Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра Системного Программирования

Зайберт Валерия Сергеевна

Разработка модуля восстановления утраченных дисков в RAID(n+m)

Курсовая работа

Научный руководитель:
руководитель Исследовательской лаборатории RAIDIX Платонов С. М.

Санкт-Петербург
2014

Оглавление

Введение

Объем данных, хранимый людьми довольно велик. Хочется иметь возможность быстрой работы с этими данными. Кроме того, хотелось бы защититься от возможных сбоев и утраты дисков или информации на них. С этой целью часто используют технологию RAID.

RAID (англ. *redundant array of independent disks* – избыточный массив независимых дисков), как понятно из названия, представляет собой объединение нескольких дисков в один массив, информация между которыми будет распределяться по определенным правилам, зависящим от типа RAID. Технология повышает скорость работы с дисками и надежность хранения информации. Также большинство видов технологии RAID позволяет восстанавливать данные после ошибок. Ошибки бывают двух типов: повреждения, место расположения которых нам известно, и скрытые повреждения, когда нам надо сначала найти место, в котором произошел сбой. Ошибки второго типа также называют SDC (Silent Data Corruption). Далее мы будем говорить об ошибках первого типа, если не сказано иного.

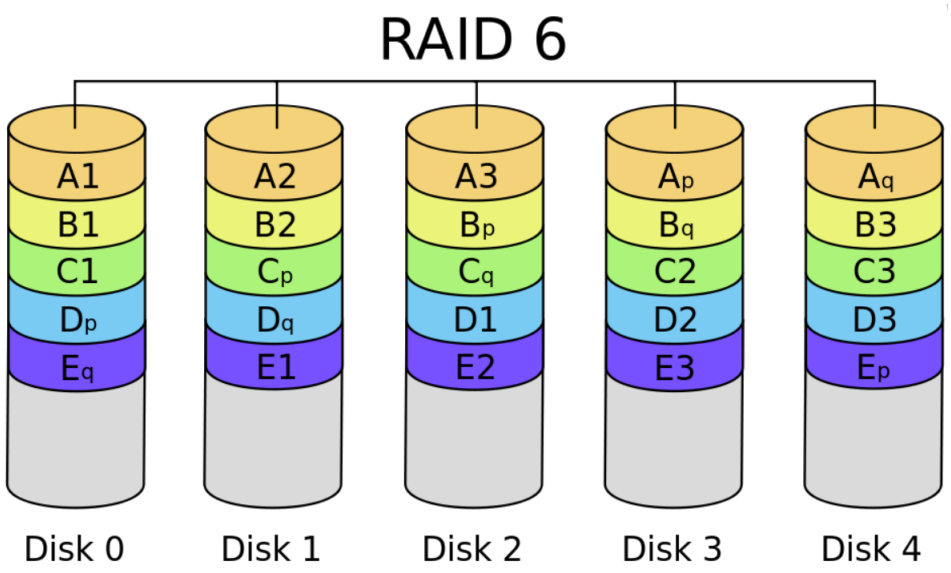


Рис. 1: Стуруктура RAID6.

Расскажем для начала об известной технологии RAID6 . В ней каждый диск разбивается на блоки одинакового размера, блоки нумеруются внутри одного диска, затем блоки с одинаковыми номерами из разных дисков объединяются в страйп (от англ. *stripe* – полоска). В каждом страйпе выбирается по два блока, в которых будут храниться специальным образом посчитанные контрольные суммы (или, как их еще называют, синдромы).

Чтобы избежать разногласий в терминологии, стоит отметить, что модуль, выполняющий работу по восстановлению данных и подсчету контрольных сумм, рабо-

тает в рамках одного страйпа. Поэтому, не умаляя общности, в нашей работе можно считать, что каждый диск состоит из одного блока. Данная технология позволяет восстанавливать до двух повреждений в каждом страйпе. То есть, если у нас вышло из строя два диска, мы сможем восстановить данные на них. Однако, если из строя вышло большее количество дисков, мы не сможем восстановить информацию

Одно из решений это объединить несколько RAID6 в RAID60, в котором данные разбиваются на части и каждая часть записывается на отдельный RAID6. Пусть мы объединили k RAID6. В этом случае в каждом RAID6 мы сможем восстановить до двух дисков, то есть всего $2 * k$ дисков.

Проанализируем скорость восстановления данных в этом случае. Пусть у нас отказал один или два диска в одном из RAID6. Тогда для восстановления нам потребуется просуммировать данные только с дисков, лежащих в том же RAID6. Если бы наш массив дисков мы не разбивали на части, нам потребовалось бы обработать информацию со всех дисков, то есть в k раз больше (если диски распределены равномерно по RAID6). То есть, скорость восстановления и расчета синдромов во всем RAID60 практически не отличается от скорости аналогичных операций в каждой его части.

Мы ускорили восстановление данных, расчет и перерасчет синдромов, но и увеличили избыточность данных. Однако, если хотя бы три поломки сосредоточенны в одном RAID6, то данные в нем будут утеряны.

Для решения этой проблемы можно увеличить число контрольных сумм. Можно реализовать модули с тремя различными синдромами, четыремя и так далее. Этот подход может иметь свои плюсы, однако это не рационально с точки зрения времени разработки. Поэтому появилась технология RAID(n+m), в которой n блоков в страйпе выделяется под хранение данных и m блоков для синдромов, причем значения n и m могут легко варьироваться в зависимости от желаний заказчика, без необходимости изменения кода. Однако, следует отметить, что в силу ограничений, накладываемых нашим полем, $n + m < 256$.

