Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Зайберт Валерия Сергеевна

Разработка модуля помехоустойчивого кодирования для системы хранения данных

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Луцив Д. В.

Рецензент:
разработчик исследовательской лаборатории "Raidix" Маров А. В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Zaibert Valeriia

Development of erasure coding module for data storage systems

Graduation Thesis

Admitted for defence.
Head of the chair:
Professor Andrey Terekhov

Scientific supervisor:
Senior Lecturer Dmitry Luciv

Reviewer:
Developer of RAIDIX research lab Aleksei Marov

Saint-Petersburg
2015

Оглавление

Введение

Системы хранения данных (СХД) — это комплексное решение по организации надежного хранения информационных ресурсов, а также предоставления гарантированного и быстрого доступа к ним. На сегодняшний день различные СХД используются во всех областях, работающих с большим потоком информации, будь это онлайн-сервисы, медиа-индустрия, медицинские учреждения и так далее.

СХД являются комплексным программно-аппаратным решением. Для обеспечения возможности хранения большого объема информации в систему подключено множество дисков, возможно, разнесенных по удаленным узлам сети. Для логического объединения нескольких дисков используют семейство технологий RAID (redundant array of independent disks)[6]. Конкретную технологию выбирают в зависимости от требований к надежности и скорости работы системы и ее размеров.

Изначально, чтобы предотвратить поломки только на одном диске системы, использовалось зеркалирование, то есть хранение нескольких копий данных на разных носителях. Однако этот подход сильно увеличивает стоимость системы за счет резкого увеличения избыточности данных, а значит, и увеличения числа носителей.

Другой поход к решению проблемы надежности хранения — использование помехоустойчивого кодирования [20]. Этот подход изначально использовался в телекоммуникациях и заключается в следующем. Есть источник, который передает данные по зашумленному каналу, и приемник, который прослушивает этот канал и пытается эти данные извлечь.

Так как при передаче данных часть информации может быть утрачена или изменена, приемник должен уметь восстанавливать эти данные. Для этого перед передачей данные кодируются, чтобы соответствовать определенным условиям. Если декодер со стороны приемника видит, что данные не соответствуют этим условиям, то он понимает, что данные необходимо восстановить, и делает это.

Зашумленным каналом может быть как канал связи, так и носители данных (диски), которые могут отказать во время эксплуатации под воздействием внешних факторов.

Чтобы не работать с отдельными битами данных, в качестве минимальной единицы информации обычно рассматривают кодовые слова. При этом в качестве кодовых слов удобно рассматривать элементы полей Галуа[16]. Размер полей жестко не ограничен, однако часто работают с полями $GF(2^8)$. То есть, одно кодовое слово — это один элемент поля, который можно закодировать одним байтом.

Ошибки в помехоустойчивом кодировании бывают двух типов. Первый тип ошибок — стирания, их еще называют отказами или просто ошибками. Это те ошибки, о местоположении которых системе известно заранее. Например, заранее известно, какой диск сломался и был заменен. Второй тип ошибок — скрытые повреждения

(Silent Data Corruption, SDC). Про наличие этих ошибок мы заранее не знаем, и их требуется найти.

Однако не все виды помехоустойчивого кодирования способны искать SDC. Иногда от этой возможности отказываются умышленно, чтобы не замедлять работу системы. Если отказаться от поиска SDC, то можно уменьшить количество чтений с дисков. Предположим, что мы отказались от поиска SDC или этот поиск и вовсе невозможен. Пусть применяемый нами метод помехоустойчивого кодирования позволяет нам восстановить информацию с k дисков. Тогда при наличии в системе медленных дисков и при быстрой реализации восстановления данных можно восстановить информацию с отстающих носителей не дожидаясь ответа от них. Подобный подход может применяться и в распределенной системе, когда в роли носителей выступают удаленные узлы. Связь с некоторыми узлами может быть медленной и можно восстановить части запрашиваемой информации, хранящиеся на медленных узлах.

Один из самых распространенных видов помехоустойчивых кодов — это коды Рида-Соломона, предложенные в 1960 году сотрудниками лаборатории Массачуссетского технологического института Ирвином Ридом и Густавом Соломоном. Впервые эти коды были применены в 1982 году в производстве компакт-дисков.

В этой работе будет рассмотрено применение только кодов Рида-Соломона к задаче хранения данных. Кроме них существуют LDPC коды[1], LRC коды[14], турбокоды[4] и множество других. Многие из них хуже подходят для задач хранения данных из-за отсутствия возможности поиска скрытых повреждений, медленных операций кодирования или декодирования или других подобных ограничений.

Постановка задачи

Целью этой работы было реализовать возможность помехоустойчивого кодирования для системы хранения данных компании «Raidix»[19] с высоким уровнем надежности. Для этого было необходимо выполнить следующие задачи.

1. Рассмотреть различные подходы к помехоустойчивому кодированию и использованию кодов Рида-Соломона.
2. Провести обзор существующих реализаций.
3. Изучить и реализовать алгоритмы кодирования, декодирования и поиска скрытых повреждений.
4. Разработать модуль, организующий работу с этими алгоритмами
5. Протестировать модуль и внедрить в ядро Linux

1. Теоретическое введение

1.1. Поля Галуа

Поле Галуа $GF(p^q)$ — конечное поле, состоящее из p^q элементов и представимое в виде $GF(p^q) = \mathbb{Z}_p[x]/\langle f(x)\rangle$, где p — простое число, а $f(x)$ — неприводимый многочлен степени q с коэффициентами из $\mathbb{Z}_p$ [16].

Над полем определена операция сложения двух элементов $a(x)$ и $b(x)$ как операция сложения многочленов, коэффициенты которых берутся по модулю p. Операция умножения $a(x)b(x)$ определена как умножение многочленов по модулю $f(x)$ с коэффициентами по модулю p. Обратный элемент для элемента поля $a(x)$ — такой элемент $a^{-1}(x)$, что $a(x)a^{-1}(x) = 1$, где 1 — нейтральный элемент поля относительно умножения.

