САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного программирования

Овсянникова Василиса Павловна

Исследование методов векторизации расчётов в RAID

Курсовая работа

Научный руководитель:
Разработчик исследовательской лаборатории RAIDIX Маров А. В.

Санкт-Петербург
2015

Оглавление

Введение

Объёмы информации, хранимой человеком, растут с каждым годом. Системы хранения данных (СХД) [9] - программно-аппаратные решения по организации надёжного хранения информационных ресурсов и предоставления гарантированного доступа к ним, - активно используются в самых различных областях, таких как научные исследования, криминалистика, медицина и многие другие. Поэтому задача безопасного хранения данных и обеспечения быстрого доступа к ним является очень актуальной. Одним из решений является RAID (Redundant Array of Independent Disks) [7] - массив из нескольких дисков, управляемый контроллером.

Существует множество модификаций RAID:

- **RAID 0**: данные делятся на столько частей, сколько дисков в массиве, и последовательно записываются на них. Отказоустойчивость не предусмотрена.
- **RAID 1** (зеркалирование): одни и те же данные записываются на два диска одновременно. Такой вариант хранения данных надёжен, но влечёт очень большую избыточность - 50%.
- **RAID 2**: массивы этого типа основаны на использовании кода Хэмминга [6]. Диски делятся на две группы: для данных и для их восстановления. Если для хранения данных используется $2^n - n - 1$ дисков, то для второй группы необходимо n дисков. Данная модификация позволяет восстанавливать на лету один утраченный диск и обнаруживать утрату двух, однако из-за большой избыточности не получила широкого распространения.
- **RAID 3**: модификация RAID 2, заменившая способность восстановления утраченного диска на лету на меньшую избыточность. RAID 3 может восстановить один утраченный диск, но не сразу же после его утраты, зато избыточным является только один диск. Главный недосток такого способа хранения данных - большая нагрузка на избыточный диск, влекущая к частому выходу этого диска из строя.
- **RAID 4**: в массиве из n дисков $n - 1$ диск используется для хранения данных и один - для хранения контрольной суммы. RAID 4 может восстановить один утраченный диск, и избыточность в таком типе хранения данных очень мала - один диск. Минусом данной модификации является низкая скорость записи.
- **RAID 5**: по сравнению с предыдущими способами хранения данных, RAID 5 допускает параллельную запись, поскольку блоки данных и контрольные суммы

циклически записываются на все диски массива. Скорость записи эта модификация снижает, но даёт большой выигрыш при чтении. Так же RAID 5 позволяет восстановить до одного утраченного диска.

- **RAID 6**: данная модификация так же позволяет параллельную запись, однако избыточность в ней на один диск больше, так как в RAID 6 хранятся две контрольные суммы, так же называемые синдромами. RAID 6 обеспечивает восстановление одного или двух утраченных дисков, а так же позволяет разрешить одно SDC (Silent Data Corruption) - ошибку, при которой нам неизвестен номер испорченного диска. Ранее предполагалось, что номера утраченных дисков нам известны.

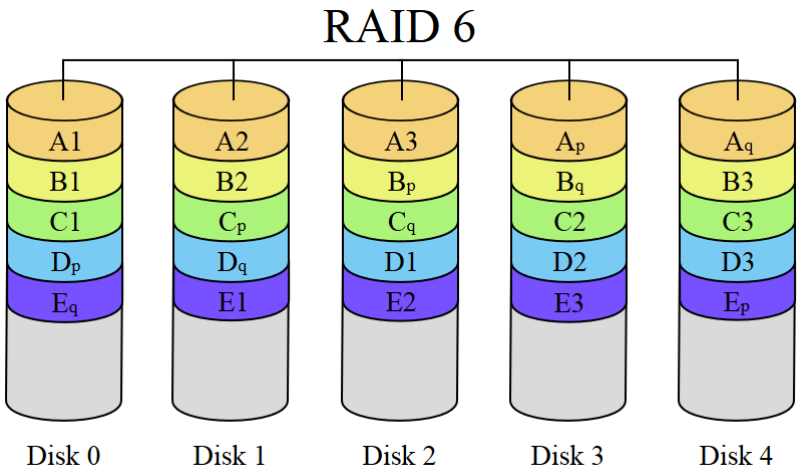

Рис. 1: Структура RAID 6.

Можно видеть, что технология RAID 6 предоставляет больше возможностей для восстановления данных, поэтому нами была выбрана именно эта модификация RAID-массива.

