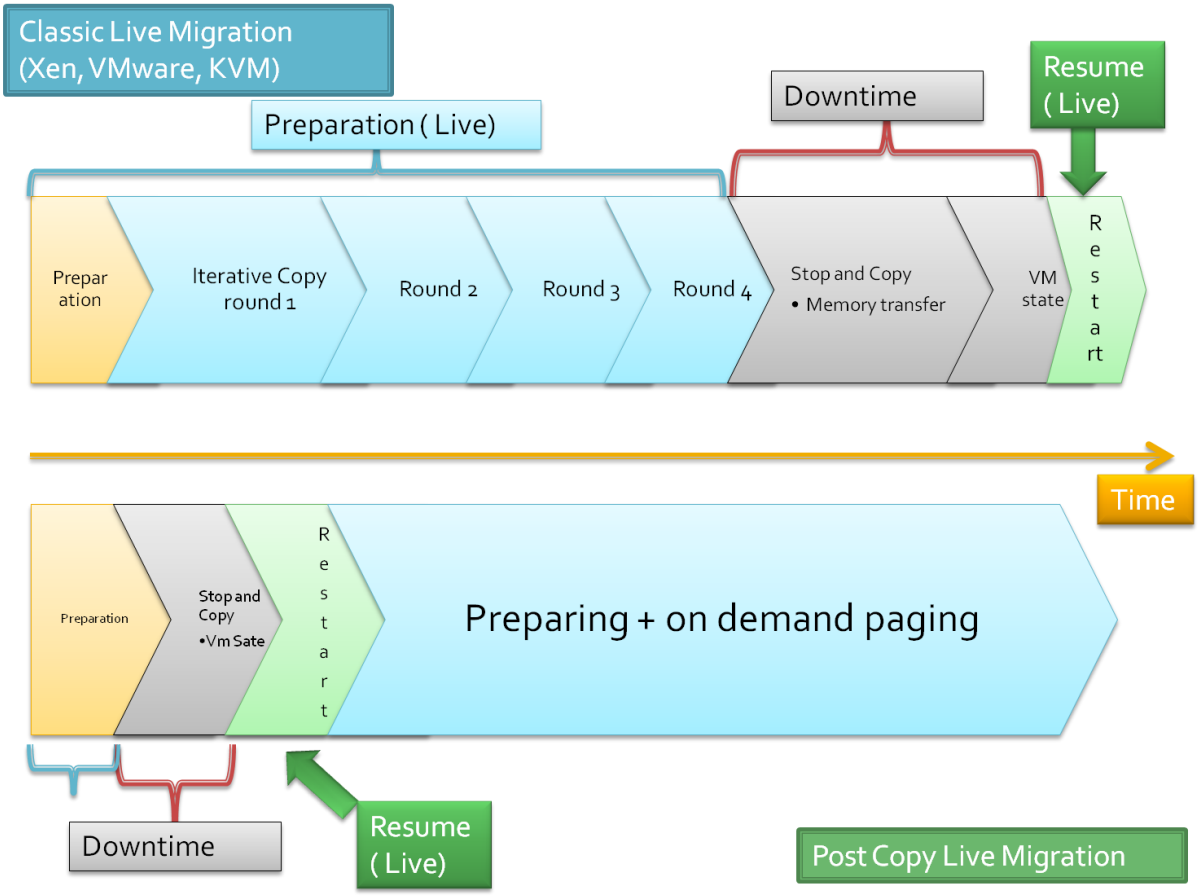


Рис. 1. Общая схема pre-copy и post-copy миграций.

Второй метод (Рис. 1) является диаметральной противоположностью первого. Сначала останавливается ВМ, затем на целевой сервер передается текущее состояние ВМ и минимальный набор памяти, необходимый для работы ВМ. После чего ВМ возобновляет свою работу на целевом сервере. Оставшиеся на хосте страницы памяти пересылаются на целевой сервер в определенном порядке.

Третий подход (Рис.2) заключается в следующем: с момента запуска ВМ на хосте в лог-файл записываются все изменения, которые происходят с ВМ. Во время миграции на целевой сервер передается не вся память, используемая ВМ, а только лог-файл и последующие в нем изменения. На целевом сервере запустим новую ВМ и эмулируем все изменения, которые происходили с ВМ на хосте, согласно записям в лог-файле.

