Санкт-Петербургский государственный университет

Бочкарев Арсений Петрович

Выпускная квалификационная работа

Реализация intrinsic-функций для процессорной архитектуры RISC-V в OpenJDK

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5162.2020 «Технологии программирования»*

Научный руководитель:
доцент кафедры информатики, к. ф.-м. н., Луцив Д. В.

Рецензент:
Ведущий инженер-программист ООО «Синтакор» Гаевский Ю. И.

Санкт-Петербург
2024

Saint Petersburg State University

Arseny Bochkarev

Bachelor's Thesis

Implementation of intrinsic functions for RISC-V processor architecture in OpenJDK

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2020 «Programming Technologies»*

Scientific supervisor:
C.Sc., docent D.V. Luciv

Reviewer:
Leading Engineer at "Syntacore" Y.I. Gaevsky

Saint Petersburg
2024

Оглавление

1. Введение

RISC-V — это открытая [17] процессорная архитектура, в основе которой лежит небольшой набор команд, с возможностью использования различных расширений [48] для увеличения производительности.

Компания Syntacore занимается созданием и сопровождением процессорных ядер на микропроцессорной архитектуре RISC-V. В число её работ входит также поддержка инструментов разработки для данной архитектуры. Одним из таких инструментов является OpenJDK — один из наиболее популярных [45] на 2023 год наборов утилит для разработки на языке Java. Приоритетное направление для работы — ускорение методов, использующихся в работе с криптографией.

Одной из особенностей среды исполнения OpenJDK (виртуальной машины «HotSpot») является концепция уровней компиляции [39]. Согласно ей, исполнение Java методов начинается в интерпретаторе Java байт-кода сразу после загрузки пришедшего виртуальной машине класса, без дополнительных оптимизаций. При достаточно частом исполнении метода, он будет скомпилирован компилятором C1, с использованием некоторых оптимизаций, например, «constant folding» или «value numbering» [16]. При последующем исполнении этого метода, он будет скомпилирован уже более мощным компилятором C2, который, например, способен более эффективно выделять регистры или проводить «global code motion» [40]. Ещё одной из возможных оптимизаций является подмена сгенерированного по Java-байткоду метода на его вручную написанную в нативных кодах специализацию [21], которая, как ожидается, работает более эффективно, чем порождённая компилятором версия. Такой подход в OpenJDK называется «intrinsification» («интринсификация»), а сами функции, написанные в ассемблерных кодах конкретной архитектуры — intrinsic-функциями.

Несмотря на то, что в OpenJDK существует порт для RISC-V, на данный момент он не поддерживает все существующие криптографические intrinsic-функции: например, в некоторых из таких «ручных» реализаций этих функций для оптимизаций могут применяться ин-

струкции из расширений архитектуры, которые совсем недавно были ратифицированы, и поддержка которых как в OpenJDK, так и в процессорах ещё не успела появиться[1]. Поэтому, для ускорения работы HotSpot необходимо расширить текущую поддержку криптографических intrinsic-функций в OpenJDK для архитектуры RISC-V.

[1]Подробнее в разделе 3

2. Постановка задачи

Целью работы является ускорение криптографических методов в OpenJDK применительно к микропроцессорной архитектуре RISC-V при помощи intrinsic-функций.
Для её выполнения были поставлены задачи, приведённые ниже.

1. Выполнить обзор существующих в OpenJDK криптографических intrinsic-функций.

2. Разработать набор intrinsic-функций для платформы RISC-V для недостающих криптографических методов OpenJDK. Убедиться, что работа intrinsic-функций полностью корректна.

3. Путём замеров производительности методов на микробенчмарках продемонстрировать ускорение их работы после реализации intrinsic-функции.

3. Обзор

3.1. Существующие intrinsic-функции

Все имеющиеся в OpenJDK intrinsic-функции описаны в файле `vmIntrinsics.hpp` [36]. В нём описываются базовые обозначения для интринсиков и вводятся сигнатуры для каждого из них, видно, в каких модулях они используются. Интересующие домены — crypto и zip (контрольные суммы). Код intrinsic-функций можно найти в файле `stubGenerator_riscv.cpp` [34]. Решение об их генерации определяется на основе свойств платформы, на которой запускается виртуальная машина. Для RISC-V (на момент весны 2024) эта информация получается [2] путём специального системного вызова при старте JVM, на основе которого делается вывод об используемых расширениях процессора. Далее, в файле `vm_version_riscv.cpp` определяется [38], будет ли разрешено сгенерировать и использовать ту или иную intrinsic-функцию. По состоянию на осень 2023 года в OpenJDK не велось работ по реализации для RISC-V следующих использующихся в криптографии интринсиков [13]:

- Adler32:
  - `updateBytesAdler32`,
- CRC32:
  - `updateBytesCRC32`,
  - `updateBytesCRC32C`,
- Poly1305:
  - `poly1305_processBlocks`,
- GHASH:
  - `ghash_processBlocks`,

[2] Подробнее в файле `vm_version_linux_riscv.cpp` [37]

- AES
  - `galoisCounterMode_AESCrypt`,
  - `aescrypt_decryptBlock`,
  - `aescrypt_encryptBlock`,
  - `cipherBlockChaining_decryptAESCrypt`,
  - `cipherBlockChaining_encryptAESCrypt`,
  - `counterMode_AESCrypt`,
  - `electronicCodeBook_decryptAESCrypt`,
  - `electronicCodeBook_encryptAESCrypt`.

Для каждой из них (кроме intrinsic-функций AES режима ECB) в OpenJDK существуют соответствующие реализации для микропроцессорной архитектуры AArch64 [20], использующие инструкции, аналогичные вводимым различными расширениями RISC-V. Для AES в режиме ECB существуют intrinsic-функции для платформы x86. Все intrinsic-функции семейства AES используют внутри себя код функции по шифрованию (расшифрованию) одного блока.

Кроме того, для методов для алгоритмов CRC32 и Adler32 существуют [32] их Java Native Interface аналоги: написанные на языке C библиотечные функции, вызов которых может делать JVM. Такое происходит, если виртуальная машина после определения, что метод «горячий», не находит для него intrinsic-функции, но находит соответствующую библиотеку с нужной функцией. Сам факт такого вызова более накладен по производительности, чем вызов написанной в нативных кодах intrinsic-функции, однако всё равно может использоваться для ускорения функций, сгенерированных по «чистому» Java коду [51].

3.2. Криптографические алгоритмы в OpenJDK

3.2.1. CRC32/CRC32C

Алгоритм CRC (Cyclic Redundancy Check) — это алгоритм нахождения контрольной суммы, предназначенный для обнаружения ошибок

при цифровой передаче данных.

Базовые шаги алгоритма приведены ниже [41]:

1. Выбирается определённая константа, записываемая в виде полинома, ненулевые степени членов которого указывают на единицы в двоичной записи этой константы. Разные виды CRC-алгоритмов, как правило, отличаются лишь разными полиномами (в чём и заключается основная разница между CRC32 и CRC32C).

2. В младший разряд изначально зануленного регистра считывается текущий старший бит сообщения, в то время как из старшего разряда регистра сдвигается один бит.

3. Если выдвинутый бит равен «1», то производится операция XOR над битами в тех разрядах регистра, которые соответствуют единицам в полиноме.

4. Шаги 2 и 3 повторяются до тех пор, пока в сообщении ещё есть биты.

5. Когда всё сообщение обработано, в регистре остаётся остаток, который и является контрольной суммой CRC.

Алгоритм 1 Псевдокод CRC32 алгоритма

```
crc ← 0
while i < n do
    b ← input[i]
    upper ← crc[31]
    crc[0] ← b
    if upper then
        crc ← crc ⊕ poly
        i ← i + 1
```

Некоторые варианты алгоритма (такие как, например, «классический» CRC32, но не CRC32C [14]) перед и после выполнения всех операций проводят инвертирование бит.

Существует большое количество способов оптимизации, часто основанных на приёме предподсчёта некоторых значений и помещении их в таблицу, зависящую от изначально выбранного полинома: неоднократное применение операции XOR над сдвигающимися битами можно привести к той же операции, но над группой бит сразу и со значением, про которое известно, что исключающее ИЛИ с ним будет иметь тот же эффект, что и с поочерёдно сдвигающимся полиномом [52].