Однако помимо безопасности хранения информационных ресурсов необходимо обеспечить высокую скорость доступа к ним. С одной стороны, RAID 6 позволяет распараллелить чтение и запись, что увеличивает производительность, с другой - необходимость кодирования для записи служит её уменьшению. Мы хотим минимизировать потери производительности от кодирования, оптимизировав его алгоритмы.

В рамках нашего исследования будут рассмотрены существующие высокопроизводительные алгоритмы RAID вычислений, а также разработаны собственные, после чего будет проведён их сравнительный анализ.

1. Цель и постановка задачи

Цель данной работы - оптимизация алгоритмов кодирования в RAID 6.

Увеличивать скорость кодирования мы будем путём векторизации RAID вычислений и минимизации чтений с диска (кодирование выполняется значительно быстрее). Для этого мы будем использовать коды Рида-Соломона [1], которые в свою очередь используют арифметику полей Галуа [10].

Задача данной работы:

- Изучить существующие методы векторизации расчётов в RAID;
- Разработать и реализовать собственные алгоритмы;
- Сравнить их скорости со скоростями существующих алгоритмов.

2. Терминология

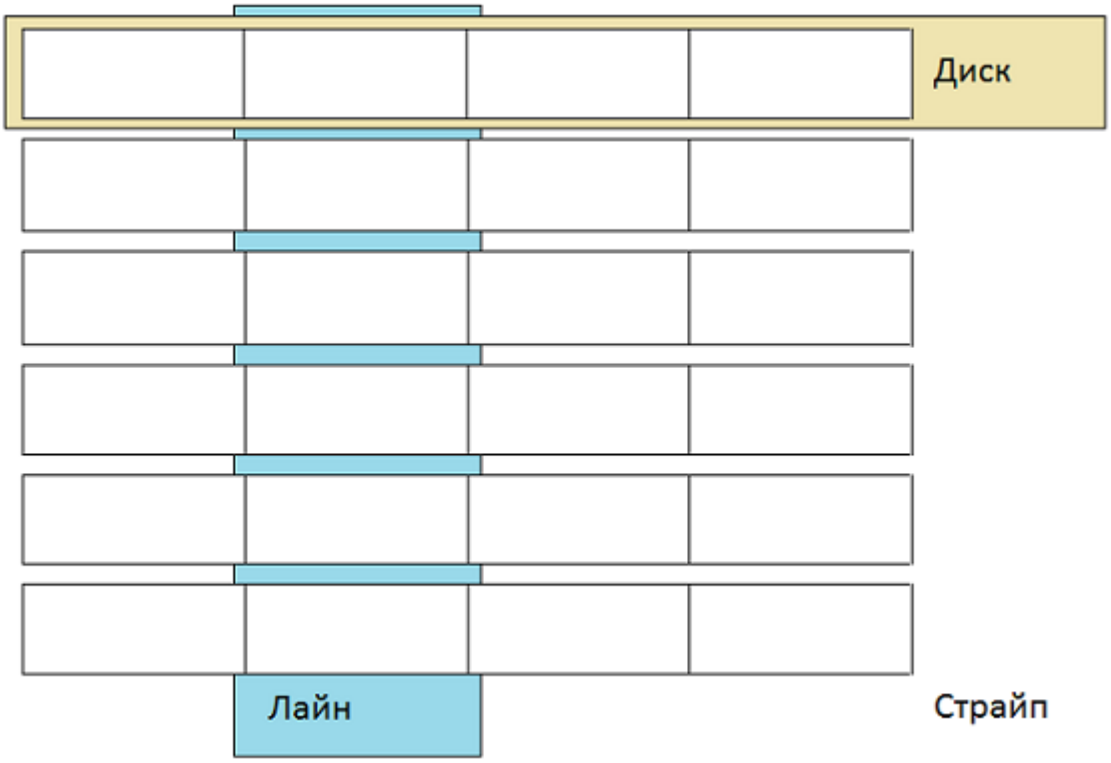

Рис. 2: Представление блоков данных в страйпе.

Страйп – основная единица обработки данных в системе хранения. Все диски в RAID-массиве разбиваются на блоки одинакового размера, и совокупность блоков с одинаковыми номерами называется **страйп**.

Диски – блоки страйпа, предназначенные для хранения данных. Обозначаются D_1, D_2, ... , D_N.

Синдром – вспомогательные блоки страйпа, использующиеся для восстановления утраченных данных. Обозначаются S_1, S_2.

Расчёт происходит по **лайнам**. Размер лайна, как правило, соответствует размеру регистра или группы регистров CPU.