Таким образом, мы должны реализовать общую структуру модуля RAID($n + m$), расширив уже имеющуюся архитеутуру RAID6. Далее, так как оптимальный расчет синдромов с использованием кодов Рида-Соломона это тема отдельной научной работы, и ей занимаются другие люди, мы сконцентрировались на восстановлении данных. Нашей задачей было найти и исследовать различные алгоритмы восстановления данных с использованием m контрольных сумм и модернизровать эти алгоритмы, чтобы они лучше подходили для решения нашей проблемы. Затем разработать прототип модуля, который позволит нам убедиться в работоспособности этих алгоритмов и применимости их на практике, и, в конце, распараллелить вычисления с использованием векторных инструкций процессора. Также следует провести дополнительные оптимизации и сравнить работу алгоритмов на практике.

1. Мотивация

Зачем необходимо уметь восстанавливать данные после большого числа отказов и делать это быстро? Многие считают, что технологии сегодня достаточно развиты и сбои происходят крайне редко. Однако, как показывают наблюдения, большие системы до 30% времени работы находятся в состоянии Dergaded mode, в котором хотя бы один диск помечен как отказавший, и либо запущен процесс восстановления, либо он отложен на потом[4][5].

Так же существует режим Advance reconstruction, суть которого заключается в следующем. Чтение с дисков — медленная операция по сравнению с работой в памяти. Более того, некоторые диски могут быть заняты чем-то еще и не ответить нам сразу. Тогда мы можем начать процесс восстановления данных на медленных дисках не дожидаясь ответа с них. Такой подход может увеличить быстродействие системы. Очевидно, что при использовании этого режима, скорость работы с массивом напрямую зависит от скорости восстановления.

2. Структура модуля RAID$(n + m)$

Мы считаем, что весь страйп лежит где-то в памяти как большой массив, и на вход модулю подается ссылка на этот участок памяти. Кроме того, на вход приходит описание этого страйпа, содержащее в себе следующие данные:

- n - число дисков с полезной информацией
- m - число дисков, отведенных для синдромов
- размер одного блока (диска)
- маска утраченных дисков
- прочая вспомогательная информация

Считается, что блоки с синдромами лежат в конце страйпа. По маске утраченных дисков мы понимаем, какую функцию необходимо запустить. Для корректной полноценной работы модуля необходимо наличие трех основных функций работы с данными.

1. Расчет синдромов.
2. Восстановление утраченных данных.
3. Поиск скрытых повреждений.

Например, для расчета синдромов мы помечаем, что мы утратили все диски с контрольными суммами. Тогда наш модуль при инициализации страйпа автоматически выберет функцию расчета синдромов и запишет ссылку на нее в структуру, описывающую страйп. Если утраченными помечены другие диски (или не помечен ни один диск), то в зависимости от выбора режима (Advance recostruction или поиск SDC) и соотношения помеченных дисков, n и m, мы выберем или функцию поиска SDC, или функцию восстановления данных и поместим ссылку на эту функцию в структуру. Затем запустится нужная функция.

3. Операции в $GF(2^8)$

Стоит напомнить, что вычисления ведутся в поле $GF(2^8)$, поэтому опреации сложения и умножения, использующиеся в вышеприведенных формулах определены другим способом[3]. Поле, в котором мы работаем это конечное поле из 256 элементов. Каждый из элементов представляет собой многочлен g с коэффицентами из $\{0, 1\}$ такой, что

$$deg\, g < 8.$$

Сответсвенно, операция сложения это привычное для нас сложение многочленов, коэффициенты которого складываются по модулю 2, а операция умножения это умножение многочленов по модулю неприводимого многочлена поля. Неприводимых многочленов в нашем поле несколько, мы выбрали тот из них, в котором минимальное количество ненулевых коэффициентов:

$$f(x) = x^8 + x^4 + x^3 + x^2 + 1.$$

Более прстой способ работы с элементами поля это представлять их как числа от 0 до 256 в двоичном представлении. К примеру, многочлен $g(x) = x^7 + x^2 + 1$ будет представлен как 10000101, или, что тоже самое $2^7 + 2^2 + 1 = 133$. Тогда сложение в нашем поле это просто операция xor.

4. Расчет контрольных сумм

Каждая из задач, описанная в предыдущей главе, может быть темой отдельного исследования. Но быстрый и правильный расчет контрольных сумм необходим в каждой из них. Мало того, что на основе уже рассчитанных и хранимых синдромов мы можем искать скрытые повреждения и восстанавливать данные, но и в ходе восстановления и поиска нам необходимо расчитывать аналогичные суммы. Поэтому эта задача особенна важна, и понимание того, как она решается и какими свойствами обладают синдромы, сильно влияет на нашу способность анализировать алгоритмы восстановления и понимать их. Именно поэтому далее мы оговорим свойства этих сумм и приведем формулы их расчета.

Вычисления в RAID$(n+m)$ основываются на систематических кодах Рида-Солмона, поэтому на синдромы накладываются следующие свойства. Пусть $D_i \in GF(2^8)$ — блоки с данными, а $S_i \in GF(2^8)$ — блоки с синдромами. Переобозначим

$$(D_0, \ldots, D_{n-1}, S_0, \ldots, S_{m-1}) = (Y_0, \ldots, Y_{N-1})$$

Контрольные суммы от всей последовательности блоков расчитываются следующим образом:

$$\tilde{S}_j = \sum_{i=0}^{N-1} (Y_i a^{j(N-i-1)}), \qquad j = 0, \ldots, m-1 \tag{1}$$

где a - примитивный элемент поля. Синдромы должны быть подсчитаны таким образом, чтобы все эти контрольные суммы равнялись нулю при условии, что ни один диск не испорчен. Имея такого рода синдромы, можно произвести восстановление утраченных дисков и поиск скрытых повреждений.

