МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ МАШИН, КОМПЛЕКСОВ И КОМПЬЮТЕРНЫХ СЕТЕЙ

DOI 10.54398/20741707_2022_2_68
УДК 004.42

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ НИЗКОУРОВНЕВОГО ПРОЦЕССОРНОГО КОДА С ИСПОЛЬЗОВАНИЕМ WEBASSEMBLY

Статья поступила в редакцию 08.04.2022, в окончательном варианте – 26.04.2022.

Бородин Олег Валерьевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
магистрант, ORCID: 0000-0002-3769-0100, e-mail: oleg.borodin.1998@mail.ru
Егунов Виталий Алексеевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
кандидат технических наук, доцент, ORCID: 0000-0001-9087-3275, e-mail: vegunov@mail.ru
Плотников Владислав Павлович, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
магистрант, ORCID: 0000-0002-1540-0512, e-mail: vladplotnikov34@gmail.com

В работе рассматривается WebAssembly – средство, которое позволяет с помощью языка JavaScript в современном браузере получать доступ к виртуальной стековой машине инструкций. Использование WebAssembly или WASM решает проблему исполнения браузерного кода на низком процессорном уровне, открывая доступ к возможностям таких языков, как C, C++, Rust и многие другие. Приводится ряд примеров, которые демонстрируют плюсы использования технологии. Рассматриваются несколько различных функций, анализируется их запуск как в стандартной JavaScript-реализации, так и с использованием WASM, анализируется полученное время исполнения, считается ускорение. Приводится обзор существующих решений, где в качестве альтернатив приводятся ранее актуальные в сообществе варианты решения проблемы низкоуровнего выполнения кода. На основе полученных практических результатов анализируются достоинства и недостатки технологии WebAssembly. Подводятся итоги, выделяются реальные сценарии использования технологии.

Ключевые слова: JavaScript, WASM, WebAssembly, Web Development, Loaded Background Calculations, JavaScript Performance, Processor Native Code

FEATURES OF USING LOW-LEVEL PROCESSOR CODE USING WEBASSEMBLY

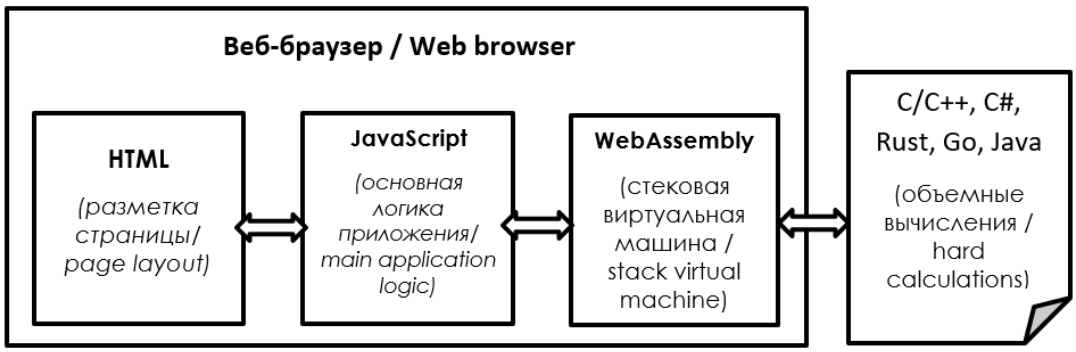
The article was received by the editorial board on 08.04.2022, in the final version – 26.04.2022.

Borodin Oleg V., Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,
master student, ORCID: 0000-0002-3769-0100, e-mail: oleg.borodin.1998@mail.ru
Egunov Vitaly A. Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,
Cand. Sci. (Engineering), ORCID: 0000-0001-9087-3275, e-mail: vegunov@mail.ru
Plotnikov Vladislav P., Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,
master student, ORCID: 0000-0002-1540-0512, e-mail: vladplotnikov34@gmail.com

The paper considers WebAssembly, a tool that allows using the JavaScript language in a modern browser to access a virtual instruction stack machine. Using WebAssembly or WASM solves the problem of executing browser code at a low processor level, opening access to the capabilities of languages such as C, C++, Rust, and many others. A few examples are given that demonstrate the advantages of using the technology. Several different functions are considered, their launch is analyzed both in the standard JavaScript implementation and using WASM, the resulting execution time is analyzed, acceleration is considered. An overview of existing solutions is provided, where alternatives are previously relevant in the community solutions to the problem of low-level code execution. Based on the obtained practical results, the advantages, and disadvantages of the WebAssembly technology are analyzed. The results are summed up, the real scenarios of using the technology are highlighted.

Keywords: JavaScript, WASM, WebAssembly, Web Development, Loaded Background Calculations, JavaScript Performance, Processor Native Code

Graphical annotation (Графическая аннотация)



Введение. Для веб-разработки, которая с каждым годом приобретает всё большую популярность и технологические новшества, остаётся актуальным по-прежнему один очень важный вопрос – быстрое исполнение кода в браузере. Современный браузер является мультиинструментом разработчика – он позволяет не только просматривать страницы, но и отлаживать код, следить за состоянием приложения, его ресурсами, сетью, безопасностью. Соответственно, нет сомнений в том, что множество улучшений в сфере веб-технологий может появиться и со стороны браузера, который уже давно функционально намного шире, чем многим кажется. Языком-монополистом в области веб-технологий и как минимум во всех популярных браузерах является JavaScript, то есть на данный момент все манипуляции вокруг веб-программирования по крайней мере на клиентской стороне будут опираться на данный язык.

JavaScript – это современный независимый язык, который обладает очень хорошей поддержкой со стороны разработчика, а также очень большим количеством активно используемых и развивающихся фреймворков и библиотек. Как известно, JavaScript изначально создавался как скриптовый язык, с помощью которого можно выполнять небольшие клиентские задачи в браузере, он не был приспособлен под быструю комплексную работу. Более того, JS однопоточен, хотя сейчас и есть некоторые способы, позволяющие частично обойти это ограничение. Конечно же, на основе этого нельзя сказать, что JavaScript неэффективен, но тем не менее у языка есть определенные внутренние ограничения и особенности, которые сильно ограничивают перспективы ускорения. Соответственно, было бы приоритетно то решение, которое имеет отличную совместимость с JavaScript, поскольку в современных условиях отказаться от него просто невозможно, но и между тем большую скорость исполнения.

Технология WebAssembly, о которой пойдет речь в данной работе, как раз является таким решением, она представлена новым форматом байт-кода, который доступен к исполнению во всех современных браузерах. Технология дает доступ к использованию в браузере таких языков, как C, C++, Rust и многих других, за счет сборки кода предварительно в набор низкоуровневых инструкций. Полученный формат является компактным, с одной стороны, при этом имеет производительность ближе к процессорной, нежели сам JavaScript. В то же время он также позволяет работать непосредственно и с JavaScript, который в данном случае обеспечивает передачу данных между браузером и WASM [1].

