Программные системы и вычислительные методы
Правильная ссылка на статью:
Карпович В.Д., Государев И.Б. Исследование производительности WebAssembly в среде исполнения Node.js // Программные системы и вычислительные методы. 2025. № 2. DOI: 10.7256/2454-0714.2025.2.74049 EDN: QOVQLO URL: https://nbpublish.com/library_read_article.php?id=74049

Исследование производительности WebAssembly в среде исполнения Node.js

Карпович Владимир Дмитриевич

ORCID: 0009-0009-5397-3731

студент; факультет Программной Инженерии и Компьютерной Техники; Национальный исследовательский университет ИТМО

109369, Россия, г. Москва, р-н Марьино, ул. Перерва, д. 26 к. 1, кв. 94

mywinter4@yandex.ru

Государев Илья Борисович

ORCID: 0000-0003-4236-5991

кандидат педагогических наук

доцент; Мегафакультет компьютерных технологий и управления; Национальный исследовательский университет ИТМО

190000, Россия, г. Санкт-Петербург, Кронверкский проспект, 49

goss@itmo.ru

Статья из рубрики "Математическое и программное обеспечение новых информационных технологий"

DOI:

10.7256/2454-0714.2025.2.74049

EDN:

QOVQLO

Дата направления статьи в редакцию:

10-04-2025

Дата публикации:

18-04-2025

Аннотация: Современные среды исполнения, такие как браузеры, Node.js и пр. предоставляют разработчикам инструменты, позволяющие выходить за рамки

традиционного JavaScript. Объектом данного исследования выступает современный подход к созданию веб-приложений, в которых возможно выполнение и совместное использование компонентов, написанных на разных языках программирования, в результате применения WebAssembly. Предметом исследования является тестирование и анализ результатов тестов, направленных на измерение производительности JavaScript и WebAssembly-модулей в среде выполнения Node.js, с акцентом на сравнение эффективности выполнения вычислительных задач, взаимодействия с памятью, обработки данных и межъязыкового взаимодействия. Автор подробно рассматривает такие аспекты темы, как интеграция WebAssembly в приложения, оценка его преимуществ при решении ресурсоёмких задач, таких как обработка изображений, объективность, репрезентативность и воспроизводимость тестирования В работе используется прикладной, экспериментальный подход. Было произведено сравнение скорости выполнения операций при использовании чистого JavaScript и WebAssembly-модулей. Для оценки эффективности были использованы данные о времени ответа на запрос, о потреблении ресурсов системы приложением. Научная новизна данной работы заключается в разработке и теоретическом обосновании подходов к тестированию веб-приложений с использованием технологии WebAssembly. В отличие от большинства существующих исследований, сосредоточенных на производительности и безопасности WebAssembly в браузерной среде, в данной работе внимание акцентировано на автоматизированном тестировании WebAssembly-модулей вне браузера, что до настоящего времени оставалось слабо проработанным направлением. Предложен методологический подход к организации тестирования WebAssembly-модулей в среде выполнения Node.js, включая принципы структурирования тестов, интеграции с JavaScript-компонентами и анализ результатов выполнения. Такой подход позволяет учитывать специфику серверного окружения, где WebAssembly всё активнее применяется — в частности, при разработке высоконагруженных вычислительных модулей, кросс-языковой логики и безопасного изолированного исполнения. Научная новизна также заключается в выведении критериев, позволяющих оценивать пригодность тех или иных компонентов приложения для переноса в WebAssembly с точки зрения тестируемости, что даёт разработчикам дополнительный инструмент принятия архитектурных решений. Предложенные идеи подтверждены экспериментальной частью, включающей примеры реализации тестирования сценариев взаимодействия между WebAssembly и JavaScript.

Ключевые слова:

WebAssembly, JavaScript, оптимизация производительности, вычислительно интенсивные задачи, обработка изображений, интеграция технологий, серверные вычисления, экспериментальный анализ, алгоритмическая оптимизация, компиляция и интерпретация

ВВЕДЕНИЕ

В современных условиях развития веб-технологий требования к производительности веб-приложений постоянно растут. Увеличение объёма обрабатываемых данных и сложности выполняемых алгоритмов делает актуальным вопрос поиска эффективных решений в области оптимизации кода. JavaScript, как один из основных языков программирования для разработки веб-приложений, предоставляет разработчикам мощные инструменты и возможности; однако, своей интерпретируемой природой он ограничен в производительности по сравнению с компилируемыми языками.

С появлением WebAssembly возникли новые перспективы для повышения производительности веб-приложений. WebAssembly представляет собой низкоуровневый бинарный формат, который позволяет запускать код, написанный на языках программирования, таких как C, C++, Rust и других, непосредственно в среде выполнения JavaScript. Эта технология создает возможность для разработчиков писать производительный код, использующий преимущества компиляции, что позволяет избежать недостатков JavaScript в задачах, требующих высокой вычислительной мощности.

Актуальность изучения WebAssembly обусловлена его растущей популярностью и внедрением в широкий спектр приложений, от игр до обработки видео и графики. Тем не менее, эта технология еще недостаточно исследована, и образуется пробел в литературе, касающейся тестирования и производительности WebAssembly по сравнению с традиционными подходами. Цель данной работы заключается в анализе возможности интеграции WebAssembly в существующие приложения на JavaScript, а также в оценке его влияния на производительность.

ЭТАПЫ

В ходе проведения тестирования производительности JavaScript и WebAssembly в среде Node.js были выделены следующие этапы:

– выбор тестового приложения: было определено, что для оценки производительности подходящей задачей будет обработка изображений, так как это вычислительно интенсивная операция, требующая относительно больших ресурсов. В частности, была выбрана задача наложения фильтра путём применения алгоритма свёртки;

– разработка тестового приложения: был реализован простой веб-сервер, включающий функционал обработчика изображений как на JavaScript, так и на WebAssembly. Для реализации функции обработки изображений на WebAssembly использован язык C, который был скомпилирован в Wasm-файл с помощью инструмента Emscripten;

– настройка тестового окружения: для создания стабильной тестовой среды и минимизации влияния внешних факторов был использован инструмент контейнеризации Docker; в контейнерах были подняты инструменты для снятия, хранения и отображения метрик, веб-сервер с тестируемым функционалом, ПО для нагрузочного тестирования;

– проведение тестов: в вышеупомянутом тестовом окружении были запущены тесты, использующие различные изображения, и последовательно отправляющие запросы к серверу; во время тестирования собирались метрики о работе приложения и состоянии системы;

– анализ результатов: после завершения тестирования собранные данные были проанализированы, на основании результатов анализа были сделаны выводы о том, в каких случаях использование WebAssembly может быть уместным.