Рассмотрим поле $GF(2^q)$. Это поле удобно использовать в помехоустойчивом кодировании, так как информация представляется в виде 0 и 1. Тогда каждому элементу поля можно однозначно сопоставить вектор длины q с элементами из $\{0, 1\}$. Например, при $q = 6$, многочлен $x^5 + x^3 + 1$ представим в виде (101001). Часто принимают $q = 8$, в этом случае один элемент поля можно закодировать одним байтом.

В этих условиях операция сложения элементов поля может быть представлена как операция *xor* двух байтовых переменных.

1.2. Коды Рида-Соломона

1.2.1. Классический подход к кодированию

Один из распространенных способов кодирования — коды Рида-Соломона, являющиеся частным случаем БЧХ кодов. Коды Рида-Соломона — циклические коды, позволяющие находить и исправлять ошибки в данных. Рассмотрим систематические коды Рида-Соломона. Пусть пересылаемое сообщение состоит из n информационных кодовых слов. Для возможности восстановления ошибок к этим информационным блокам добавляются m проверочных, называемых *синдромами*. В этом случае кодирование описывается следующим образом [16].

Берется *порождающий полином* кодов Рида-Соломона — полином минимальной степени над полем $GF(2^q)$, корнями которого являются идущие подряд по возрастанию степени примитивного элемента поля. Пусть a — примитивный элемент поля. Тогда порождающий полином для кода, исправляющего $\lfloor m/2 \rfloor$ скрытых повреждений, выглядит следующим образом

$$g(x) = (x + a)(x + a^2)(x + a^{m-1}) \tag{1}$$

Пусть имеется n кодовых слов $(D_0, D_1, \ldots, D_{n-1})$, которые мы хотим передать по за-

шумленному каналу, при этом $D_i \in GF(2^q)$. Для этого набора кодовых слов берется полином

$$G(x) = (D_0 x^{n-1} + D_1 x^{n-2} + \cdots + D_{n-2} x + D_{n-1}) x^m \tag{2}$$

Для кодирования этот полином делится с остатком на порождающий полином $g(x)$

$$G(x) = g(x)Q(x) + R(x) \tag{3}$$

Тогда коэффициенты полинома $R(x) = S_0 x^{m-1} + S_1 x^{m-2} + \cdots + S_{m-1}$ будут синдромами. Итоговый полином, коэффициенты которого передаются по каналу имеет вид

$$M(x) = G(x) + R(x) \tag{4}$$

То есть, по каналу связи передается сообщение вида $(D_0, D_1, \ldots, D_{n-1}, S_0, \ldots, S_{m-1})$. Для полинома $M(x)$ выполняется $M(1) = 0, M(a) = 0, \ldots, M(a^{m-1}) = 0$ Если произошли какие-то повреждения, то посчитав значения нового искаженного полинома $\tilde{M}(X)$ в корнях полинома $M(X)$, образующего код, получим

$$\tilde{M}(1) = C_0, \tilde{M}(a) = C_1, \ldots, \tilde{M}(a^{m-1}) = C_{m-1} \tag{5}$$

По этим значениям строится полином локаторов ошибок, корни которого помогут определить места повреждений.

1.2.2. Упрощенный метод кодирования

На практике часто используют более простой метод кодирования, отказываясь от возможности поиска скрытых повреждений [9]. Кодирование осуществляется путём умножения *матрицы кодирования* H на вектор $D = (D_0, D_1..D_{n-1})$, состоящий из информационных кодовых слов. Все кодовые слова и элементы матрицы являются элементами поля $GF(2^8)$. В итоге умножения получается вектор, состоящий из синдромов $S = (S_0, S_1, ...S_{m-1})$.

$$HD = S \tag{6}$$

Для декодирования рассматривают *расширенную матрицу кодирования*, первые n строк которой — единичная матрица I. Тогда из (6) следует:

$$\begin{pmatrix} I \\ H \end{pmatrix} D = \begin{pmatrix} D \\ S \end{pmatrix} \tag{7}$$

Пусть произошло допустимое количество ошибок, то есть хотя бы n блоков остались верными. Тогда из левой части уравнения (7) и из матрицы в правой части возьмем только те n строк, которые соответствуют безошибочным блокам. Обозначим получившуюся в левой части подматрицу расширенной матрицы кодирования

как $\tilde{H}$, а вектора в правой части как $\tilde{D}$ и $\tilde{S}$ соответственно:

$$\tilde{H}D = \begin{pmatrix} \tilde{D} \\ \tilde{S} \end{pmatrix} \tag{8}$$

Матрица $\tilde{H}$ — квадратная. Предположим, что она обратима. Тогда

$$D = \tilde{H}^{-1} \begin{pmatrix} \tilde{D} \\ \tilde{S} \end{pmatrix} \tag{9}$$

Таким образом мы сможем найти изначальный вектор D, а значит, и истинные значения всех информационных блоков, если количество ошибок не больше m. Получившуюся матрицу $\tilde{H}^{-1}$ называют *матрицей декодирования.* Далее рассмотрим наиболее часто используемые матрицы кодирования.