5. Постановка задачи

Итак, пусть у нас есть информационные блоки $D_0, \ldots, D_{n-1} \in GF(2^8)$ и подсчитаны синдромы $S_0, \ldots, S_{m-1} \in GF(2^8)$, так что контрольные суммы, описанные в (1) равны нулю при условиии отсутсвия повреждений. Пусть номера отказавших блоков $k_1, \ldots k_l$, где $l < m$. Эти номера нам известны либо из структуры описывающей страйп, либо мы нашли их при помощи функции поиска скрытых повреждений.

Для восстановления l блоков достаточно l синдромов. До возникновения повреждений все контрольные суммы от последовательности блоков $(Y_0, \ldots, Y_{N-1})$ равнялись нулю, значит после утраты данных эти суммы равны соответсвенно $\tilde{S}_0, \ldots, \tilde{S}_{l-1}$, причем для подсчета этих сумм мы не учитываем значения в отказавших блоках или считаем их нулями.

Сложив систему, полученную из формул расчета синдромов до повреждений (1), с системой, полученной после пересчета синдромов при возникновении повреждений, получим:

$$\begin{cases} (Y_{k_1} + \tilde{Y}_{k_1}) + \ldots + (Y_{k_l} + \tilde{Y}_{k_l}) = \tilde{S}_0, \\ (Y_{k_1} + \tilde{Y}_{k_1})a^{N-k_1-1} + \ldots + (Y_{k_l} + \tilde{Y}_{k_l})a^{N-k_l-1} = \tilde{S}_1, \\ (Y_{k_1} + \tilde{Y}_{k_1})a^{2(N-k_1-1)} + \ldots + (Y_{k_l} + \tilde{Y}_{k_l})a^{2(N-k_l-1)} = \tilde{S}_2, \\ \ldots \\ (Y_{k_1} + \tilde{Y}_{k_1})a^{(l-1)(N-k_1-1)} + \ldots + (Y_{k_l} + \tilde{Y}_{k_l})a^{(l-1)(N-k_l-1)} = \tilde{S}_{l-1}. \end{cases} \quad (2)$$

Здесь $\tilde{Y}_{k_i}$ - известные значения после повреждения, а Y_{k_i} - искомые значения данных до сбоя. Их мы и должны бли найти.

6. Алгоритмы восстановления

6.1. Обращение матрицы

Алгоритм, основанный на обращении матрицы это стандарнтый подход, используемый в большинстве реализаций RAID$(n+m)$.

Пусть $k_0, \ldots, k_{l-1}$ — номера отказавших блоков, где l — их количество. Зная эти номера, обнулим знаения в отказавших блоках. Для уменьшения числа операций можем не учитывать эти блоки вовсе при расчете синдромов, просто исключив из матриц соответсвующий столбцы и блоки Y_{k_i}. Тогда расчет синдромов производится следующим образом:

$$\begin{pmatrix} 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\ a^{N-1} & \cdots & a^{N-k_i} & a^{N-k_i-2} & \cdots & a & 1 \\ a^{2(N-1)} & \cdots & a^{2(N-k_i)} & a^{2(N-k_i-2)} & \cdots & a^2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a^{(l-1)(N-1)} & \cdots & a^{(l-1)(N-k_i)} & a^{(l-1)(N-k_i-2)} & \cdots & a^{l-1} & 1 \end{pmatrix} \begin{pmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{k_i-1} \\ Y_{k_i+1} \\ \vdots \\ Y_N \end{pmatrix} = \begin{pmatrix} \tilde{S}_0 \\ \tilde{S}_1 \\ \vdots \\ \tilde{S}_{l-1} \end{pmatrix} \tag{3}$$

Далее, используя систему (2) и подсчитанные контрольные суммы, мы можем восстановить утраченные данные следующим образом:

$$\begin{pmatrix} Y_{k_0} \\ Y_{k_1} \\ \vdots \\ Y_{k_{l-1}} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ a^{N-k_0-1} & a^{N-k_1-1} & \cdots & a^{N-k_{l-1}-1} \\ a^{2(N-k_0-1)} & a^{2(N-k_1-1)} & \cdots & a^{2(N-k_{l-1}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a^{(l-1)(N-k_0-1)} & a^{(l-1)(N-k_1-1)} & \cdots & a^{(l-1)(N-k_{l-1}-1)} \end{pmatrix}^{-1} \begin{pmatrix} \tilde{S}_0 \\ \tilde{S}_1 \\ \vdots \\ \tilde{S}_{l-1} \end{pmatrix} \tag{4}$$

Таким образом, стандартный алгоритм восстановления будет заключаться в умножении на обращенную матрицу Вандермонда. Нам потребуется $l(N-l)$ умножений и $l(N-l-1)$ сложений для расчета контрольных сумм и l^2 умножений и $l(l-1)$ сложений на этапе восстановления. Так же нам необходимо найти обратную матрицу, что довольно трудоемко.

Однако, можно воспользоваться тем, что наша матрица является матрицей Вандермонда, а про обращение таких матриц известны некоторые полезные свойства. Мы не будем подробнее рассматривать этот алгоритм тут, но чуть позже приведем количество операций, необходимых для восстановления по этому алгоритму, и покажем его в сравнении с остальными методами на практике.

6.2. Алгоритм Форни

Следующий алгоритм основан больше на свойствах нашего поля, а именно на действиях с полиномами в конечных полях Галуа[1].