Векторизация расчётов подразумевает одновременное выполнение вычислений на одном лайне.

3. Алгоритм расчёта синдромов и восстановления утраченных дисков

При записи данных в RAID 6 выполняется расчёт контрольных сумм. Для этого используются коды Рида-Соломона [1] и арифметика полей Галуа [10].

Для кодирования нами было выбрано поле Галуа $GF(2^8)$ по следующим причинам:

- $GF(2^8)$ позволяет закодировать до 256 дисков. $GF(2^4)$ позволяет закодировать всего лишь 16 дисков, что очень мало; $GF(2^{16})$ позволяет закодировать до 65536 дисков, однако кодирование в таком случае значительно усложняется, а для современных нужд достаточно и 256 дисков.
- Выбор для $GF(2^n)$ числа n, кратного двум, обусловлен тем, что операции с регистрами выполняются с данными, размерами кратными байту. Например, один регистр SSE [8] - это 16 байт или 8 двойных байт, или 4 переменные типа float, или 2 переменные типа double.

Синдромы считаются по следующим формулам:

$$S_1 = D_1 + \ldots + D_N$$

$$S_2 = D_1 x^{N-1} + \ldots + D_{N-1} x + D_N$$

Для упрощения расчётов можно считать S_2 с использованием схемы Горнера:

$$D_1 x^{N-1} + \ldots + D_{N-1} x + D_N x^0 = (((D_1 x + D_2)x + D_3)x + \ldots + D_N)$$

Один утраченный блок страйпа D_j восстанавливается по следующей формуле:

$$D_j = S_1 + S'_1$$

S'_1 - это S_1, пересчитанный для страйпа с утраченным диском.

Утраченные блоки страйпа D_j и D_k, такие, что j<k, восстанавливаются по следующим формулам:

$$D_j = \frac{(S_2 + S'_2)x^{-(N-k-1)} + S_1 + S'_1}{(x^{k-j} + 1)}$$

$$D_k = S_1 + S'_1 + D_j$$

S'_1 и S'_2 - это S_1 и S_2, пересчитанные для страйпа с утраченными дисками.

В качестве неприводимого многочлена мы используем $f = x^8 + x^4 + x^3 + x^2 + 1$.

Поскольку скорость чтения и записи данных напрямую зависит от скорости выполнения арифметических операций в полях Галуа, мы будем оптимизировать выполнение этих операций.

4. Инструменты

4.1. Язык

Разработка велась на языке программирования C [5]. Этот выбор обусловлен несколькими причинами:

- Лёгкость написания и отладки кода по сравнению с языком Ассемблера;
- Наличие Intrinsics-инструкций [3] - гибкого и удобного инструмента для использования функций Ассемблера, минуя написание кода на языке Ассемблера;
- В качестве компилятора будем использовать gcc [2], поскольку он является открытым и реализует различные оптимизации;
- Исследование проводится для компании RAIDIX, ПО которой выпускается в виде модулей для ядра Linux.

4.2. Векторные вычисления

Нами использовались команды, использующие расширения SSE [8] и AVX [4], поскольку разработка велась под архитектуру Intel x86.

SSE - это векторное расширение, включающее в себя 8 (16 для 64-битных систем) 128-битных регистров XMM0-XMM7 (XMM15) и набор инструкций для них. С помощью SSE можно распараллелить вычисления, разделив 128-битный регистр на 16 8-битных, 8 16-битных, 4 32-битных или 2 64-битных, и обрабатывая весь регистр за одну инструкцию.

AVX - это векторное расширение, дополняющее 128-битные регистры XMM0-XMM15 до 256-битных регистров YMM0-YMM15. По сравнению с SSE, в наборе инструкций AVX есть инструкции, не требующие выравнивания памяти.

5. Реализованные алгоритмы векторизации

Нами было реализовано четыре алгоритма векторизации RAID вычислений, два из которых реализовывалось впервые.

5.1. Алгоритм, использующий маску

Данный алгоритм рассматривает часть одного блока страйпа, принадлежащую лайну, как шестнадцать последовательных многочленов из $GF(2^8)$. Ширина лайна - 128 бит.

Более подробное описание алгоритма можно найти в статье [1].

Плюсы алгоритма:

- Малое количество операций
- Простые операции - логические операции, сдвиг

Минусы алгоритма:

- При каждом умножении необходимо генерировать маску

5.2. Алгоритм компании RAIDIX

В данном алгоритме мы каждый блок страйпа разбиваем на группы по 8 регистров. Коэффициенты одного элемента поля $GF(2^8)$ располагаются в соответствующих регистрах. Ширина лайна - 128*8 = 1024 бит.

