Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)

ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ

КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ
(Специализация 010956 «Математические и информационные технологии»)

МАТЕМАТИЧЕСКОЕ МОДЕЛИРОВАНИЕ ИЗМЕНЕНИЯ РАБОЧЕГО НАБОРА ПРОЦЕССА ДЛЯ ИТЕРАТИВНОЙ ЖИВОЙ МИГРАЦИИ

Магистерская диссертация
студента 873 группы
Тихомирова Павла Олеговича

Научный руководитель
к.ф.-м.н. Емельянов П.В.

Долгопрудный 2014

Оглавление

Введение 3

1 CRIU (Checkpoint/Restore In Userspace) 6
1.1 История . . . 6
1.1.1 Конкурирующие проекты . . . 6
1.1.2 История CRIU . . . 9
1.2 Архитектура . . . 10
1.2.1 Сохранение моментальных снимков . . . 11
1.2.2 Восстановление из снимков . . . 14
1.3 Тестирование . . . 15
1.3.1 ZDTM . . . 15
1.4 Использование . . . 16

2 Дедупликация 17
2.1 Проблема использования оперативной памяти при миграции . . . 17
2.2 Алгоритм . . . 18

3 Миграция 22
3.1 Определения . . . 22
3.2 Существующие решения . . . 23
3.3 мат-модель Миграции . . . 25
3.3.1 Оценка и аппроксимация . . . 25
3.3.2 Модель процесса живой миграции . . . 26
3.3.3 Оптимизация . . . 27
3.3.4 Выбор функции . . . 28

4 Практическая часть 32
4.1 POSIX-Таймеры . . . 32
4.1.1 Использование файловой системы procfs . . . 33
4.1.2 Системные вызовы . . . 34

4.1.3 Проблема overrun 35
4.1.4 Тестирование 35
4.2 Дедупликация 35
4.2.1 Разряженный файл 36
4.2.2 Файловая система tmpfs 36
4.2.3 Архитектура 37
4.2.4 Тестирование 38
4.3 Живая миграция 40
4.3.1 Мониторинг 40
4.3.2 Отслеживание грязной памяти 41
4.3.3 Alglib 42
4.3.4 p.HAUL 42
4.3.5 Тестирование 42

Заключение 45

Введение

Данная работа является одновременно теоретическим и практическим исследованием в Информатике в области Виртуализации ПО, направленным на оптимизацию алгоритма миграции процессов, и внедрения полученного решения в продукты Parallels.

Для абстрагирования архитектуры программ, работающих на вычислительных устройствах, от конкретных принципов их работы придумано понятие Операционной Системы. Данное понятие позволяет избавить программистов от необходимости разрабатывать программный продукт под каждую конфигурацию оборудования в отдельности. ОС предоставляет единое API(Application Programming Interface) для управления ресурсами некоторого класса вычислительных систем. Таким образом, одна программа может работать на различных устройствах в рамках одной ОС ведя себя совершенно одинаково - используя один интерфейс. Так появилась идея взять программу во время исполнения на одной вычислительной системе и перенести ее на другую систему, чтобы программа продолжила работу с того же места на котором она остановилась на первой системе. Этот процесс перенесения программы называется Миграцией программы.

Идея миграции находит применение в широком спектре задач и моделей использования вычислительных систем. Таких как: обновление ядра без перезагрузки системы, балансировка нагрузки, осуществление технической поддержки сервера прозрачно для пользователей, расширенные возможности отладки программ, безусловно, являющихся актуальными для современных компьютерных систем. Для примера рассмотрим ситуацию технической поддержки сервера. Пусть на физической машине работает веб-сервер, и необходимо провести технические работы. Для их проведения раньше пришлось бы выключить сервер, при этом пользователи не смогли бы использовать предоставляемые им услуги. Если же на время технических работ мигрировать веб-сервер на резервную машину, то проведение технических работ будет прозрачно для пользо-

вателей. Новое время простоя сервера уже не будет зависеть от времени необходимого для проведения технических работ, а только от времени миграции сервера, которое может быть гораздо меньше.

Ранее, решения данных задач можно было добиться только на специально устроенной для этого системе, например с измененным ядром, или потребовалось бы иметь заранее модифицированную переносимую программу, которая позволяла и помогала проделывать над собой такие операции, или была необходима динамическая линковка библиотек. Эти ограничения связаны с возникновением ряда проблем на практике. А именно: использование только старых, поддерживаемых ядер, что приводит к урезанию(ограничению) возможностей мигрируемых программ; отсутствие применимости к стороннему ПО с закрытым исходным кодом; снижение безопасности, встраивание библиотек может непредсказуемо повлиять на работу процесса, то есть ограничивается круг “подходящих” программ.

Для решения этого круга проблем был создан проект Checkpoint / Restore In Userspace (CRIU), в котором используется минималистичный подход при внесении необходимых изменений в ядро Линукс, а основная функциональность вынесена из ядра в пространство пользователя. В CRIU поддерживаются все новые ядра, для сохранения и последующего восстановления процесса не требуется предварительной его подготовки. Эта система оптимизируется по скорости работы и находится в активной разработке. Таким образом, в условиях появления универсального решения, которое можно использовать для миграции программ, появляется необходимость рассмотрения вопроса о Живой миграции. В предыдущих решениях этот процесс не был контролируемым, то есть в них старались уменьшить время миграции, чтобы не получить разрывов соединений, но не было проведено оценок реальной возможности Живой миграции, и внимание на этом не заострялось. Конкретная задача данной работы - это создание механизма контроля процесса миграции и предсказания поведения этого процесса на базе CRIU.

Оптимизация сохранения и восстановления программ в проекте CRIU и разработка модели Live-миграции приведет к появлению эффективного механизма решения вышеупомянутого списка задач. Таким образом,

цели магистерской диссертации:

1. Подготовительная работа с CRIU, изучение системы;
2. Дедупликация для оптимизации пространственной сложности задачи;
3. Моделирование Live-миграции процессов и групп процессов на основе CRIU;
4. Написание программной реализации алгоритма прогнозирования остановки миграции для обоснования и тестирования математической модели.

Глава 1. CRIU (Checkpoint/Restore In Userspace)

CRIU (Checkpoint/Restore In Userspace) – быстро развивающийся проект, разрабатываемый компанией Parallels, для операционной системы Linux. Цель CRIU позволить сохранять состояние программы в качестве контрольной точки(снимка), после чего восстановить и продолжить работу программы с этой точки. CRIU не является первым проектом пытающимся реализовать данную функциональность, к примеру в OpenVZ используется похожий механизм для живой миграции, а в Parallels Virtuozzo для возобновления после обновления ядра. Но у CRIU есть ключевое отличие от ранее реализованных проектов — в нем основная работа по сохранению и восстановлению перенесена в пространство пользователя. В отличие от предшественников, которые из-за большого объема кода не стали частью основного ядра, у CRIU есть все необходимое, что-бы привнести в Linux эту функциональность.

1.1. История

Рассмотрим современные, конкурентные проекты реализации функциональности сохранения и восстановления программ.

1.1.1. Конкурирующие проекты

BLCR(Berkeley Lab Checkpoint/Restart)

Проект BLCR [2] разрабатывается в Национальной Лаборатории имени Лоуренса Беркли в Калифорнийском Университете[1] США с 2003 г. Это программное обеспечение с открытым исходным кодом под лицензией GPL. Задачей которого является реализация технологии сохранения и восстановления процессов (Checkpoint/Restart) ориентированное на задачи высокопроизводительных вычислений на MPI. BLCR реализована из двух частей: загружаемый модуль ядра, для Linux 2.4 и 2.6 ядер на базе архитектур x86 и x86-64, и библиотека расширений которую нужно

[1] Lawrence Berkeley National Laboratory in University of California

связать с программой, чтобы совершать сохранение и восстановление. В силу архитектурных особенностей проект столкнулся с рядом проблем практического применения.

То что реализация BLCR не была добавлена в основное ядро Линукс создало большую проблему разработчикам, так как приложения не стоят на месте они могут требовать использования новых ядер с поддержкой нового функционала. И для применимости к этим программам необходимо постоянно портировать код BLCR под новый Linux.

> "By far the most challenging aspect of implementing BLCR was to keep it working as the Linux kernel continued to evolve."
> [2]Paul H Hargrove and Jason C Duell

Данный проект может работать только с приложением в которым заранее статически произведена привязка специальной библиотеки [3], которая может повлиять на работу программы. Таким образом, может быть замедлена работа программы и созданы проблемы безопасности. Кроме того данный проект не поддерживает работу с открытыми сетевыми соединениями. Но поддержка сохранения и восстановления сетевых соединений является критичным моментом для реализации живой миграции.

Итак основные минусы этого решения, по сравнению с подходом CRIU:

1. Добавить в ядро код проекта не получилось, потому-что он был слишком большим и сложным;
2. Необходимо связывание приложения с специальной библиотекой BLCR;
3. Библиотека потенциально может навредить программе;
4. Решение проблемы сохранения и восстановления сетевых соединений не предлагается.

DMTCP(Distributed MultiThreaded CheckPointing)

DMTCP разрабатывался в Северо-Восточном Университете[2] США в 2004-2006 г.г. DMTCP основан на предыдущей работе MultiThreaded CheckPointing [5] и выпускается под лицензией LGPL. Это инструмент для сохранения и восстановления состояния нескольких приложений реализован полностью в пространстве пользователя, тестировалась поддержка приложений на Open MPI, MatLab, Pythion, Perl.

> "Checkpointing is added to arbitrary applications by injecting a shared library at exectution time. This library: Launches a checkpoint management thread in every user process which coordinates checkpointing. Adds wrappers around a small number of libc functions in order to record information about open sockets at their cration time."
>
> [4]Jason Ansel, Kapil Arya and Gene Cooperman

В данном проекте используется динамически подключаемая библиотека, которая создает для каждого процесса управляющую нить, которая производит процесс сохранения приложения. Такой подход потенциально может деструктивно повлиять на работу приложения и существенно замедлить работу системы. Это является основным минусом этого решения по сравнению с CRIU в котором не производится потенциально деструктивного внедрения библиотеки в работающее приложение, и нет ее замедляющего влияния на приложение.

OpenVZ(Open Virtuozzo)

OpenVZ это решение контейнерной виртуализации на уровне операционной системы под лицензией GPL разрабатываемое при поддержке компании Parallels с 2005 г. OpenVZ позволяет на одном физическом сервере размещать множество изолированных копий операционной системы с общим ядром. Система обеспечивает высокопроизводительное использование физических ресурсов, и масштабируемость.