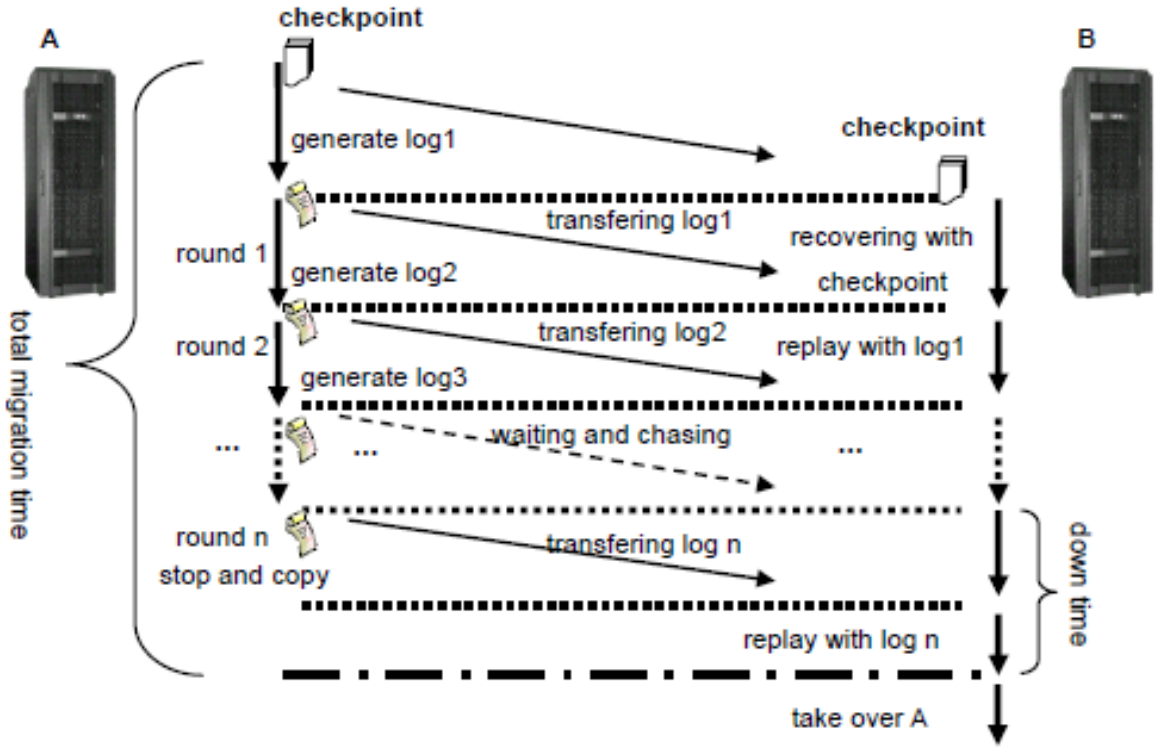


Рис. 2. Общая схема trace and replay миграции

Каждый из этих подходов обладает своими достоинствами и недостатками. Далее алгоритмы описаны более подробно.

2.1 Pre-copy

Данный алгоритм является самым используемым. Поэтому, он описан во многих статьях [1,2,10,11].

Основной проблемой, с которой пришлось столкнуться разработчикам при реализации данного алгоритма, оказалась ситуация, в которой частота внесения изменений в память выше, чем частота передачи страниц памяти. Это наблюдается в интенсивных программах (High Performance Computing). Решением данной проблемы может служить алгоритм, который анализирует скорость внесения изменений и скорость передачи данных. Например, в [1] описан алгоритм, согласно которому задается начальное, минимальное и максимальное значение пропускной способности сети, которое может быть полностью использовано процессом