Так как на данный момент рассмотрение проблемы быстрого исполнения кода в браузере далеко не новое явление, то к решению проблемы также прилагаются некоторые сопутствующие требования, которые опираются на современное сообщество разработчиков:

- кроссплатформенность, поддержка мобильных устройств, нескольких операционных систем;
- скорость исполнения в пределах скорости машинного кода процессора, о чем упоминалось ранее;
- решение на базе самого браузера, то есть без установки;
- безопасность;
- удобство использования разработчика, инструменты отладки.

Таким образом, в данной работе будут рассмотрены имеющиеся способы и подходы в использовании машинного кода в веб-программировании, а также фактическом использовании других языков, отличных от JavaScript, для решения задач в среде веб-разработки. Результаты данного исследования могут быть использованы в качестве руководства для разработчиков, которые хотят ускорить ресурсоёмкие операции на клиенте, используя доступные и удобные средства. Ведь оптимизация на стороне клиента для конечного пользователя влечет за собой определенную выгоду использования – более быструю загрузку самого приложения, ускорение ряда операций, либо появление совершенно новой функциональности, которая доступна только за счет использования какого-то уникального API, доступного на базе совсем другого языка. Все перечисленные пункты чаще всего подразумевают сокращение времени ожидания клиента, а значит, улучшение опыта использования со стороны пользователя – соответственно, это подчеркивает необходимость проведения исследования

и его актуальность. Основной же задачей в данном случае является исследование способов применения, выявление достоинств и недостатков, а также использование современной технологии WebAssembly.

Обзор существующих решений. Ниже будет приведён перечень технологий, которые так или иначе применялись в решении поставленной проблемы частично или полностью. Для начала перечислим решения, которые уже не поддерживаются по тем или иным причинам, – это ActiveX, Adobe Flash, Microsoft Silverlight, Native Client и Portable Native Client.

ActiveX представляет собой фреймворк, определяющий программные компоненты, которые могут быть использованы и написаны на разных языках программирования. Приложение собирается из нескольких подобных компонентов и может использовать их функциональность.

В основе технологии лежит использование решений Microsoft, основывающихся на технологиях OLE (Object Linking and Embedding) и COM (Component Object Model), использующихся для связывания и внедрения объектов в другие объекты и документы. Технология обеспечивает соединение между различными слоями приложения, что позволяет им вместе работать через Internet, а также иметь систему, которая направляет программный трафик.

В настоящее время ActiveX официально считается вредоносной из-за существенных проблем с безопасностью. Microsoft отказалась от поддержки ActiveX в Internet Explorer 10 и в Windows 8, а в 2015 г. поддержка ActiveX была полностью прекращена.

Adobe Flash – это платформа компании Adobe Systems, использовалась для создания веб-приложений. Получила широкое применение в разработке игр, анимации, рекламных баннеров, была одной из лидирующих технологий воспроизведения видео и аудио на веб-страницах. Среди основных серьезных недостатков можно отметить чрезмерную нагрузку на центральный процессор, связанную с невысокой эффективностью виртуальной машины Flash Player, а также рядом уязвимостей, одной из которых является угроза перехвата flash-приложением содержимого буфера обмена. В 2017 г. поддержка данного продукта была прекращена.

Microsoft Silverlight – программная платформа, использовавшаяся для написания и запуска многофункциональных интернет-приложений RIA. Она похожа на Adobe Flash, поскольку также имеет модуль для браузера, который использовался для демонстрации видео, воспроизведения аудио, показа анимации, работы с векторной графикой. Поддержка этой технологии окончательно прекратилась в 2015 г.

Native Client (NaCl) и Portable Native Client (PNaCl) – технология для запуска кода на платформах x86, x86-64, ARM и MIPS, позволяющая безопасно запускать машинный код непосредственно в браузере независимо от операционной системы. Эта технология также может быть использована для создания защищённых плагинов для браузера, частей какого-либо приложения либо самих приложений.

Помимо создания барьера против нежелательных побочных эффектов, модули NaCl переносимы как между операционными системами, так и между веб-браузерами и поддерживают функции, ориентированные на производительность, такие как инструкции по обработке потоков и векторизации, расширение набора инструкций, таких как SSE, а также использование встроенных функций компилятора и написанного вручную ассемблера.

При этом сами разработчики выделяли потенциально слабые с точки зрения безопасности системные компоненты технологии и сами это рассматривали [2]

- внутренняя песочница: бинарная проверка;
- внешняя песочница: перехват системных вызовов ОС;
- загрузчик бинарных модулей во время выполнения службы;
- сервисные интерфейсы трамплина во время выполнения;
- интерфейс связи IMC;
- интерфейс NPAPI.

Разработчики подробно рассматривали аспекты безопасности и внедряли различные механизмы, включающие уровни защиты на основе собственной уверенности в надежности различных компонентов и достижения наилучшего баланса между производительностью, гибкостью и безопасностью.

Но, несмотря на множественные усилия, технология не нашла поддержки других браузеров кроме Chrome, в итоге в 2017 г. Google объявила об отказе от PNaCl в пользу WebAssembly, на которой делается акцент в данной работе [3].

У всех вышеописанных решений были те или иные проблемы с безопасностью, либо же они не выдержали испытания временем и конкуренции. В современных условиях эти технологии не прижились, и их поддержка и развитие прекратились. Таким образом их можно разделить на две группы:

- выполнение родного кода прямо в браузере;
- исполнение кода в рамках виртуальной машины.

Примерами технологий из первой группы являются ActiveX, NaCl. К минусам здесь можно отнести отсутствие портируемости, потенциальные или подтвержденные проблемы с безопасностью.

Примерами технологий из второй группы являются Java Applets, Flash, Silverlight. Здесь к минусам можно отнести необходимость наличия плагина и/или специальной среды выполнения, другими словами, отсутствует возможность запуска сразу «из коробки»

Далее можно выделить более современное решение, которое в сравнении с рассмотренными ранее решениями делает уверенный шаг вперед – это asm.js. Asm.js является подмножеством JavaScript, с более высокой оптимизацией. При работе с asm.js вводятся определенные ограничения – только конструкции «if» и «while», данные только в числовом формате, то есть строки и объекты в использовании недоступны, функции только в именованном формате [4].

Скрипты, написанные на этом подмножестве, подлежат эффективной компиляции: типы данных переменных определяются статически с использованием вывода типов. Используется в основном в качестве промежуточного языка для компиляции с таких языков, как C/C++ и используется в связке с такими инструментами, как Emscripten или Mandreel.

Emscripten позволяет произвести оптимизацию кода и превратить его в asm.js-нотацию, одной из примечательных черт которой является наличие множества побитовых ИЛИ (табл. 1).