[2]Northeastern University

Это решение включает в себя реализацию сохранения и восстановления контейнера и Живую миграцию контейнеров. Вся функциональность OpenVZ реализована как отдельное ядро операционной системы на базе Linux. Так как функциональность полностью была реализована в ядре операционной системы, то ее поддержка была затруднена постоянным изменением ядра Линукс. На данный момент поддерживается несколько веток OpenVZ ядер основанных на 2.6.* ядрах Линукс. Учитывая темпы развития ядра было принято решение внести код OpenVZ в ядро Линукс.

Из-за объемности кода и печального опыта предыдущих попыток внесения функциональности в ядро, появилась идея проекта CRIU - сохранение и восстановление (в основном) в пространстве пользователя.

Linux Checkpoint/Restart by Oren Laadan

Checkpoint/Restart by Oren Laadan [6] это решение разрабатываемое Ореном Лааданом в Колумбийском университете с 2008 года. Данный проект был попыткой внесения функциональности сохранения и восстановления приложений в ядро Linux. В общем и целом было произведено примерно 20 попыток послать патчи проекта в ядро Линукс. После долгих попыток сообщество ядра так и не приняло проект из-за слишком большого объема изменений которые нужно было внести в ядро, и вовремя появившийся проект CRIU перехватил инициативу.

1.1.2. История CRIU

В 2011 году компанией Parallels были анонсированы планы внедрения своих систем контейнерной виртуализации в основное ядро Linux. Идея проекта была в том чтобы выделить необходимый минимум изменений, которые необходимо привнести в ядро, а всю остальную функциональность вынести в пространство пользователя.

Стартовая версия проекта CRIU была разработана Павлом Емельяновым, лидером команды OpenVZ, и представлена Linux сообществу 15 июля 2011 года. Она была первым толчком для убеждения Linux сообщества в том что идея принесет конечный результат. Многие, в связи с неудачными попытками предшественников по привнесению данного

функционала в ядро, отнеслись к идее скептически в том числе и Линус Торвальдс.

> "... this is a project by various mad Russians to perform c/r mainly from userspace, with various oddball helper code added into the kernel where the need is demonstrated. So rather than some large central lump of code, what we have is little bits and pieces popping up in various places which either expose something new or which permit something which is normally kernel-private to be modified. The overall project is an ongoing thing. I've judged that the size and scope of the thing means that we're more likely to be successful with it if we integrate the support into mainline piecemeal rather than allowing it all to develop out-of-tree. However I'm less confident than the developers that it will all eventually work! So what I'm asking them to do is to wrap each piece of new code inside CONFIG_CHECKPOINT_RESTORE. So if it all eventually comes to tears and the project as a whole fails, it should be a simple matter to go through and delete all trace of it."
>
> [1]Linus Torvalds, Andrew Morton, LKML

Но, не смотря на это, начало было положенно и первые изменения были внесены в основное ядро Linux. Далее были внесены изменения для отслеживания грязных страниц, добавлен режим восстановления для сокетов. Командой CRIU вплоть до 3 июля 2013 года было внесено более 150 патчей.

В версии ядра 3.11 уже присутствует весь необходимый для работы CRIU функционал, за небольшим исключением незначительных дополнительных возможностей. И дальнейшее развитие направленно на увеличение стабильности - поиск ошибок и повышение эффективности процедур создания контрольных точек и восстановления из них.

1.2. Архитектура

Основная функциональность CRIU делится на две части [13]:

1. dump - сохранение снимка состояния процесса[3];

2. restore - восстановление процесса из снимка этого состояния.

Также есть дополнительная функциональность направленная на оптимизацию работы системы и на повышение удобства взаимодействия с системой. Есть инструмент pre-dump - не полное сохранение, а отдельно, только снимка памяти процесса. Сам по себе, такой снимок не хранит информацию достаточную для восстановления процесса, но может быть использован для восстановления при итеративной миграции. Он позволяет подготовить до начала полного сохранения снимка часть памяти к восстановлению, то-есть уменьшить время простоя на финальной итерации. Так же в CRIU есть page-server - сервер для получения страниц памяти процесса, напрямую из процедуры сохранения состояния, это необходимо для передачи страниц памяти при миграции. Service для удобства использования CRIU через RPC(Remoute Procedure Call). Дедупликация - возможность очистки сохраненных снимков состояния для уменьшения расхода памяти при создании итеративных снимков, основанных на предыдущих итерациях.

1.2.1. Сохранение моментальных снимков

Полное сохранение состояния дерева процессов требует сохранить: структуру самого дерева, память которую используют процессы, пространства имен; очереди, пайпы, файловые дескрипторы, таймеры и все другие объекты ядра используемые программой.

CRIU делает уклон в сторону реализации данной функциональности в пространстве пользователя. По этому, чтобы добыть необходимые данные используется API (Application Programing Interfaces) ядра Linux и только при невозможности сохранения каких-то объектов обычными средствами, в этом аспекте расширяется функциональность ядра.

Рассмотрим интерфейсы используемые для сохранения состояния процессов в CRIU:

1. Файловая система Procfs;

[3]Для простоты изложения часто упоминается один процесс, но CRIU позволяет также работать с группами процессов - то есть с некоторым под деревом дерева процессов системы

2. Механизм контроля процессов ptrace;

3. Системные вызовы из контекста сохраняемого процесса;

4. Netlink socket interface - интерфейс мониторинга и управления сетевыми сокетами.

Файловая система Procfs

Файловая система proc - это особая файловая система в Linux. Это виртуальная файловая система, то-есть содержащиеся в ней файлы не хранятся на диске а являются своеобразными интерфейсами для получения текущего состояния ядра Linux. По умолчанию в Linux системах эта файловая система монтируется в директорию /proc. Если провести листинг этой директории то можно увидеть большое количество фалов, каждый из них предоставляет возможность узнать о процессах, соответствующую информацию. При попытке чтения этих файлов ядро создает необходимые данные с актуальной информацией о процессе. При сохранении состояния процесса CRIU использует интерфес proc для получения такой информации о процессе, как список созданных процессом таймеров (/proc/<pid>/timers), список открытых файлов по номерам файловых дескрипторов (/proc/<pid>/fd/<id>), информация об этих файлах: сдвиг в файле, режим доступа и статус (/proc/<pid>/fdinfo/<id>), там же находится информация о дескрипторах нотификации о событиях, получения процессом сигналов, и многое другое. Полученные из proc данные позволяют получить и сохранить в образы, широкий спектр необходимых для восстановления данных, также они позволяют облегчить дальнейший сбор информации об объектах.

Напрмер для posix-таймеров, реализация поддержки которых была вводным заданием в проекте, чтение /proc/<pid>/timers дает возможность узнать количество созданных таймеров, ID каждого таймера, каким образом вызвавший процесс должен быть уведомлен о срабатывании таймера, а также с какими часами связан таймер[4]. Все это необходимо

[4] Нехватка последнего для восстановления таймеров была обнаружена в течение разработки и добавлена в ядро Linux, сейчас эта функциональность уже находиться в основном ядре.

для воссоздания posix-таймеров а также для получения дополнительной информации на последующих этапах сохранения снимка.

Механизм контроля процессов Ptrace

Prace это интерфейс ядра Линукс, который позволяет одному процессу “наблюдателю” рассматривать и контролировать исполнение другого процесса - “наблюдаемого”. Он позволяет изучать и изменять память и регистры наблюдаемого процесса. В основном его используют в программах для отладки(gdb, dbx), и отслеживания системных вызовов.

В CRIU он используется для подключения к работающему приложению, его замораживания, записи и чтения блоков памяти в специально найденное свободное место в памяти приложения. При сохранении программы, после чтения из procfs информации о ядерных объектах, происходит подключение к приложению, записываются нужные данные в память процесса, и запускается, так называемый, паразитный код, который используя эти данные до собирает информацию об объектах, к которой извне нет доступа.

Далее происходит выход из контекста процесса, это подразумевает копирование полученных данных из памяти приложения и освобождения всей занятой CRIU памяти. Вся собранная информация записывается в снимки, и дальше необходимо разморозить процесс. Для разморозки используется хитрый механизм, который восстанавливает состояние регистров и вычищает последний собственный код из памяти приложения путем создания специального прерывания. Данный подход позволяет получить необходимую дополнительную информацию и при этом не оставить следов подключения к приложению.

Интерфейс Syscall

В режиме паразитного кода в контексте приложения используется компактный код для извлечения дополнительных неизвестных данных о состоянии процесса с использованием обычных системных вызовов, тоесть вызовов, которыми обычное приложение могло бы считать нужную ему для работы информацию.

Интерфейс Netlink

Командой CRIU была разработана функциональность получения информации сетевых сокетов(tcp repair mode) и внедрена в ядро Линукс в интерфейс Netlink. Она позволяет сохранять полное состояние сетевых сокетов, что важно для живой миграции в которой требуется восстанавливать и сохранять сетевые соединения.

1.2.2. Восстановление из снимков

При восстановлении приложения из снимка необходимо полное воссоздание состояния в котором пребывало сохраненное приложение. Для этого используются все те же четыре интерфейса ядра Linux.

Файловая система procfs позволяет не только считывать информацию о состоянии процесса, но и помогать при восстановлении объектов ядра. Например, proc позволяет получить возможность восстановления некоторых объектов с заданным ID. Путем записи в специальный файл требуемого ID объекта. При следующем создании такого объекта ядро создаст объект с данным ID, с ограничением, конечно, что этот ID должен быть изначально не занят. Такой функционал существует в ядре Линукс например для семафоров(/proc/sys/kernel/sem_next_id), для разделенной памяти(shm_next_id) и сообщений(msg_next_id). Все это возможно в основном благодаря изменениям ядра Линукс внесенными командой CRIU.

С помощью системных вызовов воссоздаются объекты ядра используемые приложением.

В конце восстановления используется Ptrace для того, чтобы аналогично как и на сохранении снимков “разморозить” приложение и очистить его контекст от следов CRIU, который его воссоздавал.

Для воссоздания сетевых сокетов был добавлен в ядро механизм починки сокетов(tcp repair mode) который позволяет их полностью воссоздавать используя интерфейс Netlink.