1.2.3. Матрица Вандермонда

Одна из встречаемых в литературе матриц кодирования — это матрица Вандермонда[8][12]. Пусть мы работаем в поле $GF(2^8)$, a — примитивный элемент в этом поле ($a = 00000010$). Тогда матрица кодирования представляется следующим образом:

$$H = \begin{pmatrix} 1 & \cdots & 1 & 1 \\ a^{n-1} & \cdots & a & 1 \\ a^{2(n-1)} & \cdots & a^2 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ a^{(m-1)(n-1)} & \cdots & a^{m-1} & 1 \end{pmatrix} \tag{10}$$

В дальнейшем будем обозначать такую матрицу Вандермонда размером $m \times n$ как $W_{m \times n}$. Использование матрицы Вандермонда очень удобно при кодировании, потому что можно уменьшить количество операций сложения и умножения, используя схему Горнера. Рассмотрим вычисление i-ой контрольной суммы:

$$\begin{aligned} S_i = a^{(n-1)i} D_0 + \cdots + a^{2i} D_{n-3} + a^i D_{n-2} + D_{n-1} = \\ = \Big(\big(\ldots (D_0 a^i + D_1) a^i + D_2 \big) a^i + \ldots \big) a^i + D_{n-2} \Big) a^i + D_{n-1} \end{aligned} \tag{11}$$

Следовательно, каждую сумму можно вычислить всего за n операций умножения и $n - 1$ сложения. Ранее мы делали предположение, что матрица $\tilde{H}$, построенная по расширенной матрице кодирования — обратима, однако, если рассматривать матрицу Вандермонда в качестве матрицы кодирования, то не для каждого набора сбойных блоков можно построить обратимую матрицу. Таким образом, существуют комбинации ошибок, которые мы не сможем исправить.

Однако матрица Вандермонда обратима, значит, если взять $\tilde{H} = H$, то данные

можно восстановить. Это можно сделать тогда, когда не отказал ни один из синдромов. Также при маых размерностях матрицы Вандермонда (при $m = 3$) матрица $\tilde{H}$ всегда оказывается обратимой.

1.2.4. Матрица Коши

Еще один пример матриц, использующихся в качестве матрицы кодирования — это матрица Коши [11].

Для этой матрицы доказано, что любая ее квадратная подматрица обратима [15]. Следовательно, если взять матрицу Коши как матрицу кодирования, то матрица из произвольных n строк расширенной матрицы кодирования будет обратима. Значит, любую комбинацию ошибок можно исправить, если их число допустимо.

2. Подготовка данных к обработке

Информация, записываемая на СХД, разбивается на блоки равного размера, например, по 4Кб. Для каждых n информационных блоков расчитывается m контрольных сумм, которые записываются в дополнительные m проверочных блоков в конце. Полученные $n + m$ блоков образуют *страйп*, началом которого всегда является первый информационный диск. Они распределяются по дискам: каждый блок попадает на свой диск. Страйпы размещают со сдвигом, как показано на рисунке 1. Чтение с дисков также осуществляется страйпами, которые потом обрабатываются системой, и нужные данные передаются пользователю.

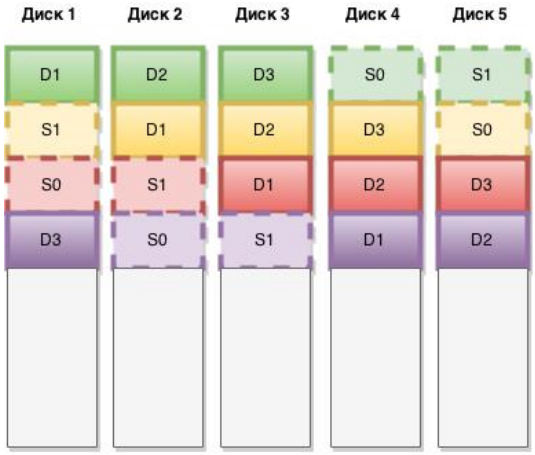


Рис. 1: Расположение страйпов по диска системы. $n = 3, m = 2$

На вход алгоритмам кодирования, декодирования и поиска скрытых повреждений будут подаваться отдельные страйпы. Для этого каждый страйп описывается структурой, в которой хранится информация о количестве каждого вида блоков, ссылка на матрицу декодирования, указатель на участок памяти, в которую считали страйп.

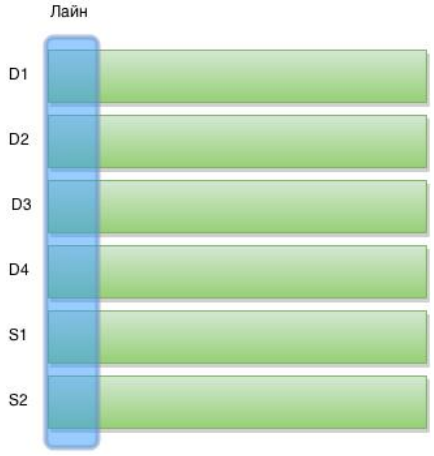


Рис. 2: Лайн. $n = 4, m = 2$

За одну итерацию алгоритмы обрабатывают массив из $m + n$ кодовых слов, по од-

ному кодовому слову с каждого блока. Назовем этот набор *лайном*. Если расположить блоки друг под другом (рис. 2), то вертикальные срезы и есть лайны.

Обрабатывать 4Кб данных по одному кодовому слову, то есть по одному байту за итерацию — очень долго. Поскольку операции на каждой итерации алгоритмов однотипны и не зависят от результатов выполнения других итераций, можно использовать векторные инструкции процессора SSE или AVX, позволяющие за одну итерацию обработать лайн шириной в 16 или 32 байта соответственно. Это позволит в разы ускорить кодирование или декодирование данных. Будем использовать SSE4.2 и AVX2, потому что они содержат инструкции, необходимые нам для реализации операций в полях Галуа. Все что далее будет сказано про SSE, верно и для AVX, если учесть то, что их размер отличается в два раза.