Таблица 1 – Представление кода на asm.js

C/C++	Asm.js

int add(int a, int b){
return a + b;
}
	function add(a, b){
	return a | 0 + b | 0
	}

Это побитовое ИЛИ делает кое-что интересное со значениями. В этом случае оно действует как неявное приведение числа с плавающей запятой к числу int. Можно ознакомиться с данным примером в консоли:

```
var x = 3.5 | 0
console.log(x) // 3
```

Если скомпилированный код asm.js выполняет некоторую визуализацию, то, скорее всего, он обрабатывается WebGL и визуализируется с использованием OpenGL. Таким образом, весь конвейер (рис. 1) технически использует JavaScript и браузер, но почти полностью обходит фактический, нормальный путь выполнения кода и рендеринга, который использует JavaScript на веб-странице [5].

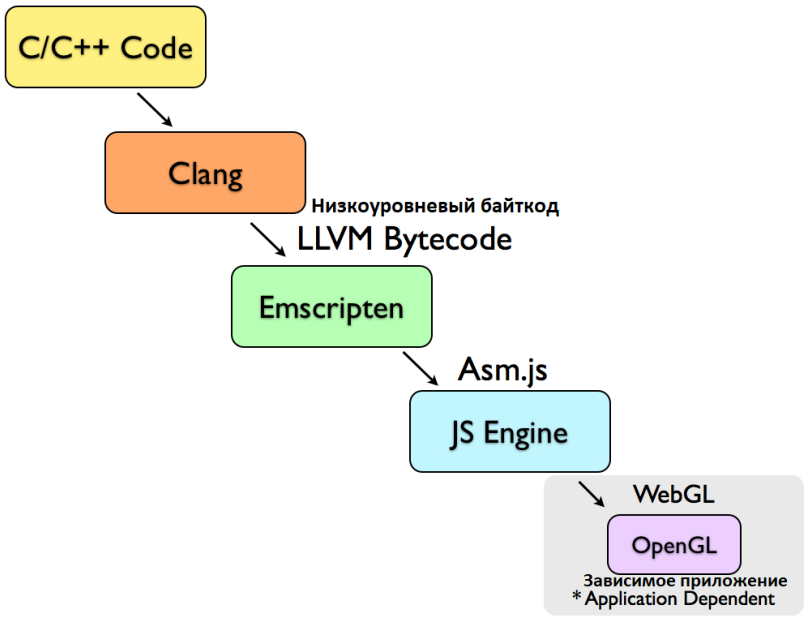


Рисунок 1 – Компиляция и исполнение asm.js

Как уже отмечено, asm.js – это подмножество JavaScript, которое сильно ограничено в том, что он может делать и как он может работать. Это сделано для того, чтобы скомпилированный код asm.js мог работать как можно быстрее, делая как можно меньше предположений, преобразуя код asm.js непосредственно в сборку. Вот основные моменты, которые позволяют asm.js быть быстрее и эффективнее JavaScript:

- asm.js отказывается от абстракций высокого уровня, таких как объекты JavaScript;
- asm.js не создает мусора, поэтому не нужно тратить время на сбор мусора;
- asm.js использует преимущества рабочих потоков, открывая более быстрый асинхронный код;
- asm.js позволяет использовать некоторые библиотеки C/C++, такие как OpenCV и Qhull.

Тем не менее, несмотря на все преимущества asm.js, в данной работе отдается большее предпочтение

в сторону WebAssembly, который детально рассматривается далее. Asm.js мог работать в основном с естественной скоростью, но на самом деле он никогда не работал стабильно во всех браузерах. Причина в том, что кто-то пытался оптимизировать его одним способом, кто-то – другим, с разными результатами. Со временем все стало сходиться, но основная проблема заключалась в том, что asm.js не был фактическим стандартом: это была неофициальная спецификация подмножества JavaScript, написанная одним поставщиком, которая лишь постепенно вызывала интерес и признание со стороны других.

WebAssembly, с другой стороны, был разработан совместно со всеми разработчиками основных браузеров. В отличие от JavaScript, который можно было сделать быстрым, только используя творческие методы, или asm.js, который можно было сделать быстрым, используя простые методы, хотя не все браузеры делают это, WebAssembly предоставляет более формальный подход к тому, как его оптимизировать [6].

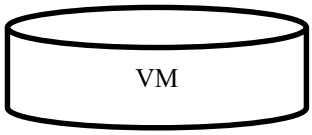
Таким образом по рассмотренным решениям сформируем таблицу и добавим в нее непосредственно WebAssembly (табл. 2).

Таблица 2 – Сравнение решений, решающих проблему браузерного эффективного исполнения кода

Технология	Годы поддержки	Языки	Архитектура	Платформа
ActiveX	1996–2015	C++, Delphi, Visual Basic, C#/VB.NET	Механизмы OLE и COM	Microsoft Windows, macOS, Solaris
Adobe Flash	1996–2017	*ActionScript*	Виртуальная машина, песочница	Microsoft Windows, Android, Linux, macOS, Solaris, BlackBerry OS
Microsoft Silverlight	2007–2015	C#, C++, JavaScript, Extensible Application Markup Language, Visual Basic	RIA (rich internet application), песочница	Microsoft Windows, macOS, Symbian OS
Native Client (NaCi) и Portable Native Client (PNaCi)	2011–2017	C, C++, JavaScript, Python, Ruby, Lua, Go	Песочница	Windows, Linux, macOS, Chrome OS
asm.js	2013 – наст. время	C, C++, JavaScript, Rust, Lua, Perl, Python, Ruby	Подмножество JavaScript, AOT (ahead-of-time) компиляция	Кроссплатформенное
WebAssembly	2017 – наст. время	C, C++, C#, JavaScript, Python, Go, Rust, Java, PHP, Lua, COBOL and other	Виртуальная машина, песочница	Кроссплатформенное

Основные особенности WebAssembly. WASM или WebAssembly – это бинарный формат, который совместим с браузером. Внутри технологии это – виртуальная машина, а на выходе – скомпилированное представление с поддерживаемого высокоуровневого языка. При этом WASM – не отдельный новый язык, а скорее инструмент. В качестве примера можно представить Java байт-код для виртуальной машины Java – это результат компиляции, то есть запускаемый блок кода (табл. 3). WASM-формат поставляется в браузер и там же обрабатывается, однако само исполнение формально осуществляется не браузером, а движком самого JavaScript. Отсюда же следует, чтоб браузер не единственный вариант использования, есть, например, среда NodeJS, которая служит чаще всего для реализации серверной составляющей приложений.

Таблица 3 – Информация о WebAssembly

WebAssembly		
бинарный формат:	виртуальная машина:	результат компиляции:
		(module (table $table0 0 funcref) (memory $memory (;0;) (export "memory") 1) (func $fib (;0;) (export "fib") (param $var0 i32) (result i32) (local $var1 i32) block $label0 local.get $var0 …

Сильные стороны технологии, за счет которых она выигрывает у рассмотренных решений ранее, можно обобщить в 3 крупных пункта – актуальность, семантика и представление.