1.3. Тестирование

CRIU это проект призванный увеличить надежность системы, по этому его тестированию отведено особенное внимание. Система CRIU находится под постоянным тестированием, на все реализованные в CRIU функциональные части одновременно готовятся модульные тесты и внедряются в автоматически тестирующую систему ZDTM + Jenkins. Так же существуют некоторые отдельные тесты которые тестируют например RPC и сохранность памяти. Пакет тестирования дает существенные показатели покрытия кода, примерно 90% покрытия функций и 70% процентов покрытия кода.

1.3.1. ZDTM

ZDTM это сокращение от Zero Down Time Migration [15]. Это система автоматического тестирования унаследованная проектом CRIU от OpenVZ, разработанная для тестирования работоспособности всех компонент отвечающих за миграцию. Система состоит из множества атомарных тестов - каждый переводит процесс в некоторое состояние: открывает файл, выделяет сегмент памяти, отправляет данные в пайп, запускает таймеры, и др. Потом происходит сохранение и восстановление теста, а затем тест проверяет сохранность состояния: нужный файл открыт, выделенный сегмент памяти на месте, данные приходят из пайпа, таймеры срабатывают через правильный интервал.

Данная система также имеет различные опции, например можно запускать тесты в выделенном пространстве имен, или хранить снимки в памяти, а не на диске. Система непрерывной интеграции через некоторые интервалы пересобирает проект из главного репозитория и прогоняет на нем тесты с различными параметрами. Такой подход позволяет поддерживать высокое качество кода CRIU.

Проект также проходил тестирование членами команды CRIU на реальных приложениях, среди них: Apache, Nginx, MySQL, MongoDB, Oracle, Make, gcc, tar, gzip, screen, LXC контейнеры, Java, VNC сервер.

1.4. Использование

Возможный спектр вариантов использования CRIU довольно широк.

1. Создание снимков для долго загружающихся приложений в конце загрузки, для последующего быстрого запуска, путем восстановления из снимка.

2. Прелинковка больших библиотек.

3. Создание снимка “за секунду до” для возможности расширенной работы по устранению ошибок, или в качестве исчерпывающего сообщения об ошибке, для последующего воссоздания проблемы на системе специалиста.

4. Перенос приложений на высокопроизводительный кластер или на низкопроизводительную систему в зависимости от ситуации.

5. Создание резервных снимков, для восстановления при неисправностях.

6. При обнаружении тупиков и условий гонки, откат на предыдущий снимок и запуск в “безопасном” режиме.

И основной целью которую воплощает CRIU является реализация живой миграции приложений.

Глава 2. Дедупликация

2.1. Проблема использования оперативной памяти при миграции

Для оптимизации процесса миграции была написана дополнительная функциональность для CRIU для дедупликации образов памяти. Данная функциональность имеет непосредственную важность для осуществления живой миграции, поскольку для ускорения миграции, снимки памяти процесса решено хранить в оперативной памяти, иначе бы процесс занимал на порядки больше времени. Но при итеративной миграции программы создающей высокую нагрузку на оперативную память, за время передачи данных одного снимка может накапливаться большой объем новых измененных страниц памяти. При большом количестве итераций общий размер образов постоянно растет, даже при относительно постоянном размере рабочего набора, размер снимков может стать на порядок больше чем размер первого снимка. Так как, со временем, все данные не будут помещаться в оперативной памяти системы, появится отгрузка данных на диск. И это нивелирует преимущества использования быстрой оперативной памяти и уменьшит скорость миграции, которая должна быть оптимальной для достижения живой миграции.

При итеративной миграции, при получении новых измененных страниц данных, соответствующие данные в старых образах, в рамках живой миграции уже не нужны, так как восстанавление напрямую из старых образов проводить не предполагается. Старые снимки будут использованны только для «подхвата» не измененных страниц во время окончания миграции при восстановлении программы. Таким образом освобождение памяти от старых страниц в родительских образах не повлияет на восстановление в конце.

2.2. Алгоритм

Был разработан алгоритм и внедрен в процесс снятия итеративного снимка, который при получении нового блока данных находил старые данные для этого блока, и очищал от них родительские снимки. Эта функциональность возможна благодаря механизму Sparce-файлов (разряженных файлов), он полностью поддерживается файловой системой tmpfs. Таким образом можно взять файл лежащий в оперативной памяти(на tmpfs) и в необходимом месте освободить определенный объем памяти. То-есть размер файла уменьшится, но при чтении из этого места файловая система будет имитировать, что оно заполнено нулями.

Теоретическая эффективность этого алгоритма основана на том, что для реального восстановления памяти процесса необходимо хранить не больше чем размер рабочего набора процесса, то-есть при использовании этой технологии можно сократить многократно увеличивающийся объем памяти до размера одного полного рабочего набора процесса.

Этот же подход применим и при восстановлении процесса из снимка, так как снимок и процесс существуют в одной оперативной памяти то чтобы минимизировать ее использование был реализован и внедрен в процесс восстановления алгоритм дедупликации на восстановлении. При чтении страниц из образов непосредственно после восстановления страницы в пространство памяти программы, эта страница освобождается в системе итеративных образов, что позволяет на всем промежутке восстановления программы поддерживать постоянное количество суммарно занимаемой программой и образами памяти. По сравнению с решением без дедупликации пространственная сложность уменьшается в два раза. Что для высоко загруженных программ является весьма эффективной оптимизацией.

Также для оптимизации алгоритма и его применения в реальных условиях была добавлена реализация объединения смежных блоков для последующей очистки большими кусками, менее 1Мб. Так как погрешность в 1Мб допустима при современных объемах оперативной памяти, и это позволит существенно сократить количество системных вызовов.

На приведенной схеме 4.1 показан пример прохода алгоритма дедупликации при создании снимка. Здесь прямоугольники с скругленными

краями схематично отображают пространство виртуальной памяти процесса. А яркие цветные прямоугольники это заполненные данными блоки памяти. Так алгоритм при нахождении нового блока памяти(салатовый), сразу после его сохранения делает спуск по дереву образов путем поиска в глубину. То-есть он находит во втором снимке максимальный пересекающийся с новым блоком пропуск или блок данных, дальше алгоритм спускается на следующий родительский снимок, для блока данных алгоритм также отправляет данные в накопительный блок очистки, так как это старые данные, которые больше не нужны. Для каждого снимка есть блок очистки который накапливает подряд идущие(в файле) данные, для последующей совместной очистки(до 1Мб). То-есть в первом снимке, например 2 и 3 блоки помеченные на удаление, лежат в файле подряд и будут удалены одновременно.

В итоге совокупность внедренных алгоритмов дает возможность для живой миграции процесса использовать объем памяти приблизительно равный объему памяти самого процесса, то-есть разрешает проблему прямого использования оперативной памяти, которое на порядок ускоряет процесс миграции.