Составим вспомогательный полином, с корнями $a^{-(N-k_0-1)}, \ldots, a^{-(N-k_{l-1}-1)}$:

$$\sigma(x) = \prod_{i=0}^{l-1}(1 + xa^{N-k_i-1}). \tag{5}$$

Введем еще один вспомогательнй полином $S(x)$:

$$S(x) = \sum_{i=0}^{l-1} \tilde{S}_i x^i. \tag{6}$$

Распишем полином (6), подставив в него формулу для $\tilde{S}_j$ из (1). Получим

$$\begin{aligned} S(x) = \sum_{j=0}^{l-1} \tilde{S}_j = &\big[(Y_{k_0} + \tilde{Y}_{k_0}) + \ldots + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})\big] + \\ &+\big[(Y_{k_0} + \tilde{Y}_{k_0})a^{N-k_0-1} + \ldots + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})a^{N-k_{l-1}-1}\big]x + \\ &+\big[(Y_{k_0} + \tilde{Y}_{k_0})a^{2(N-k_0-1)} + \ldots + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})a^{2(N-k_{l-1}-1)}\big]x^2 + \\ &+ \ldots + \\ &+\big[(Y_{k_0} + \tilde{Y}_{k_0})a^{(l-1)(N-k_0-1)} + \ldots + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})a^{(l-1)(N-k_{l-1}-1)}\big]x^{l-1}. \end{aligned} \tag{7}$$

Раскроем скобки и сгруппируем слогаемые при $(Y_{k_i} + \tilde{Y}_{k_i})$:

$$\begin{aligned} S(x) = (Y_{k_0} + \tilde{Y}_{k_0})[1 + xa^{N-k_0-1} + x^2a^{2(N-k_0-1)} + x^{l-1}a^{(l-1)(N-k_0-1)}] + \ldots + \\ (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})[1 + xa^{N-k_{l-1}-1} + x^2a^{2(N-k_{l-1}-1)} + x^{l-1}a^{(l-1)(N-k_{l-1}-1)}]. \end{aligned} \tag{8}$$

Рассмотрим выражение в квадратных скобках. Заметим, что это сумма геометрической прогрессии с начальным значением 1 и знаменателем xa^{N-k_i-1} из l слагаемых. Значит, сумма равна:

$$1 + xa^{N-k_i-1} + x^2a^{2(N-k_i-1)} + x^{l-1}a^{(l-1)(N-k_i-1)} = \frac{(xa^{N-k_i-1})^l + 1}{xa^{N-k_i-1} + 1} \tag{9}$$

Подставим полученное значение в каждую квадратную скобку в (8):

$$S(x) = (Y_{k_0} + \tilde{Y}_{k_0})\left[\frac{(xa^{N-k_0-1})^l + 1}{xa^{N-k_0-1} + 1}\right] + \ldots + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})\left[\frac{(xa^{N-k_{l-1}-1})^l + 1}{xa^{N-k_{l-1}-1} + 1}\right]. \tag{10}$$

Перемножим два введенных многочлена:

$$\begin{aligned} S(x)\sigma(x) =&(Y_{k_0} + \tilde{Y}_{k_0})\left[\frac{(xa^{N-k_0-1})^l + 1}{xa^{N-k_0-1} + 1}\right]\prod_{i=0}^{l-1}(1 + xa^{N-k_i-1}) + \ldots + \\ &(Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})\left[\frac{(xa^{N-k_{l-1}-1})^l + 1}{xa^{N-k_{l-1}-1} + 1}\right]\prod_{i=0}^{l-1}(1 + xa^{N-k_i-1}). \end{aligned} \tag{11}$$

В каждом слагаемом знаменатель можно сократить с одним из множителей $\sigma(x)$:

$$\begin{aligned} S(x)\sigma(x) = (Y_{k_0} + \tilde{Y}_{k_0})\big[(xa^{N-k_0-1})^l + 1\big]\prod_{i=1}^{l-1}(1 + xa^{N-k_i-1}) + \ldots + \\ + (Y_{k_j} + \tilde{Y}_{k_j})\big[(xa^{N-k_j-1})^l + 1\big]\prod_{i=0, i\neq j}^{l-1}(1 + xa^{N-k_i-1}) + \ldots + \\ + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})\big[(xa^{N-k_{l-1}-1})^l + 1\big]\prod_{i=0}^{l-2}(1 + xa^{N-k_i-1}). \end{aligned} \tag{12}$$

Степень получившегося полинома

$$deg\, S(x)\sigma(x) = l(l-1).$$

Построим еще один многочлен

$$\Omega(x) = S(x)\sigma(x)\, mod\, x^l. \tag{13}$$

Это выражение так же называют «ключевое уравнение». Если заметить, что

$$deg \prod_{i=0, i\neq j}^{l-1}(1 + xa^{N-k_i-1}) = l - 1,$$

то Ω можно представить следующим образом:

$$\begin{aligned} \Omega(x) = (Y_{k_0} + \tilde{Y}_{k_0})\prod_{i=1}^{l-1}(1 + xa^{N-k_i-1}) + \ldots + \\ + (Y_{k_j} + \tilde{Y}_{k_j})\prod_{i=0, i\neq j}^{l-1}(1 + xa^{N-k_i-1}) + \ldots + \\ + (Y_{k_{l-1}} + \tilde{Y}_{k_{l-1}})\prod_{i=0}^{l-2}(1 + xa^{N-k_i-1}). \end{aligned} \tag{14}$$

Напомним, что в нашем поле сложение это операция xor. Тогда, подставив $a^{-(N-k_0-1)}$ в $\Omega(x)$, получим, что обнулятся все слагаемые кроме первого. Подставив $a^{-(N-k_1-1)}$,