миграции. В течение каждой итерации подсчитывается количество страниц, которые были переданы, и количество страниц, в которые были внесены изменения. В зависимости от отношения этих величин начальное значение пропускной способности может инкрементироваться на определенную величину. Если во время какой-либо итерации пропускная способность, отведенная процессу миграции, достигла максимального значения, и при этом количество пересылаемых страниц по-прежнему меньше количества изменяемых, то исполнение ВМ приостанавливается и начинается stop-and-copy стадия. Альтернативное решение рассмотрено в [9]. В данной статье предлагается не ограничивать сверху пропускную способность сети, которая может быть отведена для процесса миграции, однако при этом следует следить - не возникает ли network contention. Например, можно начинать процесс миграции с максимально допустимой (ограничения задаются лишь каналом передачи) пропускной способности и измерять реальную пропускную способность, которую получил миграционный процесс. Если реально полученная пропускная способность ниже 80% от ожидаемой, то можно снизить выделяемую пропускную способность для миграции.

Так же стоит отметить, что недостатком данного метода является то, что общий процесс миграции занимает довольно продолжительное время. Это объясняется тем, что при практически любом стечении обстоятельств некоторые страницы памяти приходится пересылать несколько раз. Однако, тут есть некоторые оптимизации, которые будут описаны в третьей части данной статьи. Достоинством этого алгоритма является минимальное downtime.

2.2 Post-copy

Главной проблемой этого метода является то, что пока все страницы памяти, используемые ВМ на хосте не будут переданы на целевой сервер, неизбежно будут возникать PF (page fault). Решение этого вопроса описано

в статье [4]. Во-первых, во время подготовки живой миграции можно определить набор страниц, к которым процесс чаще всего обращается. Этот набор называется WWS (Writable Working Set). После того, как исполнение ВМ возобновится на целевой машине, в первую очередь следует передавать станицы из этого набора. Во-вторых, авторы этой статьи предлагают метод эффективной обработки ситуаций, в которых возникает PF на страницах, которые не включены в WWS. Принцип его работы крайне прост: страница, которая была причиной PF, становится центром раздувающегося «пузыря». Это значит, что после того, как произошел PF на целевом хосте, эта страница доставляется туда вне всякой очереди, а вслед за ней будут передаваться те страницы, которые ближе всего расположены к ней на исходной машине. Эффективность данного метода объясняется тем, что крайне часто программы, обращающиеся к какой-то странице в памяти, в скором времени обращаются к ближайшим «соседям» этой страницы.

В целом стоит отметить, что большим плюсом этого алгоритма является то, что каждая страница памяти передается по сети всего один раз, что в свою очередь обеспечивает выигрыш в полном времени миграции по сравнению с pre-copy. В то же время данный алгоритм увеличивает downtime.

2.3 Trace and Replay

Наконец, опишем недостатки и преимущества последнего алгоритма [5]. Определяющие ограничения этого алгоритма: скорость воспроизведения лог-файла на целевом сервере должна быть выше скорости записи в лог-файл на исходном хосте, кроме того скорость передачи лог-файла должна быть выше, чем скорость записи в лог-файл. К счастью, эти проблемы решаются сами собой, т.к. скорость записи в лог-фал составляет порядка MB/sec, в то время как скорость передачи порядка GB/sec. При этом скорость воспроизведения ВМ на целевом хосте можно сделать выше

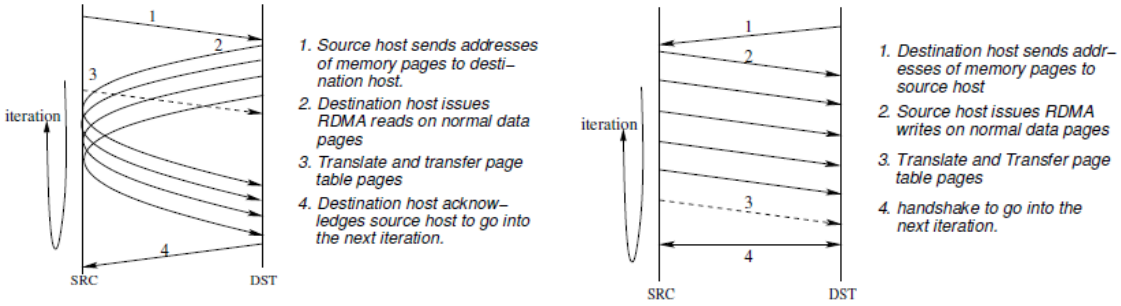
оригинальной скорости за счет отсутствия ожиданий от внешних устройств, пропуска инструкция типа Halt и т.д.