3.2.2. Adler32

Adler-32 — хеш-функция, вычисляющая значение контрольной суммы в соответствии со спецификацией библиотеки «ZLIB» для массива байт или его фрагмента.

Алгоритм вычисления контрольной суммы строится вокруг двух различных чисел A и B, зависящих от входных данных, которые в конце алгоритма конкатенируются.

1. A = 1, B = 0;
2. A равняется сумме всех входных байт в строке плюс один;
3. B — сумма всех отдельных значений A на каждом шаге;
4. Все суммы делаются по модулю 65521.

3.2.3. Poly1305

Алгоритм Poly1305 генерирует код аутентификации сообщения (имитовставку) из сообщения и одноразового ключа.

Принимая в качестве входных параметров 16-байтный случайно сгенерированный секретный ключ r и L-байтное сообщение m, Poly1305 работает по следующему алгоритму [3].

1. Разбивает сообщение на последовательные 16-байтные сегменты.
2. К каждому 16-байтовому фрагменту сообщения добавляется единица до 17 байт для использования в качестве коэффициента полинома.

3. Вычисляется значение полинома в точке r по модулю $2^{130} - 5$.

У данного алгоритма существуют оптимизации, главным образом связанные с делением по модулю. Множественное деление на число, не помещающееся целиком в 64-битный регистр — относительно дорогостоящая операция, поэтому, используя тот факт, что число $2^{130} - 5$ — простое, можно интерпретировать его как $2^{26} - 1$, компенсируя это умножением складывающихся чисел на 5 при переносе во время сложения [47].

3.2.4. GHASH

GHASH (функция Галуа) — функция по вычислению неизвестной заранее части тега аутентификации, использующаяся в алгоритмах шифрования GCM. После выполнения передачи зашифрованного сообщения, тег аутентификации используется для проверки целостности сообщения.

Функция Галуа работает следующим образом [46].

1. Пусть X — это последовательность из m разбитых по 128 битам входных данных X_i;

2. Также одним из входных параметров является заранее известный ключ хэширования H;

3. Операции в GHASH применяются внутри конечного поля $\mathrm{GF}(2^{128})$, определяемого полиномом GCM: $x^{128} + x^7 + x^2 + x + 1$;

4. Вычисление полинома-результата $Y_m = \sum_{i=1}^{m} X_i \times H^{m-i+1}$, являющегося основой тега аутентификации.

3.2.5. AES

AES — распространенный [2] алгоритм шифрования, работающий с блоками входных данных длиной 128 бит. Перед началом шифрования блоков по секретному ключу, подающемуся на вход, генерируется новый набор ключей для проведения нескольких циклов (раундов)

шифрования. Основной алгоритм по шифрованию конкретного блока состоит из четырёх отдельных процедур, представленных ниже [53].

1. `SubBytes()`: каждый байт входных данных заменяется соответствующим элементом в фиксированной 8-битной таблице поиска;

2. `ShiftRows()`: состояние шифруемых данных представляется как матрица размером 4 на 4, затем байты в каждой строке state циклически сдвигаются влево;

3. `MixColumns()`: обрабатывает данные по колонкам в матрице 4 на 4, трактуя каждую из них как полином третьей степени. В нём каждая колонка перемножается с фиксированным многочленом $c(x) = 3x^3 + x^2 + x + 2$ по модулю $x^4 + 1$;

4. `AddRoundKey()`: для каждого байта шифруемых данных производится операция XOR с ключем текущего цикла шифрования.

В зависимости от длины ключа, данные шаги могут повторяться от 10 до 14 раз. Для расшифрования используются обратные версии этих процедур.

В OpenJDK поддержано 4 режима шифрования для AES [20].

- Cipher Block Chaining Mode: Для каждого блока входных данных проводится XOR с предыдущим результатом шифрования;

- Counter Mode: на вход алгоритма подаётся счётчик, с которым происходит XOR;

- Galois Counter Mode: шифрование с присоединенными данными. Внутри него используется алгоритм GHASH;

- Electronic Codebook Mode: Независимое шифрование блоков входных данных. Intrinsic-функции поддержаны только для платформы x86 [50].

3.3. Поддерживаемые в OpenJDK платформы

В OpenJDK поддержаны многие распространённые микропроцессорные архитектуры, например, x86, PowerPC или ARM [25]. Зачастую реализации intrinsic-функций для них схожи, как, например, для алгоритма подсчёта контрольной суммы CRC32 для платформ PowerPC [23] и 64-битной архитектуры Arm (AArch64) [22].

Arm может называться наиболее схожей с RISC-V архитектурой, поддержанной в OpenJDK, благодаря тому, что она, так же, как и RISC-V, является RISC-архитектурой [49], а также благодаря своей модульности [1] и сходству по функциональности многих из инструкций двух этих архитектур.

3.4. Расширения RISC-V

Одно из главных преимуществ RISC-V — расширяемость [48] архитектуры. Это означает, что у процессорных ядер существуют различных расширения, добавляющие в базовый набор команд новые инструкции. В то же время, те процессоры, которым эти инструкции не нужны в силу специфики их применения, реализовывать их не обязаны [48]. В данной работе рассматриваются следующие расширения RISC-V.

- «`Zba`» — расширение для продвинутых манипуляций со сдвигами битов [44].
- «`Zbc`» — расширение для работы с операциями умножения без переноса со сдвигами битов [44].
- «`V`» — векторное расширение RISC-V, для которого в QEMU при настройке рабочей среды была в качестве наибольшей длины одного вектора было выбрано значение в 128 бит и максимальный размер элементов 64 бит (согласно характеристикам платы «Kendryte K230»[3]). К его особенностям можно отнести возможность динамически изменять количество элементов векторов и их длину [18].

[3]Подробнее в разделе 4.3

- «Zvbb» — расширение RISC-V для работы с битами элементов внутри элементов векторов [19].
- «Zvbc» — расширение для работы с умножением без переноса на векторах [19].
- «Zvkned» — расширение для работы с блоками AES [19] с использованием векторных регистров.

В связи с тем, что векторно-криптографические расширения RISC-V, включающее в себя «Zvbc», «Zvbb» и «Zvkned» были ратифицировано лишь в сентябре 2023 года [43], на рынке на данный момент (весна 2024 г.) отсутствуют процессоры, в которых они реализованы. По этой причине тестирование производительности intrinsic-функций, использующих инструкции из этого семейства, в полной степени невозможно.

4. Алгоритм работы

4.1. Основной алгоритм

Разработку intrinsic-функций можно разделить на 4 этапа, описанных ниже.

1. **Поиск существующих реализаций для других платформ**. В случае существования реализаций для других платформ, их порт на целевую архитектуру можно использовать в качестве стартовой точки разработки. Как ясно из раздела 3.3, для RISC-V предпочтительно начинать с анализа intrinsic-функций с платформы AArch64. Как правило, уже реализованные на других архитектурах методы для оптимизации используют какие-либо свойства алгоритмов, о которых компилятор не может догадаться: например, при векторизации Adler32 для AArch64 используется [35] таблица из чисел от 16 до 1, которая впоследствии перемножается с входными данными.

2. **Использование профилировщиков**. Для каждого из интринсифицированных методов при запуске микробенчмарка использовался профилировщик «Perfasm» с опцией «cycles» для отслеживания наиболее «горячих» мест в нативном коде, чтобы, при наличии такой возможности, оптимизировать в первую очередь их.

3. **Оптимизация при помощи стандартных техник**. Далее необходимо попробовать применить известные методы оптимизации, которые в ином случае могут применять компиляторы самостоятельно для ускорения работы методов. Например, это может быть развёртка цикла («loop unrolling», возможно, частичная) или вынесение независимого кода из цикла («loop-invariant code motion»).

4. **Оптимизации с использованием специфики RISC-V**. После применения общих компиляторных техник дальнейшее ускорение

может быть достигнуто за счёт особенностей архитектуры, то есть использования таких инструкций, которые в конечном итоге ведут к ускорению метода. Например, в расширении «`Zba`» инструкция «`sh2add`» может использоваться для доступа к элементам массива вместо двух инструкций: сдвига на 2 бита влево и сложения.