обнулятся все слагаемые кроме второго и так далее. Таким образом, получим:

$$\Omega\big(a^{-(N-k_j-1)}\big)= (Y_{k_j} + \tilde{Y}_{k_j}) \prod_{i=0, i\neq j}^{l-1} (1 + a^{-(N-k_j-1)} a^{N-k_i-1}) =$$

$$= (Y_{k_j} + \tilde{Y}_{k_j}) \prod_{i=0, i\neq j}^{l-1} (1 + a^{k_j-k_i}). \quad (15)$$

Введем обозначение

$$\lambda_j = \left(\prod_{i=0,\, i\neq j}^{l-1} (1 + a^{k_j-k_i}) \right). \quad (16)$$

Воспользовавшись этим обозначением, можно получить искомые данные.

$$Y_{k_i} = \tilde{Y}_{k_i} + \Omega(a^{-(N-k_i-1)})\lambda_i. \quad (17)$$

Однако, нам не известны $\Omega(a^{-(N-k_i-1)})$. Чтобы найти эти значения, посмотрим на $\Omega(x)$ следующим образом:

$$\Omega(x) = S(x)\sigma(x)\, mod\, x^l =$$
$$= (\tilde{S}_{l-1}x^{l-1} + \tilde{S}_{l-2}x^{l-2} + \ldots + \tilde{S}_1 x + \tilde{S}_1)(\sigma_l x^l + \sigma_{l-1}x^{l-1} + \ldots + \sigma_1 x + \sigma_0)\, mod\, x^l. \quad (18)$$

Так как мы берем всё по модулю x^l, то все коэффициенты, возникающие при степенях x больших или равных l, нас не интересуют, и мы не будем их вычислять. Рассмотрим коэффициенты при степенях x меньших l:

$$(\tilde{S}_{l-1}x^{l-1} + \tilde{S}_{l-2}x^{l-2} + \ldots + \tilde{S}_1 x + \tilde{S}_1)(\sigma_l x^l + \sigma_{l-1}x^{l-1} + \ldots + \sigma_1 x + \sigma_0)\, mod\, x^l =$$
$$= x^{l-1}(\tilde{S}_{l-1}\sigma_0 + \tilde{S}_{l-2}\sigma_1 + \ldots + S_0\sigma_{l-1}) + x^{l-2}(\tilde{S}_{l-2}\sigma_0 + \ldots + S_0\sigma_{l-2}) + \ldots + x^0\tilde{S}_0\sigma_0 =$$
$$= \Omega_{l-1}x^{l-1} + \Omega_{l-2}x^{l-2} + \ldots + \Omega_1 x + \Omega_0) = \Omega(x). \quad (19)$$

Тоже самое можно записать в матричном виде:

$$\begin{pmatrix} \sigma_{l-1} & \sigma_{l-1} & \cdots & \sigma_1 & \sigma_0 \\ \sigma_{l-2} & \sigma_{l-3} & \cdots & \sigma_0 & 0 \\ \sigma_{l-3} & \sigma_{l-4} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_0 & 0 & \cdots & 0 & 0 \end{pmatrix} \begin{pmatrix} \tilde{S}_0 \\ \tilde{S}_2 \\ \vdots \\ \tilde{S}_{l-2} \\ \tilde{S}_{l-1} \end{pmatrix} = \begin{pmatrix} \Omega_{l-1} \\ \Omega_{l-3} \\ \vdots \\ \Omega_1 \\ \Omega_0 \end{pmatrix} \quad (20)$$

Или формулой:

$$\Omega_j = \sum_{i=0}^{j} \tilde{S}_i\, \sigma_{j-i}, \qquad j = 0, \ldots, l-1. \quad (21)$$

Тогда восстановление отказавших блоков можно выполнять следующим образом. Посчитаем $\tilde{S}_i$, зная номера отказавших блоков, находим σ_j из (5). Это можно сделать

несколькими способами, например, по теореме Виета или непосредственно умножением, но мы далее рассмотрим более быстрые алгоритмы. Затем из (21) находим Ω_j. Далее по формуле (17) восстанавливаем отказавшие блоки. Значения λ_j их значения могут храниться в заранее рассчитанных таблицах, либо рассчитываться непосредственно перед восстановлением с использованием таблиц логарифмов и степеней.

Преимущество данного способа перед предыдущим заключается в том, что нам не придется обращать матрицу. Подсчитаем количество необходимых векторных операций.

Для расчета каждого синдрома мы сначала генерируем строку, на которую мы умножим данные. Строка соответствует строке матрицы (3), без учета элементов, которые соответствуют сбитым дискам. Таким образом, мы имеем $(N-l)l$ умножений и $(N-l-1)l$ сложений на каждые 16 байт диска. На основе этой информации вычислялись коэффициенты Ω с использованием $(l-1)(l-2)/2$ сложений и $l(l-2)/2$ соответственно. Для нахождения значения многочлена Ω в каждой из точек мы применили схему Горнера с l сложениями и l умножениями. И, наконец, вычисление истинных значений сбитых блоков занимает l операций умножения.

Стоит иметь в виду, что за это количество операций, необходимое для нахождения полинома Ω, его значений в точках и восстановления, алгоритм обрабатывает по 16 байт с каждого блока при использовании технологии процессора SSE. Это число может быть увеличено при использовании регистров большего размера.

В итоге мы получили $l(N-\frac{l+5}{2})+1$ операций сложения и $l(N-\frac{l-1}{2})$ операций умножения на каждую итерацию. За одну итерацию обработывается по 16 байт с каждого диска, подробнее о том, как проходят эти итерации будет сказано позднее в главе о реализации алгоритма Форни.

