САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Зайберт Валерия Сергеевна

Разработка модуля вычисления синдромов и восстановления утраченных дисков в RAID-массиве с использованием арифметики поля $GF(2^8)$

Курсовая работа

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

Санкт-Петербург
2013

Оглавление

Введение **3**

1. Выбор поля **4**

2. Формат данных и формулы расчета **5**

3. Реализация **7**

4. Замеры **8**

5. Дальнейшие исследования **10**

Заключение **11**

Введение

Системы хранения данных (СХД) это комплексное решение для хранения больших объемов информации, а так же быстрого и бесперебойного доступа к ней. Существует множество различных технологий, обеспечивающих выполнение таких требований. Одна из них RAID, что расшифровывается как Redundant Array of Independent Disks— «отказоустойчивый массив из независимых дисков», концепция которой состоит в объединении нескольких дисков для обеспечения отказоустойчивости. Кроме того, RAID распараллеливает процесс чтения и записи на все диски, что увеличивает скорость доступа к информации. В частности, в данной работе рассматривается RAID6, использующий для восстановления сбойных данных две разные контрольные суммы, называемые синдромами. Данная технология позволяет восстановить до двух отказавших дисков в дисковом массиве.

Целью нашей исследовательской работы было сравнить новый алгоритм с ранее существующими, и постараться уменьшить размер исходных файлов без значительной потери в производительности или даже с ее приростом.

Задачей данной работы было реализовать эффективное вычисление синдромов при помощи кодов Рида-Соломона в поле $GF(2^8)$, восстановление данных на дисках и сравнение с предыдущими результатами. Использование кодов Рида-Соломона позволит нам в будущем реализовать Silent Data Corruption, то есть восстановление данных в тех случаях, когда нам не известно место сбоя [5][6]. К тому же операции, позволяющие реализовать данный метод, не трудоемки.

1. Выбор поля

Существует много различных способов кодирования информации в помощью двоичных кодов, применяемых для защиты информации от ошибок. Одним из способов являются коды Рида-Солмона, которые используются в нашей задачи. Этот подход позволяет нам восстанавливать данные после многократных ошибкок. Также использование кодов Рида-Соломона позволит нам в будущем реализовать Silent Data Corruption, то есть восстановление данных в тех случаях, когда нам не известно место сбоя в RAID-массиве[5][6]. В добавок, операции, которые используются в данном методе, дают возможность нетрудоемкой реализации.

Коды Рида-Соломона работают с полями Галуа порядка p^m, где p - простое число. В нашем случае, $p = 2$. С точки зрения практического подхода, можно считать, что каждый элемент поля это вектор длины m, состоящий из 0 и 1. Иногда также удобно представлять элементы поля как полиномы степени m с коээфицинтами 0 или 1.

Теперь необходимо определиться, какое именно поле нам нужно выбрать. Мы остановились на $GF(2^8)$ из-за следующих его преимуществ:

- большая часть работ в этой области проводилась именно с этим полем. Таким образом, можно сравнить, выигрывает ли наше решение в скорости за счет непосредственно алгоритмов расчета, распараллеливания и способов представления информации;

- данное поле дает возможность обрабатывать до 255 дисков. Если брать поля меньшей размерности, то максимальное количество дисков резко падает. К примеру, поле $GF(2^4)$ позволяет использовать только 15 дисков;

- За счет сравнительно небольшого поля мы имеем возможность хранить некоторые, необходимые для работы алгоритма значения в заранее рассчитанных таблицах и эти таблицы будут иметь небольшой размер

2. Формат данных и формулы расчета

У нас имеется массив дисков. Каждый диск разбивается на блоки одинакового размера так, что размер блока является делителем размера диска. Все блоки с одинаковыми номерами образуют страйп (Stripe). Эти блоки будем обозначать как $D_0, D_1, \ldots, D_{N-1}$, где N - количество дисков без учета дисков для хранения синдромов. Для каждого страйпа часть блоков выделена специально для хранения синдромов. Так как для RAID6 мы используем два синдрома, то необходимо выделить два блока. Не умаляя общности можно считать, что это последние блоки страйпа. Остальные блоки хранят исходную информацию.

Для вычисления контрольных сумм в RAID6 используются формулы вида:

$$P = D_{N-1} + D_{N-2} + \ldots + D_0$$

$$Q = x^0 D_{N-1} + x^1 D_{N-2} + \ldots + x^{N-1} D_0$$

где x - примитивный элемент поля, например, в нашем случае $x = \{02\}$ [2].