3. Существующие открытые библиотеки помехоустойчивого кодирования

3.1. Описание библиотек

Существует множество реализаций алгоритмов кодирования Рида-Соломона. Большинство из них используются в системах хранения данных. Нами было изучено несколько открытых библиотек, реализующих вышеописанные алгоритмы. Все рассмотренные библиотеки реализуют упрощенный способ кодирования Рида-Соломона, а значит, не способны осуществлять поиск скрытых повреждений. Для ускорения вычислений в функциях умножения матриц на вектора авторы этих библиотек используют SSE2 инструкции. Приведем краткое описание каждой из библиотек.

FECpp

FECpp[13] реализована на языке C++. Разработана на основе продукта fec от Луиги Риззо. Библиотека использует матрицу Вандермонда для кодирования. Если матрица декодирования оказалась необратимой, то библиотека порождает исключение, и восстановление оказывается невозможным [6].

При расчете синдромов необходимо в качестве параметра передавать самостоятельно реализованную функцию-обработчик, которая и сохранит полученные контрольные суммы в конец страйпа. Иначе суммы нигде не сохранятся.

Перед вызовом функции декодирования необходимо создать структуру данных типа map, где в качестве ключа будет номер блока, а в качестве значения — его содержимое. На вход алгоритму декодирования опять требуется передать функцию обработчик, которая будет записывать восстановленные блоки в соответствующие ячейки страйпа. Все функции-обработчики пользователь библиотеки должен реализовать сам.

Из-за этих особенностей работы с библиотекой она кажется неудобной и очень непривычной. Кроме того, не указано максимальное число синдромов и информационных блоков в страйпе.

Zfec

Библиотека, реализованная на C, с возможностью использования из командной строки и имеющая API для языков C, Python и Haskell[5]. Она также основывается на продукте fec, поэтому использует теже формулы для расчетов, что и FECpp.

К сожалению, нет подробной документации. Способ правильного использования функций приходится брать из примеров. Для декодирования требуется создание массива с номерами правильных блоков и двух вспомогательных массивов ссылок на це-

лые блоки и утраченные.

Holostor

Holostor[2] также реализована на языке C++. Библиотека очень проста в использовании. Создается сессия (аналог дескриптора страйпа) с параметрами страйпа (число дисков, число синдромов, размер блока). В качестве матрицы кодирования используется матрица Коши. В функцию кодирования достаточно передать указатель на сессию и страйп. В функцию декодирования помимо этого необходимо передать маску отказавших дисков. В конце важно не забыть закрыть сессию.

Библиотека имеет достаточно подробную документацию. Из минусов стоит отметить, что разработчиками библиотеки заявлено максимальное количество дисков 17, а синдромов 4. Однако количество дисков, на которых получилось запустить функции библиотеки, несколько меньше (только 15).

Jerasure

Jerasure[7] — библиотека на языке C от Джеймса Планка. Состоит из двух отдельных библиотек: одной для реализации операций над полем и второй для реализации непосредственно кодов Рида-Соломона.

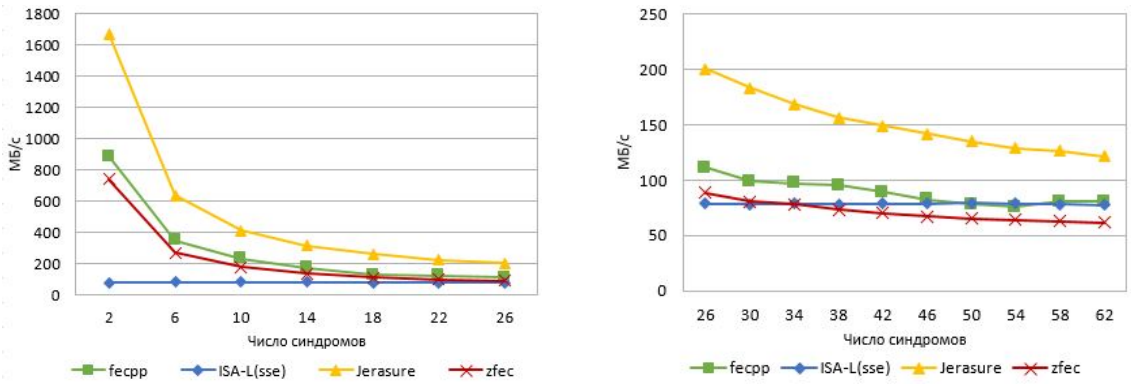
Библиотека содержит подробную документацию, включающую как общие принципы расчета синдромов и восстановления блоков, так и подробный мануал по использованию функций библиотеки.

Jerasure допускает использование различных полей $GF(2^w)$, где $w \in \{8, 16, 32\}$ и задается пользователем. Создание матриц кодирования и декодирования вынесены в отдельные функции, что позволяет переиспользовать матрицу кодирования для многих тестов и увеличить производительность кодирования.

Библиотека поддерживает возможность использования матрицы Коши для кодирования/декодирования данных и предоставляет удобный интерфейс вывода отладочной информации на экран.

ISA-L

ISA-L[3] — библиотека от Intel, написанная на C и ассемеблере, с хорошей документацией, с возможностью ручного или автоматического переключения между наборами SSE и AVX инструкций. ISA-L также позволяет использовать матрицу Коши вместо матрицы Вандермонда. Создание матрицы кодирования тоже вынесено в отдельную функцию, но матрицу декодирования пользователь должен составлять сам, однако предоставляется функция обращения матрицы.