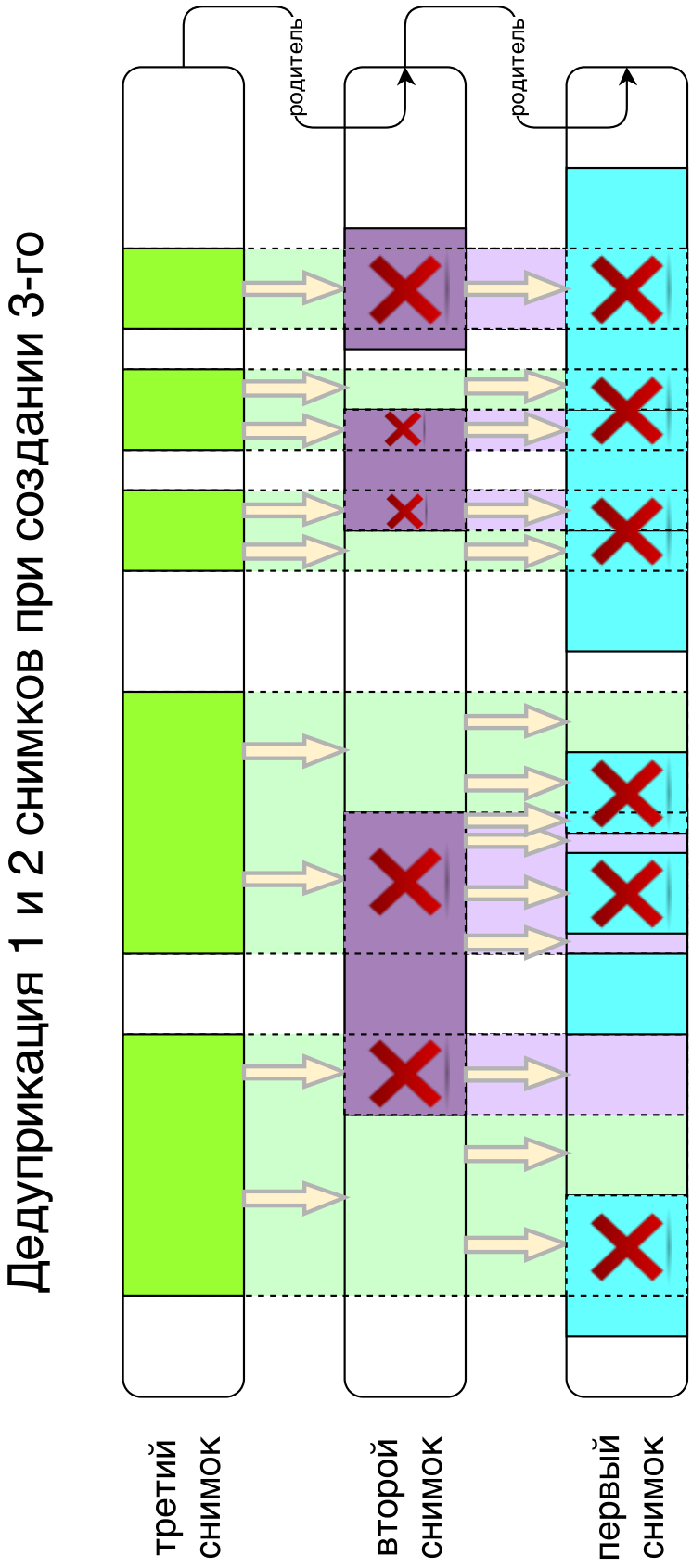

Рис. 2.1: Схема дедупликации

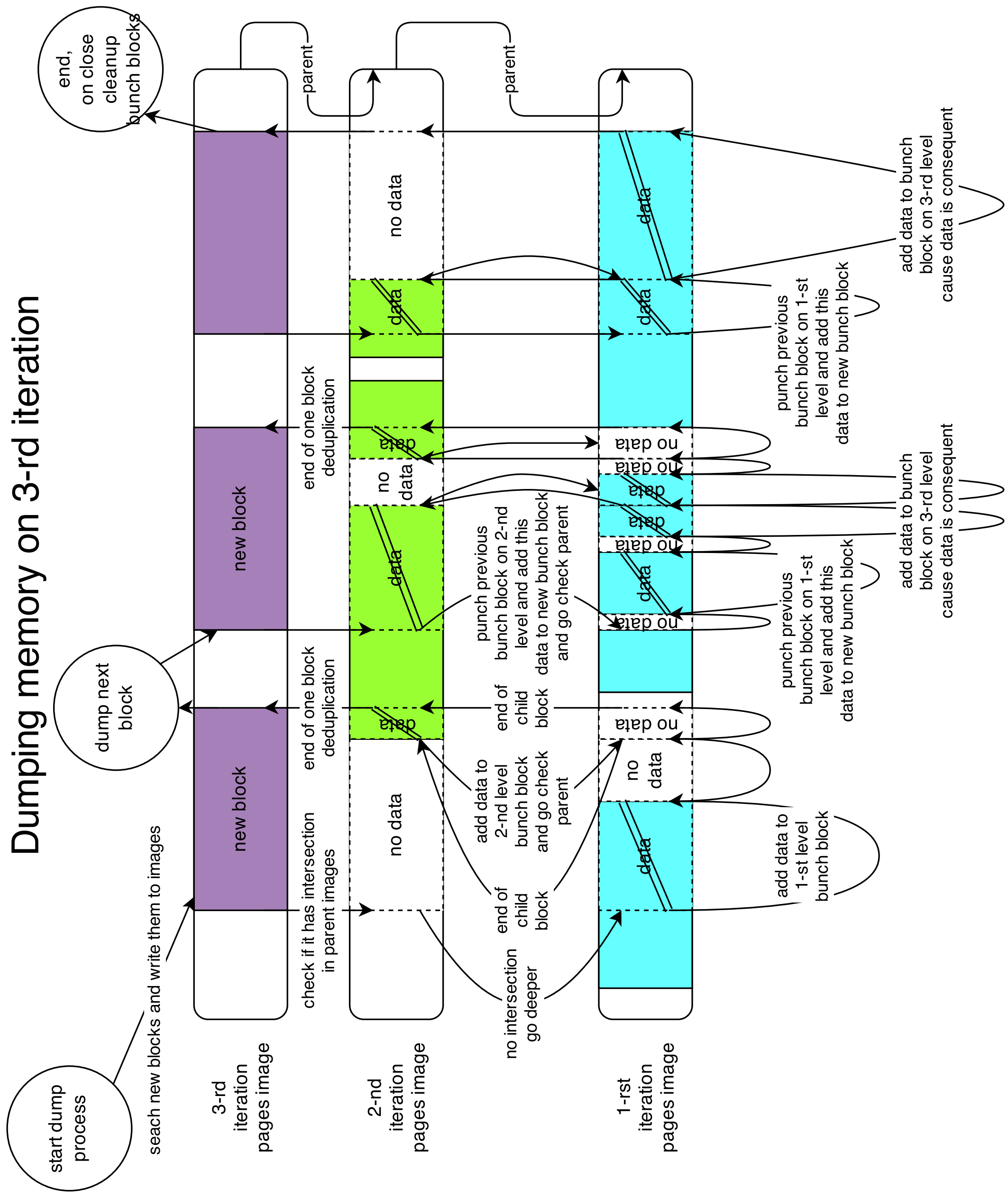

Рис. 2.2: Подробная схема дедупликации

Глава 3. Миграция

3.1. Определения

Миграция приложения - это процесс перемещения выполняющегося приложения с одного вычислительного узла на другой, при котором приложение начинает работу на целевом узле с того-же состояния в котором оно пребывало сразу перед перемещением, и этот процесс не приводит к изменению результата работы приложения.

Так как объемы памяти современных процессов достаточно велики [18], появляется необходимость оптимизировать процесс передачи памяти при миграции приложения, иначе процесс миграции займет слишком много времени. Таким образом для миграции памяти был предложен механизм Итеративной миграции. В простом процессе миграции в начале процесс останавливается, потом сохраняется снимок памяти процесса, этот снимок передается на целевой узел и происходит восстановление памяти процесса на целевом узле из снимка. Итеративная миграция - это процесс миграции, который отличается от простого процесса миграции в следующем: после снятия снимка памяти процесс сразу же запускается на начальной машине и продолжает работу, пока передается снимок. Соответственно, когда снимок памяти передан, на начальной машине появятся изменения по сравнению с снимком. Теперь программа вновь останавливается и создается снимок памяти, но только тех страниц которые изменились. И такой процесс продолжается до некоторого момента. Потенциально такой процесс может снизить время остановки при миграции за счет того что на последней итерации размер передаваемой памяти будет меньше.

Для оценки применимости миграции было введено понятие Живой миграции. Живая миграция - это такая миграция, при которой за время миграции не произойдет разрыва внешних соединений программы. Например, для сетевых соединений, время разрыва равно пяти минутам. Но некоторые соединения более высокого уровня могут использовать схему Keep-alive, что еще сужает время допустимое при миграции.

Так же существует понятие Незаметная Живая Миграция(seamless live-migration), которое определяет миграцию, при которой влияние миграции на работу приложения будет незаметно для пользователя. К этому варианту миграции необходимо стремится, то-есть минимизировать видимое влияние миграции для пользователя.

3.2. Существующие решения

Рассмотрим существующие решения для миграции программ и методы определения момента окончания итераций при миграции. В основном для осуществления миграции используются схожие алгоритмы, в которых для миграции программы с одного узла на другой необходимо:

1. сохранить снимок состояния программы, то-есть полное исчерпывающее описание происходящего в ее рамках процесса;

2. восстановить программу из снимка состояния, таким образом, чтобы она начала работу с того-же места, имела то же состояние памяти, были открыты те же дескрипторы файлов, грубо говоря необходимо, что бы программа не заметила ни каких изменений при этом процессе.

Например, вычислительная программа восстановленная из снимка должна не потерять произведенный прогресс и продолжить вычисление задачи с той же точки. И процесс сохранения и восстановления не должен повлиять на результат ее работы. Собственно осуществление этих двух функций и есть задача CRIU.

Проект CRIU направлен на решение ряда проблем которые возникали при предыдущих подходах к осуществлению миграции. Одним из самых важных ограничений предыдущих подходов было то, что вся работа по сохранению и восстановлению программы осуществляется в ядре операционной системы. Но внести функциональность, в связи с объемностью кода, в основное ядро не представлялось возможным. Что мешает реальному применению таких систем, из-за невозможности постоянной поддержки ядра, развивающегося быстрыми темпами, и соответственно программ использующих новый функционал. Примерами

таких решений служат OpenVZ, Berkeley Lab Checkpoint/Restart, Linux Checkpoint/Restart by Oren Laadan. В ряде предыдущих решений используется другой подход: необходимо или предварительное привязывание к специальной библиотеке или динамическое уже в процессе работы программы, в первом случае подход тяжело применим к программам с закрытым кодом, и в обоих случаях такая библиотека может повлиять на работу программы или замедлить ее, что является серьезной проблемой безопасности. Примерами служат Berkeley Lab Checkpoint/Restart и Distributed MultiThreaded CheckPointing.

В данных решениях в связи с вышеупомянутыми ограничениями, которые приводят к узкой применимости или замедлению миграции процесс миграции является оптимистическим, то-есть он не является оптимальным, а полагается на ряд эвристик. В CRIU же появляется возможность более точно рассмотреть этот процесс.

Во многих современных средствах миграции используется эвристические алгоритмы [14]. Есть три ключевых используемых эвристики:

1. Ограничение максимального количества итераций. Например в KVM есть предел в 30 итераций [14].

2. Остановка при отскоке, то-есть если на какой-то итерации объем переданной памяти был меньше чем объем «грязной» памяти, таким образом ситуация ухудшилась и процесс останавливается.

3. Остановка при достижении некоторого необходимого минимума измененных страниц.

Эти эвристики имеют право на существование, например пункт 3. Действительно, когда измененных страниц стало относительно мало, процесс может завершится успешной живой миграцией. Единственный минус этого пункта, это то, что предел задан статически, но он должен являться некоторой характеристикой состояния программы и соединения.

Остановка при отскоке имеет недостаток, так как при случайных флуктуациях насыщенности рабочего набора, весьма вероятно раннее завершение итераций, не достигнув оптимума.

Ограничение на количество итераций это важный параметр который не позволяет процессу миграции зависать на вечно, но и увеличивает вероятность того, что живая миграция не произойдет. Для живой миграции важно достижение некоторого минимума насыщения, иначе появляется большая вероятность неуспешного окончания процесса миграции с разрывом внешних соединений программы.

Есть работа посвященная определению производительности миграции в гипервизоре Xen [17], в которой были выявлены некоторые зависимости времени миграции при эвристических критериях от параметров. Но работа не предлагает их использования для оптимизации миграции и не точна, в связи с грубым использованием средних значений параметров.

3.3. мат-модель Миграции

При итеративной миграции для достижения живой миграции, предлагается оценивать функцию насыщения рабочего набора процесса на основе измерений, полученных в процессе мониторинга. Полученная функция используется для моделирования процесса итеративной миграции. Минимизируется суммарное время простоя при миграции и находится оптимум количества итераций. Вычисляется минимальное время простоя, достижимое при миграции.

3.3.1. Оценка и аппроксимация

Для оценки функции насыщения по экспериментальным данным предлагается использовать метод наименьших квадратов с оценочной функцией:

$$M(t) = (a \times t + b)/(c \times t + d); \tag{3.1}$$

В силу ограниченности графика насыщения рабочего набора, можно считать c = 1, a - пропорционально размеру рабочего набора(WS), и так как изначально график проходит через t = 0, M = 0, то b = 0. Таким

образом, можно рассматривать функцию:

$$M(t) = a \times t/(t + d); \tag{3.2}$$

Пусть $(t1, m1), ..., (t_n, m_n)$ экспериментальные данные, m_i - объем грязной памяти в момент времени t_i. Тогда запишем условие минимума суммы квадратов отклонений:

$$F = \sum_{i=\overline{1,n}} \left[m_i - \frac{a \times t_i}{t_i + d} \right]^2 \to min; \tag{3.3}$$

Данное уравнение не решается аналитически. Оно решается, например, с помощью алгоритма Левенберга-Марквардта [10]. Таким образом получается аналитическая апроксимация графика насыщения рабочего набора.

