САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Веселков Иван Дмитриевич

Разработка модуля вычисления синдромов и восстановления утраченных дисков в RAID-массиве с использованием арифметики поля $\mathrm{GF}(2^{256})$

Курсовая работа

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

Санкт-Петербург
2013

Оглавление

Введение

В нашем современном мире, где связаться с другом из другой части планеты так же просто, как сделать несколько щелчков мышкой, информация представляет из себя основную ценность. Информация – это данные с фондовой биржи, статистика пользователей сайта, докторская диссертация ученого, всё это информация. И, естественно, для своих владельцев она обладает большой значимостью. Брокер должен всегда иметь данные под рукой. Рекламодатель хочет, чтобы все это хранилось в удобном для него виде. А ученому достаточно того, что он может не беспокоиться за сохранность своей работы.

Для этого они используют СХД, системы хранения данных. Такие системы призваны обеспечивать быстрый и постоянный доступ к необходимой информации. Существует несколько способов организации таких систем. Один из них – объединение дисков в RAID-массивы. Такие массивы несложны в построении и настройке, и обладают всеми необходимыми качествами для хранения информации. Поэтому они очень распространены в мире.

Но эффективность их работы зависит от различных нюансов строения массива. Например RAID-6 обеспечивает большую скорость работы за счет одновременного считывания небольшого количества информации с нескольких дисков и отказоустойчивость за счет хранения контрольных сумм.

В данной работе рассматривается подход к вычислениям в RAID с использованием арифметики конечных полей $GF(2^{256})$. Использование такого поля позволяет реализовать параллельную обработку большого числа элементов за очень малое число векторных операций. Также в работе исследуется влияние технических ограничений на производительность алгоритма. Разработаны код-генераторы для автоматического написания кода функций восстановления двух отказавших дисков. Проведены тесты корректности и замеры производительности получившихся функций.

1. Основные термины и технологии

Система хранения данных (СХД) – это комплекс, который позволяет надёжно хранить большой объем данных и предоставляет к нему бесперебойный и быстрый доступ. В случае поломки одного или нескольких жестких дисков данные могут быть утрачены. Для их восстановления используется RAID, избыточный массив независимых дисков. Вычисляются контрольные суммы (синдромы), для которых требуется дополнительное дисковое пространство.

Страйп (Stripe), основная единица обработки данных в системе хранения, разбивается на буферы (Buffers) одинакового размера, обозначаемые $D_0, D_1, D_2 ... D_{N-1}$. Для организации вычислений буфера разбиваются на блоки (Blocks) одинакового размера. Размер блока является делителем размера буфера. Количество буферов N равно количеству дисков данных в массиве. В рамках одного страйпа буфера могут называться дисками. Для обеспечения отказоустойчивости в страйп вводятся дополнительные буфера, предназначенные для хранения синдромов. Их размер совпадает с размером буферов данных.

Наиболее надежным и экономным по занимаемому дисковому пространству является структура RAID-6, дисковый массив с чередованием, использующий две контрольные суммы, вычисляемые двумя независимыми способами, таким образом мы можем восстанавливать данные при выходе из строя до двух жестких дисков в дисковой группе. За счет чередования достигается параллельное выполнение процесса чтения/записи на все диски, что обеспечивает высокую скорость доступа к данным.

2. Постановка задачи

С помощью кода Рида-Соломона можно эффективно реализовать вычисление синдромов и восстановление утраченных дисков с элементами в поле $GF(2^{256})$, а также сравнить производительность с полями размера 2^8 и 2^{16}. [5]

При работе с элементами из поля $GF(2^{256})$ технология RAID-6 имеет несколько преимуществ:

- Эффективность операции умножения на x по сравнению с непараллельной реализацией

За счет реализации вычислений и использования технологий SSE при одном умножении количество инструкций совпадает с количеством единиц в двоичном представлении образующего многочлена поля. В нашем случае на умножение на x большого числа элементов расходуется всего три процессорных инструкции. [7]

- Увеличение максимального количества дисков по сравнению с полями $GF(2^8)$ и $GF(2^{16})$.

Больший размер элемента поля позволяет работать с огромным набором дисков, около $1.16*10^{77}$, что примерно равно $0,12\%$ от количества всех атомов в обозримой Вселенной. Это дает нам возможность полностью пренебречь их количеством.