Этот алгоритм выигрывает по сравнению двумя предыдущими, т.к. в течение миграции передается гораздо меньший объем данных. Как следствие, резко сокращается и время миграции и downtime.

3. Существующие оптимизации

3.1 RDMA

Можно ускорить процесс передачи данных с одного сервера на другой. Например, вместо Ethernet соединения использовать более скоростное соединение, поддерживающее технологию RDMA[9], что расшифровывается как Remote Direct Memory Access. С помощью этой технологии можно с целевого хоста иметь доступ к памяти ВМ на исходной машине, без участия в этом ОС ВМ (это экономит время, т.к. не происходит дополнительных переключений контекста) и наоборот. Это осуществляется при помощи функций RDMA Read и RDMA Write (Рис. 3). На основе каждой из этих функций можно построить протокол для реализации одной итерации живой миграции. Например, опишем подобный протокол для RDMA Read: исходный хост передает целевому серверу физические адреса страниц, которые буду переданы в этой итерации. Целевой хост, читает содержимое по этим адресам при помощи RDMA Read, записывает в свой буфер и в это же время принимает от исходной машины page tables, чтобы потом верным образом структурировать полученный страницы. Аналогичным образом можно реализовать протокол, основанный на функции RDMA Write.



(a) Migration over RDMA read

(b) Migration over RDMA write

Рис. 3. Миграция при помощи RDMA

3.2 Хэширование и компрессия

Если использовать pre-copy или post-copy алгоритм, то можно уменьшать объемы переданной информации при помощи алгоритмов хэширования и компрессии[3]. Особенно это будет полезным в том случае, когда с одного хоста необходимо мигрировать сразу несколько ВМ [6], или же с одного хоста мигрировать ВМ одновременно на несколько серверов [8]. Все страницы памяти, используемые этими ВМ, хэшируются. Если у двух страниц совпадают значения хэш-функции на них, то для таких страниц запускается побитовое сравнение. Если страницы идентичны, то вместо того, чтобы пересылать каждую страницу, достаточно переслать одну, и идентификатор второй, который будет включать в себя информацию о том, какой ВМ принадлежит страница, из какой страницы следует восстанавливать текущую и т.д. Компрессия, в свою очередь, очевидным образом уменьшает количество данных для передачи, сокращая при этом общее время миграции. Однако, используя эти методы необходимо помнить о том, что вычисление хэш-функций и компрессия могут вызывать существенный overhead в работе исходного хоста, т.к. являются ресурсозатратными.

3.3 Delta compression

Основная идея данного подхода состоит в том, что во время миграции хранится не весь набор страниц, а страницы и изменения, которые были с ними. Чтобы стало немного понятней, лучше объяснить, как это устроено. Рассмотрим какую-нибудь страницу, в неё вносятся изменения, после чего имеем измененную страницу. Учитывая, что страницы хранятся в памяти в бинарном виде, крайне быстро будет вычислить изменения, которые произошли со страницей. Сделать это можно при помощи битовых операций. Преимущество от хранения страницы и внесенных в неё изменений (которые в памяти хранятся в виде точно такого же размера страницы) в том, что страница с изменениями крайне эффективно компрессируется. Во время миграции создается кэш переданных страниц, если страница содержится в этом кэше, то высчитывается дельта между переданной ранее и данной страницей, дельта компрессируется и передается на целевой сервер. Если же страницы нет в этом кэше, то она добавляется в него и пересылается целиком. Единственная проблема состоит в том, как определять размер кэша. Авторы [12] предлагают определять этот размер опытным путем.

3.4Scheduling

Еще одним способом оптимизации является внесение изменений в планировщик операционной системы на исходном хосте. В статье [7] описывается алгоритм, который понижает приоритет процесса, подлежащего миграции. Условно говоря, каждому процессу добавляются флаги, значения которых увеличиваются на единицу, если количество измененных за данную итерацию страниц больше, чем за прошлую итерацию. Значения этих флагов обрабатываются стандартным планировщиком системы, в результате чего приоритет этого процесса падает. Как следствие, он получает меньше процессорного времени и,

соответственно, делает меньше изменений в страницах. Как итог, живая миграция происходит за меньшее количество итераций.