3.3.2. Модель процесса живой миграции

На основе найденной приближенной функции можно смоделировать процесс миграции, предполагая что эта функция слабо меняется во времени. То-есть на каждой итерации есть некоторый размер накопленной памяти для передачи, время передачи этих данных можно оценить по их размеру и средней скорости передачи данных соединения. По полученному времени используя приближенную функцию можно определить объем новых данных который будет накоплен к началу следующей итерации. И так получается модель реального итеративного процесса происходящего при итеративной миграции. Где M_i - объем грязной памяти накопленной на i-той итерации, $t(M)$ время на передачу блока памяти объемом M, a и d - найденные при аппроксимации параметры.

$$M_{n+1} = \frac{a \times t(M_n)}{t(M_n) + d} \tag{3.4}$$

Этот итеративный процесс сходится. Так как приближенная функция насыщения рабочего набора процесса непрерывна, монотонно возрастает и ограничена $\mathrm{M(t)} < 1$, значит получающаяся последовательность времени передачи данных на каждой итерации, монотонно убывает и ограничена снизу нулем, что и значит сходимость процесса.

Так же необходимо учесть, что времени передачи блока данных не пропорционально размеру этого блока. Есть некоторая задержка l(latency) при передаче данных в реальной сети. То-есть если M - размер данных, v - теоретическая скорость соединения, то время на передачу будет:

$$t(M) = v \times M + l \tag{3.5}$$

$$M_{n+1} = \frac{a \times (v \times M_n + latency)}{v \times M_n + latency + d} \tag{3.6}$$

Таким образом проведя достаточное количество итераций, можно определить приближенный инфимум необходимого для передачи времени, и количество итераций для его достижения с некоторой заданной точностью.

3.3.3. Оптимизация

При рассмотрении модели итераций пока не учитывалось, что для снятия снимка на каждой итерации требуется некоторый интервал времени. Если включить его в модель, то можно посчитать такую характеристику процесса как суммарное время простоя мигрируемой программы - $T_{\sum freezetime}$.

Время для снятия снимка можно считать не изменяющимся от итерации к итерации, так как основную нагрузку создает то, что при снятии снимка необходимо делать полный обход памяти процесса и проверять каждую страницу на предмет ее изменения с момента предыдущего снимка.

Тогда суммарное время простоя на i итерации будет:

$$T_{\sum freezetime}(i) = (i-1) \times \Delta t + t(i). \tag{3.7}$$

Где Δt — время простоя для снятия снимка, $t(i)$ — время для передачи грязной памяти на i итерации. Так как $t(i)$ сходится к инфимуму при i стремящемуся к бесконечности а $(i-1)\Delta t$ линейно возрастает, то у последовательности $T_{\sum freezetime}(i)$ сумм, существует минимум.

То-есть используя предложенную модель можно определить количество итераций в процессе миграции необходимое для минимизации суммарного времени простоя. Суммарное время простоя, в свою очередь

характеризует влияние процесса миграции на среднюю скорость работы процесса. Таким образом минимизация этого влияния является важным параметром для живой миграции, так как живая миграция это миграция процесса при которой не происходит разрыва соединения.

3.3.4. Выбор функции

Так же для аппроксимации графика насыщения рабочего набора можно использовать другие "подходящие" аналитические функции. Процесс насыщения рабочего набора приложения в начале имеет большую скорость насыщения, это обосновано тем, что вся память которая ему доступна в начале "чистая" и каждый запрос на запись добавляет грязные страницы. Но со временем когда большая часть памяти уже стала "грязной" вероятность, что запрос на запись попадет в "чистую" страницу становится пропорционально меньше, в предположении, что распределение вероятности попасть в некоторую определенную страницу равномерное. Таким образом аппроксимирующая функция должна быть выпукла вверх. Так же функция должна монотонно возрастать, так как размер грязной памяти не может уменьшится[1] и быть ограничена сверху. Были рассмотрены различные функции например:

$$M(t) = a \times atan(b \times t); t \subset [0, \infty] \tag{3.8}$$

$$M(t) = a \times e^{-\frac{1}{b \times t}}; t \subset [0, \infty] \tag{3.9}$$

$$M(t) = \begin{cases} a \times t, & t \subset [0, b]; \\ a \times b, & t \subset [b, \infty]; \end{cases} \tag{3.10}$$

$$M(t) = \frac{a \times t}{t + b}; t \subset [0, \infty] \tag{3.11}$$

[1] Хотя на практике это не так, например если приложение освобождает память, но в рамках этой работы рассмотрена упрощенная модель взаимодействия с памятью.