После каждого из этапов работы производится апробация полученных результатов, в случае возникновения функциональных ошибок проводится отладка.

4.2. Апробация

Корректность результатов проверяется двумя путями.

1. Запуском функциональных тестов — как специализированных, так и общих для всего OpenJDK.

2. Запуском микробенчмарков для соответствующих intrinsic-функций и замерами производительности на них. Использовался тот же набор микробенчмарков, что и для замеров производительности на платформе AArch64. Успехом для такого вида тестирования будет считаться результат, по выбранной метрике превосходящий код, генерируемый компилятором C2 из OpenJDK.

    - Для части intrinsic-функций, использующих векторно-криптографические расширения RISC-V, данный пункт не является необходимым. В этом случае для достижения результатов достаточно обеспечения функциональной корректности на эмуляторе.

4.3. Замеры производительности

Замеры в большей части экспериментов проводились на плате «Alibaba T-head RVB-ICE». В случае расширения «`Zba`», когда используемых процессорных расширений платы «T-head» не хватало для поддержки всех инструкций, применяющихся в реализации

intrinsic-функции, использовалась плата «StarFive VisionFive2», где данное расширение поддержано. Аналогичная ситуация с работой векторных инструкций, которых нет ни на одной из описанных плат. В этом случае применялась плата «Kendryte K230», для которой поддержка расширения «`V`» есть. Подробные характеристики плат находятся в отдельном репозитории [11]. Замеры производились с использованием набора библиотек JMH, который не только показывает количество операций за единицу времени, но и строит доверительный интервал с уровнем доверия 99% для этих результатов. В качестве метрики выбирается количество операций в миллисекунду при работе на тестовом стенде.

5. Реализация

Все intrinsic-функции генерируются в свежевыделенную память при запуске JVM [34]. Для каждой из них запоминается адрес, по которому совершается вызов [33] с заранее определёнными параметрами в начале исполнения кода интринсифицированного метода.

Использование всех добавленных intrinsic-функций регулируется соответствующими ключами в файле «`vm_version_riscv.cpp`». Например, для отключения использования нативной реализации функции «`poly1305_processBlocks`» в случае измерения её производительности достаточно передать опции «`-XX:+UnlockDiagnosticVMOptions`» и «`-XX:DisableIntrinsic=_poly1305_processBlocks`», чтобы в запущенной виртуальной машине Java эта интринсифицированная специализация метода не использовалась.

Для первоначальной проверки корректности на функциональных тестах использовался инструмент «`JTreg`» с тестами из OpenJDK. В случае ошибок при реализации intrinsic-функции в ассемблерных кодах отладка производилась при помощи инструмента GDB, в котором точки останова проставлялись по тому адресу, в который код метода был сгенерирован. Как правило, этот адрес находится в функциях вида «`inline_*`» в файле «`library_call.cpp`».

Adler32

Для платформы AArch64 последняя реализация подсчёта контрольной суммы по алгоритму Adler32 использует векторные инструкции. Однако, в JDK9 присутствует [24] версия без их использования. Первоначально портированная в качестве прототипа скалярная версия при использовании профилировщика показала, что большую часть времени процессор провёл, исполняя инструкции в цикле по счётчику 16 байт (практически весь вывод целиком состоит из этого цикла: время, проведённое в остальных частях функции, было настолько мало, что оно даже не было распечатано). Это значит, что основное внимание во время оптимизации должно быть уделено ему (подробные результаты работы

пролифировщика находятся в секции 7).

После векторизации в целях оптимизации наибольшего цикла была применена его частичная раскрутка, в которой использовалось то свойство алгоритма, что после считывания каждых 16 байт данных расчёты по ним могут производиться независимо от предыдущих. После расчётов эти промежуточные результаты записывались каждый в отдельный векторный регистр. После всех подсчётов происходило финальное суммирование.

Для дальнейшей оптимизации большого цикла можно было избавиться от лишнего счётчика, воспользовавшись тем, что на каждой итерации производилось взятие данных из памяти, после которого указатель на эти входные данные сдвигался. Если перед входом подсчитывать конечный адрес, откуда будут браться начальные данные, то в качестве счётчика для цикла выступал бы указатель, в качестве верхней границы для него — этот конечный адрес. Таким образом одну инструкцию по прибавлению единицы в цикле, выполняющимся в среднем наибольшее число раз, можно выкинуть.

Данные оптимизации [5] позволили выиграть на больших данных около 30% разницы (полные результаты замеров в секции 7).

В RISC-V Vector Extension существуют инструкции смены размерности векторов («`vsetvli`»). Поскольку при подсчёте контрольной суммы необходимо умножать вектора с возможным переполнением, это означает, что смена размерности должна применяться при взятии результата. Однако частое использование такой операции может быть дорогим [42], поэтому её оптимизации стоит уделить пристальное внимание.

При работе инструкции умножения с расширением («`vwmul.vv`») для 16 активных элементов по байту каждый результат разделяется между двумя регистрами: изначальным регистром-результатом, и следующим за ним [18]. Используя это, можно, задействуя большее количество векторных регистров, а также изменив порядок исполнения части инструкций, скомпенсировать это ограничение, не меняя размер регистров при умножении чисел, а перенеся эту смену на момент взятия результатов. Листинги кода с пояснениями находятся в приложении в

секции 7.

Избавление [4] от одной из инструкций смены размерности в раскрученном цикле принесли около 19% разницы в производительности (подробнее в разделе 7).

Poly1305

Была реализована согласно алгоритму, используя ускоренное умножение с переносом. Основную сложность при реализации составлял тот факт, что данные запакованы в пять 26-битных частей, лежащих в трёх регистрах, поэтому необходимо было вручную следить за всеми переносами.

Хотя для реализации для платформы AArch64 не использовалось специфических процессорных расширений, для ускорения специализации этой intrinsic-функции для RISC-V можно, например, использовать ассемблерную инструкцию «`sh2add`» для ускорения операции умножения на 5 (интерпретируя её, как сдвиг влево на 2 позиции и сложение). Подобное её применение [10] может дать до 6% прироста в производительности (подробнее в разделе 7).

CRC32

За основу была взята табличная версия алгоритма CRC32 без использования семейства инструкций по умножению без переноса, поскольку расширение, добавляющее их, отсутствует на многих популярных платах для работы [15]. Наибольшую часть метода занимает цикл по счётчику 16 байт, в котором выполняется загрузка в регистры частей сообщения, а затем, после поиска в таблице, выполняется XOR регистра, аггрегирующего результат, с найденным значением. После того, как длина сообщения для обработки станет меньше 16 байт, исполнение перейдёт на аналогичный цикл меньшей длины с меньшим объёмом загружаемой в регистр информации.

По выводу профилировщика для первоначально реализованной вер-

сии intrinsic-функции видно[4], что наибольшую часть времени, проведённого внутри функции, процессор исполнял цикл по счётчику 16 байт. Это значит, что его оптимизации должно было быть уделено пристальное внимание.

Один из возможных путей ускорения цикла — его частичная раскрутка [8]: увеличение числа одинаковых действий во время одной итерации цикла позволит уменьшить число прыжков. В исходной версии для AArch64, например, она не была проведена.

Кроме того, для дополнительного ускорения было произведено [7] избавление от расчётов в регистре-счётчике цикла, аналогично случаю Adler32.

Результаты до и после применения двух вышеуказанных оптимизаций находятся в приложении в разделе 7. Разница на больших данных составила примерно 12%.

Для последующей оптимизации применялась [6] инструкция «`sh2add`», благодаря которой, при используемом расширении «`Zba`», выполняется одна ассемблерная инструкция, вместо двух для доступа к элементам таблицы. Данная операция делается, чтобы прибавлять счётчик с учётом размера элементов массива к адресу его начала. Разница в производительности для такой оптимизации на больших данных достигает 20% (подробные результаты в разделе 7). За счёт этого достигается ускорение относительно существовавшей ранее в OpenJDK версии, поэтому интринсик предназначен для использования только на платах с расширением «`Zba`».