(a) $2 \leq m \leq 26$ (b) $26 \leq m \leq 96$

Рис. 3: График зависимости скорости кодирования от числа синдромов при $n = 96$

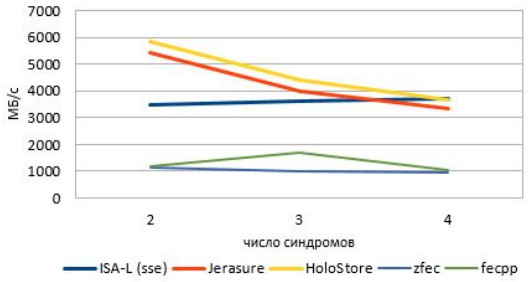


Рис. 4: График зависмости скорости кодирования от числа синдромов при $n = 6$

3.2. Сравнение библиотек

Недостатком этих библиотек является отсутствие возможности поиска срытых повреждений, которые могут встречаться в СХД. Кроме того, первые две библиотеки не гарантируют нам восстановление данных на дисках при некоторых отказах, с которыми могли бы справиться их аналоги. Библиотеки ISA-L и Jerasure часто можно встретить в открытых реализациях СХД, например, CEPH[10] и Sheepdog[11], поэтому мы уделили особое внимание им при тестировании.

Нами было написано тестовое окружение для каждой из библиотек с учётом особенностей их API. На графиках (рис. 3 и 4) показана зависимость производительности от числа синдромов m при фиксированном $n = 96$. Для каждого значения числа m запускался набор из 1000 тестов. Каждый тест рассчитывал синдромы, затем случайным образом выбирал m блоков и помечал их как отказавшие. Затем замерялась скорость построения матрицы декодирования и скорость обработки одного страйпа алгоритмом декодирования.

Для каждого набора тестов отбрасывалось 5% минимальных и максимальных результатов, по оставшимся значениям бралось среднее, которое и показано на графиках.

Тестирование проводилось на машине со следующими параметрами:

- OS: Ubuntu 12.04 LTS
- OS type: 64-bit
- Processor: Intel® CoreTM i3-4000M CPU @ 2.40GHz × 4
- Memory: 3.8 GB

Несмотря на то, что ISA-L часто используется в СХД, она показала относительно низкую скорость декодирования. Изучив ее подробнее, мы выяснили, что ее узкое место — обращение матрицы, реализованное по методу Жордана-Гаусса. Обращение матрицы можно векторизовать и получить заметный прирост производительности.

4. Реализация основных алгоритмов

4.1. Кодирование

В нашем методе кодирования мы постараемся совместить классический подход к кодам Рида-Соломона и упрощенный метод кодирования с использованием матриц кодирования и декодирования [12]. Для этого распишем полином $M(x)$ из (4):

$$M(x) = D_0 x^{m+n-1} + D_1 x^{m+n-2} + \cdots + D_{n-1} x^m + S_0 x^{m-1} + \cdots + S_{m-1} \tag{12}$$

Используя (12), можно переписать условие (5) в матричном виде

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ a^{m+n-1} & a^{m+n-2} & \dots & a & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a^{m(m+n-1)} & a^{m(m+n-2)} & \dots & a^m & 1 \end{pmatrix} \begin{pmatrix} D_0 \\ \vdots \\ D_{n-1} \\ S_0 \\ \vdots \\ S_{m-1} \end{pmatrix} = \begin{pmatrix} C_0 \\ \vdots \\ C_{m-1} \end{pmatrix} \tag{13}$$

Обозначим $N = m + n$, $D = (D_0, D_1, \dots, D_{n-1})$, $S = (S_0, S_1, \dots, S_{m-1})$ и заметим, что матрица в левой части — матрица Вандермонда. Тогда

$$W_{m \times N} \begin{pmatrix} D \\ S \end{pmatrix} = \begin{pmatrix} C_0 \\ \vdots \\ C_{m-1} \end{pmatrix} \tag{14}$$

Значит, необходимо выбрать такую матрицу кодирования, чтобы посчитанные с ее помощью синдромы S удовлетворяли следующему требованию

$$W_{m \times N} \begin{pmatrix} D \\ S \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \tag{15}$$

В дальнейшем элементы вектора S, которые хранятся на дисках, будем называть синдромами, а значения $C_0, \dots, C_{m-1}$, полученные в правой части — контрольными суммами.

Рассмотрим i-ую строку данного соотношения:

$$a^{i(N-1)} D_0 + \cdots + a^{im} D_{n-1} + a^{i(m-1)} S_0 + \cdots + a^0 S_{m-1} = 0. \tag{16}$$

Напомним, что в качестве операции сложения в нашем поле используется xor.

Тогда

$$a^{i(N-1)}D_0 + \cdots + a^{im}D_{n-1} = a^{i(m-1)}S_0 + \cdots + a^0 S_{m-1} \tag{17}$$

Вынесем a^m за скобки:

$$a^m(a^{i(n-1)}D_0 + \cdots + a^0 D_{n-1}) = a^{i(m-1)}S_0 + \cdots + a^0 S_{m-1}. \tag{18}$$

Перепишем снова в матричном виде:

$$([a^m, \ldots, a^m]I_m)W_{m\times n}D = W_{m\times m}S, \tag{19}$$

Здесь I_m — еденичная матрица размером $m \times m$. Матрица $W_{m\times m}$ является обратимой, а значит

$$W^{-1}_{m\times m}([a^m, \ldots, a^m]I_m)W_{m\times n}D = S. \tag{20}$$

Таким образом, матрица кодирования $H = W^{-1}_{m\times m}a^m_{m\times m}W_{m\times n}$.

4.2. Группировка итераций

Алгоритм кодирования принимает на вход страйп и рассчитывает синдромы лайн за лайном, где ширина лайна равна размеру SSE регистра. Обработка каждого лайна заключается в умножении матрицы кодирования на вектор данных. Элементы матрицы кодирования занимают 1 байт, но для алгоритма быстрого умножения [10] однобайтовый элемент матрицы кодирования представляется как две SSE-переменные.

Есть два очевидных способа реализовать умножение матрицы на вектор. Первый способ заключается в последовательном обходе строк матрицы и умножении каждой из них на вектор данных. Во втором способе сначала полагаем все синдромы равными нулю, потом берем элемент вектора данных, умножаем на элементы из соответствующего столбца матрицы кодирования и результат добавляется к соответствующему синдрому.