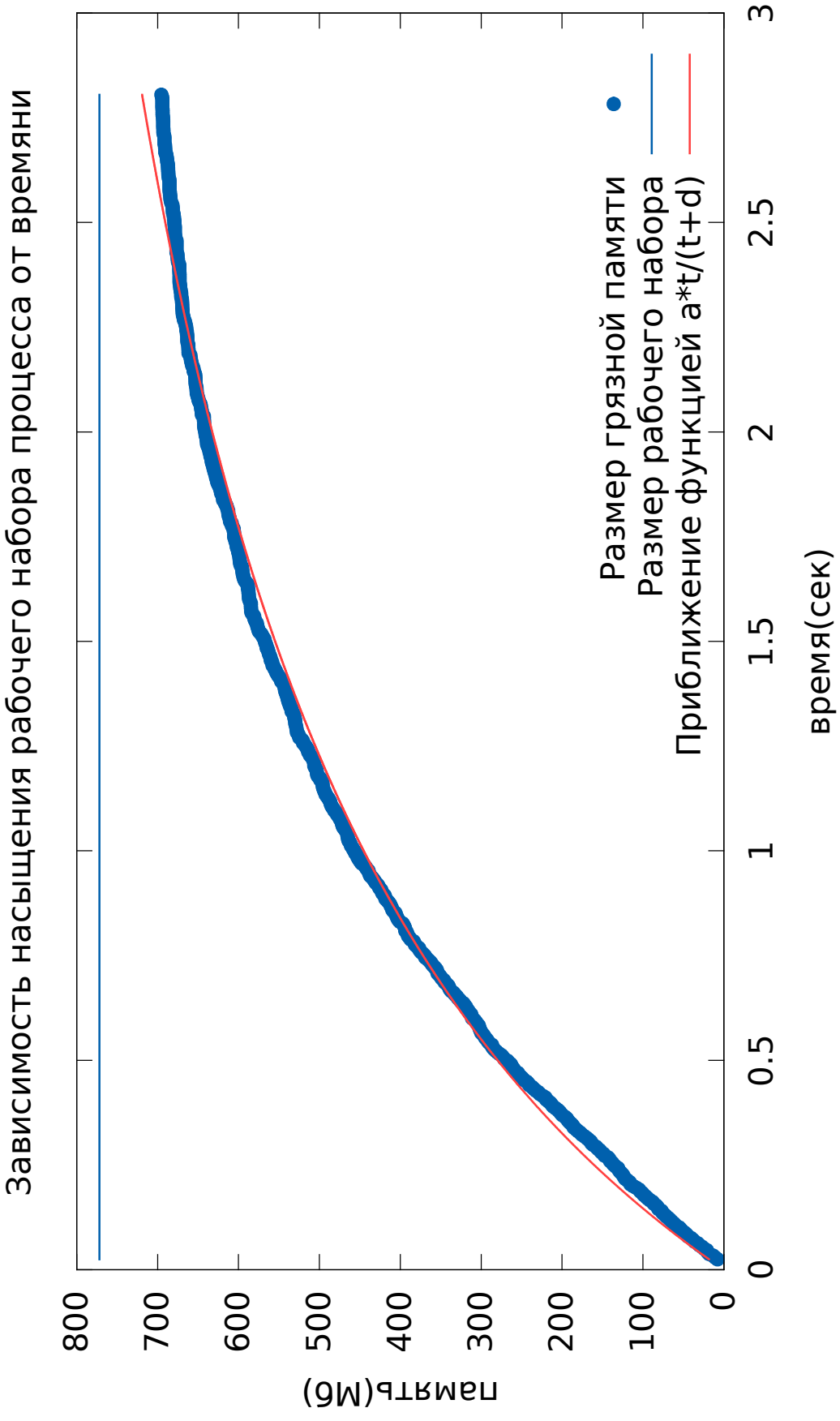

Рис. 3.1: Мониторинг и аппроксимация

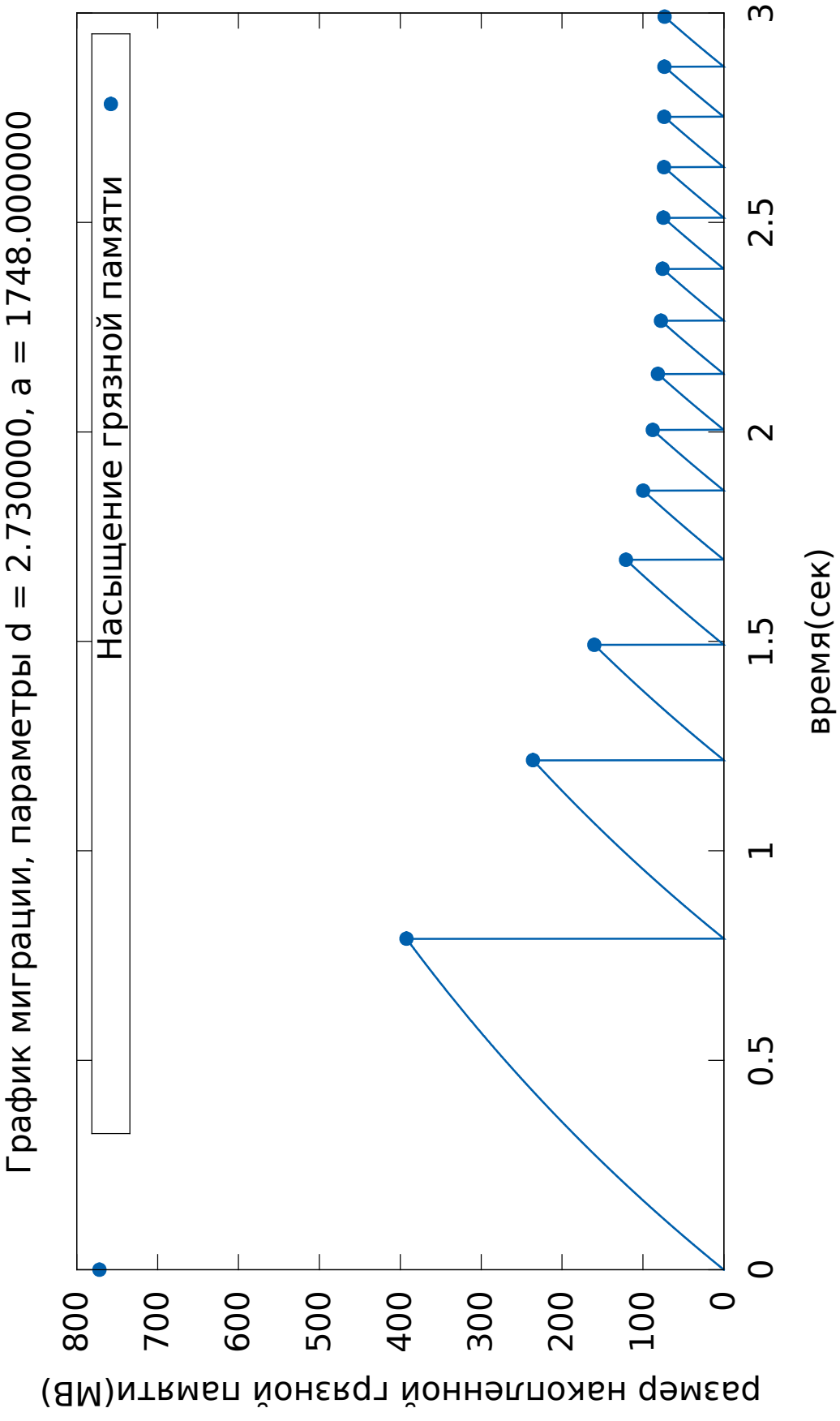

Рис. 3.2: Модель миграции

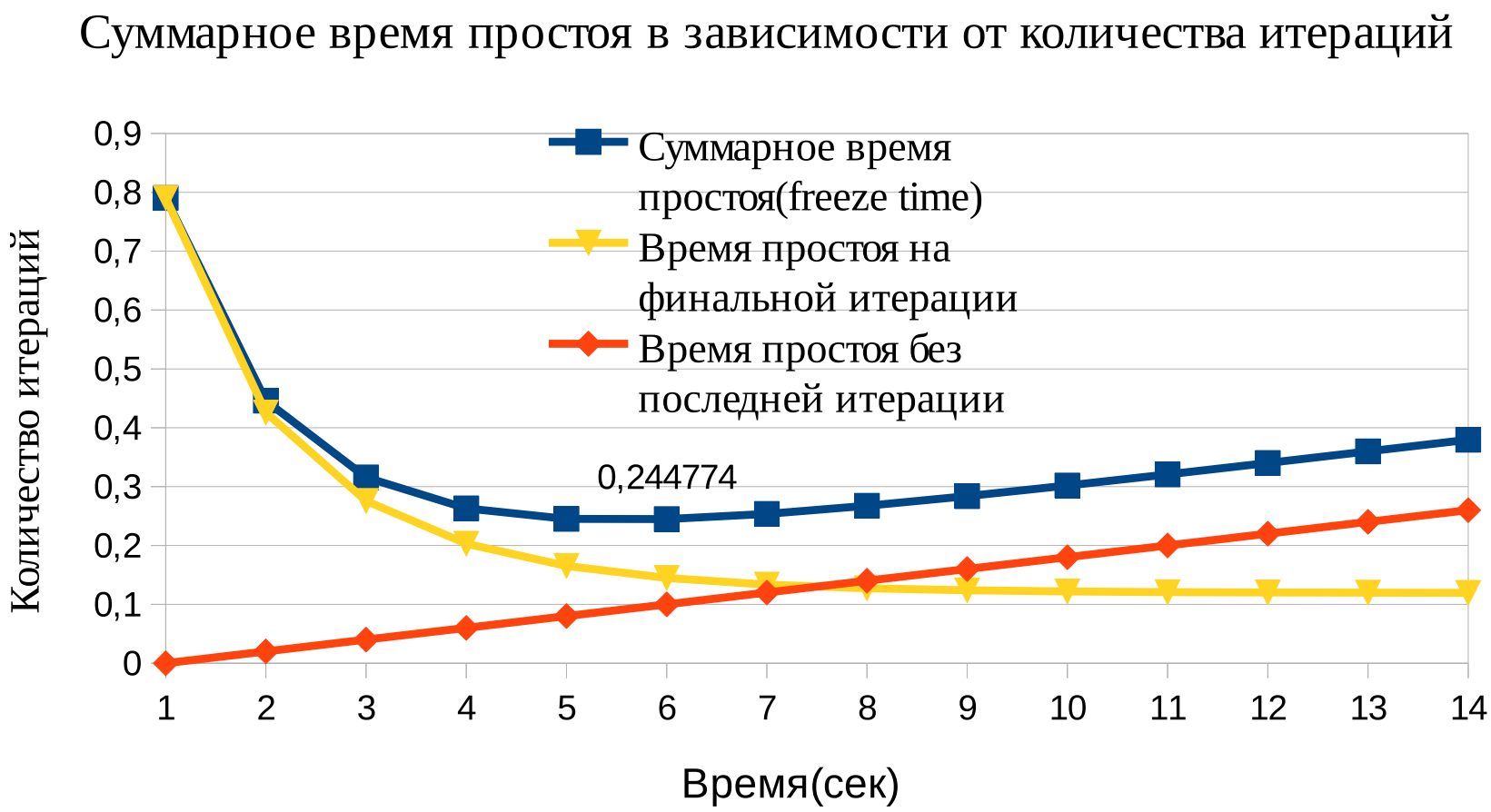

Рис. 3.3: Оптимизация миграции

Глава 4. Практическая часть

В рамках данной дипломной работы были решены три практических задачи:

1. Добавление поддержки Posix-таймеров в CRIU;
2. Добавление дедупликации итеративных снимков в CRIU;
3. Написание системы предсказания окончания миграции.

Первая задача является вводной задачей для изучения архитектуры проекта CRIU, вторая задача имеет непосредственную важность для использования оперативной памяти при создании и хранении итеративных снимков, что необходимо для ускорения процесса миграции. Третья задача заключалась в написании алгоритма на основе полученной математической модели с целью ее обоснования.

4.1. POSIX-Таймеры

В CRIU, в рамках подготовительного изучения системы и вводного задания была добавлена поддержка posix-таймеров. То-есть добавлена функциональность сохранения и восстановления таймеров стандарта POSIX.

POSIX-таймеры это интервальные таймеры поддерживаемые операционной системой Линукс. Они имеют ряд преимуществ перед обычными таймерами UNIX:

1. Процесс может иметь несколько таймеров.
2. Лучшая точность таймеров. Таймеры могут использовать значения с точностью до наносекунд.
3. Уведомление о завершении работы таймера может быть сделано либо с помощью любого произвольного сигнала (реального времени) или с помощью потоков. Для уведомление о завершении работы

таймера в обычных таймерах UNIX есть лишь ограниченный набор сигналов.

4. Таймеры POSIX.1b предоставляют поддержку различных типов часов, таких как CLOCK_REALTIME, CLOCK_MONOTONIC, и так далее, которые могут иметь различные источники с различным разрешением. Обычный таймер UNIX, с другой стороны, связан с системными часами.

Функциональность сохранения и восстановления posix-таймеров разрабатывалась с оглядкой на реализацию подобной функциональности для itimers(обычные интервальные таймеры), в которой, в процессе работы, были найдены и исправлены ошибки, и на основе функциональности для POSIX-часов. В начале CRIU при сохранении состояния дерева процессов, для каждого процесса, для объектов ядра для которых это необходимо, вызывает модули получения информации из файловой системы proc.

4.1.1. Использование файловой системы procfs

Файловая система procfs является основным используемым интерфейсом Для сохранения состояния posix-таймеров из пространства пользователя, весной 2013 года был принят патч [7] в ядро Линукс, отображающий основную информацию объектов struct k_itimer хранящих состояние posix-таймеров, с помощью интерфейса seq_file. Таким образом читая файл /proc/<pid>/timers получаем список таймеров используемых процессом. Для каждого posix-таймера, считывается уникальный номер таймера - ID, информация о способе прихода оповещения о срабатывании таймера и информация о часах с которыми связан таймер. Все это сохраняется в динамический список и CRIU переходит на следущий этап.

Патч в ядро Linux

Для отображения ClockID - уникального номера часов к которым привязан posix-таймер, необходимого для восстановления таймера, был

написан патч [8] в ядро Линукс. Так как предыдущая реализация /proc/<pid>/timers не содержала этой критичной информации и другие интерфейсы ядра не предоставляли к ней доступа, то полностью восстановить таймеры было не возможно. По этому было решено расширить функциональность ядра.

Пример содержимого файла, для процесса с двумя таймерами:

```
cat /proc/<pid>/timers
ID: 21
signal: 12/          (null)
notify: signal/pid.25907
ClockID: 1
ID: 10
signal: 10/00000000006062a0
notify: signal/pid.25907
ClockID: 0
```

4.1.2. Системные вызовы

После считывания основной информации о таймерах из файловой системы Procfs, используя интерфейс ptrace, CRIU встраивается в сохраняемый процесс. Список с информацией о posix-таймерах необходим для работы с ними из под контекста сохраняемого процесса, соответственно был добавлен перенос этих данных в память процесса используя аналогичные механизмы для других объектов. Далее зная какие есть таймеры можно используя системные вызовы timer_gettime() и timer_getoverrun() получить значения оставшегося времени до срабатывания таймера и количество итераций таймера, когда таймер не сработал из-за блокировки сигналов. После этого данные возвращаются в контекст CRIU, а контекст сохраняемого процесса чистится от инъекций памяти и приводится в свое состояние до подключения к нему CRIU. Все данные сохраняются в снимки с использованием формата Google Protobuf.

При восстановлении воссоздается дерево процессов и для каждого процесса по очереди восстанавливаются все per-process объекты. Для восстановления posix-таймеров используются системные вызовы работы с таймерами timer_create() и timer_settime(), с параметрами полученными при сохранении процесса, для воссоздания и запуска таймеров.

