Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 21.Б10-мм

Оптимизация алгоритма CRC-32 под RISC-V

Пономарев Николай Алексеевич

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
ст. преподаватель кафедры ИАС К. К. Смирнов

Санкт-Петербург
2023

Оглавление

Введение

Большинство современных устройств работает на процессорах от крупных корпораций, таких как Intel, ARM, IBM. Закрытость архитектуры и сложность реализации таких процессоров не позволяют использовать их для обучения инженеров. Для этих целей университет Беркли с 1982 года [9] разрабатывает собственные RISC архитектуры. Последняя из них, RISC-V, была разработана в 2010 году с целью не только быть удобной для обучения студентов, но и быть применимой в реальных системах. Она заинтересовала не только исследователей, но и множество компаний, в том числе Seagate, Alibaba и Nvidia [4]. Для массового применения архитектура должна обладать развитой экосистемой, в том числе иметь оптимизации для часто используемых алгоритмов.

Одним из таких алгоритмов является алгоритм проверки целостности данных CRC (Cyclic Redundancy Check), который имеет широкий спектр применения. В том числе применяется в ядре Linux, где его используют файловые системы, такие как ext4 и Btrfs, а так же алгоритмы сжатия ядра gzip и bzip2.

CRC — параметризуемый алгоритм, который имеет множество вариантов. В данной работе речь пойдет об оптимизации 32-битного CRC, чаще называемого CRC-32.

Ускорению данного алгоритма посвящены работы [6, 7]. Одной из самых быстрых оптимизаций является оптимизация от компании Intel [3], использующая специальную инструкцию процессора для умножения многочленов. Кроме того, во многих процессорах (в том числе, процессорах Intel и ARM) существует инструкция для вычисления CRC-32[1].

Базовый набор инструкций RISC-V не содержит в себе необходимых инструкций, но новые расширения для RISC-V позволяют реализовать оптимизацию Intel на данной платформе.

[1]Однако обычно эти инструкции считают CRC-32 по другому многочлену, имеющему более узкое применение

1. Постановка задачи

Целью данной работы является реализация оптимизации CRC-32 с использованием аппаратных инструкций умножения многочленов на архитектуре RISC-V.

Для достижения данной цели были поставлены следующие задачи:

- Изучить варианты оптимизации алгоритма CRC-32
- Выбрать целевую платформу для проведения измерений с учетом необходимых расширений процессора
- Адаптировать одну из существующих реализаций под RISC-V
- Выполнить замеры производительности оптимизированного кода

2. Обзор

2.1. Алгоритм CRC

Алгоритм CRC был впервые представлен в 1961 году [8] для обнаружения ошибок при передаче данных по сетям без сложных математических вычислений.

Рассмотрим математические основы CRC. Многочлен с коэффициентами над полем $\mathbb{F}_2$ может быть представлен как последовательность битов, где 1 на позиции i означает наличие слагаемого x^i, а 0 его отсутствие. Например, $x^5 + x^2 + x + 1$ может быть записан как 100111_2. Данное представление единственно, что позволяет сопоставлять бинарные последовательности с многочленами.

В таком случае операции над многочленами можно свести к операциям над битам. Заметим, что сложение и вычитание многочленов над полем $\mathbb{F}_2$ являются операцией исключающего ИЛИ (далее xor или $\oplus$) над битовым строками.

$$(x^5 + x^2 + x + 1) + (x^6 + x^4 + x^2 + 1) = 0100111_2 \oplus 1010101_2 = \\ = 1110010_2 = x^6 + x^5 + x^4 + x$$

Умножение битовых последовательностей идентично умножению бинарных чисел, только без переноса единицы, поэтому умножать можно используя сдвиги влево (далее $\ll$) и xor.

$$(x^5 + x^2 + x + 1) \cdot (x^6 + x^4 + x^2 + 1) = 0100111_2 \cdot 1010101_2 = \\ = 1010101_2 \oplus (1010101_2 \ll 1) \oplus (1010101_2 \ll 2) \oplus (1010101_2 \ll 5) = \\ = 101100001011_2 = x^{11} + x^9 + x^8 + x^3 + x + 1$$

Деление тоже сводится к сдвигам: «прикладываем» делитель к началу делимого, если бит равен 1, то записываем 1 в частное и совершаем xor, иначе двигаемся вправо записывая все встреченные нули в частное, продолжаем до тех пор пока длина делимого не станет меньше длины

делителя.

$$(x^6 + x^4 + x^2 + 1) : (x^5 + x^2 + x + 1) = 1010101_2 : 0100111_2 = 10_2$$

Остаток при этом равен

$$(1010101_2 \oplus (100111_2 \ll 1)) \oplus 100111_2 = 11011_2$$

Пусть M — сообщение которые мы хотим защитить контрольными суммами. Зафиксируем многочлен $P(x)$ степени n. Умножим M на x^n и вычислим остаток от деления на $P(x)$. Данный остаток и будет называться CRC.

$$R(x) = M(x) \cdot x^n \mod P(x)$$

2.2. Реализации CRC-32

Пусть $P(x)$ любой многочлен степени 32. Тогда вычисление CRC-32 выполняется прямым способом: сообщение M сдвигается на 32 бита влево, а затем описанным выше способом делится на $P(x)$. Данная реализация оперирует битами, но компьютеры обычно оперируют более крупными блоками информации.