С точки зрения количества операций оба подхода одинаковы, но ни один из них не использует регистры максимально полезно при большом количестве синдромов или информационных дисков. В первом случае следует поместить в регистры строку матрицы, и не вытеснять ее, пока она не умножится на весь столбец данных. Но, если информационных дисков слишком много, то есть строка матрицы слишком длинная, то она не поместится на регистры. Преимущество второго способа в том, что один информационный блок будет считываться только один раз. На регистрах помимо значения из одного информационного блока можно было бы хранить текущие значения синдромов, но их может оказаться слишком много.

Нами был предложен другой подход к умножению матрицы на вектор, являющийся модификацией второго способа. Будем вычислять контрольные суммы группами.

По возможности будем группировать расчет синдромов по 8 в одной группе. Первые 8 синдромов попадут в первую группу, вторые 8 во вторую и так далее. Если в какой-то момент остается меньше 8 синдромов, то создаются группы из оставшихся размером 4, 2 или 1, в зависимости от числа оставшихся синдромов.

Для каждой группы синдромов размером $q \in \{8, 4, 2, 1\}$ берутся соответствующие q строк матрицы кодирования, заводится q переменных для каждого из синдромов, которые инициализируются нулями и способны поместиться на регистры. Далее идет вычисление синдромов, как это описано во втором способе. Вычисления повторяются, пока все синдромы не будут посчитаны.

Реализованный метод позволяет ускорить операцию умножения матрицы на вектор на 35% по сравнению с обычным подходом.

4.3. Декодирование

4.3.1. Векторное обращение матрицы

В описанных библиотеках обращение матрицы осуществляется методом Жордана-Гаусса. При этом обращение не использует векторные инструкции процессора, как это сделано для реализации умножения матриц на вектора данных.

Предлагается реализация алгоритма обращения матрицы методом Жордана-Гаусса с использованием векторных операций SSE. Для её реализации берется матрица, которую надо обратить, и строки расширяются нулями так, чтобы размер был кратен размерам SSE регистра, то есть 16 байтам. Затем осуществляется метод Жордана-Гаусса, где все операции со строками производятся не по одному элементу строки, а сразу с шестнадцатью, за счёт SSE функций.

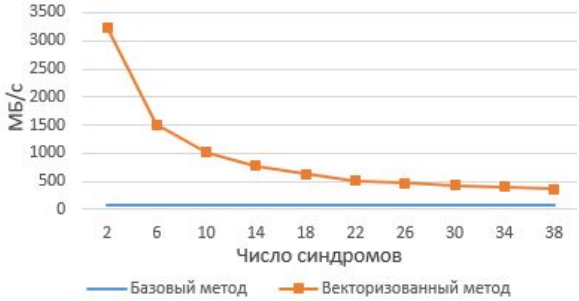
Этот подход мы используем сначала в паре с библиотекой ISA-L для улучшения её производительности. Мы заменим библиотечный вызов функции обращения матрицы в тестовом окружении на нашу, что позволит легко протестировать предложенный алгоритм обращения, не изменяя тестового окружения. Тесты проводились так же, как это было сделано для тестирования открытых библиотек.

На графиках (рис.5) виден заметный прирост производительности при переходе с обычного метода обращения матриц на векторизованный. Тесты проводились для библиотеки ISA-L, так как она чаще всего встречается в открытых реализациях СХД и предоставляет свои функции для обращения матрицы.

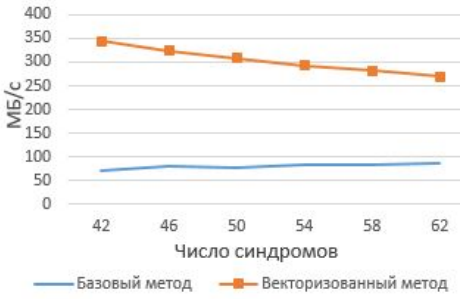
Этот подход мы и будем использовать в реализации стандартного метода декодирования.

4.3.2. Реализация алгоритмов декодирования

В предыдущих работах [18] нами рассматривалось три алгоритма декодирования:



(a) $2 \leq m \leq 26$



(b) $26 \leq m \leq 96$

Рис. 5: График зависимости скорости кодирования от числа синдромов при $n = 96$

- стандартный алгоритм декодирования;
- алгоритм Форни;
- алгоритм «R».

Два из них описаны и реализованы в рамках курсовой работы [18], обоснование третьего алгоритма можно найти в [12]. Алгоритм «R» основан на свойствах матрицы Вандермонда, которую мы используем в качестве проверочной. Используя ее свойства можно при помощи интерполяционных многочленов построить матрицу декодирования. Затем умножив матрицу декодирования на вектор из n не отказавших блоков, можно восстановить информацию.

Мы реализовали алгоритм «R» и применили к нему метод группировки, описанный ранее. Затем были проведены тестирование и замеры производительности. Тесты проводились так же, как и при тестировании открытых библиотек. В алгоритмах, где строится матрица декодирования, замерялась скорость построения этой матрицы и скорость декодирования, а в алгоритме Форни только скорость декодирования. Алгоритм «R» с проведенными улучшениями показал лучшую скорость декодирования. Он и будет использован в нашем модуле.

4.4. Поиск скрытых повреждений

Последняя функциональная часть модуля — поиск SDC. Алгоритм будет состоять из нескольких шагов:

- быстрая проверка на наличие скрытых повреждений;
- построение полинома локаторов ошибок;
- нахождение корней полинома и мест SDC;
- запуск алгоритма декодирования.

Распишем каждый шаг и его смысл подробнее.

Алгоритмы, позволяющие искать скрытые повреждения, — достатчно трудоемки. Однако, используя систематическое кодирование, можно довольно быстро проверить, есть ли какие-либо повреждения. Для этого достаточно воспользоваться формулой (5). В нашем случае, мы можем использовать для этого умножение матрицы Вандермонда на вектор с данными и синдромами, как это показано в формуле (14).