Основной сложностью при восстановлении posix-таймеров является то, что ядро Линукс не предоставляет интерфейса для восстановления таймера с определенным идентификационным номером, по этому предложенный алгоритм использует то что таймеры ядром создаются в порядке возрастания ID(timer_create()). То-есть, сохраненные таймеры сортируются в порядке возрастания ID, и CRIU пытается восстанавливать их в этом порядке. Пусть например наименьший ID = 5, тогда чтобы его восстановить создаются таймеры 1-4, а потом воссоздается пятый. В конце все лишние таймеры удаляются.

4.1.3. Проблема overrun

При восстановлении posix-таймеров найдена проблема восстановления overrun. То-есть не существует системного интерфейса чтобы во первых узнать точное количество итераций таймера когда сигналы заблокированы, во вторых нет интерфейса для восстановления этого значения, по этому для некоторых ситуаций одно измерение таймера попадающее на момент сохранения и восстановления может быть фальсифицировано, но далее таймер будет работать как и должен. Для исправления ситуации в будущем необходимо изменить интерфейсы ядра для работы с overrun.

4.1.4. Тестирование

Тестирование реализации поддержки таймеров стандарта POSIX проводилось с помощью специального теста из пакета ZDTM, который был написан для подобной функциональности реализованной в OpenVZ и адаптирован с учетом проблемы overrun.

4.2. Дедупликация

Для осуществления живой миграции в CRIU предусмотрен механизм создания итеративных снимков. При создании итеративных снимков памяти в CRIU, на каждой итерации накапливаются новые данные, то-есть суммарный объем занимаемый итеративными снимками все время увеличивается. Что по прохождении большого количества итераций может

привести к росту объема снимков на порядок. Так как живая миграция подразумевает максимальную оптимальность передачи, сохранения и восстановления, для хранения снимков предлагается использовать оперативную память. Но указанный рост объема памяти занимаемого снимками препятствует этому решению. Для решения проблемы нехватки памяти и вследствие не оптимальности процесса миграции был предложен и реализован алгоритм дедупликации итеративных снимков. Для осуществления этой функциональности использован API Sparce file.

4.2.1. Разряженный файл

Разряженный файл(Sparce file) - тип файла, который хранит данные в файловой системе более эффективно если в нем существуют последовательности подряд идущих нулевых байт [11]. Вместо таких последовательностей хранятся только их метаданные. При чтении все замененные блоки представляются как реальные блоки заполненные нулевыми байтами.

Большинство современных файловых систем поддерживают разряженные файлы, например: NTFS, ext4, tmpfs. В основном такие файлы используются для хранения образов дисков и снимков баз данных. Преимуществом использования разряженных файлов является эффективность использования памяти - место выделяется, только если оно действительно нужно. Минусом же является фрагментация файловой системы.

Используемым интерфейсом для работы с разряженными файлами является функция fallocate() [9]. Она реализована в операционной системе Линукс, и позволяет напрямую манипулировать пространством файла, выделять и освобождать пространство. Используя флаг FALLOC_FL_PUNCH_HOLE можно освобождать выбранный блок в файле.

4.2.2. Файловая система tmpfs

Это специальная файловая система во многих UNIX-подобных операционных системах. Позволяет монтировать файловую систему размещая ее в оперативной памяти вместо физического диска [11]. Применяется в

данной работе для хранения образов памяти, это позволяет убрать задержку записи на диск в процессе передачи памяти при миграции, что потенциально увеличивает скорость миграции на порядок учитывая характеристики скорости чтения/записи для современных дисков и оперативной памяти. Она полностью поддерживает функционал разряженных файлов.

4.2.3. Архитектура

В CRIU память сохраняется блоками pagemap entry - записями, которые образуются из зон виртуальной памяти(Virtual Memory Area), соответствующие записям страницы записываются в файл снимка подряд в порядке соответствующем порядку записей. Для прохода по данным записей CRIU вычитывает записи, и проматывает файл страниц на соответствующий кусок данных. Часть записей имеют пометку in_parent, то-есть, для их поиска нужно спустится в родительский снимок, для них в текущем снимке не хранится данных. Это те страницы которые не изменились с момента предыдущего снимка.

При сохранении очередного снимка, во-первых открывается дерево родительских образов. При поступлении записи с грязными страницами, вызывается рекурентная функция дедупликации, которая идет в родительский образ, находит все записи пересекающиеся с грязной записью, и если они не помечены in_parent, то запускает на них функцию освобождения блока. Дальше для найденных пересекающихся записей вызывается функция дедупликации, и так до изначального снимка. Таким образом для всех новых записей все старые записи будут очищены и память занимаемая ими отдана системе. Есть условие, что алгоритм, при использовании для итеративной миграции, должен быть применен на каждой итерации.

При восстановлении из итеративных снимков процесс спуска в родительский снимок уже реализован, остается только использовать на уже восстановленных блоках функцию освобождения блока.

Функция освобождения блока использует накопление, то-есть освобождает блок не сразу, а запоминает его, при приходе подряд идущих соседних блоков они запоминаются как один большой блок. Накопление

идет до 1Мб, чтобы одновременно получить хорошую производительность и эффективно очищать снимки для достижения минимального размера занимаемой памяти все время.

Используя данный алгоритм, так же очищаются записи которые были освобождены сохраняемой программой, так как очевидно они будут пройдены мимо алгоритмом, так как они не пересекаются или пересекаются только частично с записями в снимке-ребенке. При таком проходе мимо на них запускается рекурсивная функция дедупликации.

В результате был реализован алгоритм поиска старых страниц памяти встроенный в CRIU в процедуры снятия итеративного снимка и восстановления из снимков. Для этого были изменены функции чтения снимков на восстановлении и добавлена аналогичная функция в сохранении. В процессе реализации были найдены и исправлены ошибки в подсистеме работы с памятью, применены оптимизации.

4.2.4. Тестирование

Для всех компонентов функционала дедупликации были написаны тесты сохранности памяти при различных сценариях использования памяти и интегрированы в ZDTM. На основе системы непрерывной интеграции Jenkins, постоянно запущено автоматическое тестирование с случайно использующими память тестами.

Так же проведены тесты использования памяти в процессе миграции. Они показали эффективность использования памяти при использовании функции дедупликации, и обратную ситуацию иначе.

Методика заключалась в следующей последовательности действий:

1. монтируется файловая система tmpfs для хранения снимков на целевой виртуальной машине;

2. устанавливается тестовое приложение, активно загружающее RAM исходной виртуальной машины;

3. производится две итеративные миграции тестового приложения на вторую виртуальную машину, соответственно с включенной и выключенной дедупликацией;

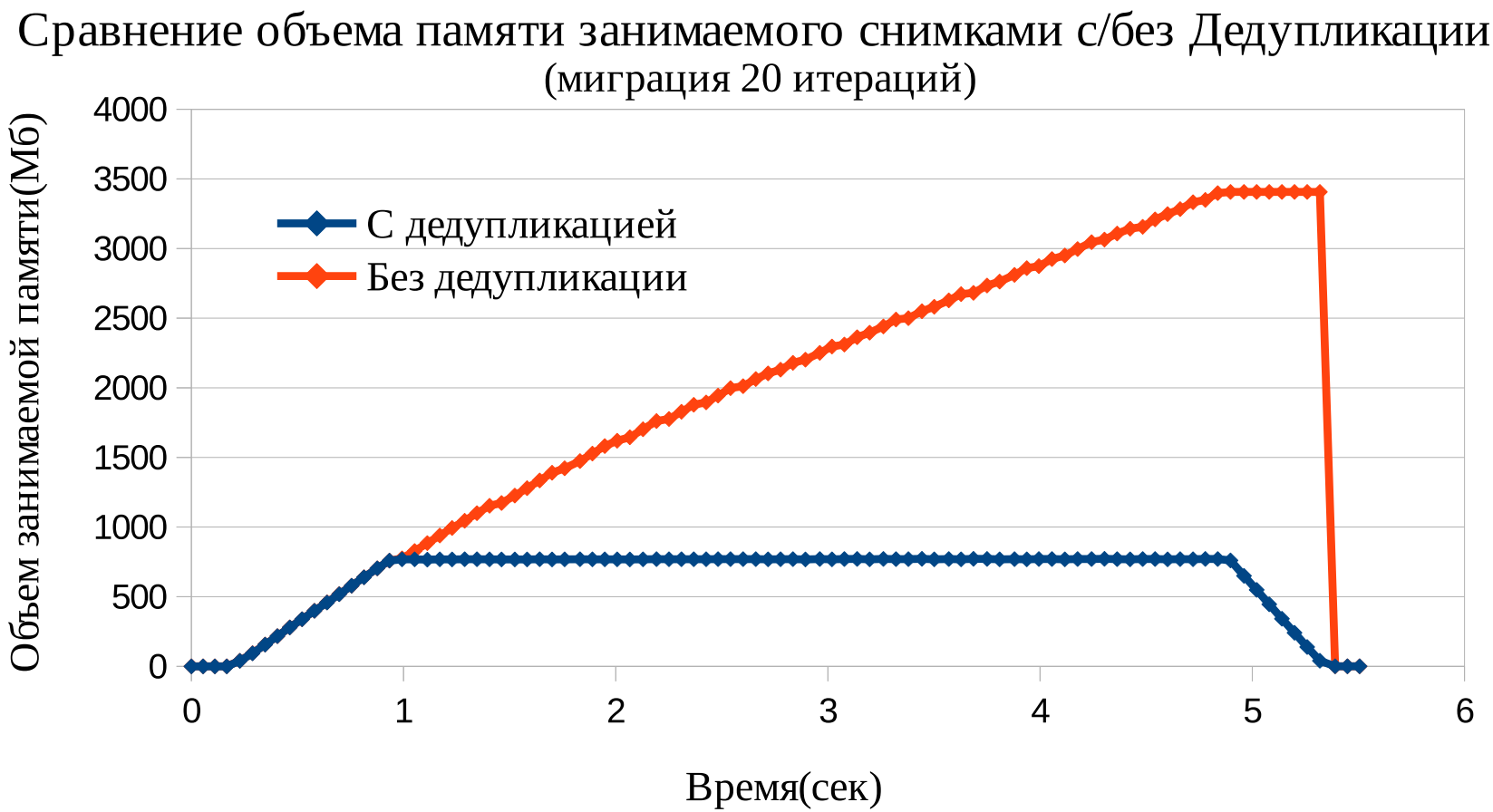

Рис. 4.1: Тестирование дедупликации

4. измеряется с помощью программы disk usage занимаемое образами пространство.