Под актуальностью можно понимать активное развитие технологии, высокое упоминание в сообществе разработчиков, более простую интеграцию с современными инструментами, множество доступных примеров и документации.

Под семантикой имеется в виду независимость от языка, платформы, аппаратной части, а также быстрое выполнение, безопасность и детерминированность.

Представление – отличный формат, который достаточно компактный, простой для генерации, быстрый при декодировании, быстрый при компиляции, подходит для потоковой передачи, а также допускает параллелизм.

WebAssembly отличен от JavaScript, но он не предназначен для замены, он используется для дополнения и работы вместе с JavaScript, позволяя веб-разработчикам использовать сильные стороны обоих языков.

JavaScript – это гибкий язык высокого уровня, достаточно выразительный для написания масштабируемых веб-приложений и не только, часто он используется в робототехнике, программировании различных плат, игровой индустрии и мобильной разработке. У него много преимуществ – он динамически типизируется и имеет огромную экосистему, которая предоставляет мощные фреймворки, библиотеки и другие инструменты [7].

WebAssembly – это низкоуровневый язык, похожий на ассемблер, с компактным двоичным форматом, который предоставляется языкам с низкоуровневыми моделями памяти, такими, как, например, C и Rust. Цель компиляции заключается в том, чтобы они могли работать в вебе, то есть структурно WASM можно представлять как связующую прослойку между такими языками и браузером. Также WebAssembly имеет высокоуровневую цель поддержки языков с моделями памяти со сборкой мусора в будущем [8].

В глобальном смысле WebAssembly – это виртуальная стековая машина с памятью, исполняющая инструкции. Простота данной концепции позволяет быть доступной ей для современного процессора. Схема взаимодействия браузера в качестве среды исполнения, канала сообщений представленного JavaScript и непосредственно формата WASM, полученного при компиляции из другого языка, представлена на рисунке 2.

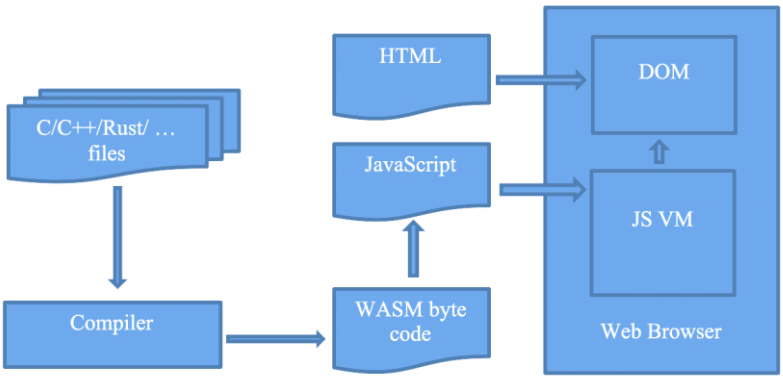


Рисунок 2 – Схема работы WebAssembly в браузере

Предлагаемые решения. В первую очередь ознакомиться с данной технологией можно с помощью простого онлайн-инструмента WasmFiddle, представленного на рисунке 3.

Интересовать здесь могут 4 области:

- в левом верхнем углу – исходный код на языке C;
- в левом нижнем углу – результат компиляции, доступный в нескольких форматах (текстовом, JavaScript-массиве чисел, Firefox x86-формате и др.);
- в правом верхнем – собранная и подготовленная к выполнению на JavaScript сборка;
- в правом нижнем – результат выполнения, вывод.

Для того чтобы опробовать WasmFiddle, будем использовать рекурсивную функцию расчёта n-го числа Фиббоначи:

```
int fibRecursive(int number) {
   if (number == 0) return 0;
   else    {
      if ((number == 1) || (number == -1)) return 1;
      else   {
         if (number > 0) return fibRecursive(number - 1) + fibRecursive(number - 2);
         else return fibRecursive(number + 2) - fibRecursive(number + 1);
      }
   }
}
```

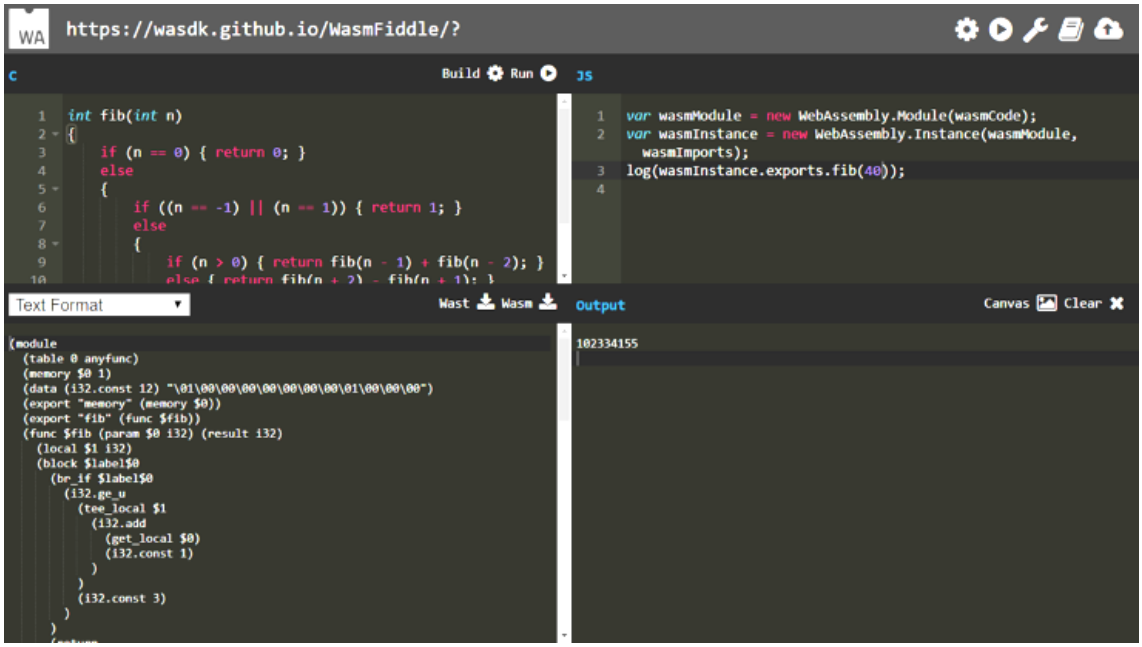


Рисунок 3 – Интерфейс WasmFiddle

На выходе компиляции результатом становится WASM-файл, его текстовое представление позволяет разобраться в том, что именно содержит сборка, какие таблицы, операции и код. С его помощью можно определять, что именно экспортируется наружу в виде страницы памяти и функций. Также это представление используется для отладки (табл. 3). В готовом виде для использования в JavaScript-среде это будет выглядеть следующим образом (один из вариантов представления WASM):

const wasmCodeArray = new Uint8Array([0, 0, 1, 1, 11, 10, 7, 112, 7, 2, 11, 109, 114, 10, 0, 0, 1, 134, 128, 0, 10, 0 , 96, 1, 11, 1, 127, 12, 127, 3, 13, 12, 128, 22, 10, 1 … 0, 65, 12, 11, 12, 1, 11, 1, 11, 10, 10, 1, 1, 0, 1, 1, 0, 0]);

Здесь WASM описывается в виде массива чисел, но на практике WASM-файл будет значительно больше, и его загрузка должна быть выполнена соответствующим образом с какого-либо источника, например со стороны сервера.