4. Компрессия

4.1 Общие положения

В данной работе в качестве оптимизации существующего алгоритма миграции в QEMU была выбрана компрессия. Поэтому имеет смысл сказать несколько слов о том, что это такое и как это использовать.

Компрессия или сжатие данных — это алгоритмическое преобразование данных, производимое с целью уменьшения занимаемого ими объёма. Применяется данная технология для более рационального использования устройств хранения и передачи данных. Сжатие основано на устранении избыточности, содержащейся в исходных данных. Простейшим примером избыточности является повторение в тексте фрагментов (например, слов естественного или машинного языка). Подобная избыточность обычно устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины. Другой вид избыточности связан с тем, что некоторые значения в сжимаемых данных встречаются чаще других. Сокращение объёма данных достигается за счёт замены часто встречающихся данных короткими кодовыми словами, а редких — длинными. Сжатие данных, не обладающих свойством избыточности (например, случайный сигнал или белый шум, зашифрованные сообщения), принципиально невозможно без потерь.

В основе любого способа сжатия лежит модель источника данных, или, точнее, модель избыточности. Иными словами, для сжатия данных используются некоторые априорные сведения о том, какого рода данные сжимаются. Не обладая такими сведениями об источнике, невозможно

сделать никаких предположений о преобразовании, которое позволило бы уменьшить объём сообщения. Модель избыточности может быть статической, неизменной для всего сжимаемого сообщения, либо строиться или параметризоваться на этапе сжатия (и восстановления). Методы, позволяющие на основе входных данных изменять модель избыточности информации, называются адаптивными. Неадаптивными являются обычно узкоспециализированные алгоритмы, применяемые для работы с данными, обладающими хорошо определёнными и неизменными характеристиками. Подавляющая часть достаточно универсальных алгоритмов являются в той или иной мере адаптивными.

Все методы сжатия данных делятся на два основных класса:

1. Сжатие без потерь
2. Сжатие с потерями

При использовании сжатия без потерь возможно полное восстановление исходных данных, сжатие с потерями позволяет восстановить данные с искажениями, обычно несущественными с точки зрения дальнейшего использования восстановленных данных. Сжатие без потерь обычно используется для передачи и хранения текстовых данных, компьютерных программ, реже — для сокращения объёма аудио- и видеоданных, цифровых фотографий и т. п., в случаях, когда искажения недопустимы или нежелательны. Сжатие с потерями, обладающее значительно большей, чем сжатие без потерь, эффективностью, обычно применяется для сокращения объёма аудио- и видеоданных и цифровых фотографий в тех случаях, когда такое сокращение является приоритетным, а полное соответствие исходных и восстановленных данных не требуется.

4.2 Используемый алгоритм

В данной работе использовалась модификация одного из алгоритмов Зива-Лемпела – LZRW1. Этот алгоритм является адаптивным, и при этом реализует сжатие без потерь. Опишем его более подробно.