ТЕСТОВОЕ ПРИЛОЖЕНИЕ

Результаты исследования не смогут быть объективными и отражающими действительность, если эксперименты будут проводится над приложением, не подходящим для моделируемой ситуации. В рамках данного исследования предполагается, что WebAssembly будет использован в качестве дополнения к JavaScript как к основному языку программирования; WebAssembly будет использован точечно там, где использование JavaScript нецелесообразно, нерационально или неэффективно.

Таким образом, предполагается, что данный инструмент сможет расширить спектр ситуаций, в которых целесообразно использовать Node.js в качестве основной платформы для разработки [1].

Из этого следует, что приложение, над которым проводятся эксперименты, должно содержать в себе подлежащий оборачиванию в Wasm-формат участок. В случае, когда функционал может быть эффективно реализован нативными средствами Node.js, будем считать, что использование сторонних инструментов не оправдано.

В качестве основного функционала, который можно подвергнуть оптимизации с помощью WebAssembly, была выбрана обработка изображений. Наложение сложного фильтра – высоконагруженная операция, связанная с большим объёмом данных, регулярно применяемая в реальных приложениях; такая предметная область позволит достаточно объективно изучить влияние WebAssembly на производительность и оптимизацию приложения [2].

Архитектурно, приложение представляет собой HTTP-сервер с публичным API, позволяющим загружать, скачивать и обрабатывать изображения. Технологический стек включает в себя платформу Node.js, библиотеку для роутинга express и TypeScript.

Маршруты для обработки изображения представлены в трёх версиях: с алгоритмом, реализованном на JS и реализованном на C и скомпилированном в wasm-код в однопоточном и многопоточном режимах. Модуль WebAssembly инициируется единожды и затем вызывает экспортированную функцию обработки.

Исходный код приложения выложен в репозиторий на GitHub и доступен по ссылке https://github.com/Winter4/image-processor-backend [3].

АЛГОРИТМ ОБРАБОТКИ ИЗОБРАЖЕНИЯ

Обработка изображения производится с помощью алгоритма на основании свёртки. Свёртка — это математическая операция, основанная на концепции ядра (также называют фильтром или маской). Ядро — это квадратная матрица с обязательно нечётным размером, содержащая числа, которые определяют веса пикселей [4].

Свёртка выполняется следующим образом: центр ядра накладывается на текущий пиксель. Каждый элемент ядра умножается на свой пиксель, и результаты суммируются. Полученная сумма заменяет текущий (центральный) пиксель. После этого ядро сдвигается на следующий пиксель, и процесс повторяется. В итоге формируется новое изображение, где каждый пиксель является результатом свёртки. Все вычисления должны производиться с исходным изображением, не включающим уже «свёрнутые» пиксели.

Формула свёртки для изображения с одним каналом на пиксель выглядит следующим образом:

$$I'(x, y) = \Sigma^{k}_{i=-k} \Sigma^{k}_{j=-k} I(x + i, y + j) * M(i, j),$$

где x, y – координаты текущего пикселя; I'(x, y) – значение пикселя результирующего изображения в координатах (x, y); i, j – величина смещения по осям x и y соответственно, относительно координат текущего пикселя; k - половина стороны матрицы, округлённая вниз до целого; I(x + i, y + j) – значение пикселя исходного изображения в координатах (x + i, y + j); M(i, j) – значение элемента ядра в

координатах (k + i, k + j).

При обработке краёв изображения, где ядро выходит за границы, используются различные методы: заполнение нулями, отражение соответствующих пикселей на другой стороне ядра, повторение ближайшего пикселя или циклическое повторение, при котором края «замыкаются» друг на друге так, будто изображение циклично повторяется.

Если пиксель представляет собой совокупность чисел с одинаковым диапазоном значений, например, 3 цветовых канала в системе RGB, свёртка будет применяться для каждого отдельного канала. То есть, для вычисления нового значения R-канала текущего пикселя необходимо произвести вычисления с R-каналами окрестных пикселей, далее – с G- и B-каналами соответственно. Если пиксель представляет собой совокупность чисел с разными диапазонами значений, например, RGBL, где L – яркость от 0 до 9, то есть два варианта: нормализация значений, произведение вычислений и последующая денормализация, или применение отдельного ядра для вычисления нового значения яркости. Применение единого ядра к ненормализованным значениям с большой вероятностью приведёт к отклонениям от ожидаемого результата, однако теоретически допустимо.

Свёртка является универсальным инструментом, лежащим в основе обработки изображений и даже более сложных систем, таких как свёрточные нейронные сети, используемые в задачах распознавания объектов и других областях анализа данных.

ТЕСТОВОЕ ОКРУЖЕНИЕ

Тестовое окружение состоит из сервера приложения, ПО для снятия метрик, ПО для хранения и анализа метрик, ПО для визуализации метрик и ПО для нагрузочного тестирования.

В качестве ПО для снятия метрик был использован Cadvisor – инструмент с открытым исходным кодом, разработанный Google, для мониторинга и анализа производительности контейнеров. Cadvisor автоматически обнаруживает контейнеры, работающие на хосте, собирает статистику о их производительности и предоставляет данные в реальном времени. Хоть в основном инструмент рассчитан на Docker, его можно использовать со всеми платформами контейнеризации, поддерживающими cgroups [5].

Для хранилища метрик был выбран Prometheus. Это популярная система мониторинга и оповещения с открытым исходным кодом, разработанная для сбора метрик из различных источников, их хранения и анализа. Изначально был создан в SoundCloud, и в процессе стал фактическим стандартом в индустрии. Prometheus работает по принципу pull-модели, то есть сам опрашивает целевые источники данных через HTTP; формат метрик также стал стандартом. В качестве языка запросов Prometheus использует PromQL. Метрики хранятся в собственной временной базе данных, где данные организованы по временным рядам с метками в формате "ключ-значение". Для оповещений Prometheus интегрируется с другим ПО, к примеру Alertmanager, что позволяет настраивать гибкие правила уведомлений и отправлять их через Slack, email и пр [6].

За визуализацию метрик отвечает Grafana - платформа с открытым исходным кодом для визуализации данных и работы с ними. Она позволяет подключаться к различным источникам данных, таким как Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL и многим другим, чтобы создавать настраиваемые панели (дашборды) с визуализацией. Grafana поддерживает широкий набор графических компонентов, включая графики,

гистограммы, тепловые карты, таблицы, что делает её универсальным инструментом для анализа данных. Есть возможность задавать гибкие фильтры, применять шаблоны и использовать переменные для упрощения работы с дашбордами в сложных системах. Поддерживаются настройки оповещений: можно задать пороговые значения для метрик и получать. Grafana широко применяется в DevOps для мониторинга инфраструктуры, серверов, приложений и баз данных, а также в бизнесе для построения аналитических отчетов. Благодаря своей модульности, масштабируемости и интеграции с различными системами Grafana является ключевым инструментом для визуализации данных в условиях современной разработки [7].