Недостатком метода является тот факт, что нам потребуется дополнительно хранить довольно много результатов вспомогательных вычислений.

6.3. Сравнение алгоритмов

Ниже представлена таблица, иллюстрирующая количество операций, необходимых для восстановления сбитых дисков тем или иным алгоритмом. Количество операций приведено для расчета одной итерации (за итерацию по 16 байт, используя SSE инструкции, или по 32 с использованием AVX). В каждом алгоритме есть еще вспомогательные вычисления, которые можно провести для всего страйпа сразу, так как они не зависят от данных на дисках. Пометки об этих действиях можно найти в последнем столбце.

Название алгоритма	Число умножений	Число сложений	Дополнительные действия
Стандартный алгоритм	Nl	$l(N-2)$	Обращение матрицы Вандермонда размером $l \times l$
По свойствам матрицы Вандермонда	$N(l-1)$	$l(N-l-1)$	Построение вспомогательной матрицы
Алгоритм Форни	$l(N+\frac{l-1}{2})$	$l(N+\frac{l+5}{2})+1$	Построение полинома по заданным корням

Видно, что различные методы требуют разное количество операций сложения и умножений, к тому же, дополнительные действия иногда довольно трудоемки. По этим причинам нельзя с точностью сказать, какой подход даст лучший результат. Далее в этой статье мы рассмотрим реализацию алгоритма Форни и сравним его по производительности со страндартным алгоритмом и алгоритмом, использующим свойства матрицы Вандермонда.

7. Реалиация алгоритма Форни

Любой алгоритм восстановления, расчета синдромов или поиска скрытых повреждений, основанный на систематических кодах Рида-Солмона происходит по одной и той же логике. Каждый блок данных представляется как последовательность ячеек определенного фиксированного одинакового размера. Размер этих ячеек выбирается таким образом, чтобы размер блока был ему кратен. Ячейки с каждого блока логически объединяются в линию подобно тому, как блоки объединялись в страйпы. Один проход алгоритма обрабатывает эту линию, то есть по одной ячейке с каждого блока.

Для примера обратимся к RAID6 и рассмотрим расчет самой простой контрольной суммы:

$$S_0 = D_{n-1} + D_{n-2} + \ldots + D_1 + D_0. \tag{22}$$

Тогда расчет происходит следующим оюразом. Мы читаем по одной ячейке из каждого блока, накапливая сумму, и результат пишем в соответствующую ячейку блока с первой контрольной суммой. Если мы читали первые ячейки, то и результат запишется в первую ячейку синдрома, если вторые, то и результат запишется во вторую ячейку и так далее, пока не достигнем конца блока.

Более формально: пусть каждый блок состоит из w ячеек: $Y_i = (y_{i,0}, \ldots, y_{i,w-1})$. Тогда

$$S_{0,j} = D_{n-1,j} + D_{n-2,j} + \ldots + D_{1,j} + D_{0,j} \qquad j = 0, \ldots w-1. \tag{23}$$

По аналогичной схеме считаются все котрольные суммы и работает алгоритм восстановления.

Соответственно, в наших прототипах мы использовали размер ячейки по 1 байту, чтобы слагаемые помещались в char. В векторизованных с помощью SSE вариантах мы несколько перестраивали способ представления наших данных при сложении и умножении и обрабаотывали ячейки по 16 байт.

При реализации прототипа нам не важна была скорость восстановления данных, однако, в конечном варианте, достижение максимальной скорости - первоочередная задача, поэтому надо было внимательно рассмотреть все вопросы реализации, правильные ответы на которые помогут выиграть в производительности, не нанося ущерб используемой памяти. Рассмотрим подробнее проблемы, с которыми мы столкнулись.

Первый вопрос, который возникает, это вопрос о быстром построении полинома $\sigma(x)$ по его корням. Существует алгоритм, позволяющий нам решать эту задачу без использования дополнительной памяти за $O(l)$. Пусть $(q_1^{-1}, \ldots, q_l^{-1})$ - корни многочлена $\sigma(x)$. Будем строить $\sigma(x)$ итеративно:

$$\sigma_1(x) = 1 + xq_1, \sigma_i(x) = (1 + xq_i)\sigma_{i-1}(x), \qquad i = 2, \ldots, l \tag{24}$$

Таким образом, корнями полинома $\sigma_l(x)$ степени l будут $(q_1^{-1}, \ldots, q_l^{-1})$, то есть мы

построили искомый многочлен.

Распишем подробнее i-ый шаг:

$$\sigma_i(x) = (1 + xq_i)\sigma_{i-1}(x) = \sigma_{i-1}(x) + xq_i\sigma_{i-1}(x) \tag{25}$$

Если представить эти полиномы как массивы, где на j-ом месте стоит коэффициент при x^j, то умножение многочлена на x это сдвиг всех коэффициентов влево. Затем все коэффициенты следует умножить на q_j и поэлементно сложить с массивом до преобразований. Очевидно, этот алгоритм легко векторизовать с использованием инструкций SSE или AVX, что является его дополнительным достоинством.

Второй вопрос заключался в том, как считать λ_j а точнее можно ли хранить все эти заранее посчитанные значения? Казалось бы, они не зависят от данных в блоках и их можно заранее вычислить, записать в таблицу и потом к ним обращаться. Но λ_j зависят от номеров сбитых дисков и их количества, к сожалению, их число получается довольно большим, а хранить такие большие таблицы не целесообразно. Однако, можно хранить значения $(1+a^{k_j-k_i})^{-1}$. Этих значений всего 256, перемножив нужные из них в зависимости от номеров сбитых дисков можно получить искомые λ_j.