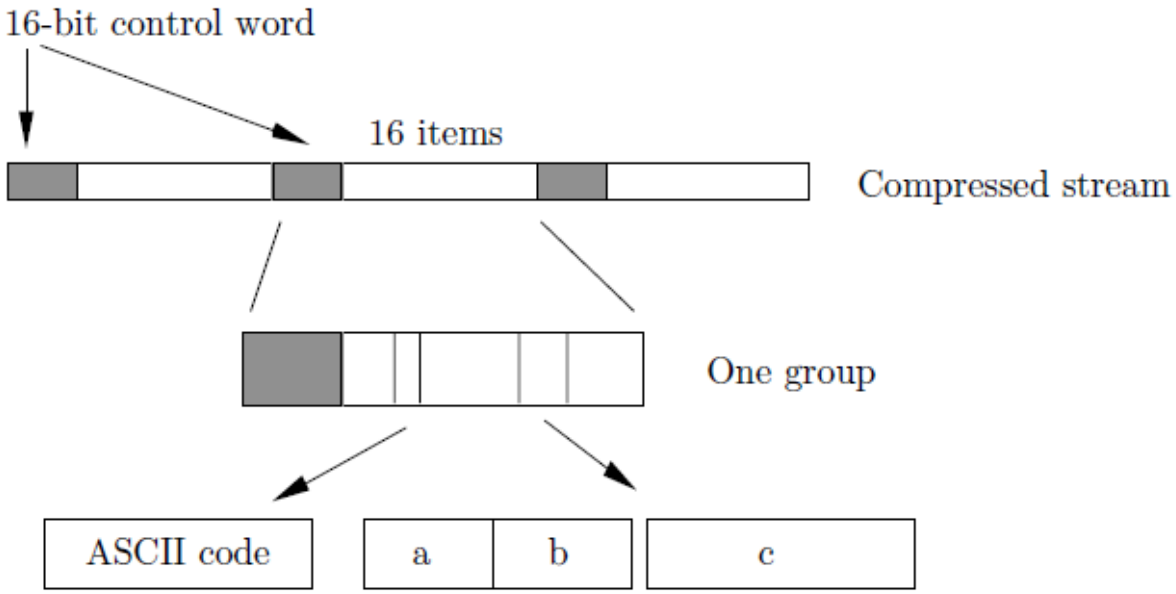


Рис. 4. Общий вид выходного буфера для алгоритма LZRW1.

Начать описание следует с выходного буфера, получаемого в результате работы кодировщика. Следует сразу отметить, что этот буфер записывается в бинарном виде. Состоит же он из «литерал», длиной 8 бит, и «копий» длиной 16 бит. Каждая «копия» - это указание на то, что часть входного буфера определенной длины уже была закодирована. Причем в «копии» хранится как адрес (в виде смещения относительного текущего адреса), по которому находится уже закодированная часть, так и её длина. Элементы выходного буфера получаются следующим образом: литералы – это просто ASCII код какого-либо символа (8 бит достаточно для того, чтобы закодировать 1 из 256 символов), а копия состоит из длины совпадающего участка (4 бита) и смещения (12 бит). Весь закодированный буфер состоит из некоторых блоков, каждый из которых начинается со своего рода 16 битной контрольного слова. Биты в этом слове указывают

на то, является ли элемент из закодированного буфера «литерой» или «копией». Таким образом, декодер в каждой итерации считывает сначала контрольное слово, проходит далее по соответствующему блоку, разделяя его на части по 8 и 16 бит, и каждую часть декодирует очевидным образом. Тем самым исходные данные восстанавливаются в полном объеме.

Теперь можно вернуться к алгоритму самого кодировщика, который изображен на рисунке 5.