Нагрузочное тестирование производится с помощью k6 — это инструмент с открытым исходным кодом для нагрузочного и производительного тестирования, разработанный создателями Grafana. k6 написан на Go и предоставляет удобный синтаксис для настройки на JavaScript. Благодаря интеграции с CI/CD системами, такими как Jenkins, GitLab CI и GitHub Actions, k6 легко можно включить в процессы автоматизации. Он также поддерживает remote write - экспорт данных в реальном времени. k6 отличается гибкостью, простотой использования, высокой производительностью и возможностью работы с большими объемами запросов [8].

Для минимального влияния сторонних факторов на результаты тестирования, необходимо изолировать все элементы. Для этого отлично подходит технология контейнеризации. Для реализации был выбран контейнерный движок Docker – из-за своей популярности и распространённости (фактически является стандартом индустрии).

Для контейнеризации приложения-обработчика необходимо написать скрипт сборки образа – Dockerfile:

Рисунок 1 – Dockerfile для сборки образа приложения

Код Dockerfile в текстовом виде представлен в приложении А. Для консистентной развёртки инфраструктуры и тестового окружения был написан docker-compose файл [9]. Полный код файла представлен в приложении Б. Для повышения объективности исследования ограничим контейнер с приложением 1 ядром центрального процессора и

2 ГБ оперативной памяти.

ТЕСТИРОВАНИЕ

Проведём тестирование приложения в условиях вышеописанного тестового окружения. Так как узким местом приложения является функционал обработки изображений, и целью исследования является оценка производительности именно этого функционала, тестирование будет проводиться путём последовательной отправки запросов в одном потоке, один за одним. Это позволит избежать накладных расходов, связанных с обработкой параллельно-идущих запросов, произвести максимально точную оценку производительности, а также отследить сбои в стабильности работы системы, такие как утечка памяти [10].

Тестирование будет проводиться в двух режимах: с использованием маленького и большого изображения. Маленькое изображение – файл формата .jpg, объёмом 314 КБ, разрешением 1680x1050 пикселей; суммарное количество пикселей 1 764 000. Большое изображение – файл формата .jpg, объёмом 10.3 МБ, разрешением 11384 x 4221 пикселей; суммарное количество пикселей – 48 051 864. Тест с использованием маленького изображения будет представлять собой последовательную отправку 5000 запросов; с использованием большого – последовательную отправку 150 запросов. Часть файлов для тестирования была взята с ресурса examplefile.com [11].

Стандартное отклонение фильтра Гаусса (сигма) возьмём равное 7, размер ядра 7x7 [12].

Однопоточный режим

Результаты теста с маленьким изображением и JS-реализацией представлены на рисунке 2:

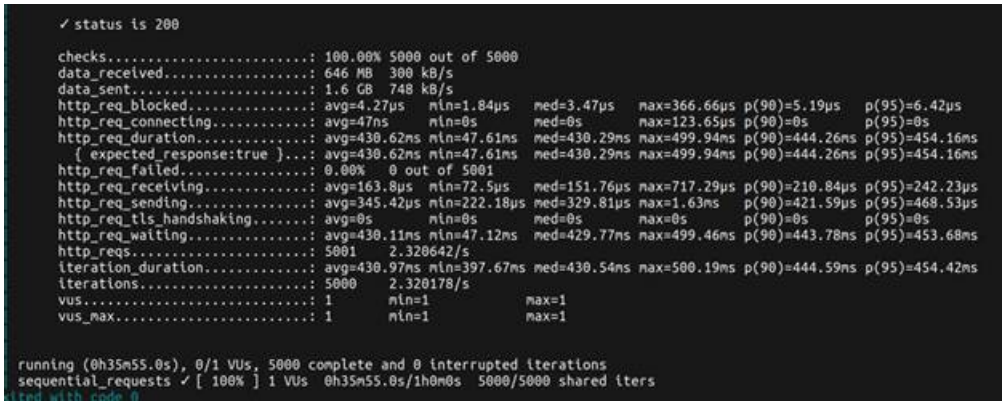

Рисунок 2 - результаты теста с маленьким изображением на JS-реализации

Показатели системы в процессе теста с маленьким изображением и JS-реализацией представлены на рисунке 3:

Рисунок 3 – показатели системы во время теста с маленьким изображением на JS-реализации

Как можно видеть, 5000 запросов были обработаны за 35 минут 55 секунд; среднее время ответа – 431 мс, это соответствует среднему показателю RPS 2.32. Средняя загрузка оперативной памяти при этом составила 40 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с маленьким изображением и Wasm-реализацией представлены на рисунке 4:

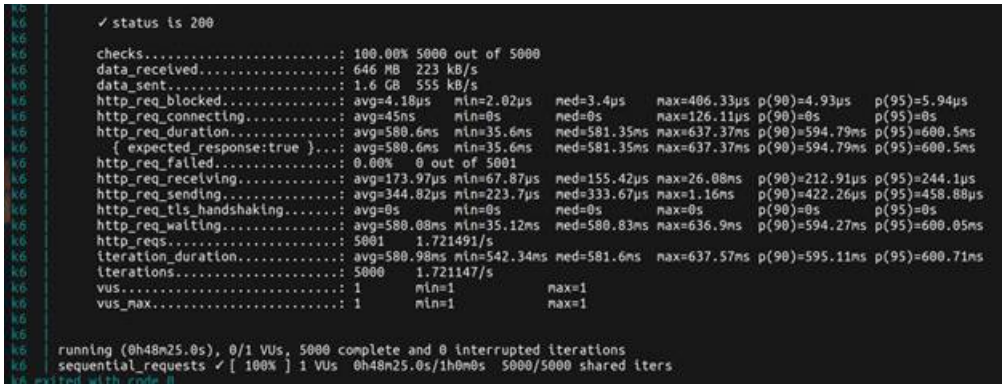

Рисунок 4 – результаты теста с маленьким изображением на Wasm-реализации

Показатели системы в процессе теста с маленьким изображением и Wasm-реализацией представлены на рисунке 5:

Рисунок 5 – результаты системы во время теста с маленьким изображением на Wasm-

реализации

5000 запросов были обработаны за 48 минут 25 секунд; среднее время ответа – 581 мс, это соответствует среднему показателю RPS 1.72. Средняя загрузка оперативной памяти при этом составила 43.7 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с большим изображением и JS-реализацией представлены на рисунке 6:

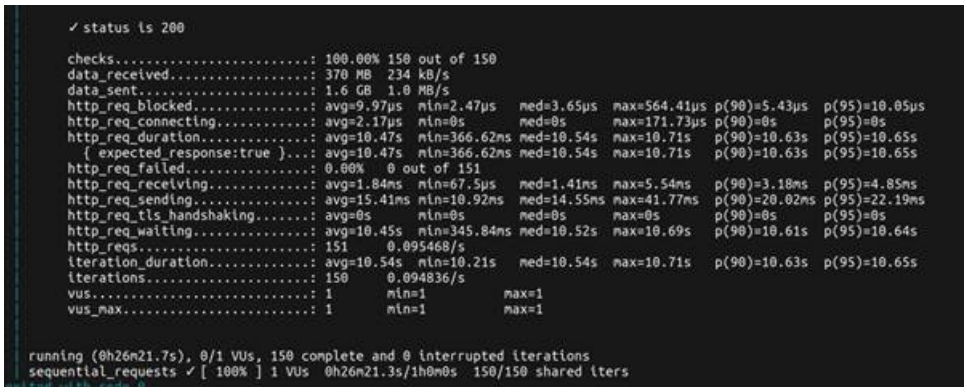

Рисунок 6 – результаты теста с большим изображением на JS-реализации

Показатели системы в процессе теста с большим изображением и JS-реализацией представлены на рисунке 7:

Рисунок 7 – показатели системы во время теста с большим изображением на JS-реализации

150 запросов были обработаны за 26 минут 21 секунду; среднее время ответа – 10.47 с, это соответствует среднему показателю RPS 0.09. Средняя загрузка оперативной памяти при этом составила 409.4 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с большим изображением и Wasm-реализацией представлены на рисунке 8:

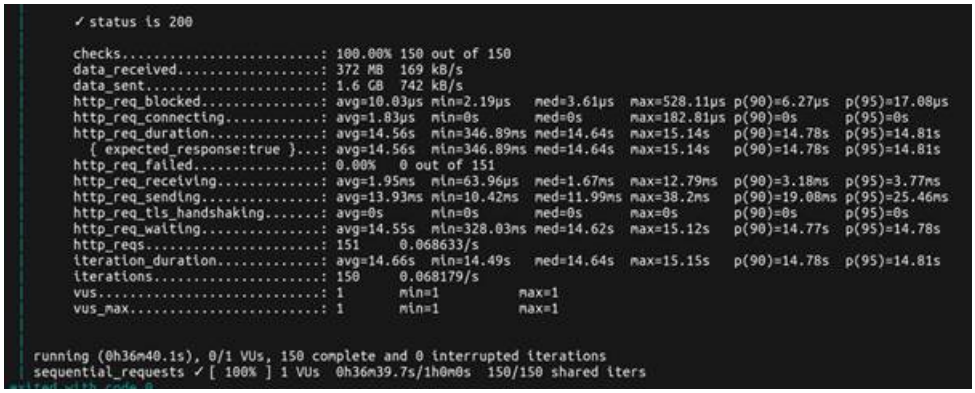

Рисунок 8 – результаты теста с большим изображением на Wasm-реализации

Показатели системы в процессе теста с большим изображением и Wasm-реализацией представлены на рисунке 9:

Рисунок 9 – показатели системы во время теста с большим изображением на Wasm-реализации

150 запросов были обработаны за 36 минут 39 секунд; среднее время ответа – 14.56 с, это соответствует среднему показателю RPS 0.06. Средняя загрузка оперативной памяти при этом составила 539.3 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Многопоточный режим

В JavaScript отсутствует нативная многопоточность. Пакет cluster из стандартного набора поставки Node.js может реплицировать приложение, однако каждая реплика в отдельности всё ещё будет однопоточной, и таким образом удельная производительность останется неизменной. В Emscripten (компилятор WebAssembly), в свою очередь, нативная поддержка многопоточности появилась в 2019 году в версии 1.38.27. Emcc использует Web Workers API [13] и SharedArrayBuffer [14] для её реализации. При этом, многопоточность инкапсулирована на уровне .wasm-файла и настроек компилятора. То есть, при репликации такого приложения, на каждой из реплик будет использована многопоточность, что потенциально повышает удельную производительность.

Для исследования эффективности применения многопоточности в emcc исходный код на .c был доработан для использования Worker API, а в настройки компилятора были добавлены соответствующие флаги. Модуль был скомпилирован с использованием пула из 4х потоков. При этом, реализация алгоритма свёртки как таковая не изменилась – ни объём вычислений, ни их сложность. Исходный код этой версии приложения доступен в том же репозитории.

Тесты будут проведены с разными ограничениями по ядрам для контейнера с приложением: 1 ядро, 2 ядра, 4 ядра. Ограничения по оперативной памяти остаются неизменными – 2 ГБ.

Результаты теста с маленьким изображением и ограничением в 1 ядро ЦПУ представлены на рисунке 10:

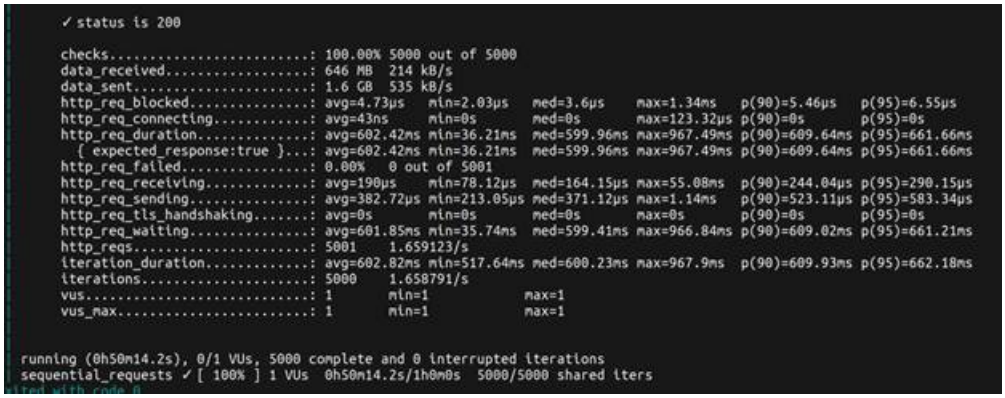

Рисунок 10 – результаты теста с маленьким изображением и ограничением в 1 ядро процессора

Показатели системы во время теста с маленьким изображением и ограничением в 1 ядро ЦПУ представлены на рисунке 11:

Рисунок 11 – показатели системы во время теста с маленьким изображением и ограничением в 1 ядро процессора

5000 запросов были обработаны за 50 минут 14 секунд; среднее время ответа – 602 мс, это соответствует среднему показателю RPS 1.65. Средняя загрузка оперативной памяти при этом составила 67.2 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с большим изображением и ограничением в 1 ядро ЦПУ представлены на рисунке 12:

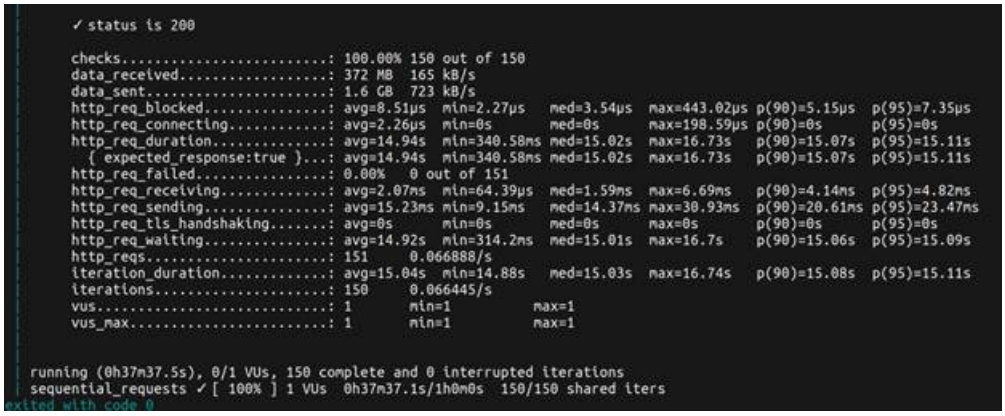

Рисунок 12 – результаты теста с большим изображением и ограничением в 1 ядро процессора

Показатели системы во время теста с большим изображением и ограничением в 1 ядро ЦПУ представлены на рисунке 13:

Рисунок 13 – показатели системы во время теста с большим изображением и ограничением в 1 ядро процессора

150 запросов были обработаны за 37 минут 37 секунд; среднее время ответа – 14.94 с, это соответствует среднему показателю RPS 0.06. Средняя загрузка оперативной памяти при этом составила 547.2 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с маленьким изображением и ограничением в 2 ядра ЦПУ представлены на рисунке 14:

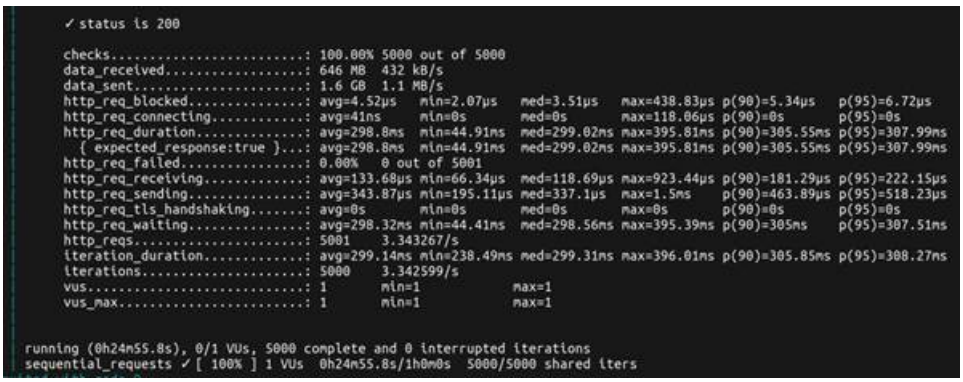

Рисунок 14 – результаты теста с маленьким изображением и ограничением в 2 ядра процессора

Показатели системы во время теста с маленьким изображением и ограничением в 2 ядра ЦПУ представлены на рисунке 15:

Рисунок 15 – показатели системы во время теста с маленьким изображением и ограничением в 2 ядра процессора

5000 запросов были обработаны за 24 минуты 55 секунд; среднее время ответа – 299 мс, это соответствует среднему показателю RPS 3.34. Средняя загрузка оперативной памяти при этом составила 67.1 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с большим изображением и ограничением в 2 ядра ЦПУ представлены на рисунке 16:

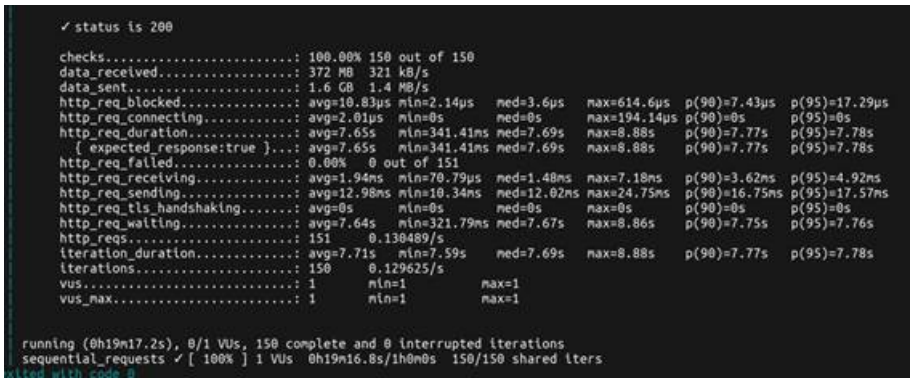

Рисунок 16 – результаты теста с большим изображением и ограничением в 2 ядра процессора

Показатели системы во время теста с большим изображением и ограничением в 2 ядра ЦПУ представлены на рисунке 17:

Рисунок 17 – показатели системы во время теста с большим изображением и

ограничением в 2 ядра процессора

150 запросов были обработаны за 19 минут 17 секунд; среднее время ответа – 7.65 с, это соответствует среднему показателю RPS 0.13. Средняя загрузка оперативной памяти при этом составила 541.6 МБ. Использование ЦПУ упиралось в лимит всё время теста.

Результаты теста с маленьким изображением и ограничением в 4 ядра ЦПУ представлены на рисунке 18:

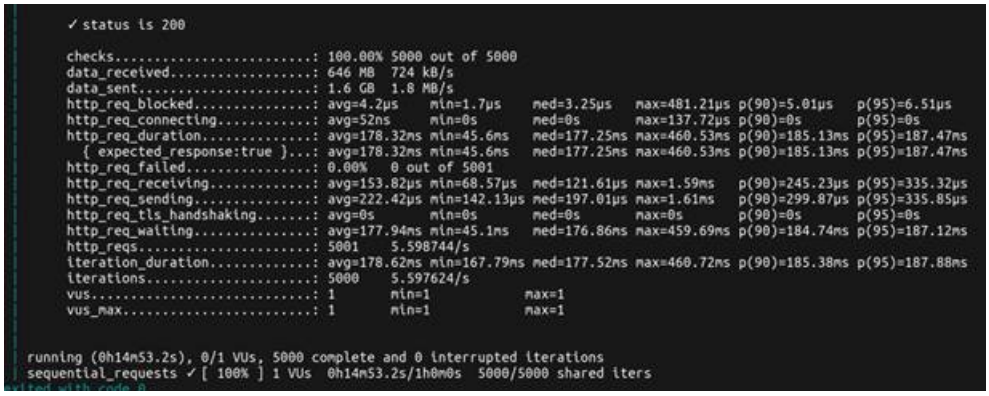

Рисунок 18 – результаты теста с маленьким изображением и ограничением в 4 ядра процессора

Показатели системы во время теста с маленьким изображением и ограничением в 4 ядра ЦПУ представлены на рисунке 19:

Рисунок 19 – показатели системы с маленьким изображением и ограничением в 4 ядра процессора

5000 запросов были обработаны за 14 минут 53 секунды; среднее время ответа – 178 мс, это соответствует среднему показателю RPS 5.6. Средняя загрузка оперативной памяти при этом составила 69.7 МБ. Использование ЦПУ стремилось к 350% всё время теста, что эквивалентно полной загрузке 3.5 ядер процессора.