Поскольку алгоритмы, используемые при подсчёте CRC32 и CRC32C, отличаются лишь предпосчитанными данными и инвертированием бит регистра-результата, добавление intrinsic-функции `updateBytesCRC32C` после `updateBytesCRC32` состояло лишь в размещении отдельной таблицы, и возможности не делать инвертирование в начале и конце работы функции.

[4]вывод Perfasm в приложении в секции 7

GHASH

Была реализована согласно алгоритму, описанному в разделе 3.2.4, с использованием умножения Карацубы и взятия остатка от деления получившегося полинома по модулю полинома GCM. В реализации использовалась как стандартное векторное расширение «`V`», так и ратифицированные в сентябре 2023 года векторно-криптографические расширения: «`Zvbb`», «`Zvbc`». Поскольку на данный момент на рынке отсутствуют микропроцессоры, способные исполнять инструкции векторной криптографии, отладка и тестирование корректности проводились на эмуляторе QEMU. Характеристики эмулируемого процессора в QEMU находятся в приложении (секция 7).

Наибольшее внимание при реализации данной intrinsic-функции уделялось реализации умножения Карацубы на векторах. В предложенном для платформы AArch64 варианте использовались инструкции семейства «`pmull`», выполняющие векторное умножение без переноса. Поддержка аналогичных им инструкций «`vclmul`» и «`vclmulh`» в OpenJDK на момент начала работы отсутствовала. Кроме того, в некоторых случаях было необходимо менять местами элементы векторов, что потребовало добавления в OpenJDK поддержки ассемблерной инструкции «`vslideup.vi`». Также, на вход intrinsic-функции данные поступают с использованием порядка «от младшего к старшему» для байт, но «от старшего к младшему» для бит [20], поэтому потребовалось также поддержать инструкции «`vrev8.v`» и «`vbrev.v`» для более удобной работы с машинными словами во входных данных.

AES

В первую очередь была реализованы функции по шифрованию и дешифрованию одного блока входных данных. В них сначала считывались сгенерированные заранее ключи раундов, затем производилась нужная операция (`encrypt`/`decrypt`) согласно базовому алгоритму. За основу были взяты реализации для платформы AArch64, использовавшие специальные инструкции для ускорения AES, запускающие отдель-

ные шаги (процедуры) шифрования. В OpenJDK на момент начала работ отсутствовала поддержка инструкций для векторно-криптографического расширения «`Zvkned`», в котором включаются команды, реализующие схожий с AArch64 функционал, направленный на ускорение алгоритмов AES. Они были добавлены согласно спецификации [19]. В дальнейшем каждый из режимов шифрования использовал этот код, но со своей спецификой.

- У режима CBC после каждого этапа шифрования/дешифрования производилась операция XOR с предыдущим результатом.
- Режим CTR выполнял шифрование, а также XOR со счётчиком, подававшимся на вход. Кроме того, в изначально перенесённой версии применялась частичная раскрутка цикла шифрования, которая также была перенесена.
- Для режима GCM после цикла шифрования из режима CTR требовалось производить подсчёт функции GHASH, для которой в точности использовался код отдельной intrinsic-функции из предыдущего раздела.
- В случае режима ECB единственной платформой, для которой он был поддержан, была x86. В нём последовательно производился запуск того же кода, который используется для шифрования одного блока входных данных, что и было реализовано.

Поскольку расширение «`Zvkned`» лишь недавно было ратифицировано, на рынке пока что отсутствуют процессоры с поддержкой AES инструкций. Поэтому запуск функциональных тестов и отладка проводились на эмуляторе QEMU.

Реализация всех описанных intrinsic-функций находится в открытом форке OpenJDK [9].

6. Замеры производительности

В данной работе рассматривается ускорение отдельных методов OpenJDK. Для каждого из них необходимо добиться результатов на микробенчмарках лучше, чем код, генерируемый C2.

Полные таблицы с результатами находятся в приложении в секции 7.

`updateBytesAdler32`

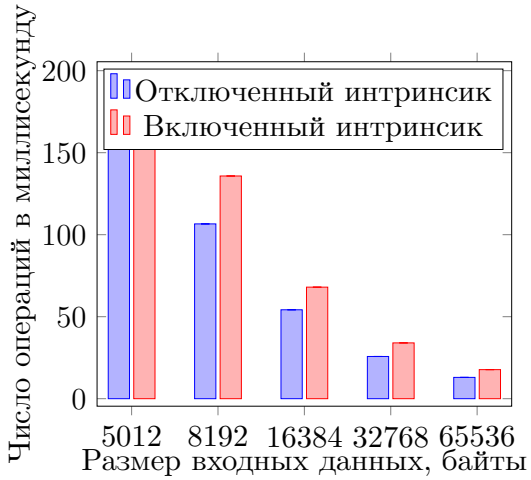

Рис. 1: Результаты работы микробенчмарка для Adler32

Обеспечена функциональная корректность на тесте `TestAdler32.java` [28] Измерение производительности проводилось на плате «Kendryte K230», поскольку требовалась поддержка векторных инструкций. Использованный микробенчмарк: «`Adler32.TestAdler32.testAdler32Update`» [12]. В случае данного метода даже на большом объёме входных данных выигрыш составляет около 32-36%, по сравнению с кодом, генерируемым компилятором C2.

poly1305_processBlocks

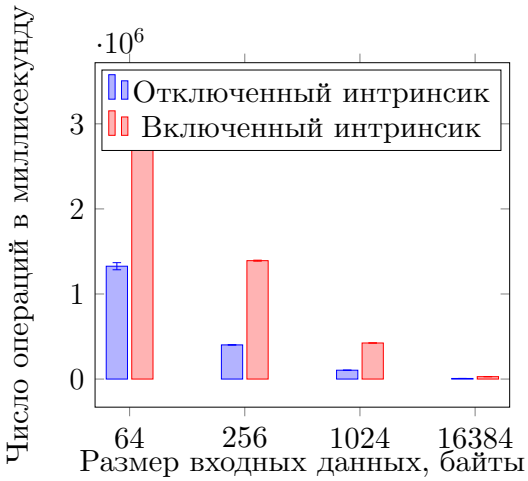

Рис. 2: Результаты работы микробенчмарка для Poly1305

Обеспечена функциональная корректность на соответствующих тестах из OpenJDK [26]. Замеры производились на плате «Alibaba T-head RVB-ICE» на существующих в OpenJDK микробенчмарках из класса «`Poly1305DigestBench`»: «`digestBuffer`», «`digestBytes`» и «`updateBytes`», все из которых внутри себя тестируют intrinsic-функцию `poly1305_processBlocks` [12]. На каждом из таких бенчмарков был получен выигрыш на всех наборах входных данных, на больших входных данных разница при включении интринсифицированного метода составляет от 248 до 260%.

`updateBytesCRC32`

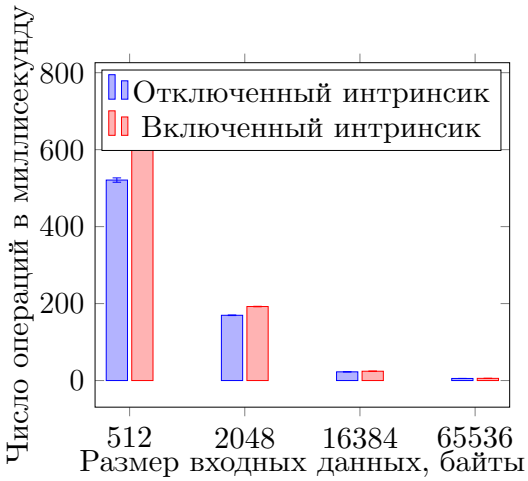

Рис. 3: Результаты работы микробенчмарка для CRC32

Обеспечена функциональная корректность на тесте `TestCRC32.java` из OpenJDK [30]. Замеры производились на плате с поддержанным расширением «`Zba`» — «StarFive VisionFive2» на микробенчмарке «`CRC32.TestCRC32.testCRC32Update`» [12]. На объёме начальных данных 64-256 байт выигрыш составляет от 217% до 80%. При дальнейшем увеличении размера входных параметров преимущество реализации intrinsic-функции в нативных кодах становится меньше, однако **не нивелируется полностью**: при большом объёме данных выигрыш в производительности становится приблизительно 3%.