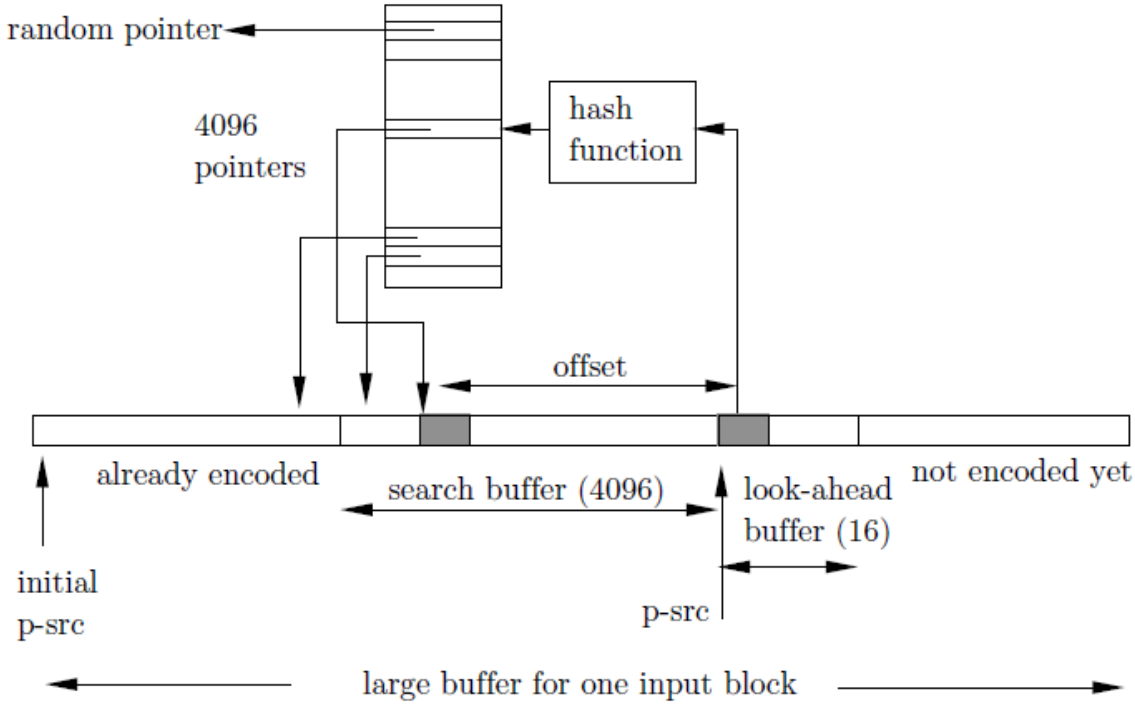


Рис. 5. Алгоритм кодировщика.

Кодировщик непосредственно оперирует с двумя буферами: look-ahead и search, размером 16 байт и 4 килобайта соответственно. В look-ahead буфер считывается часть входного буфера, а search буфер представляет собой уже закодированный текст. Стоит отметить, что для обработки обоих буферов достаточно всего лишь одного указателя, который одновременно является началом look-ahead буфера, и концом search буфера. Обозначим его p_src. Этот указатель проходит по всему входному буферу таким образом, что

элементы с меньшими адресами являются уже закодированными, а с большими адресами еще нет.

Далее опишем как кодируется look-ahead буфер. Три первых символа из этого буфера хэшируются в 12-битное бинарное число, которое интерпретируется как индекс во вспомогательном массиве размера 2^12 = 4096 байт. Изначально этот вспомогательный буфер заполнен как угодно. Каждый элемент массива – это адрес, который мы рассматриваем в качестве адреса в уже закодированной части входного буфера. Таким образом, получая значение хэш-функции от первых трех символов look-ahead буфера, мы получаем указатель. Если этот указатель выходит за границы search буфера, то в выходной поток записывается «литерал», совпадающий с первым символом из трех, а значение p_src увеличивается на 1. Тоже самое происходит, если указатель попадает внутрь search буфера, но по данному указателю находятся символы, не совпадающие с исследуемыми тремя (т.е. не совпадает даже первый символ с символом по данному указателю). Во всех остальных случаях по данному указателю ищется максимальная по вложению строка, совпадающая с подстрокой look-ahead буфера. Длина получившейся строки кодируется в 4ех битное число и вкупе с 12-битным адресом составляет «копию», которая записывается в выходной поток. Значение p_src увеличивается на длину соответствующей строки.

Согласно книге [13] данный алгоритм является одним из самых быстрых, но не самым эффективным в силу того, что не достигает максимального сжатия.

5. Результаты

В данной работе было проведено изучение существующих алгоритмов живой миграции, рассмотрены сильные и слабые стороны каждого из алгоритмов. Так же был сделан обзор возможных оптимизаций, каждая из которых обеспечивает улучшение по одной из трех метрик: общее время миграции, количество переданных по сети данных и время простоя виртуальной машины. Для реализации был выбран алгоритм компрессии из семейства алгоритмов Зива-Лемпела LZRW1. Выбор пал именно на этот алгоритм по следующим причинам: реализация других оптимизаций является на порядок более сложной задачей, чем реализация алгоритма компрессии, кроме того компрессия обеспечивает выигрыш сразу по двум из трех вышеперечисленных метрик. И наконец последняя причина заключается в следующем: данный алгоритм работает быстрее, чем стандартные алгоритмы LZ77 и LZ78, при этом несущественно уступая им в качестве сжатия.

Однако, на стадии экспериментальной проверки, я столкнулся с непреодолимыми сложностями при сборке программы, в результате экспериментальное подтверждение эффективности выбранного метода пока не получено.