Используя схему Хорнера, можно представить формулы таким образом, чтобы в них использовались только умножения на примитивный элемент поля и сложения:

$$Q = ((\ldots D_0 \ldots)x + D_{N-3})x + D_{N-2})x + D_{N-1}$$

В качестве неприводимого многочлена в поле лучше всего брать такой, чтобы единиц в нем было как можно меньше. В этом случае уменьшится количество операций при умножении на x[6]. В поле $GF(2^8)$ мы можем рассматривать многочлен $f = x^8 + x^4 + x^3 + x^2 + 1$ В нашем алгоритме распараллеливания и распределения данных у нас получилось умножать или складывать 256, при использовании AVX, и 128, при использовании SSE, элементов за то же количество операций, которое мы ранее тратили на один элемент.

Рассмотрим теперь формулу для восстановления данных (j и k - номера сбойных дисков, $k > j$):

$$D_k = P + \sum_{i=0, i \neq k}^{N-1} D_i$$

$$D_j = \frac{(P + \tilde{P}) + (Q + \tilde{Q})x^{-(n-k-1)}}{x^{k-j} + 1}$$

Здесь появляется умножение на x в некоторой отрицательной степени и деление на некий элемент поля. Так как мы находимся в поле, то

$$x^{k-j} + 1 = x^i$$

i мы сможем найти, используя предподсчитанные таблицы степеней примитивного элемента и логарифмов элементов нашего поля. Далее с помощью малой теоремы Ферма эту формулу можно свести всё к следующей:

$$D_j = ((P + \tilde{P}) + (Q + \tilde{Q})x^{2^8-1-(n-k-1)})x^{2^8-1-i}$$

Стоит отметить, что умножение на произвольный элемент поля так же сводится через факторизацию к умножениям на x и сложениям в зависимости от битов элемента, на который умножаем [4]:

$$\begin{gathered} a(x)b(x) = \\ = (a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0)(b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots + b_0) = \\ = ((\ldots(b_{n-1}a(x)x + b_{n-2}a(x))x + b_{n-3}a(x))x + \ldots + b_1 a(x))x + b_0 a(x) \end{gathered}$$

Есть два способа производить умножение:

- Массив функций, где i-ая функция умножает свой аргумент на x^i
- Обобщенное умножение. Используя факторизацию, мы перемножаем два элемента.

3. Реализация

Наша реализация алгоритма использует векторные операции и по-новому распределяет данные в блоках, за счет чего резко уменьшается число необходимых операций для обработки страйпа.

Для повышения эффективности кода мы старались уменьшить количество условных переходов и циклов, так как на них падает производительность про сравнению с линейным кодом [3][1]. Для этого для каждого числа дисков был реализован свой набор функции. Функции писались не вручную, их создавали код-генераторы. Получились довольно объемные файлы, в каждом из которых функции решающие одну и туже задачу, но для разного количества дисков. В качестве эксперимента и для уменьшения размера модуля мы попробовали различные способы сворачиваний основных этапов вычислений.

В конце работы приведены графики, на которых видна зависимость производительности от различных степеней свернутости циклов.

Для реализации нами был выбран язык C, а не ассемблер, так как первый имеет определенные плюсы, представленные ниже.

- Простота переносимости с одной архитектуры на другую. Нам нет нужды беспокоиться о том, что на разных архитектурах могут отличаться ассемблерные команды.

- Возможность использования оптимизаций компилятора. Современные компиляторы предоставляют различные возможности оптимизации, позволяющие наиболее эффективно распределять память, работать с регистрами и преобразовывать код. [3][1]

- Отсутствие необходимости распределять регистры. Для хранения каждого синдрома нам необходимо по 8 регистров. Сейчас, используя RAID6, нам нужно только два синдрома, но мы уже задействуем все 16 регистров. В дальнейшем планируется увеличивать количество синдромов, но тогда нам потребуется больше регистров, чем у нас есть. Возникает вопрос о порядке операций и грамотном распределении регистров. В предыдущем пункте указано, что этот вопрос одним из лучших способов сможет решить компилятор. [3][1]