Подробное описание в [11].

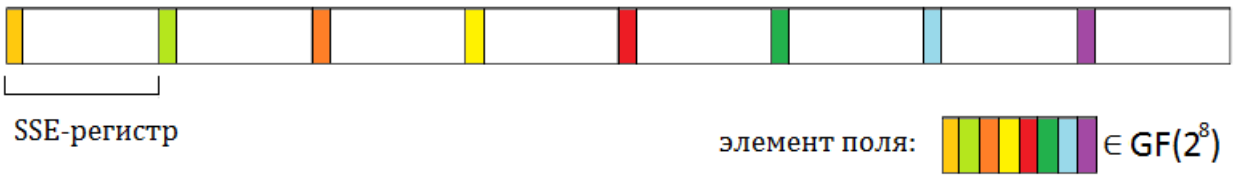

Рис. 3: Расположение многочленов в алгоритме RAIDIX.

Плюсы алгоритма:

- За три операции XOR происходит умножение на примитивный элемент поля 64 (128 - при использовании SSE, 256 - AVX) элементов поля
- Быстрый расчёт небольшого числа контрольных сумм

Минусы алгоритма:

- Неудобное расположение битов элементов поля

5.3. Алгоритм, использующий SHUFFLE

Данный алгоритм был разработан непосредственно в ходе нашего исследования.

Алгоритм рассматривает часть одного блока страйпа, принадлежащую лайну, как шестнадцать многочленов из $GF(2^8)$, расположенных таким образом: 0й, 8й, 16й и так далее до 56го бита образуют коэффициенты первого многочлена в порядке от старшего к младшему, 1й, 9й, и так далее до 57го - второй, и таким образом в 64х битах располагаются восемь многочленов. Ширина лайна - 128 бит.

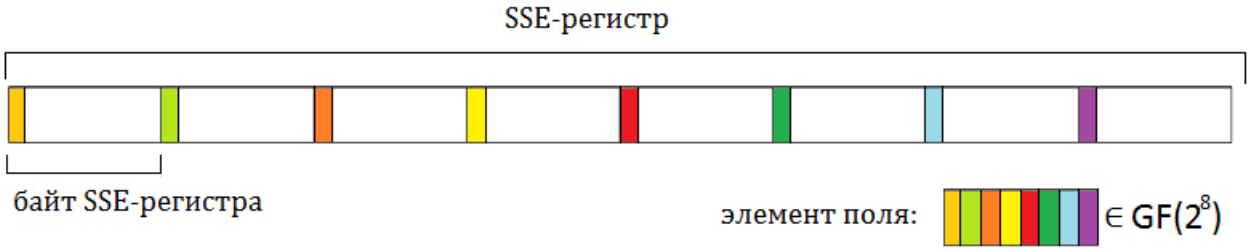

Рис. 4: Расположение многочленов в алгоритме, использующем SHUFFLE.

Плюсы алгоритма:

- Мало операций

Минусы алгоритма:

- Операция SHUFFLE дороже, чем XOR

5.4. Модификация алгоритма компании RAIDIX для работы с группами дисков

Данный алгоритм был разработан непосредственно в ходе нашего исследования.

Алгоритм рассматривает восемь последовательных дисков, принадлежащих лайну, как многочлен седьмой степени с коэффициентами из $GF(2^8)$. Ширина лайна - 128 бит.