Будем обрабатывать байт сообщения за раз, тогда алгоритм будет выглядеть следующим образом:

Листинг 1: Побайтовый CRC-32

```
uint32_t crc32(const uint8_t *M, uint32_t len_m) {
  uint32_t R = 0;
  for (uint32_t i = 0; i < len_m; ++i) {
    R ^= M[i];
    for (uint32_t j = 0; j < 8; ++j) {
      bool rightmost_bit_set = R & 1;
      R = R >> 1;
      if (rightmost_bit_set)
        R ^= P;
    }
```

```
  }
  return R;
}
```

В данной реализации CRC вычисляется не слева направо, а справа налево. Это, конечно же, меняет результат, поэтому отправитель и получатель заранее должны договориться о том, в каком порядке вычисляется CRC.

Теперь зафиксируем $P(x)$. Предыдущий подход плох тем, что содержит в себе 2 цикла: внешний, который проходит побайтово по сообщению, и внутренний, который движется по младшим 8 битам R. Вариантов последних 8 бит R всего $2^8 = 256$. Мы можем заранее вычислить их все и записать в таблицу (массив), заменив таким образом внутренний цикл.

Листинг 2: Побайтовый CRC-32 с однобайтной таблицей

```
uint32_t crc32(const uint8_t *M, uint32_t len_m) {
  uint32_t R = 0;
  for (uint32_t i = 0; i < len_m; ++i) {
    R = (R >> 8) ^ tbl[(R ^ M[i]) & 0xFF];
  }
  return R;
}
```

Предыдущий способ можно сделать еще быстрее, если вычислить таблицу для большего количества байт. Недостатком данного подхода является большой размер таблицы. Например для алгоритма, обрабатывающего сразу 16 байт требуется таблица размером 16 Кб, которая может занять четверть или даже половину L1 кэша современного процессора.

Современные процессоры предоставляют аппаратную инструкцию для вычисления CRC-32. Однако ее не всегда можно применить, т.к. существует несколько наиболее распространенных многочленов $P(x)$. Одним из часто используемых является многочлен, описанный в стан-

дарте IEEE 802.3, $P = 0x04C11DB7$[2]. Он используется при кодировании файлов PNG, в алгоритмах сжатия Gzip и Bzip2, в протоколе Ethernet и др. Другим известным многочленом является $P = 0x1EDC6F41$. Алгоритм CRC-32 по данному многочлену обычно называют CRC-32C по фамилии автора Castagnoli [1]. Он применяется в протоколе iSCSI, в файловых системах ext4 и Btrfs. Набор инструкций SSE4.2 для платформы x86 предоставляет возможность вычисления только CRC-32C. Процессоры архитектуры ARM64 умеют вычислять, как IEEE 802.3 CRC-32, так и CRC-32C.

В статье [3] предложен алгоритм вычисления CRC-32 путем последовательной свертки больших кусков данных до размера в 128 бит. При этом свертка может происходить параллельно, что еще больше ускоряет работу алгоритма. После этого происходит свертка последних 128 бит, что и является результирующим CRC. Данный алгоритм является одним из самых быстрых. По данным [7] алгоритм, обрабатывающий 16 байт с помощью таблицы на платформе x86 работает с производительностью 8 бит за цикл, в то время как данная оптимизация обрабатывает 36 бит за цикл [5]. С использованием данного подхода будет выполнена оптимизация для платформы RISC-V.

2.3. Расширения RISC-V

RISC-V обладает модульной архитектурой, что позволяет проектировать процессоры только с необходимым функционалом. RISC-V чип может быть 32 или 64-битным с базовым набором инструкций RV32I[3] и RV64I соответственно.

Для добавления дополнительной фунциональности, такой как умножение и деление или операции над числами с плавающей запятой используются так называемые расширения (англ. *extensions*). Часто используемые расширения могут объединяться в группы. Одна из них — G, она объединяет в себе базу I и расширения MAFDZicsr_Zifencei.

[2]Обычно наличие слагаемого x^{32} подразумевается, но не записывается, чтобы многочлен был 32-битным числом

[3]Полные названия расширений можно найти в таблице 1

Набор инструкций RV64GC является минимальным необходимым для запуска Linux [2].