Результаты теста с большим изображением и ограничением в 4 ядра процессора представлены на рисунке 20:

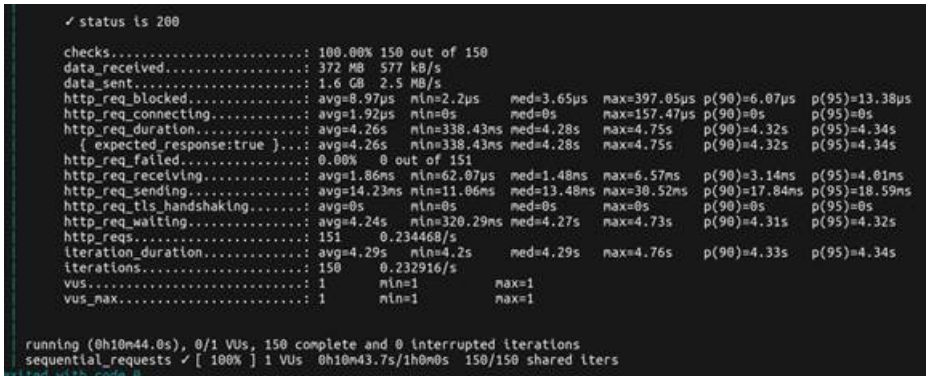

Рисунок 20 – результаты теста с большим изображением и ограничением в 4 ядра процессора

Показатели системы во время теста с большим изображением и ограничением в 4 ядра процессора представлены на рисунке 21:

Рисунок 21 – показатели системы во время теста с большим изображением и ограничением в 4 ядра процессора

150 запросов были обработаны за 10 минут 44 секунды; среднее время ответа – 4.26 с, это соответствует среднему показателю RPS 0.23. Средняя загрузка оперативной памяти при этом составила 521.8 МБ. Использование ЦПУ стремилось к 350% всё время теста, что эквивалентно полной загрузке 3.5 ядер процессора.

АНАЛИЗ

На основе проведённых тестирований можно утверждать, что в однопоточном режиме накладные расходы на использование Wasm-модуля не окупаются ни с маленьким изображением (для которого такой результат был ожидаем), ни для большого (для которого можно было прогнозировать уменьшение времени ответа). Время ответа на запрос для JS- и Wasm-решений составляет 430мс против 580мс и 10.47с против 14.56с соответственно; показатели увеличиваются на 35% и 39% соответственно. Причинами этого можно считать медленную работу с памятью в WebAssembly, т.к. специфичность задачи заключается в работе с относительно большими объёмами данных.

В многопоточном режиме ситуация меняется. Алгоритм, реализованный на Wasm в многопоточном режиме, но с ограничением в 1 ядро на контейнер, показывает близкий к однопоточному результат в 600мс против 580мс и 14.94с против 14.56с; увеличение времени ответа на запрос составляет ~40% относительно JS-реализации и ~3% относительно однопоточной Wasm-реализации. Увеличение времени ответа на запрос относительно однопоточной реализации может быть объяснено накладными расходами на запуск воркеров. Однако, как только лимит по ядрам начинает увеличиваться,

результаты улучшаются пропорционально добавлению ресурсов: при ограничении в 2 ядра Wasm-решение показывает 299мс на маленькой картинке и 7.65с на большой картинке; время ответа на запрос уменьшается на 30% и 27%. При ограничении в 4 ядра Wasm-решение начинает использовать весь потенциал своих четырёх потоков и показывает 178мс и 4.26с для маленькой и большой картинки соответственно, что означает уменьшение времени ответа на запрос на ~59%.

Также можно утверждать, что Wasm-решение потребляет стабильно больше ресурсов оперативной памяти устройства: от 30% до 65% в зависимости от конфигурации. Тем не менее, если сравнить абсолютные значения, разница не будет столь значительной: 40МБ против 67МБ и 409МБ против 547МБ. При учёте конфигураций современных вычислительных устройств данные значения можно считать сопоставимыми.

Результаты анализа сведены в таблицу и представлены в таблице 1 для удобства:

Таблица 1 – сравнение результатов

Решение	Режим	Изображение	Время ответа	Изм. врем. ответа
JS	Однопот.	Маленькое	0.43с	-
		Большое	10.47с	-
Wasm		Маленькое	0.58с	+35%
		Большое	14.56с	+39%
	Многопот. (1)	Маленькое	0.6с	+39%
		Большое	14.94с	+42%
	Многопот. (2)	Маленькое	0.3с	-30%
		Большое	7.65с	-27%
	Многопот. (4)	Маленькое	0.18с	-59%
		Большое	4.26с	-59%

Итоговые результаты: в однопоточном режиме WebAssembly показывает ухудшение результата на 38% в среднем.

В многопоточном режиме WebAssembly показывает почти такой же результат или лучше, в зависимости от выделенных ядер процессора. При использовании одного ядра результат хуже на 39% и 42%; при использовании двух ядер наблюдается улучшение результата на 28.5% в среднем; при использовании четырёх ядер результат улучшается на 59%.

На основании этого можно сделать вывод, что в однопоточном режиме предпочтительным вариантом является JavaScript, так как он быстрее справляется с обработкой задач. WebAssembly раскрывает свой потенциал только при использовании многопоточности, где увеличение числа ядер приводит к значительному улучшению производительности. Использование WebAssembly оправдано для вычислительно интенсивных задач и обработки больших изображений, особенно в условиях достаточного количества вычислительных ресурсов.

ЗАКЛЮЧЕНИЕ

В ходе исследования было успешно проведено тестирование производительности JavaScript и WebAssembly. Было разработано приложение, подходящее для проведения тестирования; подготовлено и настроено тестовое окружение; проведены тесты производительности, собраны метрики о состоянии приложения в процессе тестирования

и о загрузке ЦПУ и ОЗУ системы в процессе тестирования; собранные данные были проанализированы, на их основании сделаны выводы о целесообразности и эффективности различных решений; сформулированы рекомендации по использованию JavaScript и WebAssembly; все результаты были задокументированы и систематизированы.