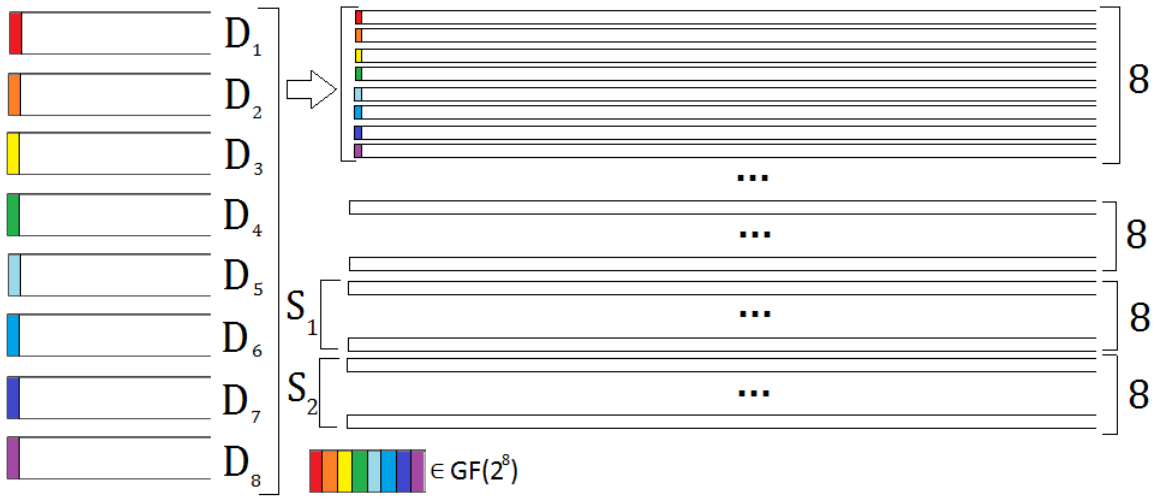

Рис. 5: Расположение многочленов в модифицированном алгоритме RAIDIX.

Плюсы алгоритма:

- Умножение на примитивный элемент поля содержит на 27% меньше операций XOR, чем в алгоритме RAIDIX
- Восстановление двух утраченных дисков в большинстве случаев осуществляется с помощью одной контрольной суммы
- Восстановление одного утраченного диска требует в 8 раз меньше обращений к дискам

Минусы алгоритма:

- Большая избыточность (20% дисков используется под синдромы, тогда как остальные алгоритмы используют для хранения синдромов только 3% дисков)
- Количество дисков в массиве должно быть кратным характеристике поля

6. Тестирование

Тестирование производилось следующим образом:

1. Для выбранного количества блоков страйпа N и размера страйпа выделяется область памяти нужного размера, которая впоследствие заполняется случайными данными. Потом она логически разделяется на N одинаковых блоков, эмулирующих различные жёсткие диски.

2. Запускается функция расчёта синдромов, которые сохраняются в блоках страйпа, предназначенных для хранения синдромов.

3. Проверяется работа функций восстановления утраченных блоков страйпа:

   (a) Случайным образом выбираются номера блоков страйпа, чье повреждение будет эмулироваться.

   (b) Данные из этих блоков страйпа сохраняются в отдельном массиве.

   (c) Эти блоки страйпа заполняются нулями.

   (d) Вызывается функция восстановления блоков страйпа, которая использует вычисленные синдромы.

   (e) Проверяется совпадение данных, сохранённых в массиве, и восстановленных функцией.

Для измерения времени используется таймер в наносекундах. Для каждой функции тест проводился 10000 раз, затем отбрасывалось по 5% самых маленьких и самых больших по величине результатов. Из оставшихся выбиралось среднее арифметическое значение.

Характеристики тестового сервера:

ОС: Scientific Linux release 6.1 (Carbon)

CPU: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz

RAM: 12 GB

7. Результаты

На представленных графиках приведены скорости различных RAID-вычислений для каждого из реализованных алгоритмов.