- Оптимальное использование процессорного кэша

Цифра 256 была выбрана не случайно. Длина страйпа имеет вполне определённое значение в 4 Кбайта. Если мы разобьем его на переменные по 16 байт, чтобы можно было их обрабатывать с помощью технологии SSE, то получим как раз 256 переменных. Будем представлять их совокупность как многочлен поля $GF(2^{256})$.

Для вычисления контрольных сумм в RAID-6 используются формулы вида

$$P = D_0 + D_1 + D_2 + ... + D_{n-1}$$

$$Q = g_0 * D_0 + g_1 * D_1 + g_2 * D_2 + ... + g_{n-1} * D_{n-1}$$

Где g_i - произвольнй элемент поля [1]. Используя схему Хорнера, можно представить формулу вычисления Q так, чтобы в ней использовались только операции умножения на примитивный элемент поля и сложения.

$$Q = D_0 + x * (D_1 + x * (D_2 + \ldots + x * (D_{n-2} + x * D_{n-1}) \ldots))$$

Умножение на произвольный элемент поля также сводится к комбинации сложений и умножений на примитивный элемент. Количество операций сложения зависит от битов элемента, на который происходит умножение

$a(x) * b(x) = (a_{n-1} * x^{n-1} + a_{n-2} * x^{n-2} + \ldots + a_1 * x + a_0) * (b_{n-1} * x^{n-1} + b_{n-2} * x^{n-2} + \ldots + b_1 * x + b_0) = b_0 * a(x) + x * (b_1 * a(x + \ldots + x * (b_{n-3} * a(x) + x * (b_{n-2} * a(x) + x * b_{n-1} * a(x))) \ldots)$

b_i– это биты элемента, на который производится умножение. То есть в зависимости от значения этих битов будут осуществляться умножения на x и сложения. [6]

Для восстановления утраченных дисков D_j и D_k использовались следующие формулы:

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

$$D_k = P + \bar{P} + D_j$$

Где $\bar{P}$ и $\bar{Q}$, пересчитанные контрольные суммы, при D_j и $D_k = 0$;

При восстановлении данных появляется операция умножения на ^{n}. Она была реализована через умножение на произвольный многочлен $a(x)$, при $a(x) = x^n$

3. Инструменты

Был написан генератор кода для создания отдельных функций для каждого количества дисков (от 5 до 128) с последующим объединением их в массив. Это вызвано тем, что условные переходы понижают производительность по сравнению с вычислением напрямую. Генератор позволил автоматизировать написание большого количества похожих функций.

Для написания генератора был выбран язык язык C, по следующим причинам:

1. Субъективно, реализовать на нём было проще, нежели мы бы пытались написать это на языке ассемблера.

2. При вычислении векторных операций нам требуется большое количество регистров под переменные. Компилятор берет работу по их распределению на себя, причем выполняет, конечно, оптимальнейшим способом. [3] [4]

3. В случае изменения архитектуры выполняющего процессора нам будет достаточно переписать определяющие функции (сложение, умножение на примитивный элемент поля).

4. Оптимизация, которую нам предоставляет компилятор, по времени затраченному на робту молуля является лучшей, чем мы бы реализовывали её самостоятельно.

Для всех возможных конфигураций систем (от 5 до 128) были осуществленны замеры длительности работы алгоритма в тактах процессора. Для этого была приминена функция `__rdtsc()`.

Используемая конфигурация тестовой машины:

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

- Оперативная память: 39.4 Гб

Генераторы кодов собирались в gcc 4.7, оптимизация –O3.