Получено два графика: график размера занимаемой образами памяти без дедупликации и график размера занимаемой памяти с дедупликацией снимков.

Из результатов тестирования видно, что при итеративной миграции без использования дедупликации памяти, занимаемое образами пространство в несколько больше чем при ее использовании.

Данный алгоритм, реализованный как компонент CRIU, показал теоретически предсказанную пространственную эффективность, то-есть позволил сократить суммарное использование памяти снимками до размера одного рабочего набора программы. Что осуществлено, не только во время создания снимков, но и во время восстановления, это можно заметить на конце графика “С дедупликацией” на пятой секунде, во время восстановления приложения происходит синхронное сокращение занимаемой снимками памяти, за счет уже восстановленных и, таким образом, ненужных блоков.

Тестирование дедупликации так же принесло пользу в аспекте проверки работы механизма сохранения памяти в снимки в CRIU. Были найдены ряд ошибок реализации, и исправления этих недостатков были внесены в код проекта.

4.3. Живая миграция

Был разработан алгоритм прогнозирования момента окончания итеративной миграции и реализован для CRIU с целью обоснования. В задаче прогнозирования можно выделить две подзадачи:

1. Мониторинг динамики насыщения рабочего набора процесса;
2. Оптимизация времени простоя при миграции, на основе аналитически проанализированных данных мониторинга.

4.3.1. Мониторинг

Существуют различные способы мониторинга используемой процессом памяти (например, утилиты ps, page-collect), но они не достаточно

точны и неприменимы для анализа памяти процесса в произвольный момент времени. Такие способы в основном дают возможность получить информацию об общем размере виртуальной памяти, используемой процессом, и о размере предоставленной физической памяти, но для анализа этого не достаточно.

Командой CRIU в апреле 2013г. в ядро Linux была добавлена функциональность, позволяющая отслеживать использование памяти процессом, называемая soft-dirty tracking. С помощью soft-dirty tracking, используя файловую систему procfs, можно, включить отслеживание памяти на уровне ядра системы. После этого можно получать битовую маску памяти процесса, проанализировав которую, выделить страницы памяти, изменившиеся при работе процесса и оставшиеся в том же состоянии, что и в момент включения опции.

Это позволяет проводить мониторинг насыщения рабочего набора, т.е. именно то, что необходимо для миграции. Изначально эта функциональность была добавлена для определения «грязной» памяти при создании итеративных снимков программ. Но ее можно также успешно использовать для подготовительного мониторинга.

4.3.2. Отслеживание грязной памяти

Отслеживание грязной памяти реализовано в ядре операционной системы Линукс, как один из интерфейсов файловой системы procfs [12]. Для включения отслеживания грязной памяти процесса достаточно записать 4 в файл /proc/<pid>/clear_refs. При этом ядро сбросит биты в метаданных страниц в состояние “чистая” страница, и также отметит все страницы процесса как защищенные от записи. Таким образом если процесс будет читать некоторую страницу своей памяти произойдет page-fault(страничная ошибка), который будет обработан ядром, и страница будет помечена как “грязная”. Произошедшие страничные ошибки будут обработаны без дополнительных задержек, так как данные страницы процесса уже содержится в памяти. По этому, данное отслеживание страниц это очень быстрый и эффективный механизм.

Далее для определения состояния грязной памяти необходимо прочитать файл /proc/<pid>/pagemap, и просматривать в нем состояние, так

называемых, soft-dirty bits. В CRIU для сбора данных о грязной памяти процесс останавливается, так как требуется получить консистентный снимок состояния. Для целей мониторинга же можно и нужно использовать не консистентное, а приближенное состояние грязной памяти. Это позволяет не влиять на работу исследуемого процесса и так как производится большое количество измерений, то отклонение от реальности при измерении будет минимально.

4.3.3. Alglib

После того как собраны данные об использовании грязной памяти процессом, данные аппроксимируются выбранной функцией $M(t) = at/(t+d)$;. Это осуществляется методом наименьших квадратов с помощью алгоритма Левенберга-Маквардта. В реализации системы для этого используется свободное программное обеспечение ALGLIB - библиотека для численного анализа.

4.3.4. p.HAUL

Для осуществления живой миграции существует надстройка над CRIU написанная на языке Python, это набор скриптов которые с использованием CRIU сервиса осуществляют миграцию программ. Для миграции достаточно запустить сервер p.haul на принимающей стороне, на исходном сервере запустить клиент и выбрать необходимую программу. Используемый алгоритм итеративной миграции аналогичен алгоритму используемому в OpenVZ.

4.3.5. Тестирование

Тестирование прогнозирования живой миграции проводилось между двумя виртуальными машинами в рамках одной хостовой системы. Каждая виртуальная машина имела выделенное ядро процессора, чтобы исключить взаимовлияние установок.

Было осуществлено тестирование скорости для передачи данных, определены задержка и коэффициент пропорциональности, согласно используемой модели. Была запущена копия исследуемой программы, которая

создает высокую нагрузку на оперативную память, записывая в произвольные регионы случайные данные. Модуль определения окончания миграции провел мониторинг насыщения рабочего набора процесса, осуществил предсказание поведения по модели используя полученные входные параметры затем построил график модели и определил оптимум итераций.

Далее была проведена итеративная миграция исследуемой программы с помощью p.haul и CRIU. Полученные данные о итерациях в этом процессе составили второй график. На рисунке4.2 приведены оба графика, видно что отклонение реальных данных от модели на протяжении итераций не велико. И при использовании результата предсказания оптимума количества итераций, можно осуществлять миграцию с меньшим влиянием на мигрируемый процесс, чем при использовании эвристических критериев.

Конечно в данном тестировании использовался модельный процесс над которым проводились эксперименты, в дальнейшей работе предполагается исследовать работу алгоритма на реальных программах, и соответствующим образом дорабатывать реализацию и модель.

Рис. 4.2: Сравнение модели и эксперимента

Заключение

В результате данной магистерской диссертации была изучена актуальная проблема живой миграции приложений, которая имеет высокую важность для современных компьютерных систем. Приведено сравнение инструментов сохранения и восстановления и обоснование выбора, для использования в миграции программ, системы CRIU.

Для оптимизации процесса миграции программ, а в частности решения проблемы использования оперативной памяти, был разработан алгоритм дедупликации итеративных снимков памяти. Он учитывает специфику механизма хранения и создания снимков памяти, используемого как в CRIU, так и в многих других системах сохранения и восстановления. Была осуществлена эффективная реализация алгоритма и добавлена в проект CRIU. Для данной функциональности было разработано покрывающее множество тестов. При прохождении тестирования была показана теоретически предсказанная эффективность и надежность алгоритма.

Была построена модель процесса живой миграции и на ее основе, выработан алгоритм определения окончания итераций, оптимизирующий суммарное влияние процесса миграции на мигрируемый процесс. Данный алгоритм реализован в качестве прототипа. Было проведено тестирование прототипа алгоритма на тестовом стенде, в качестве мигрируемой программы был взят тестовый процесс активно использующий память. Сравнение модели и реального процесса миграции показывает, что хотя и происходят отклонения от модели, предсказание количества итераций помогает уменьшить время простоя мигрируемого процесса.

В дальнейшем планируется протестировать алгоритмы на реальных приложениях и уточнить модель и усовершенствовать реализацию, для использования на практике.

Литература

1. Linus Torvalds, Andrew Morton; Merge branch 'akpm' (aka "Andrew's patch-bomb, take two") // Linux Kernel Mailing List, 4 Apr. 2012.

2. Paul H Hargrove, Jason C Duell; Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters // Journal of Physics Conference Series, volume 46:494–499, 2006.

3. Jason C Duell; The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart // Berkeley Lab Technical Report, 17 pages, December, 2002.

4. Jason Ansel, Kapil Arya and Gene Cooperman; DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop // 23rd IEEE International Parallel and Distributed Processing Symposium, 12 pages, May, 2009.

5. Michael Rieker, Jason Ansel; Transparent user-level checkpointing for the Native POSIX Thread Library for Linux // International Conference on Parallel and Distributed Processing Techniques and Applications, 492–498, 2006.

6. Oren Laadan, Serge E. Hallyn; Linux-CR: Transparent Application Checkpoint-Restart in Linux // Proceedings of the 12th Annual Linux Symposium, 16 pages, Jul. 2010.

7. Pavel Emelyanov; posix timers: Extend kernel API to report more info about timers // Linux Kernel Mailing List [Электронный ресурс], `https://lkml.org/lkml/2013/2/14/273`, Feb. 2013.

8. Pavel Tikhomirov; posix-timers: Show clock ID in proc file // Linux Kernel Mailing List [Электронный ресурс], `https://lkml.org/lkml/2013/5/17/276`, May, 2013.

9. fallocate(2) // Linux Programmer's Manual [Электронный ресурс], http://www.kernel.org/doc/man-pages/, Apr. 2013.

10. Kaj Madsen, Hans Bruun, Ole Tingleff; The Levenberg–Marquardt Method // Methods for non-linear least squares problems, 24–29, 2nd Edition, Apr. 2004.

11. Daniel P. Bovet, Marco Cesati; Understanding the Linux Kernel // O'Reilly Media, Inc., 3nd Edition, Nov. 2005.

12. Кирилл Колышкин, Павел Емельянов; CRIU: больше, чем живая миграция для Linux контейнеров // Yet Another Conference, 2012.

13. Андрей Вагин; CRIU – новый амбициозный проект для сохранения и восстановления процессов // [Электронный ресурс] http://habrahabr.ru/post/148413/, Июль 2012.

14. Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, Anthony Liguori; kvm: the Linux Virtual Machine Monitor // Proceedings of the Linux Symposium, June 27th–30th, 2007.

15. Michael Kerrisk; LCE: Checkpoint/restore in user space: are we there yet? // Linux Weekly News [Электронный ресурс], http://lwn.net/Articles/525675/, Nov. 20, 2012.

16. Stuart Hacking, Benoıt Hudzia; Improving the live migration process of large enterprise applications // Virtualization technologies in distributed computing, 51–58, 2009.

17. Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W. Moore and Andy Hopper; Predicting the Performance of Virtual Machine Migration // University of Cambridge Computer Laboratory, 10 pages, 2010.

18. Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, Eric Roman; Optimized Pre-Copy Live Migration for Memory Intensive Applications // Proceedings of International Conference for High Performance Computing, 11 pages, 2011.

19. Тихомиров П.О.; Live-миграция приложений в CRIU // Труды 55-й научной конференции МФТИ, 2013.