updateBytesCRC32C

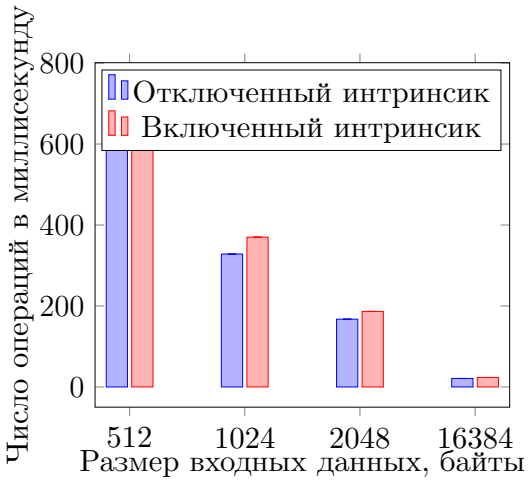

Рис. 4: Результаты работы микробенчмарка для CRC32C

Обеспечена функциональная корректность на тесте `TestCRC32C.java` из OpenJDK [29]. Для измерения производительности использовалась плата «StarFive VisionFive2», микробенчмарк «`CRC32C.TestCRC32C.testCRC32CUpdate`» [12]. Здесь ассемблерная реализация на небольших по размеру входных параметрах аналогично `updateBytesCRC32` выигрывает: разница между реализованной в JVM и в нативных RISC-V кодах в числе операций в миллисекунду составляет +33%. В это же время на больших данных ситуация становится примерно такой же, как и в случае CRC32: около 3-4% прирост производительности.

ghash_processBlocks

Обеспечена функциональная корректность на всех наборах входных данных для `TestGHASH.java` теста из OpenJDK [31].

AES

Для всех intrinsic-функций семейства AES была обеспечена функциональная корректность на соответствующем `TestAESMain.java` тесте из OpenJDK [27].

Обсуждение результатов

Все полученные результаты превосходят в производительности код, сгенерированный компилятором C2. Разница в производительности может быть сравнительно небольшой при увеличении размера входных данных по сравнению с малыми начальными данными, как это происходит в CRC32 intrinsic-функциях. В таком случае дальнейшее ускорение может быть достигнуто, например, за счёт векторизации, или, для CRC32, — с использованием расширения «`Zbc`».

Все полученные результаты использовались как аргументы в пользу интеграции реализованных intrinsic-функций в основную ветку OpenJDK. Статус принятия на момент весны 2024 года:

- Intrinsic-функция для Poly1305 была интегрирована в OpenJDK,
- CRC32 и Adler32 находятся на рассмотрении,
- CRC32C будет отправлена после принятия CRC32,
- GHASH и семейство AES intrinsic-функций будут отправлены на рассмотрение после появления на рынке плат с нужными им расширениями.

7. Результаты

В рамках данной работы были достигнуты описанные ниже результаты.

1. Выполнен обзор криптографических intrinsic-функций в OpenJDK:
   - выявлены 13 криптографических методов OpenJDK, нуждающихся в ускорении,
   - рассмотрены подходы к реализации их intrinsic-функций для различных платформ.
2. Для Java-методов, использующихся в криптографии и нуждающихся в ускорении, был реализован соответствующий им набор intrinsic-функций для платформы RISC-V. Для каждого из 13 методов была обеспечена его функциональная корректность.
3. Путём измерений производительности на платах с архитектурой RISC-V подтверждено, что все методы, для которых такие замеры возможно провести (CRC32, Adler32 и Poly1305), ускорены относительно сгенерированного виртуальной машиной кода на стандартных OpenJDK микробенчмарках.

Приложение

Вывод Perfasm в изначальном варианте реализации для Adler32

```
        ??      0x3fc83b31a8:   bltz    a2,0x3fc83b3268
3.59%   ??      0x3fc83b31ac:   ld      a4,0(a1)
3.67%   ??      0x3fc83b31ae:   zext.b  a5,a4
        ??      0x3fc83b31b2:   add     a0,a0,a5
        ??      0x3fc83b31b4:   srli    a5,a4,0x8
0.91%   ??      0x3fc83b31b8:   zext.b  a5,a5
1.83%   ??      0x3fc83b31bc:   add     a3,a3,a0
        ??      0x3fc83b31be:   add     a0,a0,a5
...
1.65%   ??      0x3fc83b31f0:   srli    a5,a4,0x30
0.97%   ??      0x3fc83b31f4:   zext.b  a5,a5
1.06%   ??      0x3fc83b31f8:   add     a3,a3,a0
1.80%   ??      0x3fc83b31fa:   add     a0,a0,a5
1.82%   ??      0x3fc83b31fc:   add     a3,a3,a0
2.84%   ??      0x3fc83b31fe:   srli    a5,a4,0x38
1.74%   ??      0x3fc83b3202:   add     a0,a0,a5
1.90%   ??      0x3fc83b3204:   add     a3,a3,a0
...
1.83%   ??      0x3fc83b3262:   addi    a2,a2,-16
1.78%   ??      0x3fc83b3264:   bgez    a2,0x3fc83b31ac
0.10%   ??      0x3fc83b3268:   addi    a2,a2,15
0.36%   ??      0x3fc83b326a:   bltz    a2,0x3fc83b327e
...
0.04%           0x3fc83b328e:   ld      s0,0(sp)
                0x3fc83b3290:   ld      ra,8(sp)
                0x3fc83b3292:   addi    sp,sp,16
                0x3fc83b3294:   ret
```

Влияние оптимизации счётчика цикла и его раскрутки на производительность Adler32

Замеры производились на плате «Kendryte K230» с использованным векторным расширением. Использованный микробенчмарк: «`Adler32.TestAdler32.testAdler32Update`». Число запусков — 25 штук

Размер данных, байты	Результат до оптимизаций	Ошибка	Результат после оптимизаций	Ошибка
64	7589.212	4.844	7177.572	13.724
128	4723.060	6.390	4724.756	6.231
256	2688.354	26.488	2813.707	2.464
512	1452.748	0.185	1557.127	1.325
1024	754.251	1.526	821.303	1.480
2048	385.121	0.125	422.749	0.333
5012	158.940	0.082	175.323	0.154
8192	99.754	0.115	117.811	0.157
16384	49.838	0.059	58.990	0.081
32768	23.943	0.049	28.827	0.066
65536	10.710	0.114	14.773	0.116

Таблица 1: Результаты оптимизаций цикла Adler32

Для использования инструкций векторного расширения необходимо явно указать ключ «`-XX:+UseRVV`» при старте виртуальной машины.

Оптимизация инструкции смены размерности векторов для Adler32

Листинг 1: Иллюстрация кода до оптимизации vsetvl

```
; Частично раскрученный цикл расчётов
for (int i = 0; i < 8; i++)
    vsetivli temp0, 16, e8, m1;
    ...
    ; Первый шаг подсчётов
    ; vbytes --- входные данные
    vwredsumu.vs unroll_regs[i], vbytes, vzero;
    ; Второй шаг подсчётов
    ; vtable = { 16, 15, 14, ..., 1 }
    vwmulu.vv vtemp1, vtable, vbytes;
    vsetivli temp0, 8, e16, m1;
    vadd.vv vtemp3, vtemp1, vtemp2;
    vwredsumu.vs unroll_regs[i + 8], vtemp3, vzero;
    ; Первые элементы unroll_regs[i + 8] и unroll_regs[i]
    ; теперь хранят результаты

vsetivli temp0, 2, e64, m1;
; Взятие результатов
for (int i = 0; i < 8; i++)
      vmv.x.s temp0, unroll_regs[i];
      vmv.x.s temp1, unroll_regs[i + 8];
      ; Результаты в temp0 и temp1
```