Далее рассмотрим задачу оптимального расчета Ω_i по формуле (20). Понятно, что реализовывать честное умножение матрицы такого рода на вектор в данном случае не желательно по памяти и производительности. Можно обойтись умножением векторов заданной длины и при подсчете каждого коэффициента Ω_i уменьшать длину, передаваемую в функцию умножения. Тем самым мы сэкономим память по сравнению со стандартным подходом, не создавая целую матрицу, и время, не производя умножения на нулевые элементы.

Теперь рассмотрим возможные подходы к непосредственному восстановлению данных, когда уже проведены все вспомогательные расчеты. Пусть у нас подсчитаны контрольные суммы и все вспомогательные значения, а именно, найдены многочлены $\Omega(x)$ и $\sigma(x)$ и значения λ_j. Тогда существует два подхода: либо восстанавливать по блокам (сначала полностью восстановить первый блок, потом второй и так далее), либо как и раньше восстанавливать, обрабоатывая по 16 байт в каждом блоке. Плюсом первого подхода является то, что нам не придется хранить $\Omega(a^{-(N-k_j-1)})$ для каждого k_j, однако, это нарушает логику нашей работы и из-за особенностей кеширования может пагубно сказаться на производительности. Как и получилось на практике.

8. Тестирование

Нашей командой были созданы векторизованные с помощью SSE реализации всех трех алгоритмов. Мы провели сравнение их производительности на практике. Тестирование происходило следующим образом.

1. Назначался алгоритм, который будет тестироваться.
2. В памяти выделялся массив блоков размером по 4 Кб, лежащих подряд. По n блоков для данных, которые заполнялись случайными последовательностями битов, и по m блоков для контрольных сумм. Причем, блоки для контрольных сумм выделялись в конце массива.
3. Запускался алгоритм расчета синдромов и последние m блоков заполнялись полученными данными.
4. В режиме проверки корректности алгоритма делалась копия всего участка памяти с блоками.
5. Случайным образом из $m + n$ дисков выбиралось m, которые помечались как отказавшие, и информация на которых заменялась на другую случайную последовательность бит.
6. Запускался таймер.
7. Вызывался выбранный алгоритм тестирования.
8. После завершения работы алгоритма таймер останавливался.
9. В режиме проверки корректности алгоритма сверялась копия страйпа, сделанная в п.4, и страйп, полученный после восстановления

Сначало запускалось по 1000 таких тестов для каждой конфигурации m и n и для каждого из алгоритмов для проверки правильности их работы. Потом выбирались некоторые пары m и n и запускалось по 1000000 тестов на каждый алгоритм для каждой из пар для замеров производительности. Объем страйпа делился на показатели таймеров в секундах, чтобы получить данные по восстановлению в Mb/s. Отбрасывалось по 5% минимальных и максимальых значений. Затем по оставшимся считалась средняя скорость.

Замеры проводились на машине:

- Intel® Core™ i3-4000M CPU @ 2.40GHz × 4
- ОЗУ: 3.8 GiB

- 64-bit
- ubuntu 12.04

Код компилировался компилятором gcc с ключом оптимизации -O2

Ниже представлены графики замеров. На графиках показана зависимость скорости работы алгоритмов от числа поврежденных дисков, совпадающем с числом синдромов, при фиксированном n и зависимость скорости работы при фиксированом числе синдромов и варьирущемся числе дисков с данными. Как видно из графиков, алгоритм, использующий свойства матрицы Вандермонда показывает наилучшую производительность. Однако, алгоритм Форни, реализованный мной, оказался хуже стандартного на практике.

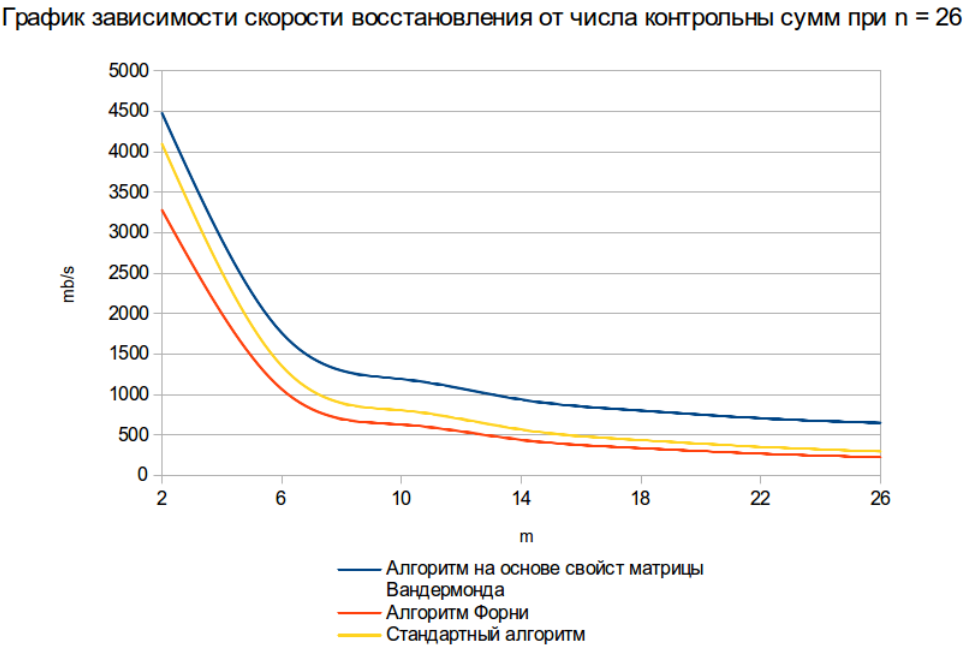


График зависимости скорости восстановления от числа информационных блоков при m = 2