Результатом исследования стали данные о производительности WebAssembly при работе в окружении Node.js, выводы на основании этих данных и рекомендации по использованию технологии в продуктовом приложении. Также, результаты этого исследования могут служить основой для дальнейших исследований в нескольких направлениях.

В частности, целесообразно более детально изучить поведение WebAssembly-модулей в различных средах выполнения, включая серверные, мобильные и встраиваемые платформы, с акцентом на устойчивость, безопасность и совместимость при автоматизированном тестировании. Это позволит выработать рекомендации по адаптации существующих подходов к более широкому кругу задач, а также выявить ограничения и потенциальные риски, связанные с использованием WebAssembly вне браузера. Также, значительный интерес представляет разработка специализированных инструментов и фреймворков, интегрирующих тестирование WebAssembly в современные процессы CI/CD. Это направление включает в себя не только автоматизацию запуска тестов, но и расширение средств анализа покрытия кода, отслеживания производительности, выявления утечек памяти и других низкоуровневых метрик. Также перспективным является исследование методов изоляции и песочницы при тестировании небезопасного или недоверенного кода, что особенно актуально в контексте микросервисной архитектуры и serverless-вычислений. В совокупности это формирует основу для системного подхода к обеспечению качества приложений, использующих WebAssembly как полноценный компонент вычислений.

ПРИЛОЖЕНИЕ А – Dockerfile для сборки образа приложения

```
FROM node:20-alpine

WORKDIR /app

COPY . .

RUN npm ci

RUN npm run tsc

ENV NODE_ENV=production

CMD ["node", "./compiled/source/api.js"]
```

ПРИЛОЖЕНИЕ Б – docker-compose файл для развёртки окружения

```
services:

app:

build: .

image: image-processor-backend:latest

container_name: backend
```

```
ports:
- "5001:5001"
healthcheck:
test: wget -nv -t1 --spider http://localhost:5001/ping/
interval: 5s
timeout: 5s
retries: 5
restart: always
deploy:
resources:
limits:
cpus: '4'
memory: 2048M
reservations:
cpus: '1'
memory: 1024M
cadvisor:
image: gcr.io/cadvisor/cadvisor:latest
container_name: cadvisor
ports:
- "9100:8080"
volumes:
- /:/rootfs:ro
- /var/run:/var/run:ro
- /sys:/sys:ro
- /var/lib/docker/:/var/lib/docker:ro
- /dev/disk/:/dev/disk:ro
prometheus:
image: prom/prometheus
container_name: prometheus
volumes:
```

```
- ./benchmark/prometheus.yml:/etc/prometheus/prometheus.yml

- prometheus-data:/prometheus

command: --web.enable-remote-write-receiver --
config.file=/etc/prometheus/prometheus.yml --storage.tsdb.path=/prometheus

ports:

- "9090:9090"

depends_on:

- app

- cadvisor

grafana:

image: grafana/grafana

container_name: grafana

volumes:

- ./benchmark/grafana/datasources:/etc/grafana/provisioning/datasources

- ./benchmark/grafana/dashboards:/etc/grafana/provisioning/dashboards

- ./benchmark/grafana/dashboards-json:/var/lib/grafana/dashboards

- grafana-data:/var/lib/grafana

ports:

- "3001:3000"

depends_on:

- prometheus

k6:

image: grafana/k6

container_name: k6

environment:

- K6_PROMETHEUS_RW_SERVER_URL=http://prometheus:9090/api/v1/write

- K6_PROMETHEUS_RW_TREND_STATS=avg

volumes:

- ./benchmark/k6:/k6

command: run --out experimental-prometheus-rw /k6/k6.js # image entrypoint is 'k6'

depends_on:
```

app:

condition: service_healthy

required: true

volumes:

grafana-data:

driver: local

name: image-processor-benchmark-grafana

prometheus-data:

driver: local

name: image-processor-benchmark-prometheus

Библиография

1. Stefan Fredriksson. WebAssembly vs. its predecessors: A comparison of technologies // diva-portal.org. Jul. 25, 2020. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1460603&dswid=-4940 (дата обращения 07.04.2025).
2. Marcus Alevärn. Server-side image processing in native code compared to client-side image processing in WebAssembly // diva-portal.org. Jul 27, 2021. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1587964&dswid=-8296 (дата обращения 07.04.2025).
3. Репозиторий с исходным кодом проекта. URL: https://github.com/Winter4/image-processor-backend (дата обращения 07.04.2025).
4. Kirill Smelyakov. Efficiency of image convolution // ieee.org. Sep 8, 2019. URL: https://doi.org/10.1109/CAOL46282.2019.9019450 (дата обращения 07.04.2025).
5. Aravind Samy Shanmugam. Docker Container Reactive Scalability and Prediction of CPU Utilization Based on Proactive Modelling // ncirl.ie. Sep 13, 2017. URL: https://norma.ncirl.ie/2884/1/aravindsamyshanmugam.pdf (дата обращения 07.04.2025).
6. Xinchen Xu, Aidong Xu. Research on Security Issues of Docker and Container Monitoring System in Edge Computing System // iopscience.iop.org. Sep 27, 2020. URL: https://doi.org/10.1088/1742-6596/1673/1/012067 (дата обращения 07.04.2025).
7. Matti Holopainen. Monitoring Container Environment with Prometheus and Grafana // theseus.fi. May 3, 2021. URL: https://www.theseus.fi/bitstream/handle/10024/497467/Holopainen_Matti.pdf (дата обращения 07.04.2025).
8. Bahrami Sepide. Automated Performance Testing in Ephemeral Environments // thesis.unipd.it. Jul 9, 2024. URL: https://thesis.unipd.it/handle/20.500.12608/66511 (дата обращения 07.04.2025).
9. David Reis. Developing Docker and Docker-Compose Specifications: A Developers' Survey // ieee.org. Dec 22, 2021. URL: https://doi.org/10.1109/ACCESS.2021.3137671 (дата обращения 07.04.2025).
10. Adeel Ehsan. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions // mdpi.com. Apr 26, 2022. URL: https://doi.org/10.3390/app12094369 (дата обращения 07.04.2025).
11. Веб-ресурс, сборник различных тестовых файлов. URL: https://examplefile.com (дата обращения 07.04.2025).

12. Mark van dek Wilk. Convolutional Gaussian Processes // proceedings.neurips.cc. 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/1c54985e4f95b7819ca0357c0cb9a09f-Abstract.html (дата обращения 07.04.2025).
13. Linus Hellberg. Performance evaluation of Web Workers API and OpenMP // diva-portal.org. Jul 6, 2022. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1681349&dswid=-9566 (дата обращения 07.04.2025).
14. Philip Lassen. WebAssembly Backends for Futhark // futhark-lang.org. June 29, 2021. URL: https://futhark-lang.org/student-projects/philip-msc-thesis.pdf (дата обращения 07.04.2025).