Листинг 2: Иллюстрация кода после оптимизации «vsetvl»

```
vsetivli temp0, 16, e8, m1;
; Частично раскрученный цикл расчётов
for (int i = 0; i < 8; i++)
    ...
    ; Первый шаг расчётов
    ; vbytes --- входные данные
    vwredsumu.vs unroll_regs[i], vbytes, vzero;
    ; Второй шаг расчётов
    ; vtable = { 16, 15, 14, ..., 1 }
    vwmulu.vv unroll_regs[i + 8], vtable, vbytes;
    ; Результат разделён  между unroll_regs[i + 8] и следующим
     ↪  за ним регистром:
    ; unroll_regs[i + 8] = { (b1 * 16), (b2 * 15), (b3 * 14),
     ↪  ..., (b8 * 9) }
    ; unroll_regs[i + 8]->successor() = { (b9 * 8), (b10 * 7),
     ↪  (b11 * 6), ..., (b16 * 1) }

; Взятие результатов
vsetivli temp0, 8, e16, m1;
for (int i = 0; i < 8; i++)
    ...
    vredsum_vs(vtemp1, unroll_regs[i + 8], vzero);
    vredsum_vs(vtemp2, unroll_regs[i + 8]->successor(),
     ↪  vzero);

    vmv.x.s temp0, unroll_regs[i];
    vmv.x.s temp1, vtemp1;
    vmv.x.s temp2, vtemp2;
    ; Результаты в temp0, temp1 и temp2
```

Влияние RISC-V-специфичных оптимизаций на производительность Adler32

Размер данных, байты	Результат до оптимизаций	Ошибка	Результат после оптимизаций	Ошибка
64	7177.572	13.724	8303.783	4.505
128	4724.756	6.231	5321.895	4.090
256	2813.707	2.464	3103.418	1.843
512	1557.127	1.325	1694.527	1.170
1024	821.303	1.480	885.590	2.340
2048	422.749	0.333	454.327	0.205
5012	175.323	0.154	187.816	0.132
8192	117.811	0.157	135.791	0.102
16384	58.990	0.081	68.019	0.038
32768	28.827	0.066	34.023	0.093
65536	14.773	0.116	17.723	0.042

Таблица 2: Результаты с включенной intrinsic-функцией **до оптимизаций**

Для использования инструкций векторного расширения необходимо явно указать ключ «`-XX:+UseRVV`» при старте виртуальной машины.

Разница в производительности расширения «Zba» для Poly1305

Используемый микробенчмарк: «Poly1305DigestBench.digestBytes», по метрике числа операций в миллисекунду. Замеры производились на плате «StarFive VisionFive2». Число запусков — 5 штук (стандартное для данного микробенчмарка в OpenJDK).

Размер данных, байты	Результат без расширения	Ошибка	Результат с расширением	Ошибка
64	438428.266	1133.615	473471.424	629.290
256	385128.011	1400.798	400511.047	3325.395
1024	251098.625	297.697	254303.255	1804.576
16384	31706.373	26.301	31675.181	25.079
1048576	520.664	0.800	521.008	1.204

Таблица 3: Результаты без использованного расширения «Zba»

Для использования «Zba» расширения необходимо явно указать ключ «-XX:+UseZba» при старте виртуальной машины.

Вывод Perfasm в изначальном варианте реализации CRC32

```
          0x3fb86a86bc:   nop
          0x3fb86a86be:   nop
...
3.17%  ?  0x3fb86a86dc:   lw     a0,0(t1)
5.23%  ?  0x3fb86a86e0:   and    a0,a0,t3
       ?  0x3fb86a86e4:   srli   t1,t0,0x8
       ?  0x3fb86a86e8:   zext.b t1,t1
       ?  0x3fb86a86ec:   slli   t4,t1,0x2
       ?  0x3fb86a86f0:   add    t1,a5,t4
       ?  0x3fb86a86f4:   lw     t4,0(t1)
1.79%  ?  0x3fb86a86f8:   and    t1,t4,t3
       ?  0x3fb86a86fc:   xor    a0,a0,t1
0.92%  ?  0x3fb86a8700:   srli   t1,t0,0x10
0.41%  ?  0x3fb86a8704:   zext.b t1,t1
       ?  0x3fb86a8708:   slli   t4,t1,0x2
       ?  0x3fb86a870c:   add    t1,a4,t4
...
0.84%  ?  0x3fb86a889c:   bgez   a2,0x3fb86a86c0
0.00%     0x3fb86a88a0:   addi   a2,a2,12
0.00%     0x3fb86a88a2:   bgez   a2,0x3fb86a85de
          0x3fb86a88a6:   addi   a2,a2,4
          0x3fb86a88a8:   bgtz   a2,0x3fb86a866c
...
0.00%     0x3fb86a88c0:   ld     s0,0(sp)
          0x3fb86a88c2:   ld     ra,8(sp)
          0x3fb86a88c4:   addi   sp,sp,16
          0x3fb86a88c6:   ret
```

Влияние оптимизации счётчика цикла и его раскрутки на производительность CRC32

Листинг 1: Diff для оптимизации счётчика CRC32

```
+  bind(L_by16_loop_entry);
+    const Register buf_end = x30; // t5
+    add(buf_end, buf, len); // buf_end will be used as endpoint for loop below
+    andi(len, len, 16-1); // len = (len % 16)
+    sub(len, len, 16); // Length after all iterations
   bind(L_by16_loop);
-    subw(len, len, 16);
     ld(tmp, Address(buf));
     update_word_crc32(crc, ..., table, ...);
     update_word_crc32(crc, ..., table, ...);
+
     ld(tmp, Address(buf, wordSize));
     update_word_crc32(crc, ..., table, ...);
     addi(buf, buf, 16);
     update_word_crc32(crc, ..., table, ...);
-    bge(len, zr, L_by16_loop);

+    ble(buf, buf_end, L_by16_loop);
     addiw(len, len, 16-4);
     bge(len, zr, L_by4_loop);
     addiw(len, len, 4);
```

Листинг 2: Diff для раскрутки цикла CRC32

```
+  const int64_t unroll = 20;
+  const int64_t unroll_words = unroll*wordSize;
...
-  bind(L_by16_loop_entry);
+  bind(L_unroll_loop_entry);
     const Register buf_end = x30; // t5
     add(buf_end, buf, len); // buf_end will be used as endpoint for loop below
-    andi(len, len, 16-1); // len = (len % 16)
-    sub(len, len, 16); // Length after all iterations
-  bind(L_by16_loop);
-    ld(tmp, Address(buf));
+  bind(L_unroll_loop);
-    update_word_crc32(crc, ..., table, ...);
-    update_word_crc32(crc, ..., table, ...);
-    ld(tmp, Address(buf, wordSize));
-    update_word_crc32(crc, ..., table, ...);
-    update_word_crc32(crc, ..., table, ...);
+    andi(len, len, unroll_words-1); // len = (len % unroll_words)
+    sub(len, len, unroll_words); // Length after all iterations
+    for (int i = 0; i < unroll; i++) {
+      ld(tmp, Address(buf, i*wordSize));
+      update_word_crc32(crc, ..., table, ...);
+      update_word_crc32(crc, ..., table, ...);
+    }
-    ble(buf, buf_end, L_by16_loop);
-    addiw(len, len, 16-4);
+    addi(buf, buf, unroll_words);
+    ble(buf, buf_end, L_unroll_loop);
+    addiw(len, len, unroll_words-4);
```

Замеры производились на плате «StarFive VisionFive2» с использованным расширением «`Zba`». Использованный микробенчмарк: «`CRC32.TestCRC32.testCRC32Update`». Число запусков — 12 штук.

Размер данных, байты	Результат до оптимизаций	Ошибка	Результат после оптимизаций	Ошибка
64	4206.654	0.547	4215.728	3.972
128	2308.843	3.565	2607.882	1.627
256	1214.727	0.305	1364.899	8.857
512	623.173	0.651	704.316	3.222
2048	158.965	0.376	180.738	0.474
16384	19.934	0.055	22.722	0.059
65536	4.730	0.007	5.327	0.019

Таблица 4: Результаты оптимизаций цикла CRC32

Разница в производительности расширения «Zba» в CRC32

Микробенчмарк: «CRC32.TestCRC32.testCRC32Update», по метрике числа операций в миллисекунду. Замеры производились на плате «StarFive VisionFive2». Число запусков — 12 штук.