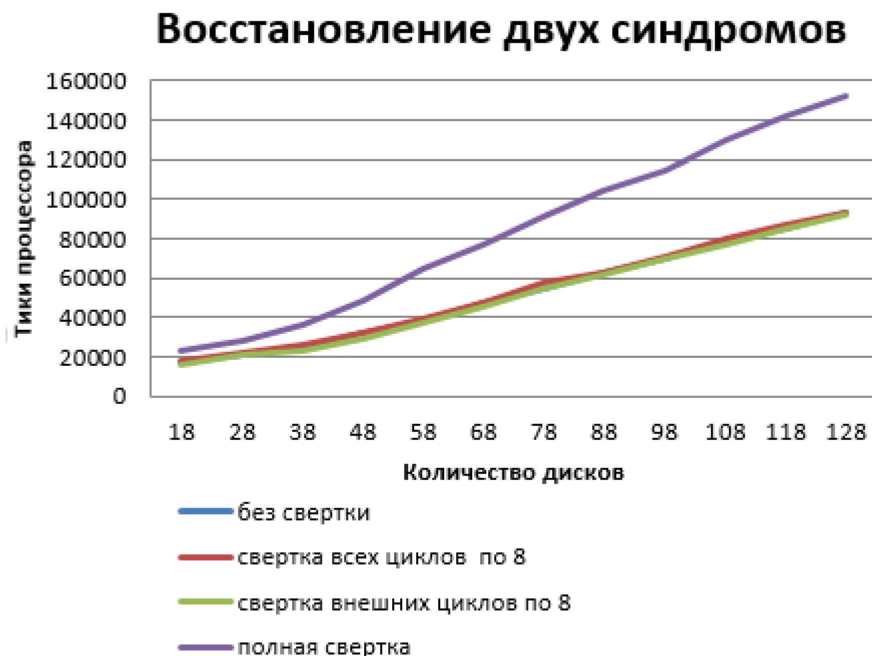
4. Замеры

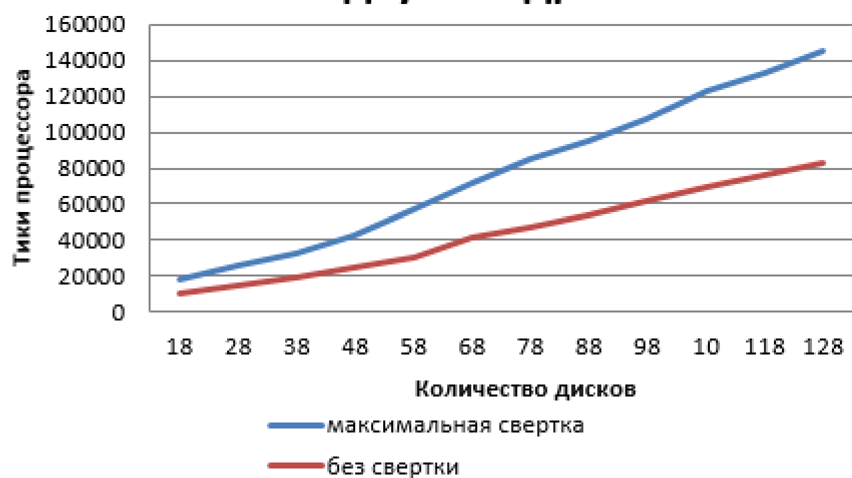
Ниже приведены графики сравнения различных реализаций для восстановления двух синдромов. Измерения производились на следующей конфигурации:

- ОС: Debian 7.0
- Тип ОС: x64
- CPU: Intel® Xeon(R) CPU E5-2620 0 @ 2.00GHz × 18
- Оперативная память: 39.4 Гб

Код собирался gcc 4.7, с ключом оптимизации –O3.

В итоге мы получили, что за счет оптимизации компилятора только один вид свертки дает отличные результаты от реализации без сверток. При восстановлении двух дисков максимальная свертка работает в среднем в два раза медленнее, но исходный код весит почти в семь раз меньше. При расчете двух синдромов максимальная свертка медленнее на 80%, а исходный код меньше в 5,3 раза. Нам хотелось уменьшить размер кода, но не ценой падения производительности в два раза.




Расчет двух синдромов
160000
140000
120000
100000
80000
60000
40000
20000
0
Тики процессора
18
28
38
48
58
68
78
88
98
10
118
128
Количество дисков
максимальная свертка
без свертки

5. Дальнейшие исследования

В ближайшее время хотелось бы попробовать исследовать различные опции компилятора, чтобы найти золотую середину между размером кода и производительностью.

В дальнейшем планируется реализовать исправление ошибок Silent Data Corruption, выполнить подсчет трех синдромов, что позволит нам восстанавливать до трех сбойных дисков в массиве. Сделать восстановление сбойных дисков с помощью обобщенного умножения и сравнить результаты.

Заключение

Получилось реализовать распараллеливание вычислений и перераспределение данных в блоках, что позволило заметно ускорить алгоритмы обработки страйпа. Исследованы различные способы свертки циклов функций расчета синдромов и восстановления данных, что позволило уменьшить размер исходного кода, произведены сравнения этих реализаций.

Список литературы

[1] Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, Second Edition. — Pearson Education, Inc, 2007.

[2] Anvin H. Peter. The mathematics of RAID-6. — 2011.

[3] The Software Optimization Cookbook High-Performance Recipes for IA-32 Platform / Richard Gerber, Aart J.C. Bik, Kevin B. Smith, Xinmin Tian. Second Edition. — Intel Press, 2006.

[4] Питерсон У., Уэлдон Э. Коды, исправляющие ошибки. — М. : Мир, 1976.

[5] Утешев А.Ю. Математика отказоустойчивых дисковых массивов. — http://pmpu.ru/vf4/codes/raid.

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