Исполнение WebAssembly в браузере происходит следующим образом: браузер отрисовывает html-страницу в обычном режиме с привязанными к ней скриптами, которые, в свою очередь, уже выполняют загрузку и подготавливают к работе WebAssembly – получается особый модуль (WebAssembly module), а затем создаётся его экземпляр, после этих действий с его помощью можно вызывать доступные для экспорта функции [9].

const module = new WebAssembly.Module(wasmCodeArray);
const wasmInstance = new WebAssembly.Instance(module, []);
console.log(wasmInstance.exports.fibRecursive (12));

Данный способ подходит для понимания работы технологии, однако использовать такой подход в выпускаемом продукте нецелесообразно, и здесь становится актуален компилятор, например Emscripten. Это наиболее предпочтительный компилятор для работы с asm.js и WebAssembly при получении их из C/C++. Также существуют компиляторы и под другие языки, например Go, C#, Python, Rust, TypeScript и многие другие.

Компиляция в asm.js: > emcc -o1 fibRecursive.c -o fibRecursive.html

Компиляция в wasm: > emcc -o1 fibRecursive.c -o fibRecursive.html –s WASM=1

Установка Emscripten довольно проста – загрузка с официального сайта, распаковка файлов и выполнение нескольких консольных команд. После этого компилятор можно свободно использовать. В результате мы получаем html-файл с основной структурой, js-файл с набором общих сервисных функций и, конечно же, wasm-файл.

Emscripten успешно развивается и уже поддерживает ряд привлекательных возможностей, среди них:

- стандартные библиотеки для C и C++;
- OpenGL, EGL - 2D/3D-графика на основе WebGL;
- SDL 2 – различный ввод (клавиатура/мышь/джойстики), видео, звук;
- OpenAL – звук;
- эмуляция файловой системы (Emscripten File System Overview);
- EM_ASM("JS code") – исполнение JavaScript-кода, сформированного в виде строки [10].

Обзор нескольких видов программ. Для исследования скорости исполнения WASM-модулей будем использовать некоторые функции, написанные на языке C/C++ и собранные в качестве модулей WebAssembly с помощью WasmFiddle, рассмотренного ранее, и противопоставленные им функции на языке JavaScript.

Среди примеров кода представлены:

- рекурсивная функция расчёта n-го числа Фиббоначи, на примере которой была рассмотрена работа с WasmFiddle;
- итеративная реализация предыдущей функции:

```
function fibIter (number) {
   let a = 0; let b = 1;
   for (let i = 2; i <= number; i++) {
      let c = b;
      b = a + b;
      a = c;
   }
   return b;
}
```

- простая функция перемножения чисел:

```
function jsMultiplyInt(a, b, size) {
   let c = 1.0;
   for (let i = 0; i < size; i++) {
     c = c * a * b;
   }
   return c;
}
```

- простая функция перемножения векторов:

```
function jsMultiplyIntVec(src1, src2, res, size) {
  for (let i = 0; i < size; i++) {
    res[i] = src1[i] * src2[i];
  }
}
```

Сама процедура тестирования будет проходить с помощью специального скрипта, нацеленного на фиксирование времени исполнения стандартной функции, а затем функции уже в WASM-формате с последующим расчетом ускорения.

JavaScript-файл будет содержать помимо самих тестируемых функций ещё и логику самого тестирования. В специальном объекте storage по уникальным ключам будет храниться время исполнения той или иной функции:

```
const jsId = 'js';
const wasmId = 'wasm'
let storage = {};
storage[jsId] = 0;
storage[wasmId] = 0;
```

Далее реализована функция, которая выполняет замер времени исполнения переданной в качестве аргумента функции и аргументов для неё, представленных массивом. Выглядит она следующим образом:

```
function execFunc(func, execFuncOptions) {
   const { arrArgs, funcCallCount } = execFuncOptions;
   func(...arrArgs); // warming up
   let elapsedTime = 0;
   for (let i = 0; i < funcCallCount; i++) {
      const start = performance.now();
      func(...arrArgs);
      const end = performance.now();
      elapsedTime += end - start;
   }
   return elapsedTime;
}
```

Здесь перед замером производится неучитываемый запуск вне цикла, для однозначного понимания работоспособности переданной функции, после производятся временные замеры с помощью специальной стандартной функции языка performance.now(), вычисляется разность между временной меткой до начала выполнения и временной меткой после выполнения вычислений, что как раз таки будет означать время исполнения самой функции, далее полученное время возвращается наружу.

Далее подготовим сами функции к тестированию и данные для них. Функция на JavaScript:

```
function jsMultiplyInt(a, b, n) {
   let c = 1.0;
   for (ley i = 0; i < n; i++)
    c = c * a * b;
   return c;
}
```

И она же (в исходном виде на C), но только в виде WASM-модуля:

```
const wasmCodeArr = new Uint8Array([1, 1, 0, 97, 115, 109, 1, 0, 0, 0, 2, 2, 6, 100, 101, 109, 1, 136, 128, 128, 128, 127, … 128, 0, 1, 0, 132, 2, 128, 128, 1, 128, 1, 0, 1, 112, 0, 0, 0, 11, 5, 131, 1, 2, 2, 65, 127, 106, 34, 10, 2, 13, 0, 11, 11, 10, 32, 3, 10, 21, 11]);
const wasmExportModule = new WebAssembly.Module(wasmCodeArr);
const wasmInstanceFunctions = new WebAssembly.Instance(wasmExportModule, []);
```

Данные для тестирования будем использовать следующие:

```
const num = 99999999;
const arr = [num, num, num];
```

Осуществим печать полученных функций в консоль:

```
const jsFunc = jsMultiplyInt;
const wasmFunc = wasmInstanceFunctions.exports.multiplyInt;
console.log(jsFunc);
console.log(wasmFunc);
```

И наконец реализация самой основной функции, test-запуск которой полностью производит нужный тест. Представлен будет основной фрагмент без промежуточных приготовлений:

```
setTimeout(() => {
        const wasmTime = execFunc(wasmFunc, execFuncOptionsWasm, wasmId);
        storage[wasmId] += wasmTime;
        setTimeout(() => {
           const jsTime = execFunc(jsFunc, execFuncOptionsJs, jsId);
           storage[jsId] += jsTime;
           const wasmAcceleration = jsTime / wasmTime;
                console.log(`WASM acceleration = (jsTime: ${jsTime} / wasmTime: ${wasmTime}) = ${wasmAcceleration}`);
           resetStorage(storage, jsId, wasmId);
        });
     });
```

Здесь стоит коснуться основных моментов:

- каждый запуск execFunc с помощью setTimeout выносится в независимый стек вызовов, то есть запуск производится в независимом пространстве, где основным вычислениям ничего мешать не будет;
- для корректной записи результатов используются ранее созданные идентификаторы jsId и wasmId, которые нужны для доступа к объекту storage;
- после вычислений производится расчет ускорения.