Размер данных, байты	Результат без расширения	Ошибка	Результат с расширением	Ошибка
64	3563.320	3.326	4206.654	0.547
128	1928.837	2.234	2308.843	3.565
256	1005.273	1.953	1214.727	0.305
512	512.550	1.718	623.173	0.651
2048	130.396	0.341	158.965	0.376
16384	16.319	0.073	19.934	0.055
65536	3.913	0.011	4.730	0.007

Таблица 5: Результаты использования расширения «Zba» для CRC32

Для использования «Zba» расширения необходимо явно указать ключ «-XX:+UseZba» при старте виртуальной машины.

Характеристики эмулируемого QEMU процессора

```
# cat /proc/cpuinfo
processor   : 0
hart        : 0
isa     : rv64imafdcvh_zicbom_zicboz_zicntr_zicsr_
            _zifencei_zihintpause_zihpm_zba_zbb_zbs_sstc
mmu     : sv39
mvendorid   : 0x0
marchid     : 0x0
mimpid      : 0x0

# uname -a
Linux buildroot 6.6.0-15029-gbe3ca57cfb77 #1 SMP Thu Nov 16
↪  18:32:05 MSK 2023 riscv64 GNU/Linux
```

Полные результаты запуска микробенчмарков

Все запуски производились по метрике числа операций в миллисекунду.

`updateBytesAdler32`

Замеры производились на плате «Kendryte K230» на микробенчмарке «`Adler32.TestAdler32.testAdler32Update`».

Размер данных, байты	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	1867.257	10.034	8303.783	4.505
128	1651.408	10.354	5321.895	4.090
256	1345.505	4.847	3103.418	1.843
512	976.550	3.889	1694.527	1.170
1024	634.572	1.256	885.590	2.340
2048	371.763	0.588	454.327	0.205
5012	168.774	0.147	187.816	0.132
8192	106.578	0.135	135.791	0.102
16384	54.216	0.097	68.019	0.038
32768	25.744	0.025	34.023	0.093
65536	12.992	0.064	17.723	0.042

Таблица 6: Результаты работы микробенчмарка для Adler32

`poly1305_processBlocks`

Замеры производились на плате «Alibaba T-head RVB-ICE» на микробенчмарках «`Poly1305DigestBench.digestBuffer`», «`Poly1305DigestBench.digestBytes`» и «`Poly1305DigestBench.updateBytes`». Число запусков для всех микробенчмарков — 18 штук.

Размер данных, байты	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	223185.086	3131.485	250676.271	1454.754
256	164903.669	884.950	221451.149	1260.833
1024	80366.730	1100.096	160449.248	251.783
16384	7755.241	36.262	25203.082	35.786
1048576	121.664	0.264	440.824	1.070

Таблица 7: Результаты работы микробенчмарка «`Poly1305DigestBench.digestBuffer`»

Размер данных, байты	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	231896.008	836.575	263467.411	3095.752
256	158705.411	5798.758	222436.665	9161.904
1024	73558.623	603.631	165043.555	3893.206
16384	6939.142	33.912	25398.762	31.423
1048576	119.978	2.635	441.013	1.061

Таблица 8: Результаты работы микробенчмарка «`Poly1305DigestBench.digestBytes`»

Размер данных, байты	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	1326415.483	41797.496	3371496.212	9625.447
256	401563.162	2606.782	1391753.631	5824.551
1024	104105.493	640.884	424069.668	223.985
16384	7131.103	43.685	28256.683	80.286
1048576	121.816	0.897	445.610	0.277

Таблица 9: Результаты работы микробенчмарка «`Poly1305DigestBench.updateBytes`»

`updateBytesCRC32`

Замеры производились на плате «StarFive VisionFive2» с использованием расширения «`Zba`» на микробенчмарке «`CRC32.TestCRC32.testCRC32Update`».

Размер данных, байты	Число запусков	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	12	1390.530	42.217	4415.416	2.822
128	12	1109.742	24.201	2756.321	0.769
256	12	805.345	12.155	1450.461	4.115
512	12	520.965	5.651	750.851	0.496
2048	12	169.591	0.747	192.352	0.599
16384	12	22.624	0.139	24.209	0.044
65536	12	5.430	0.016	5.670	0.0159
131072	40	2.729	0.003	2.841	0.001
262144	40	1.367	0.001	1.420	0.001
524288	40	0.684	0.001	0.709	0.001
2097152	40	0.170	0.001	0.176	0.001

Таблица 10: Результаты работы микробенчмарка для CRC32

`updateBytesCRC32C`

Замеры производились на плате «StarFive VisionFive2» с использованием расширения «`Zba`» на микробенчмарке «`CRC32C.TestCRC32C.testCRC32CUpdate`».

Размер данных, байты	Число запусков	Результат без intrinsic-функции	Ошибка	Результат с intrinsic-функцией	Ошибка
64	15	3197.553	9.166	4464.713	7.374
128	15	2049.627	30.381	2776.158	2.444
256	15	1211.558	16.074	1459.655	1.978
512	15	666.442	4.975	753.855	0.324
1024	15	351.452	1.110	382.750	0.184
2048	15	180.660	0.051	192.901	0.079
4096	15	91.440	0.135	96.732	0.194
8192	15	45.667	0.094	48.488	0.043
16384	15	22.750	0.058	24.185	0.042
32768	15	10.805	0.009	11.697	0.027
65536	15	5.483	0.006	5.676	0.005
131072	40	2.741	0.003	2.836	0.005
262144	40	1.371	0.001	1.420	0.001
524288	40	0.686	0.001	0.712	0.005
1048576	40	0.340	0.001	0.355	0.001
2097152	40	0.169	0.001	0.177	0.001

Таблица 11: Результаты работы микробенчмарка для CRC32C

Список литературы

[1] ARM. Armv8.x and Armv9.x extensions and features. — URL: https://developer.arm.com/documentation/102378/0201/Armv8-x-and-Armv9-x-extensions-and-features (дата обращения: 14.12.2023).

[2] Awati R., Bernstein C., Cobb M. Advanced Encryption Standard. — URL: https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard/ (дата обращения: 01.05.2024).

[3] Bernstain D. The Poly1305-AES Message-Authentication Code. — Springer, Berlin, Heidelberg, 2005. — URL: https://link.springer.com/chapter/10.1007/11502760_3 (дата обращения: 14.12.2023).

[4] Bochkarev Arseny. Adler32 RISC-V specific optimizations. — URL: https://github.com/openjdk/jdk/pull/18382/commits/3359ec2a82d5d1ac38f9ec142948d37a7cc53590 (дата обращения: 25.05.2024).

[5] Bochkarev Arseny. Adler32 general optimizations. — URL: https://github.com/openjdk/jdk/pull/18382/commits/8cef929325f4552b92d2eb6ec1f26698a97a59e8 (дата обращения: 25.05.2024).

[6] Bochkarev Arseny. CRC32 RISC-V specific optimizations. — URL: https://github.com/openjdk/jdk/pull/17046/files#diff-7a5c3ed05b6f3f06ed1c59f5fc2a14ec566a6a5bd1d09606115767daa99 (дата обращения: 25.05.2024).

[7] Bochkarev Arseny. CRC32 induction variable elimination. — URL: https://github.com/openjdk/jdk/pull/17046/commits/e0f97888fe9a4cf50bef190e4d7c30e6782347ac (дата обращения: 25.05.2024).

[8] Bochkarev Arseny. CRC32 loop unrolling. — URL: https://github.com/openjdk/jdk/pull/17046/commits/

42503b769d462c02b8893833b65397090291265c (дата обращения: 25.05.2024).

[9] Bochkarev Arseny. OpenJDK RISC-V crypto intrinsics branch. — URL: https://github.com/ArsenyBochkarev/jdk/tree/riscv_crypto_intrinsics (дата обращения: 25.05.2024).

[10] Bochkarev Arseny. Poly1305 RISC-V specific optimizations. — URL: https://github.com/openjdk/jdk/pull/16417/files#diff-97f199af6d1c8c17b2fa4f50eb1bbc0081858cc59a899f32792a2d31f93 (дата обращения: 25.05.2024).

[11] Bochkarev Arseny. RISC-V boards used specs. — URL: https://github.com/ArsenyBochkarev/OpenJDK-RISCV-Intrinsics/blob/main/docs/benchmarks/micro/cpu/riscv/riscv_boards_specs.md (дата обращения: 25.05.2024).