4. Реализация

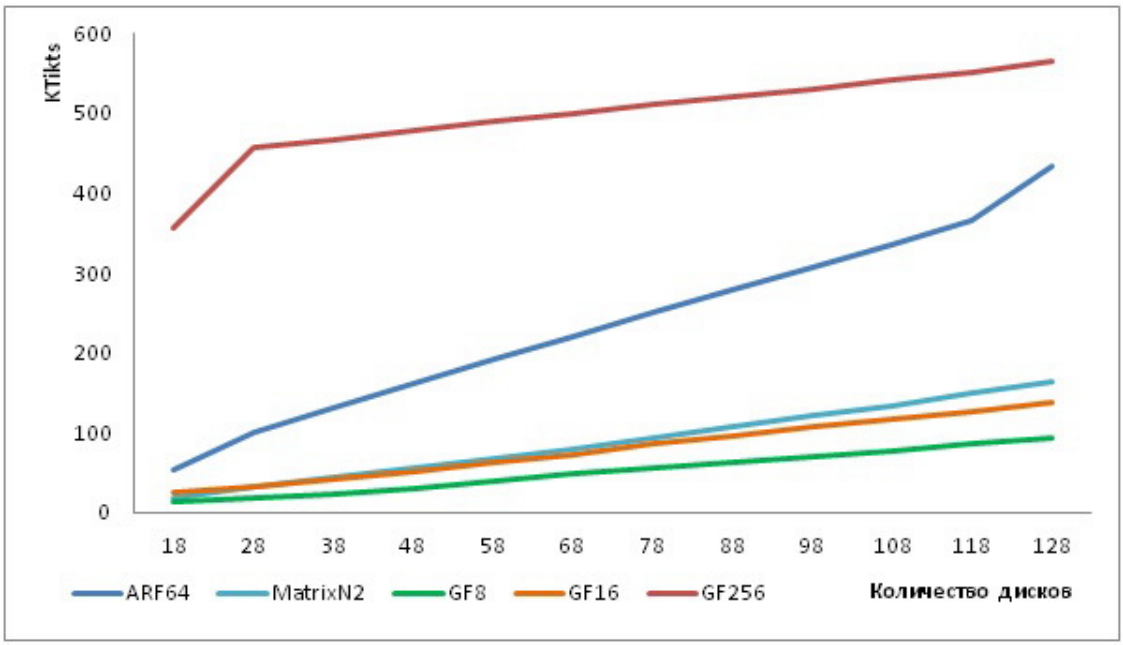

Диаграмма 1: Восстановление двух дисков.

Выше приведены замеры работы функции восстановления (Диаграмма 1) для различного количества дисков, от 18 до 128. Для сравнения приведены результаты замеров работы тех же функций из модулей, работающих с полями $GF(2^8)$ и $GF(2^{16})$.

В теории, обработка целого страйпа должна была проходить очень быстро. Например, умножение на x сильно зависит от количества значащих единиц в образующем многочлене. Таким образом, мы стараемся выбрать многочлен с их наименьшим количеством. В нашем случае использовался многочлен $x^{256} + x^{10} + x^5 + x^2 + 1$, что позволяет выполнять умножение всего за три операции сложения. [2] И, поскольку мы обрабатываем целый страйп за несколько операций, то ожидеается существенный прирост в скорости обработки целого страйпа.

Как видно из диаграммы, функция показала себя гораздо хуже, чем все другие алгоритмы. В целом, исследования показали, что это вызвано очень долгой работой функции умножения на x в произвольной степени. Для сравнения: в алгоритмах полей $GF(2^8)$ и $GF(2^{16})$ для умножения на x в произвольной степени использовался предподсчитанный

массив функций умножения на x в конкретной степени. В то время как мы реализовали умножение через последовательные сложения и умножения на примитивный элемент поля.

Немалую роль сыграло и отсутствие у процессора быстрой операции получения произвольного бита 256-битной переменной, которая часто используется в наших факторизованных операциях. В итоге нам приходится вначале выделить из нашей переменной 64 бита, а затем применять операцию сдвига. Также сильно сказалось большое количество обращений к памяти, которое, в совокупности с предыдущими факторами и привело к более долгому времени работы в сравнении с остальными алгоритмами.

5. Планы на дальнейшую работу

В дальнейшем планируется написать функции для восстановления трех дисков, а также функции для исправления скрытых повреждений информации. Возможно в дальнейшем, при увеличнии числа регистров на процессорах, у данного алгоритма появятся преимущества. Также планируется реализовать работу функций в полях $GF(2^{64})$ и $GF(2^{128})$, по скольку в этих полях мы получим уменьшение сложности операции умножения в 16 и 4 раза соответственно. При работе с $GF(2^{128})$ будет проведено исследован алгоритм использующий AVX регистры. В случае с $GF(2^{64})$ мы имеем готовую инструкцию процессора для получения произвольного бита переменной.

Список литературы

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

[2] Seroussi Gadiel. Table of Low-Weight Binary Irreducible Polynomials. — HPL-98-135, 1998.

[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] Кнут Д.Э. Искусство программирования. Т.2: Полученные алгоритмы. — Издательство Вильямс, 2004.

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

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

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