Результаты и замеры. Далее в уже рассмотренном формате произведём замеры для всех представленных ранее видов программ с помощью реализованного скрипта. Разница в тестах будет минимальная – обеспечить нужно будет лишь подачу различных аргументов на вход функциям.

Таблица 4 – Рекурсивная функция расчёта n-го числа Фибоначчи, количество запусков 10

N	JS, мс.	WASM, мс.	Ускорение
40	10106,2	5710,5	1,77
41	16361,3	9191,8	1,78
42	27074,2	15094,1	1,79
43	44069,4	24318,9	1,81
44	70758,5	38957,2	1,81

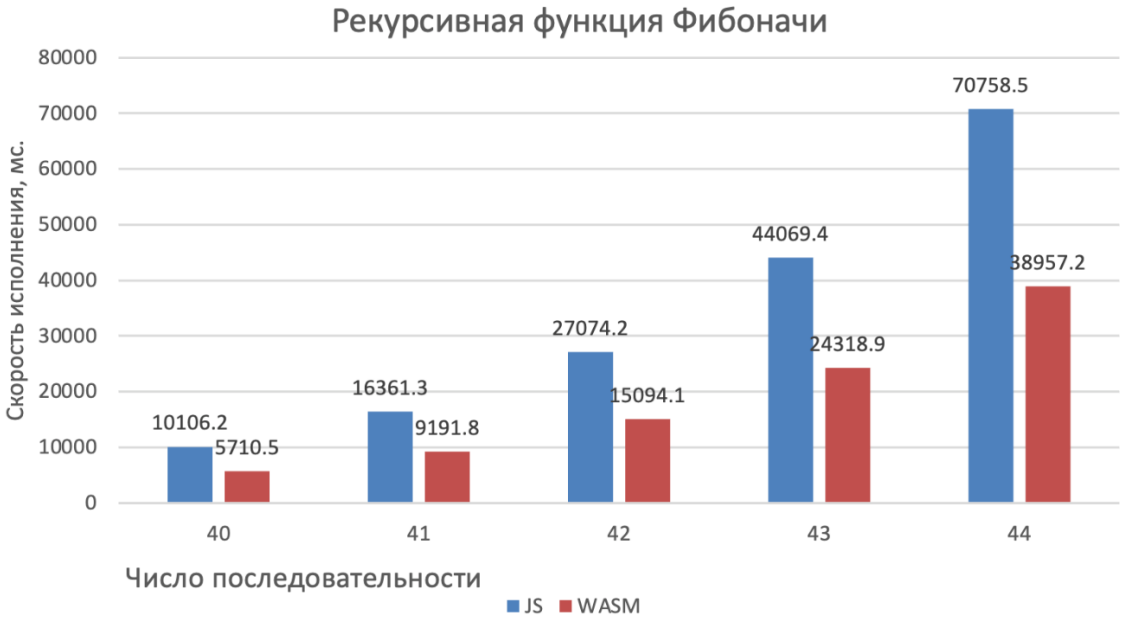


Рисунок 4 – Рекурсивная функция расчёта n-го числа Фибоначчи, количество запусков 10

Таблица 5 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 40–44

N	JS, мс.	WASM, мс.	Ускорение
40	468,9	454,0	1,03
41	477,3	441,0	1,08
42	466,89	434,0	1,07
43	468,4	479,2	0,98
44	471,0	446,1	1,05

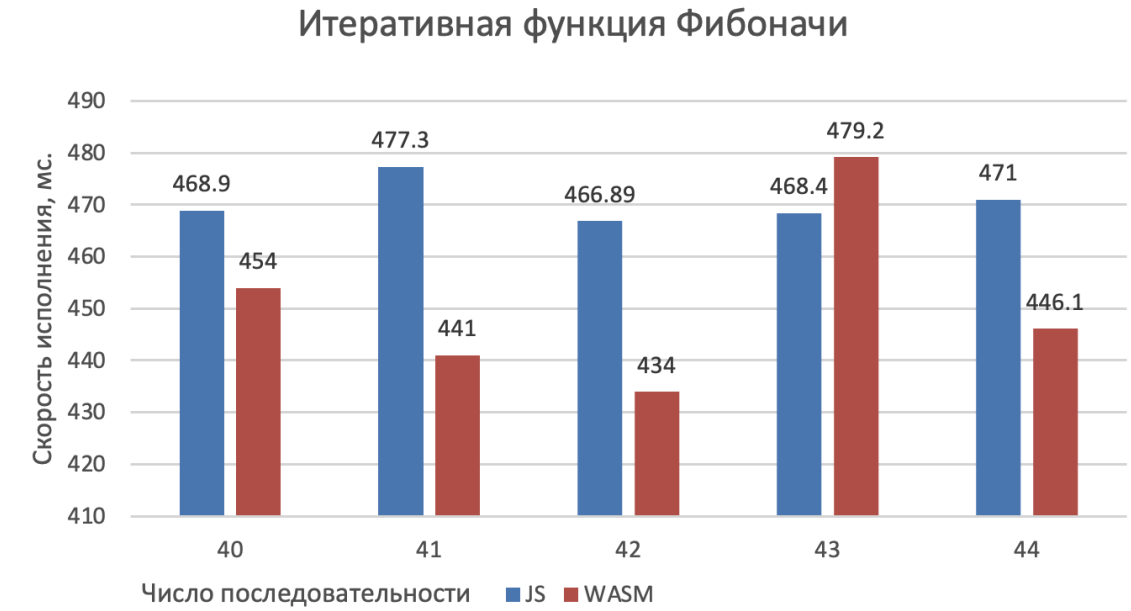


Рисунок 5 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 40–41

Таблица 6 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 100–500

N	JS, мс.	WASM, мс.	Ускорение
100	585,7	553,8	1,06
200	777,7	616,2	1,26
300	937	680,1	1,37
400	1087,8	745,2	1,46
500	1250,6	774,4	1,61