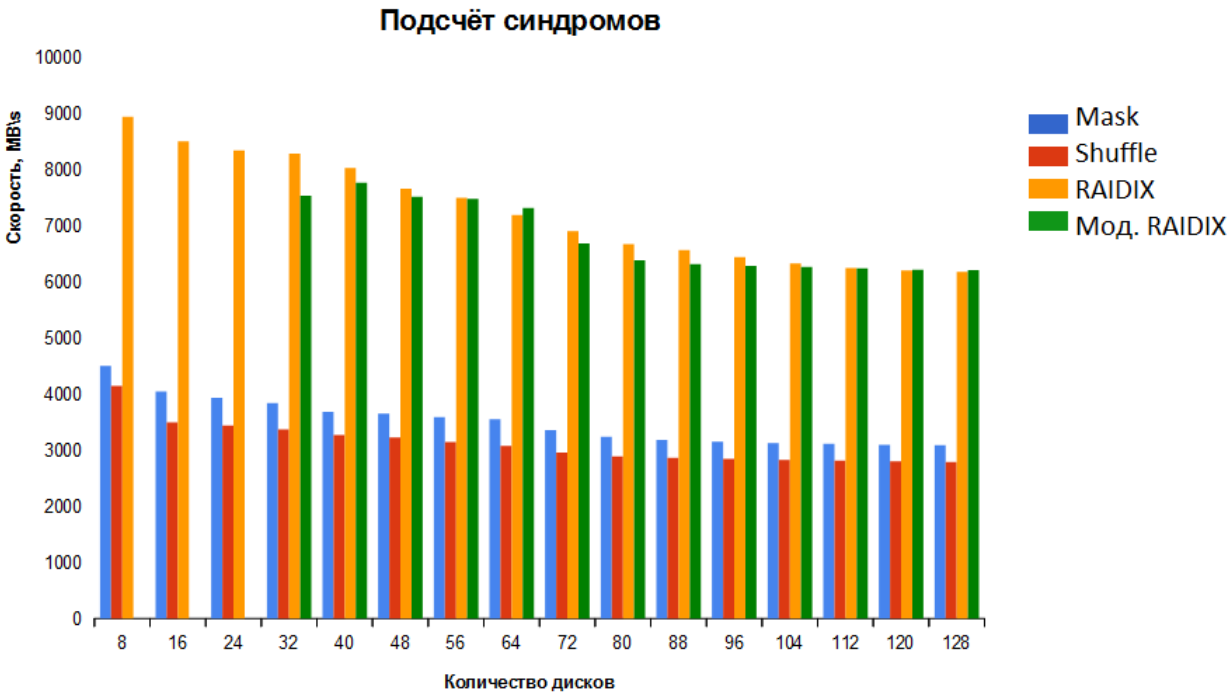

Рис. 6: Зависимость скоростей подсчёта синдромов от количества дисков.

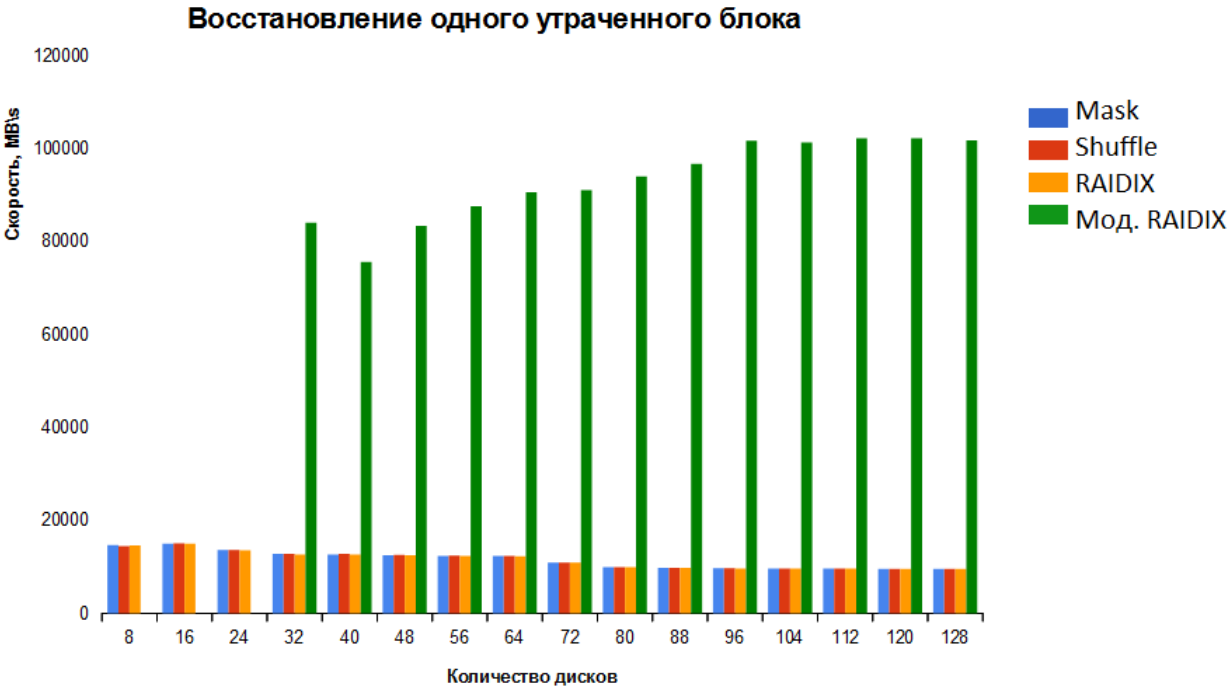

Рис.
7: Зависимость скоростей восстановления одного блока страйпа от количества дисков.