В дальнейшей работе планируется получить экспериментальные данные, дающие представление о том, на сколько эффективным оказывается выбранный алгоритм компрессии. В частности, на сколько процентов сокращается время миграции. После того, как подобные оценки будут получены, можно будет заниматься их подробным изучением, и как следствие улучшением рассмотренного алгоритма.

6. Заключение

В заключении хотелось бы еще раз отметить, что живая миграция является крайне полезным и мощным средством для управления распределенными системами и дата-центрами. Более того, у данной технологии есть огромное пространство для дальнейшего развития. Например, до сих пор не решена полностью задача о принятии решения для миграции в тех или иных условиях.

В настоящее время существует несколько способов реализации живой миграции – pre-cope, post-copy и trace and replay. Однако, стоит отметить, что во всех широко использующихся на данный момент продуктах, таких как Xen, OpenVZ, VMware и др., используется pre-copy миграция. При этом только в некоторых из них присутствуют оптимизации: в OpenVZ реализован алгоритм chekpointing (по сути это разновидность 3.3, в которой в определенные моменты времени делаются “срезы” памяти и хранятся изменения между этими “срезами”); в KVM используется алгоритм, контролирующий пропускную способность сети, выделяемую для миграции и др.

Хотелось бы поблагодарить своего научного руководителя Мелехову Анну и преподавателя по сетевым технологиям Подлесных Дмитрия за неоценимую помощь в работе.

7. Список используемой литературы

[1] Clarck, C., Fraser, k., Hand, S., Hansen, J., Jule, E., Limpach, C., Pratti, I., And warfield, A.” Live migration of virtual machines”. In Network System Design and Implementation (2005).

[2] NELSON,M., Lim, and Greg Hutchins ”Fast transparent migration for virtual machines”. In Use nix, Anaheim, Ca(2005)

[3] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan.”Live VM migration with adaptive memory compression”. In Proceedings of the 2009 IEEE International Conference on Cluster Computing (Cluster 2009), 2009

[4] M. R. Hines and K. Gopalan, “Post-copy based live VM migration using adaptive pre-paging and dynamic selfballooning”, in Proceedings of the ACM/Usenix international conference on Virtual execution environments (VEE’09), 2009

[5] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, “Live migration of virtual machine based on full system trace and replay,”in Proceedings of the 18th International Symposium on High Performance Distributed Computing (HPDC’09), 2009

[6] U. Deshpande, X. Wang, and K. Gopalan. «Live gang migration of virtual machines.» In Proceedings of HPDC, pages 135–146, 2011.

[7] Zhaobin Liu, Wenyu Qu, Weijiang Liu, and Keqiu Li. 2010. «Xen Live Migration with Slowdown Scheduling Algorithm.» In Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications andTechnologies (PDCAT '10). IEEE Computer Society, Washington, DC, USA

[8] Lei Cui, Jianxin Li, Bo Li, Jinpeng Huai, Chunming Hu, Tianyu Wo, Hussain Al--Aqrabi, and Lu Liu. 2013. «VMScatter: migrate virtual machines to many hosts.» In Proceedings of the 9th ACM SIGPLAN/SIGOPS international

conference on Virtual execution environments (VEE '13). ACM, New York, NY, USA

[9] W. Huang, Q. Gao, J. Liu, and D. K. Panda. “High Performance Virtual Machine Migration with RDMA over Modern Interconnects.” In Proceedings of IEEE International Conferenceon Cluster Computing (Cluster'07), September 17-20, 2007,Austin, Texas, USA

[10]”Optimized Pre-Copy Live Migration for Memory Intensive Applications” Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, Eric Roman Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, USA

[11]”A Quantitative Study of Virtual Machine Live Migration” Wenjin Hu, Andrew Hicks, Long Zhang, Eli M. Dow, Vinay Soni, Hao Jiang, Ronny Bull, Jeanna N. Matthews Department of Computer Science Clarkson University

[12] Petter Svard, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. 2011. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11). ACM, New York, NY, USA, 111--120.

[13] D. Salomon “Data Compression. The Complete Reference. Fourth Edition”, Computer Science Department, California State University, Northridge, CA 91330-8281, USA