Результаты процедуры рецензирования статьи

В связи с политикой двойного слепого рецензирования личность рецензента не раскрывается.
Со списком рецензентов издательства можно ознакомиться здесь.

Представленная статья на тему «Исследование производительности WebAssembly в среде исполнения Node.js» соответствует тематике журнала «ПРОГРАММНЫЕ СИСТЕМЫ I ВЫЧИСЛИТЕЛЬНЫЕ МЕТОДЫ» и посвящена актуальному вопросу интеграци WebAssembly в существующие приложения на JavaScript, а также в оценке его влияния на производительность. Актуальность изучения WebAssembly обусловлена его растущей популярностью и внедрением в широкий спектр приложений, от игр до обработки видео и графики. Тем не менее, эта технология еще недостаточно исследована, и образуется пробел в литературе, касающейся тестирования и производительности WebAssembly по сравнению с традиционными подходами.
Авторами в ходе проведения тестирования производительности JavaScript и WebAssembly в среде Node.js были выделены следующие этапы:
– выбор тестового приложения;
– разработка тестового приложения;
– настройка тестового окружения;
– проведение тестов;
– анализ результатов.
В статье представлен достаточно широкий анализ литературных российских и зарубежных источников.
Авторами самостоятельно проведено исследование, в ходе которого предполагалось, что WebAssembly будет использован в качестве дополнения к JavaScript как к основному языку программирования; WebAssembly будет использован точечно там, где использование JavaScript нецелесообразно, нерационально или неэффективно. Таким образом, по мнению авторов предполагалось, что данный инструмент сможет расширить спектр ситуаций, в которых целесообразно использовать Node.js в качестве основной платформы для разработки
Стиль и язык изложения материала является достаточно доступным для широкого круга читателей. Практическая значимость статьи четко обоснована. Статья по объему соответствует рекомендуемому объему от 12 000 знаков.
Статья достаточно структурирована - в наличии введение, заключение, внутреннее членение основной части (этапы, тестовое приложение, алгоритм обработки изображения, тестовое окружение, тестирование, анализ).
Авторами в заключительной части статьи обозначено, что было проведено тестирование производительности JavaScript и WebAssembly. Практическая значимость исследования заключается в разработке приложения, подходящего для проведения тестирования; также авторами подготовлено и настроено тестовое окружение; проведены тесты

производительности, собраны метрики о состоянии приложения в процессе тестирования и о загрузке ЦПУ и ОЗУ системы в процессе тестирования; собранные данные были проанализированы, на их основании сделаны выводы о целесообразности и эффективности различных решений; сформулированы рекомендации по использованию JavaScript и WebAssembly; все результаты задокументированы и систематизированы.
К недостаткам можно отнести следующие моменты: из содержания статьи не прослеживается научная новизна. Отсутствует четкое выделение предмета исследования.
Рекомендуется четко обозначить научную новизну исследования, сформулировать предмет. Также будет целесообразным добавить о перспективах дальнейшего исследования.
Статья «Исследование производительности WebAssembly в среде исполнения Node.js» требует доработки по указанным выше замечаниям. После внесения поправок рекомендуется к повторному рассмотрению редакцией рецензируемого научного журнала.

Результаты процедуры повторного рецензирования статьи

В связи с политикой двойного слепого рецензирования личность рецензента не раскрывается.
Со списком рецензентов издательства можно ознакомиться здесь.

Статья посвящена актуальной проблеме сравнения производительности технологий WebAssembly и JavaScript при выполнении ресурсоемких операций в серверной среде Node.js. Основное внимание автора сосредоточено на анализе эффективности обработки изображений с использованием алгоритма свертки в различных режимах работы.
Методологическая база исследования отличается комплексным подходом и включает несколько взаимосвязанных этапов. Автором разработано специализированное тестовое приложение, реализующее алгоритм обработки изображений на двух технологических платформах - JavaScript и WebAssembly. Особую ценность представляет реализация многопоточного режима выполнения для WebAssembly. Для обеспечения точности измерений создано специализированное тестовое окружение с использованием современных инструментов мониторинга и анализа: Docker для контейнеризации, Prometheus для сбора метрик, Grafana для визуализации данных и k6 для нагрузочного тестирования. Проведена серия тестов с варьированием параметров, включая размеры обрабатываемых изображений и количество доступных ядер процессора.
Актуальность исследования не вызывает сомнений, так как проблема оптимизации производительности веб-приложений остается одной из ключевых в современной разработке. WebAssembly представляет собой перспективную технологию, однако ее сравнительный анализ с традиционными решениями на JavaScript в контексте серверной платформы Node.js до настоящего времени не получил достаточного освещения в научной литературе. Представленная работа восполняет этот пробел, предлагая ценные практические выводы для разработчиков.
Научная новизна исследования проявляется в нескольких аспектах. Во-первых, автором проведено комплексное сравнение производительности WebAssembly и JavaScript в серверной среде с акцентом на возможности параллельных вычислений. Во-вторых, разработаны практические рекомендации по выбору технологического решения в зависимости от доступных аппаратных ресурсов. В-третьих, выполнен детальный анализ особенностей работы с памятью при использовании WebAssembly, что представляет особый интерес для специалистов.

Стилистика статьи соответствует академическим стандартам, текст отличается четкостью изложения и логической завершенностью. Структура работы хорошо продумана: от обоснования актуальности проблемы через описание методологии к представлению результатов и их анализу. Использование графиков и сравнительных таблиц значительно повышает наглядность представленных данных. Единственным пожеланием может служить дополнение исследования анализом влияния версий используемого программного обеспечения на полученные результаты.

Библиографический список включает релевантные современные источники, что свидетельствует о глубине проработки темы. Для большей полноты можно было бы добавить ссылки на официальную документацию рассматриваемых технологий.

Ключевые выводы исследования имеют существенное практическое значение. Установлено, что в однопоточном режиме WebAssembly демонстрирует производительность на 38% ниже по сравнению с JavaScript. Однако при задействовании многопоточности и наличии двух и более ядер процессора ситуация кардинально меняется - WebAssembly показывает превосходство до 59%. Эти результаты позволяют сформулировать четкие рекомендации по выбору технологического решения в зависимости от доступных аппаратных ресурсов.

Статья представляет значительный интерес для широкого круга специалистов, включая разработчиков высоконагруженных приложений, исследователей веб-технологий и специалистов по оптимизации производительности. Высокий научный уровень работы, практическая значимость результатов и четкость изложения делают ее достойной публикации в журнале "Программные системы и вычислительные методы".