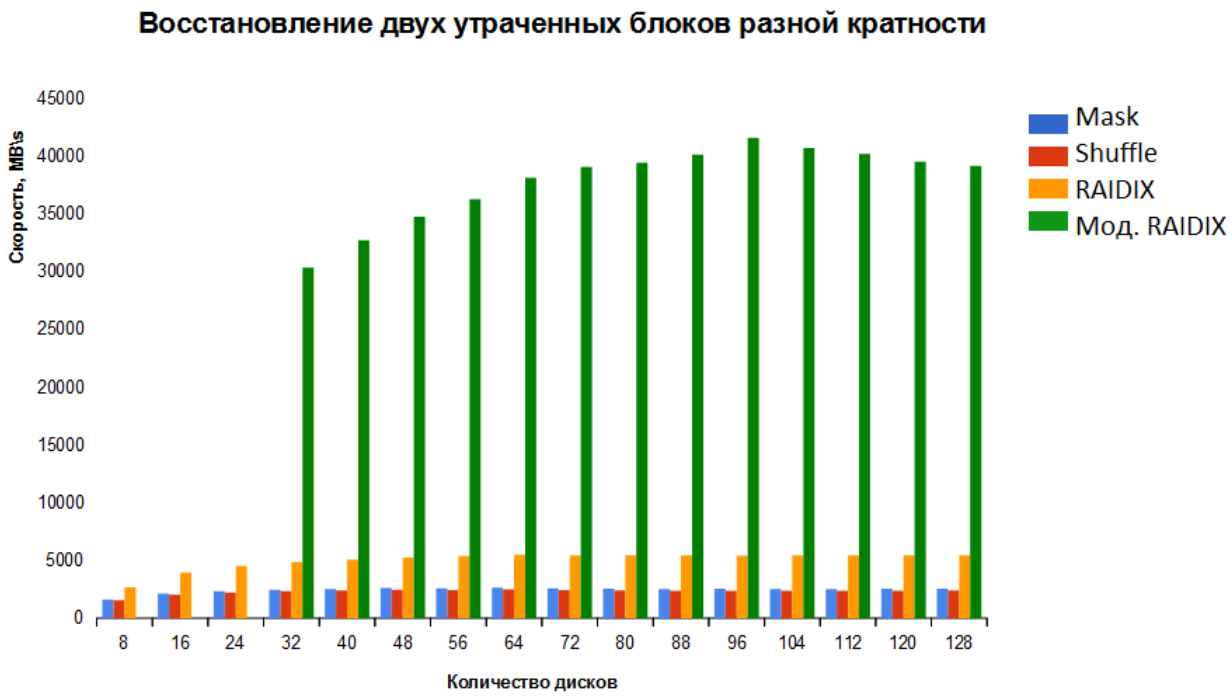

Рис. 8: Зависимость скоростей восстановления двух блоков страйпа разной кратности от количества дисков.