[12] Bochkarev Arseny. OpenJDK intrinsics benchmarks. — 2023. — URL: https://github.com/ArsenyBochkarev/OpenJDK-RISCV-Intrinsics/tree/main/benchmarks/ (дата обращения: 14.12.2023).

[13] Bochkarev Arseny. Unimplemented Intrinsics Prior to Fall 2023. — 2023. — URL: https://github.com/ArsenyBochkarev/OpenJDK-RISCV-Intrinsics/blob/main/docs/riscv_crypto_intinsics_list.md (дата обращения: 14.12.2023).

[14] Castagnoli G., Brauer S., Herrmann M. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. — IEEE, 1993. — URL: https://ieeexplore.ieee.org/document/231911 (дата обращения: 14.12.2023).

[15] DFRobot. Stepping into RISC-V: A Detailed Look at Four Common RISC-V CPU Development Boards. — URL: https://www.dfrobot.com/blog-13481.html (дата обращения: 25.05.2024).

[16] Design of the Java HotSpotTM Client Compiler for Java 6 / T. Kotzmann, C. Wimmer, H. Mössenböck et al. — 2008. — URL: https://web.stanford.edu/class/cs343/resources/java-hotspot.pdf (дата обращения: 14.12.2023).

[17] International RISC-V. Frequently Asked Questions. — URL: https://web.archive.org/web/20160219195430/http://riscv.org/faq/ (дата обращения: 14.12.2023).

[18] International RISC-V. RISC-V "V" Vector Extension. — 2021. — URL: https://drive.google.com/file/d/1AQZ3l_EGeMa2NftMO562gVZ4vj61od2H/view?usp=drive_link (дата обращения: 14.12.2023).

[19] International RISC-V. RISC-V Cryptography Extensions Volume II Vector Instructions. — 2023. — URL: https://drive.google.com/file/d/1gb9OLH-DhbCgWp7VwpPOVrrY6f3oSJLL/view?usp=sharing (дата обращения: 14.12.2023).

[20] OpenJDK. AArch64 Stubs. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/aarch64/stubGenerator_aarch64.cpp (дата обращения: 14.12.2023).

[21] OpenJDK. IntrinsicCandidate.java. — URL: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/jdk/internal/vm/annotation/IntrinsicCandidate.java (дата обращения: 25.05.2024).

[22] OpenJDK. Intrinsics for Arm platform. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/aarch64/macroAssembler_aarch64.cpp (дата обращения: 25.05.2024).

[23] OpenJDK. Intrinsics for PowerPC platform. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/ppc/macroAssembler_ppc.cpp (дата обращения: 25.05.2024).

[24] OpenJDK. JDK9 stubGenerator_riscv.cpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/aarch64/stubGenerator_aarch64.cpp (дата обращения: 25.05.2024).

[25] OpenJDK. OpenJDK available platforms. — URL: https://github.com/openjdk/jdk/tree/master/src/hotspot/cpu/ (дата обращения: 25.05.2024).

[26] OpenJDK. Poly1305 JTreg test. — URL: https://github.com/openjdk/jdk/blob/master/test/jdk/com/sun/crypto/provider/Cipher/ChaCha20/unittest/Poly1305UnitTestDriver.java (дата обращения: 25.05.2024).

[27] OpenJDK. TestAESMain.java. — URL: https://github.com/openjdk/jdk/blob/master/test/hotspot/jtreg/compiler/codegen/aes/TestAESMain.java (дата обращения: 25.05.2024).

[28] OpenJDK. TestAdler32.java. — URL: https://github.com/openjdk/jdk/blob/master/test/hotspot/jtreg/compiler/intrinsics/zip/TestAdler32.java (дата обращения: 25.05.2024).

[29] OpenJDK. TestCRC32C.java. — URL: https://github.com/openjdk/jdk/blob/master/test/hotspot/jtreg/compiler/intrinsics/zip/TestCRC32C.java (дата обращения: 25.05.2024).

[30] OpenJDK. TestCRC32.java. — URL: https://github.com/openjdk/jdk/blob/master/test/hotspot/jtreg/compiler/intrinsics/zip/TestCRC32.java (дата обращения: 25.05.2024).

[31] OpenJDK. TestGHASH.java. — URL: https://github.com/openjdk/jdk/blob/master/test/jdk/com/sun/crypto/provider/Cipher/AES/TestGHASH.java (дата обращения: 25.05.2024).

[32] OpenJDK. Zlib in JVM. — URL: https://github.com/openjdk/jdk/blob/master/src/java.base/share/native/libzip/zlib/zlib.h (дата обращения: 02.05.2024).

[33] OpenJDK. library_call.cpp in OpenJDK. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/share/opto/library_call.cpp (дата обращения: 14.12.2023).

[34] OpenJDK. stubGenerator_riscv.cpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/riscv/stubGenerator_riscv.cpp (дата обращения: 25.05.2024).

[35] OpenJDK. stubRoutines_aarch64.cpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/aarch64/stubRoutines_aarch64.cpp (дата обращения: 25.05.2024).

[36] OpenJDK. vmIntrinsics.hpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp (дата обращения: 25.05.2024).

[37] OpenJDK. vm_version_linux_riscv.cpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/os_cpu/linux_riscv/vm_version_linux_riscv.cpp (дата обращения: 25.05.2024).

[38] OpenJDK. vm_version_riscv.cpp. — URL: https://github.com/openjdk/jdk/blob/master/src/hotspot/cpu/riscv/vm_version_riscv.cpp (дата обращения: 25.05.2024).

[39] Oracle. Compilation Optimization. — URL: https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/comp-opt.htm#JRHMG119 (дата обращения: 14.12.2023).

[40] Paleczny M., Vick C., Click C. The Java HotSpotTM Server Compiler. — 2001. — URL: https://www.researchgate.net/publication/220817735_The_Java_HotSpot_Server_Compiler (дата обращения: 14.12.2023).

[41] Peterson W. W., Brown D. T. Cyclic Codes for Error Detection // Proceedings of the IRE. — IEEE, 1961. — URL: https://ieeexplore.ieee.org/document/4066263 (дата обращения: 14.12.2023).

[42] RISC-V. Cost of vsetvl instructions. — URL: https://github.com/riscv/riscv-v-spec/issues/642 (дата обращения: 03.05.2024).

[43] RISC-V. RISC-V recently ratified extensions. — URL: https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions (дата обращения: 25.05.2024).

[44] RISC-V Bit-Manipulation ISA-extensions / B. Koppelmann, P. Adelt, W. Mueller, J. C. Scheytt. — 2021. — URL: https://drive.google.com/file/d/11-dKxnp7yfl9L3HESXGCtYl90dFKGTzE/view?usp=drive_link (дата обращения: 14.10.2023).

[45] Relic New. 2023 State of the Java ecosystem. — URL: https://newrelic.com/resources/report/2023-state-of-the-java-ecosystem (дата обращения: 25.05.2024).

[46] Saarinen M.-J. GCM, GHASH and Weak Keys. — 2011. — URL: https://www.researchgate.net/publication/220335697_GCM_GHASH_and_Weak_Keys (дата обращения: 14.12.2023).

[47] Vaillant L. The design of Poly1305. — 2017. — URL: https://loup-vaillant.fr/tutorials/poly1305-design (дата обращения: 14.12.2023).

[48] Waterman A., Asanović K. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. — 2019. — URL: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view?usp=drive_link (дата обращения: 14.12.2023).

[49] Wikipedia. ARM architecture family. — URL: https://en.wikipedia.org/wiki/ARM_architecture_family (дата обращения: 14.12.2023).

[50] Wikipedia. Block cipher mode of operation. — URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation (дата обращения: 14.12.2023).

[51] Wikipedia. Java Native Interface. — URL: https://en.wikipedia.org/wiki/Java_Native_Interface (дата обращения: 02.05.2024).

[52] Williams R. A Painless Guide to CRC Error Detection Algorithms. — 1993. — URL: https://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf (дата обращения: 14.12.2023).

[53] of Standards National Institute, Technology. Advanced Encryption Standard. — 2001. — URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf (дата обращения: 14.12.2023).