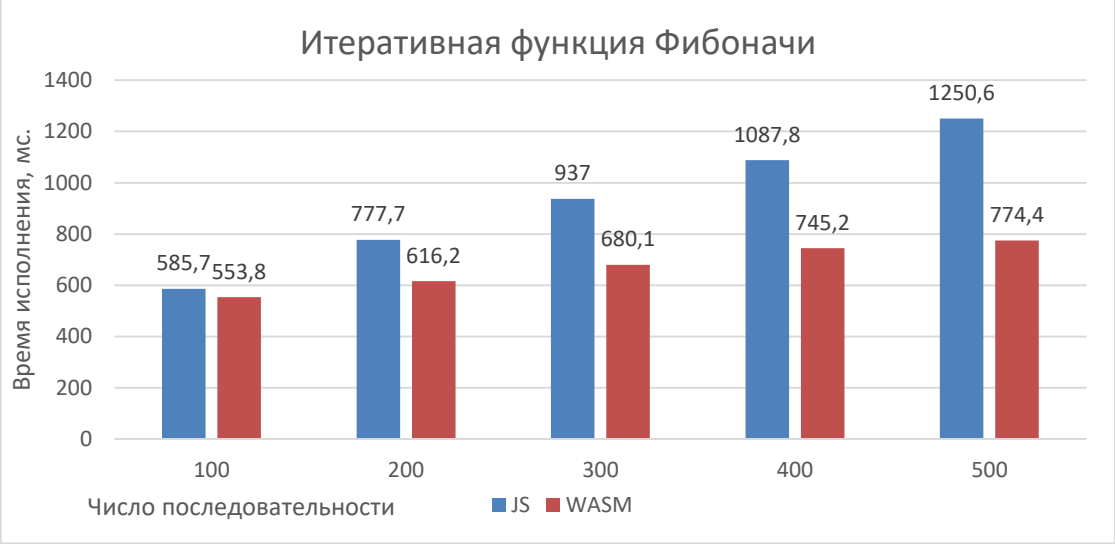


Рисунок 6 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 100–500

Таблица 7 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 1000–5000

N	JS, мс.	WASM, мс.	Ускорение
1000	2111,7	966,0	2,19
2000	3785,3	1464,5	2,58
3000	5399	1945,8	2,77
4000	6945,5	2491,3	2,79
5000	8549,5	3039,9	2,81

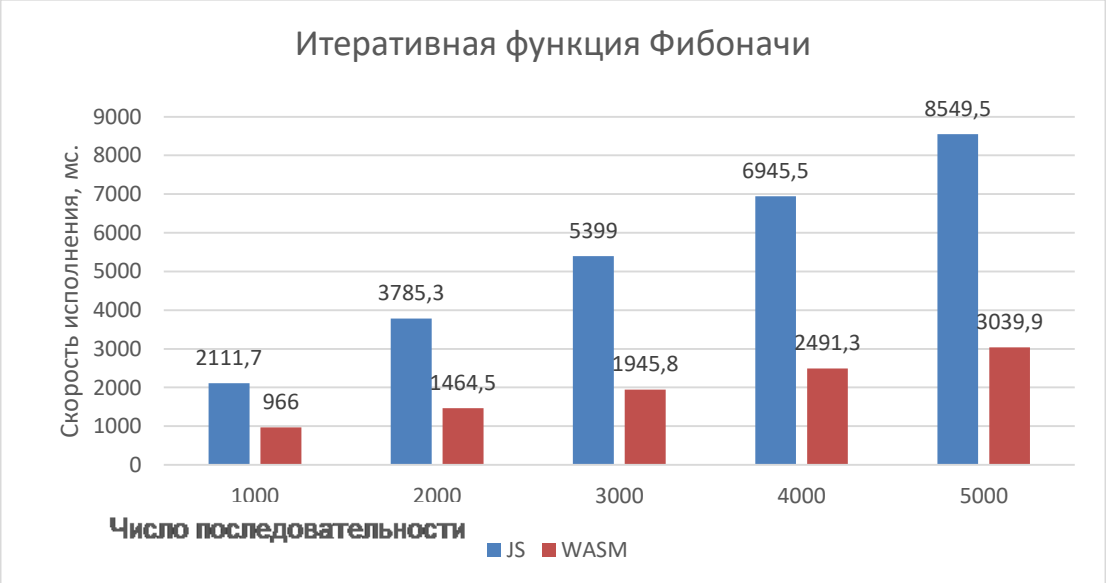


Рисунок 7 – Итеративная функция расчёта n-го числа Фибоначчи, количество запусков 1 000 000, диапазон значений 1000–5000

Таблица 8 – Функция перемножения целых чисел, где n – число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

N	JS, мс.	WASM, мс.	Ускорение
10	42,5	41,2	1,03
25	50,2	47,6	1,05
50	56,8	48,2	1,18
75	72,8	48,6	1,5
100	78	50,5	1,54
125	78,9	55,3	1,43
150	85,1	57,6	1,48

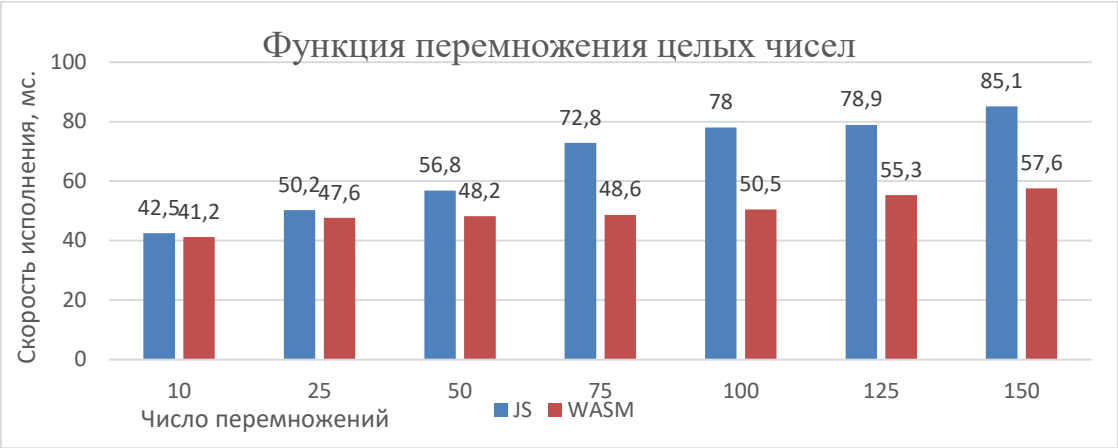


Рисунок 8 – Функция перемножения целых чисел, где n– число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

Таблица 9 – Функция перемножения целых чисел, где n – число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