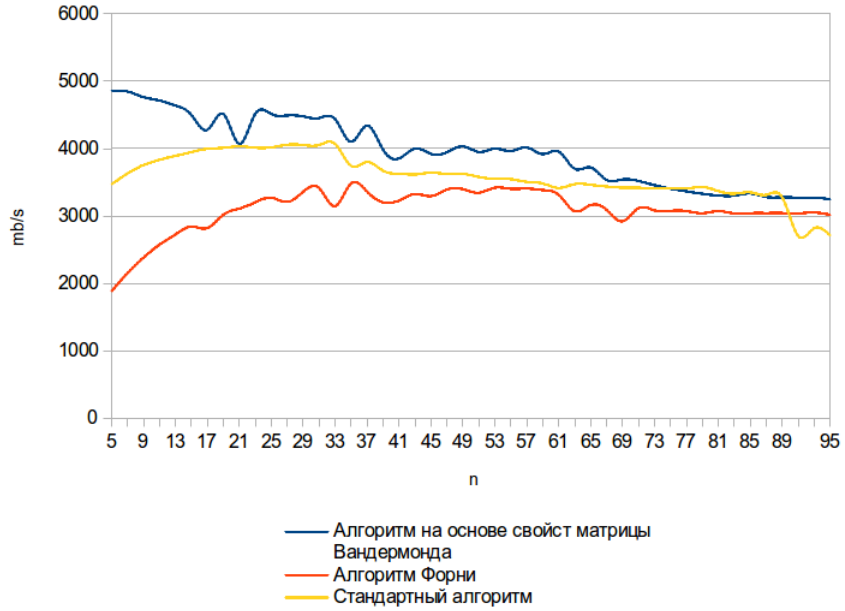
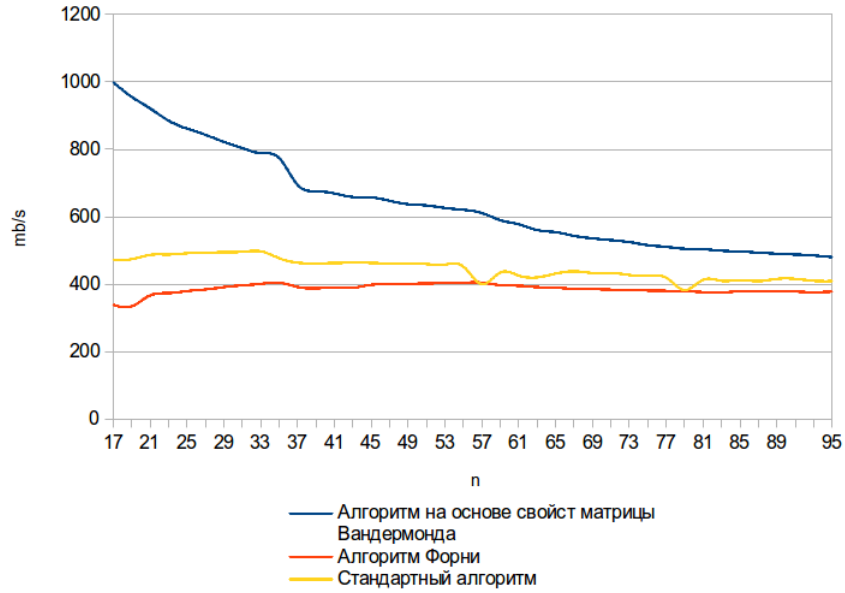


График зависимости скорости восстановления от числа информационных блоков при m = 16



Заключение

Нам удалось разработать модуль восстановления данных в RAID($m+n$). Мы предложили три алгоритма, один из которых общеупотребим, подробно изучили, создали рабочие прототипы и векторизовали их реализации с использованием SSE. Провели замеры и тесты и убедились в том, что наши алгоритмы работают правильно и .

Все открытые реализации, найденные и изученные нами, используют стандартный алгоритм, наши же алгоритмы показали заметно лучшую производительность по сравнению с ним. Выбор между ними стоит осуществлять в зависимости от того, какое количество дисков мы хотим отвести дя хранения информации, а какое отдать для обеспечения надежности системы.

Кроме того, мы разработали несколько модификаций алгоритмов, которые можно подробнее исследовать и реализовать в дальнейшем. Планируется переписать реализацию векторных операций сложения и умножения с использованием AVX2 и сравнить с полученными результатами. Также требуется реализовать модуль поиска SDC, без которого полноценная работа с RAID($n + m$) не представляется возможной.

Список литературы

[1] Э. Берлекэмп. *Алгебраическая теория кодирования.* — М., Мир, 1971

[2] Т. Кормен and Ч. Лейзерсон and Р. Ривест and К. Штайн. *Алгоритмы. Построение и анализ.* — Вильямс, 2012.

[3] А. Ю. Утешев. *Поля Галуа.* — 2012. http://pmpu.ru/vf4/gruppe/galois

[4] B. Beach. *What Hard Drive Should I Buy?* — 2014. http://blog.backblaze.com/2014/01/21/what-hard-drive-should-i-buy

[5] Eduardo Pinheiro and Wolf-Dietrich Weber and Luiz André Barroso *Failure Trends in a Large Disk Drive Population* — 2007. 5th USENIX Conference on File and Storage Technologies (FAST 2007). 17-29