Однако нам заранее может быть известно о поломках на некоторых дисках и это необходимо учесть, чтобы наша проверка на наличие SDC не указала нам на эти диски. Для этого можно занулить значения в отказавших дисках или удалить соответствующие блоки в векторе $Y = (D|S)$ и столбцы в матрице Вандермонда. Удаление поможет нам уменьшить число операций при умножении проверочной матрицы на вектор.

После умножения мы получим вектор $(C_0, \ldots, C_m)$. Если хоть одна компонента этого вектора не равна нулю, то в системе есть скрытые повреждения. Если все компоненты равны нулю, то SDC нет и можно приступить к исправлению обычных ошибок, если такие были.

Если SDC есть, то найденные значения помогут построить *полином локатора ошибок* — полином, корнями которого являются $a^{N-j_0-1}, \ldots, a^{N-j_{p-1}-1}$, где $j_0, \ldots, j_{p-1}$ — номера блоков с SDC, $N = m + n$.

Для нахождения такого полинома можно использовать алгоритм Берликампа-Месси (BMA)[17]. Алгоритм на входе принимает по одному байту с каждого блока $(C_0, \ldots, C_m)$, то есть работает лайнами, шириной в один байт. После работы BMA мы получим коэффициенты полинома.

Зная коэффициенты многочлена, можно найти его корни. Мы работаем с конечным полем из 256 элементов, а значит, корни полинома можно найти перебором. После нахождения корней полинома $a^{N-j_0-1}, \ldots, a^{N-j_{p-1}-1}$, по ним можно получить значения $j_0, \ldots, j_{p-1}$. Диски с этими номерами мы пометим как отказавшие и запустим алгоритм декодирования, который исправит эти ошибки и те, о которых было известно изначально (если такие есть).

4.4.1. Использование кэширования

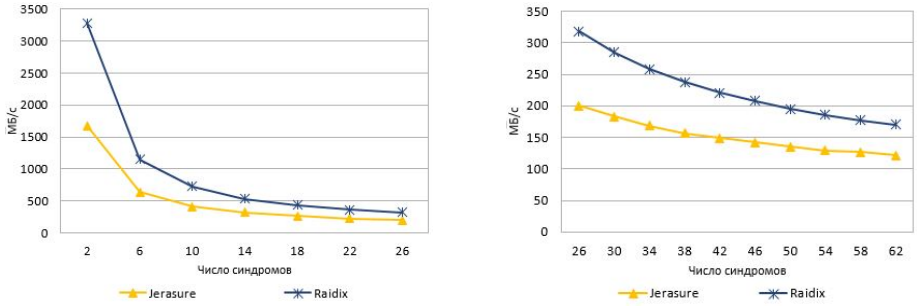
Обрабатывать по одному байту с каждого блока, размер которого вычисляется килобайтами, слишком долго. Назовем *длиной ошибки* то, сколько байт подряд на одном блоке являются не верными. С учетом особенностей работы дисковых драйверов и файловых систем, длина ошибки чаще всего больше одного байта. Значит, если мы на каком-то запуске алгоритма BMA и нахождения корней многочлена нашли на $i-$ом блоке ошибку, то и многие последующие запуски этих алгоритмов скорее всего покажут ошибку на этом же блоке. Однако мы не можем это гарантировать, и точная

длина ошибки не известна.

Чтобы ускорить процесс нахождения SDC, можно кэшировать уже обработанные значения и для этих значений запоминать номер отказавшего блока. Кэшировать значения $(C_0, \ldots, C_m)$ и полученные корни, чтобы уменьшить число запусков и алгоритма BMA и алгоритма нахождения корней, нельзя, потому что $(C_0, \ldots, C_m)$ — линейная комбинация $(D_0, \ldots, D_{n-1}, S_0 \ldots, S_{m-1})$. Их значения никак не зависят от того, где произошли SDC, и будут сильно отличаться от лайна к лайну.

Кэшировать получится только значения коэффициентов полинома. Получив полином, мы проверим, не получали ли мы его ранее. Если нет, то найдем его корни и запомним полином и корни в подходящей структуре, а потом вернем корни. Если уже находили, то найдем полученный полином в структуре и сразу вернем его корни.

Тестирование показало, что достаточно хранить только один, последний полученный полином. Такая реализация с кэшированием не сильно увеличивает используемую память, но ускоряет процесс нахождения SDC в $3 - 10$ раз по сравнению с реализацией, не использующей кэширование.



(a) $2 \leq m \leq 26$ (b) $26 \leq m \leq 96$

Рис. 6: График зависимости скорости кодирования от числа синдромов при $n = 96$

5. Тестирование

Было проведено тестирование кодирования, декодирования и поиска скрытых повреждений. Для каждой части запускалось по 10000 тестов. Корректность кодирования отдельно не проверялась, так как она проверяется в ходе декодирования и поиска SDC.

Тестирование декодирования проводилось так же, как описано в главе с открытыми библиотеками. Все тесты прошли успешно. Кроме того, мы сравнили скорость декодирования нашей реализации и библиотеки jerasure (рис. 6).

Тестирования поиска SDC было разбито на два этапа. Сначала тестировался поиск и восстановление при наличии обычных ошибок, а потом без них. Оба набора тестов начинались со создания страйпа, заполнения его блоков случайными значениями и расчета контрольных сумм. Значения блоков страйпа копировалось во временное хранилище для проверки в конце.