Выбранный способ оптимизации требует наличие инструкции для умножения многочленов. На платформе RISC-V данные инструкции предоставляет расширение B, в котором они называются clmul (carry-less multiplication) и clmulh (carry-less multiplication high). В оригинальной реализации используются векторные регистры длиной 128 бит, они могут присутствовать на процессоре RISC-V при наличие расширения V. Однако в данном расширение отсутствуют операции clmul и clmulh[4], поэтому нам требуется только поддержка набора инструкций B. clmul и clmulh работают на обычных регистрах, длина которых равна разрядности платформы, поэтому выполнение оптимизации возможно только на 64-битном варианте RISC-V.

Сокращение	Полное название
RV32I	Base Integer Instruction Set, 32-bit
RV64I	Base Integer Instruction Set, 64-bit
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Zicsr	Control and Status Register (CSR)
Zifencei	Instruction-Fetch Fence
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
V	Standard Extension for Vector Operations
Zvkb	Vector Bit-manipulation used in Cryptography

Таблица 1: Расширения RISC-V

[4]Тем не менее подобная инструкция ожидается в расширении Zvkb

2.4. Платформы RISC-V

Расширение B было утверждено RISC-V International в ноябре 2021 года. По состоянию на декабрь 2022 года единственным ядром процессора, в котором есть поддержка данного расширения, является XuanTie C908. К сожалению, ни одной платы с ним еще не представлено на рынке, поэтому требуется симулятор. Он должен удовлетворять следующим условиям:

- обладать поддержкой расширения B
- уметь исполнять 64-битные программы

В результате были найдены следующие симуляторы: Spike[5] и gem5[6].

2.4.1. Spike

Spike является функциональным симулятором RISC-V. Он разрабатывается RISC-V International и обладает поддержкой всех принятых расширений архитектуры. Тем не менее Spike не может использоваться для измерения производительности программ, т.к. не исполняет инструкции с точность до циклов.

2.4.2. gem5

gem5 является индустриальным стандартом для симуляции микроархитектуры с точность до циклов [10]. Он позволяет тонко настраивать имеющиеся компоненты системы, такие как ядра процессора, кэш, размер и тип оперативной памяти с помощью Python.

Именно gem5 будет использоваться в дальнейшем для оценки быстродействия алгоритма.

[5] https://github.com/riscv-software-src/riscv-isa-sim
[6] https://www.gem5.org/

3. Реализация

В качестве базовой реализации была выбрана реализация из ядра Linux[7]. Она использует инструкцию PCLMULQDQ из набора команд CLMUL для архитектуры x86. PCLMULQDQ оперирует на 128 битных регистрах XMM и обрабатывает 512 бит данных за раз. Особенностью этой инструкции является то, что на вход она принимает два 64-битных числа из XMM регистров, а результат является 128-битным числом и занимает целый регистр

На RISC-V доступны только регистры размера 64, поэтому существуют две инструкции: clmul и clmulh. Обе принимают на вход 64-битные числа, но первая возвращает младшие 64 бита результата, а вторая, соответственно, старшие.

Разработка велась на языке ассемблера, поэтому набор доступных регистров был ограничен. В силу этого ограничения, данная реализация обрабатывает только 128 бит за раз.

Длина входных данных должна быть кратна 128 битам. Для решения этой проблемы используется функция-обертка, которая выравнивает данные, вычисляя CRC-32 по стандартному алгоритму до границы кратной 128 битам.

В процессе адаптации код, одной из самых сложных задач было правильно использование регистров. В базовой реализации, из-за описанных выше особенностей инструкции PCLMULQDQ, значение регистра обязательно копируется в другой. Этого хотелось избежать, т.к. количество гарантированно свободных регистров ограниченно. В этом помогала остановка программы с помощью дебаггера и просмотр значения регистров.

Листинг 3: Оригинальный код

```
movdqa  %xmm1, %xmm5
pclmulqdq $0x00, CONSTANT, %xmm1
pclmulqdq $0x11, CONSTANT, %xmm5
```

[7] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/crypto/crc32-pclmul_asm.S

```
pxor    %xmm5, %xmm1
pxor    (BUF), %xmm1
```

Путем изучения содержимого регистров, удалось изменить данный код, используя меньше копирований.

Листинг 4: Адаптированный код

```
clmulh t3, CONSTANT_2, t1 # xmm5[127:64]
clmul t2, CONSTANT_2, t1 # xmm5[63:0]
clmulh t1, CONSTANT_1, t0 # xmm1[127:64]
clmul t0, CONSTANT_1, t0 # xmm1[63:0]
xor t0, t0, t2 # xmm1[63:0]
xor t1, t1, t3 # xmm1[127:64]
ld t4, 0(BUF)
xor t0, t0, t4
ld t4, 8(BUF)
xor t1, t1, t4
```

Тем не менее объем кода в этом месте увеличился вдвое, т.к. за раз можно обработать только 64 бита данных, а не 128.