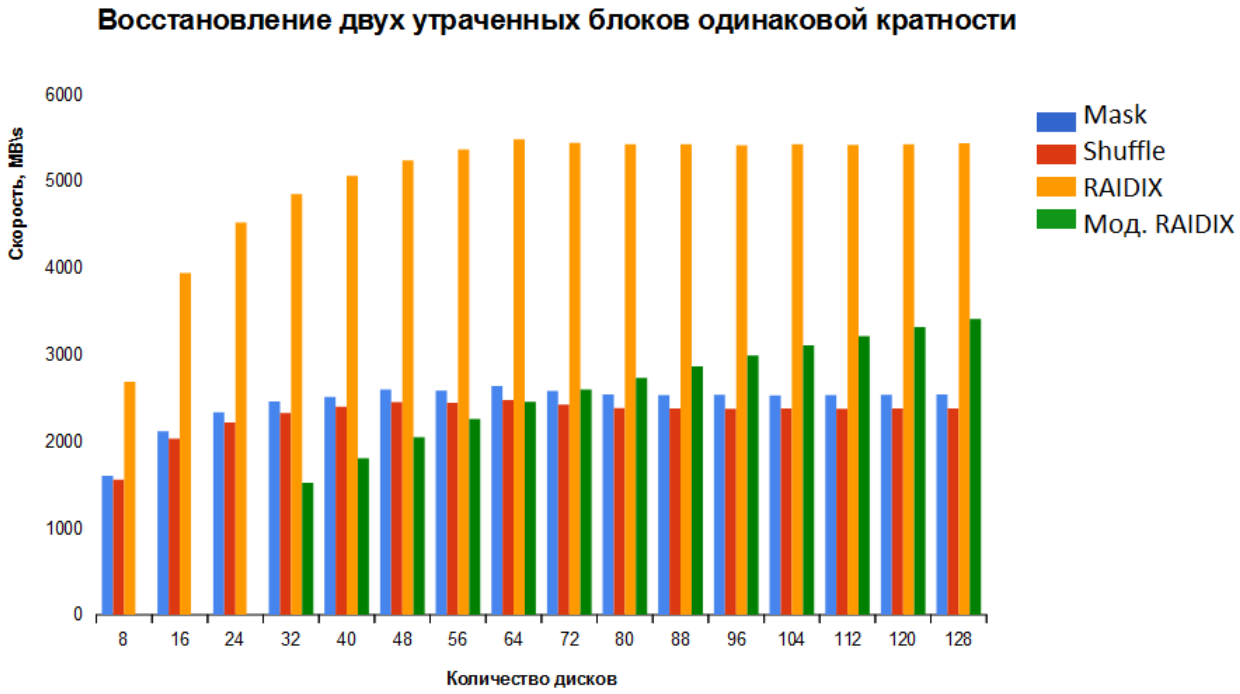

Рис. 9: Зависимость скоростей восстановления двух блоков страйпа одинаковой кратности от количества дисков.

Под дисками одинаковой кратности подразумеваются диски, чьи порядковые номера равны по модулю восемь.

8. Анализ полученных результатов

Алгоритм, использующий SHUFFLE, оправдал ожидания и показал скорости, близкие к скоростям алгоритма с маской. Его низкая производительность была предсказана на этапе составления алгоритма, и реализация подтвердила эту гипотезу.

Оценить эффективность модификации алгоритма RAIDIX простым сравнением скоростей не получится, поскольку функция восстановления двух блоков страйпа в нём показывает различную скорость в зависимости от кратности порядковых номеров утраченных дисков. Поэтому для измерения производительности этой функции посчитаем математическое ожидание её скорости.

Пусть a - скорость восстановления двух блоков страйпа различной кратности, b - скорость восстановления двух блоков страйпа одинаковой кратности, p_1 - вероятность потерять два диска различной кратности, p_2 - вероятность потерять два диска одинаковой кратности, N - количество дисков с данными в RAID-массиве.

Тогда всего возможно $N(N-1)$ комбинаций номеров утерянных дисков. Одинаковой кратности из них будут $8C_{N/8}^2$, следовательно:

$$p_2 = \frac{8C_{N/8}^2}{N(N-1)}$$

$$p_1 = 1 - p_2$$

Тогда математическое ожидание скорости работы алгоритма будет вычисляться по формуле:

$$E(\upsilon) = p_1 a + p_2 b$$

Далее приведены расчёты математического ожидания скорости восстановления двух блоков страйпа для модификации алгоритма RAIDIX. Скорость измеряется в мегабайтах в секунду.

Количество дисков	E (V модиф.RAIDIX)
32	28916,2
40	31090,9
48	32949,2
56	34352,2
64	36081,3
72	36964,6
80	37283,1
88	37939,9
96	39284,1
104	38468,1
112	37962,7
120	37325,2
128	36976,4

Рис. 10: Сравнительная таблица количества дисков в RAID-массиве и математического ожидания скорости восстановления двух блоков страйпа для модификации алгоритма RAIDIX.

Таким образом, разработанный нами алгоритм показал хорошую производительность, что открывает дальнейшие перспективы для применения его на практике.

Заключение

Были реализованы четыре алгоритма векторизации RAID вычислений, два из которых ранее не реализовывались, и разработана тестовая среда для проверки их надёжности. Осуществились сравнительные измерения скоростей новых алгоритмов и уже существующих. Был произведён анализ полученных результатов.

Список литературы

[1] Anvin H. Peter. The mathematics of RAID-6. — URL: https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf.

[2] GCC. GCC, the GNU Compiler Collection. — URL: https://gcc.gnu.org/.

[3] Intel. Intel intrinsic guide. — URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[4] Wikipedia. AVX. — URL: https://ru.wikipedia.org/wiki/AVX.

[5] Wikipedia. C. — URL: http://en.wikipedia.org/wiki/C_(programming_language).

[6] Wikipedia. Hamming code. — URL: https://en.wikipedia.org/wiki/Hamming_code.

[7] Wikipedia. RAID. — URL: https://en.wikipedia.org/wiki/RAID.

[8] Wikipedia. SSE. — URL: https://ru.wikipedia.org/wiki/SSE.

[9] Wikipedia. Storage area network. — URL: https://en.wikipedia.org/wiki/Storage_area_network.

[10] Утешев А. Ю. Поле Галуа GF(16) (версия для программистов). — URL: http://pmpu.ru/vf4/gruppe/galois/vspom4.

[11] Федоров А. Р. Способ восстановления записей в запоминающем устройстве, система для его осуществления и машиночитаемый носитель (RU 2448361).