В первом случае затем случайно выбиралось случайных $k < m/2$ блоков, которые были помечены как отказавшие и портились. Затем случайным образом выбирались $p < (m - k)/2$ блоков, которые только портились. Блоки портились путем заполнения случайной последовательностью бит. После этого запускался алгоритм поиска SDC и исправления ошибок. Если в конце значения блоков в страйпе отличалось от значения блоков во временном хранилище, то тест считался проваленным. Второй набор тестов проводился так же, но при $k = 0$.

В итоговой реализации алгоритмов все тесты прошли успешно.

6. Внедрение в ядро Linux

Внедрение в ядро необходимо, чтобы наш модуль мог взаимодействовать со всей остальной системой хранения данных, реализованной в компании Raidix, без накладных расходов на переход в userspace.

Для внедрения необходимо было:

- заменить все инструкции динамического выделения памяти на инструкции, используемые в ядре;
- вынести объявления переменных в начало области видимости (согласно ISO C90);
- уменьшить выделяемую на стеке память до допустимых размеров (2КБ);
- реализовать SSE инструкции в ядре Linux.

Последний пункт необходим, так как в ядре нет возможности подклчить библиотеки, реализующие intrinsic-инструкции SSE или AVX. Необходимо перенести реализацию используемых функций из этих библиотек в отдельные файлы. Полученную краткую версию библиотеки можно вместе с остальными исходниками модуля перенести из пользовательского интерфейса в ядро.

Размер выделяемой на стеке памяти изначально учитывался при проектировании модуля и реализации алгоритмов. Если для некоторых алгоритмов требовалось много дополнительной памяти, то она выделялась динамически, даже если это могло замедлить скорость работы.

После проведения этих изменений необходимо было провести функциональные тесты в ядре, чтобы проверить корректное взаимодействие нашего модуля и остальной системой. Тесты проводились путем записи и чтения на реальные диски через СХД.

Для проверки декодирования некоторые диски отключались от системы после записи данных на них. Пользователь СХД не должен заметить отключения дисков, то есть информация, получаемая после отключения дисков, должна совпадать с той, что мы записывали.

Для проверки поиска SDC необходимо осуществить запись сразу на диски через СХД, затем на случайный диск провести запись в обход системы. При последующем чтении система должна обнаружить изменения данных, сообщить об этом, и исправить найденные ошибки.

Со всеми описанными тестами система справилась, значит, наш модуль внедрен успешно.

Результаты

В ходе работы был разработан модуль помехоустойчивого кодирования, позволяющий находить и исправлять множественные ошибки. Для этого была проделана следующая работа.

- Рассмотрены различные подходы к помехоустойчивому кодированию.
- Проведен обзор существующих открытых библиотек помехоустойчивого кодирования, показана их неэффективность и ненадежность некоторых из них.
- Исследованы алгоритмы кодирования, декодирования и поиска скрытых повреждений.
- Алгоритмы реализованы, проведены тестирование и сравнительные замеры производительности.
- Разработан и протестирован модуль, организующий работу вышеописанных алгоритмов. Модуль внедрен в ядро Linux.

Список литературы

[1] Gallager Robert G. Low-Density Parity-Check Codes. — 1963.

[2] HoloStor. — 2011. — URL: https://code.google.com/p/holostor/ (дата обращения: 04.06.2015).

[3] Intel. ISA-L. — 2015. — URL: https://goo.gl/aHNK6M (дата обращения: 04.06.2015).

[4] Moon Todd K. Error Correction Coding: Mathematical Methods and Algorithms. — Wiley-Interscience, 2005. — ISBN: 0471648000.

[5] O'Whielacronx Zooko. zfec. — 2012. — URL: https://pypi.python.org/pypi/zfec (дата обращения: 04.06.2015).

[6] Patterson David A., Gibson Garth, Katz Randy H. A Case for Redundant Arrays of Inexpensive Disks (RAID) // SIGMOD Rec.

[7] Plank J. Jerasure. — 2014. — URL: http://web.eecs.utk.edu/~plank/plank/www/software.html (дата обращения: 06.06.2015).

[8] Plank J. S. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. — 1996. — July. — no. CS-96-332.

[9] Plank J. S. Erasure Codes for Storage Systems: A Brief Primer // ;login: the Usenix magazine. — 2013. — December. — Vol. 38, no. 6.

[10] Plank J. S., Greenan K. M., Miller E. L. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions // FAST-2013: 11th Usenix Conference on File and Storage Technologies. — San Jose, 2013. — February.

[11] Plank J. S., Xu L. Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications // NCA-06: 5th IEEE International Symposium on Network Computing Applications. — Cambridge, MA, 2006. — July.

[12] Platonov S., Marov A. Effective Method for Coding and Decoding RS Codes Using SIMD Instructions. — 2014. — September.

[13] Randombit. FECpp. — 2010. — URL: http://www.randombit.net/code/fecpp/ (дата обращения: 04.06.2015).

[14] Robinson J. LRC // RFC. — 1985. — URL: https://tools.ietf.org/html/rfc935 (дата обращения: 06.06.2015).

[15] An XOR-based Erasure-resilient Coding Scheme / J. Blömer, M. Kalfane, R. Karp et al.

[16] Берлекэмп Э. Алгебраическая теория кодирования / Под ред. Берман С.Д. — М. : Мир, 1971. — С. 473.

[17] Е.А.Крук, А.А.Овчинников. Лекции по Теории кодирования. — Санкт-Петербург : Отдел оперативной полиграфии ГУАП, 2004. — С. 62.

[18] Разработка модуля восстановления утраченных дисков в RAID(n+m) : Отчет / СПбГУ ; исполн.: В. С. Зайберт. — Санкт-Петербург, Россия : 2014.

[19] ООО «Raidix». — URL: `http://www.raidix.ru/` (дата обращения: 06.06.2015).

[20] Питерсон У., Уэлдон Э. Коды, исправляющие ошибки / Под ред. Добрушина Р.Л., Самойленко С. И. — М. : Мир, 1976. — С. 593.