Так же анализ регистров позволил оптимизировать операцию побитового сдвига. Без оптимизации, для сдвига двух 64-битных регистров, которые рассматриваются как один 128-битный, на 32 бита вправо необходимо четыре инструкции.

Листинг 5: Универсальный сдвиг на 32 вправо

```
srli t0, t0, 32 % Сдвинуть младшие биты на 32 вправо
slli t3, t1, 32 % Биты, которые необходимо
                % перенести в другой регистр
or t0, t0, t3 % Объединить биты в один регистр
srli t1, t1, 32 % Сдвинуть старшие биты на 32 вправо
```

Оказалось, что в момент выполнения данной операции старшие 32 бита являются нулями и последняя инструкция не нужна.

4. Эксперимент

Эксперименты проводились на симуляторе gem5 со следующими характеристиками:

- MinorCPU с частотой 1 ГГц и размером кэша L1 в 64 Кб
- 512 Мб ОЗУ DDR4 с частотой 2400 МГц

Исходные файлы компилировались с флагами `-O3 -static`, а затем запускались на симуляторе. В качестве стандартного алгоритма использовался вариант CRC-32 с таблицей размером 16 Кб.

Объем данных, байт	Стандартный алгоритм, тиков	Оптимизированный алгоритм, тиков
128	$308.5 \cdot 10^3$	$83 \cdot 10^3$
1024	$2277.5 \cdot 10^3$	$424 \cdot 10^3$
8192	$18 \cdot 10^6$	$3.6 \cdot 10^6$
65536	$191 \cdot 10^6$	$30.5 \cdot 10^6$

Таблица 2: Результаты измерений

При малом объеме, данные полностью помещаются в кэш и время исполнения стабильно. В ином случае возникают задержки, которые влияют на время работы.

По результатам эксперимента, можно сделать вывод о том, что оптимизация действительно работает и дает ощутимый прирост производительности на больших объемах данных.

Заключение

В рамках данной учебной практики были достигнуты следующие результаты:

- Изучены существующие способы оптимизации алгоритма CRC-32
- Выбрана целевая платформа для проведения измерений с учетом необходимых расширений процессора
- Проведена адаптация реализации для x86 под RISC-V с использованием инструкции clmul
- Произведены измерения быстродействия оптимизированного кода на симуляторе gem5.

Исходный код расположен по адресу: https://github.com/WoWaster/riscv-crc32-clmul.

Список литературы

[1] Castagnoli G., Brauer S., Herrmann M. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits // IEEE Transactions on Communications. — 1993. — Vol. 41, no. 6. — P. 883–892.

[2] Debian Wiki. RISC-V. — 2022. — URL: https://wiki.debian.org/RISC-V (дата обращения: 2022-12-14).

[3] Fast CRC computation for generic polynomials using PCLMULQDQ instruction : Rep. / Intel, Tech. Rep.(white paper) ; Executor: Vinodh Gopal, E Ozturk, J Guilford et al. : 2009. — URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf.

[4] Hsu Jeremy. RISC-V Star Rises Among Chip Developers Worldwide. — URL: https://spectrum.ieee.org/riscv-rises-among-chip-developers-worldwide (дата обращения: 2022-12-07).

[5] Intel Corporation. ISA-L Performance Report Release 2.19. — 2017. — URL: https://01.org/sites/default/files/documentation/intel_isa-l_2.19_performance_report_0_0.pdf (дата обращения: 2022-12-15).

[6] Kadatch Andrew, Jenkins Bob. Everything we know about CRC but afraid to forget. — 2010. — URL: https://google-code-archive-downloads.storage.googleapis.com/v2/code.google.com/crcutil/crc-doc.1.0.pdf.

[7] Omar Kareem. Fastest CRC32 for x86, Intel and AMD, + comprehensive derivation and discussion of various approaches. — URL: https://github.com/komrad36/CRC (дата обращения: 2022-12-06).

[8] Peterson W. W., Brown D. T. Cyclic Codes for Error Detection // Proceedings of the IRE. — 1961. — Vol. 49, no. 1. — P. 228–235.

[9] Design and Implementation of RISC I : Rep. : UCB/CSD-82-106 / EECS Department, University of California, Berkeley ; Executor: Carlo H. Séquin, David A. Patterson : 1982. — Oct. — URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/5449.html.

[10] The gem5 Simulator: Version 20.0+ / Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram et al. // CoRR. — 2020. — Vol. abs/2007.03152. — arXiv : 2007.03152.