N	JS, мс.	WASM, мс.	Ускорение
1000	311,5	143,6	2,17
2000	599	255,10	2,34
3000	859,2	344,5	2,5
4000	1115,8	440,7	2,53
5000	1426,0	559,2	2,55
6000	1676,3	651,2	2,57

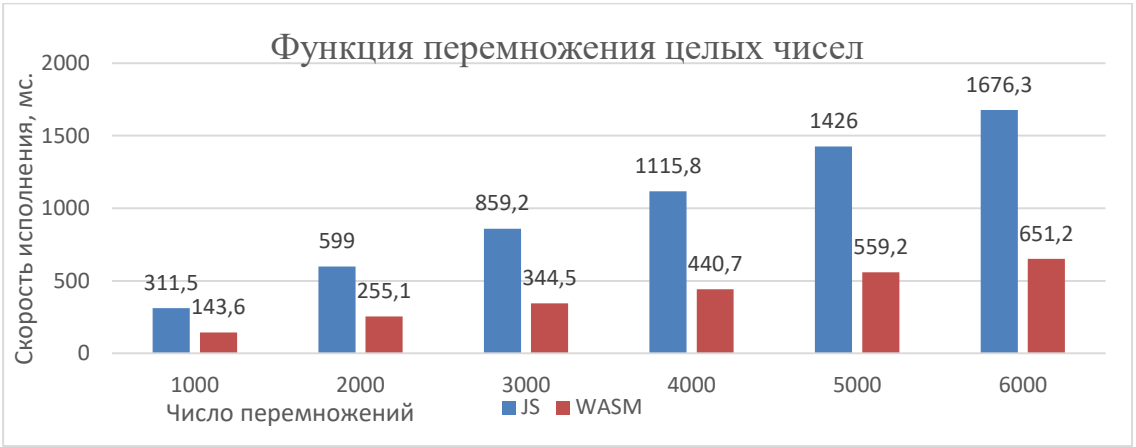


Рисунок 9 – Функция перемножения целых чисел, где n – число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

Таблица 10 – Функция перемножения целых чисел, где n – число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

N	JS, мс.	WASM, мс.	Ускорение
10000	2753,1	1063	2,58
25000	6820,6	2621,5	2,6
50000	13571,2	5105,9	2,65
75000	20556,8	7701,5	2,67
100000	27000,4	10293,6	2,62

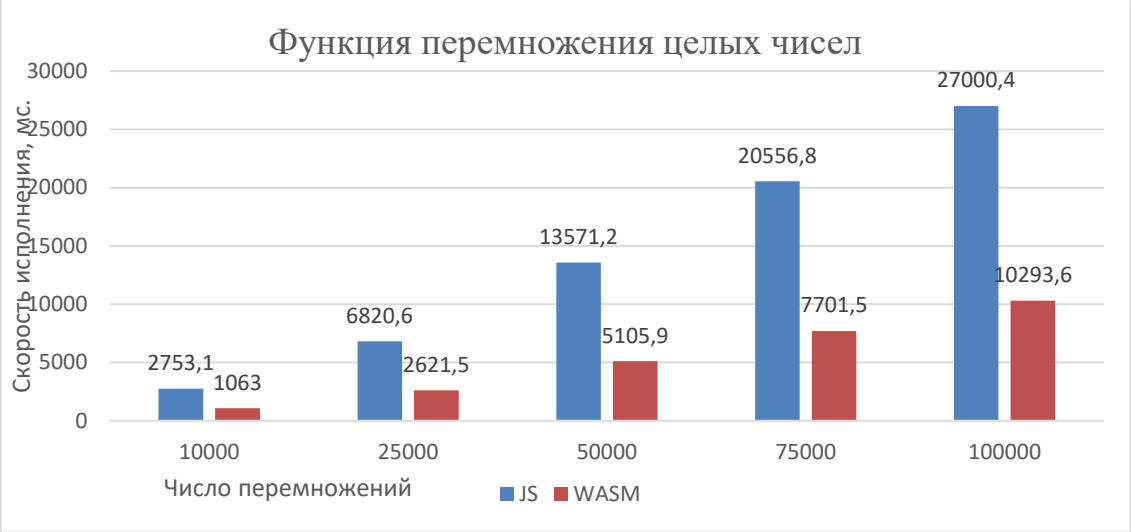


Рисунок 10 – Функция перемножения целых чисел, где n – число перемножений (аргументы: a = 33, b = –4), количество запусков 100 000

Таблица 11 – Функция перемножения векторов размерности N, количество запусков 10

N	JS, мс.	WASM, мс.	Ускорение
250	0,3	0,199	1,5
500	0,7	0,6	1,166
1000	1,4	1,7	0,823
6000	3,7	10,9	0,34
10000	3,2	13,9	0,23
16000	2,2	25,5	0,024

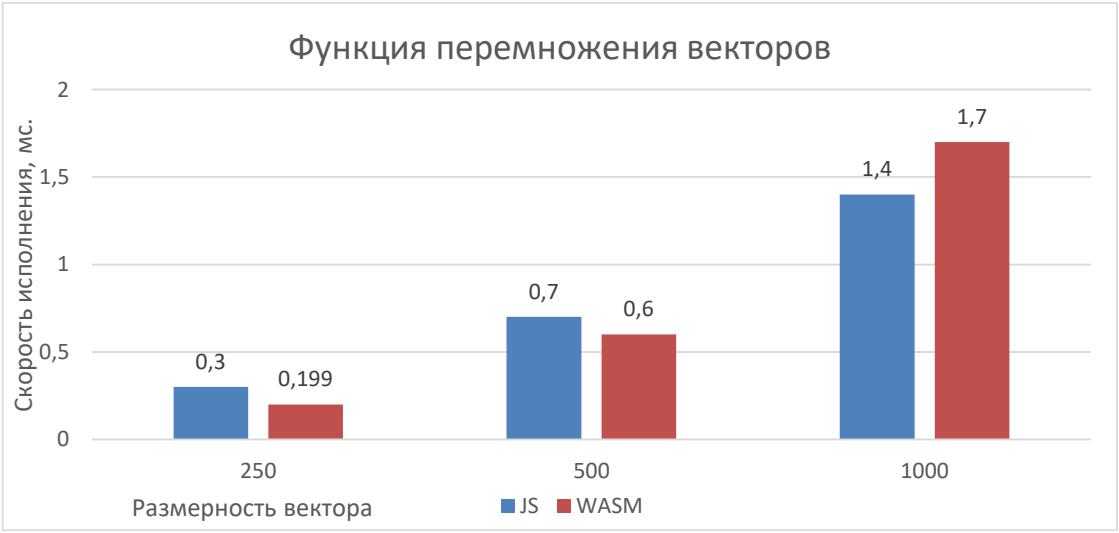


Рисунок 11 – Функция перемножения векторов размерности N, количество запусков 10, размерности 250, 500, 100

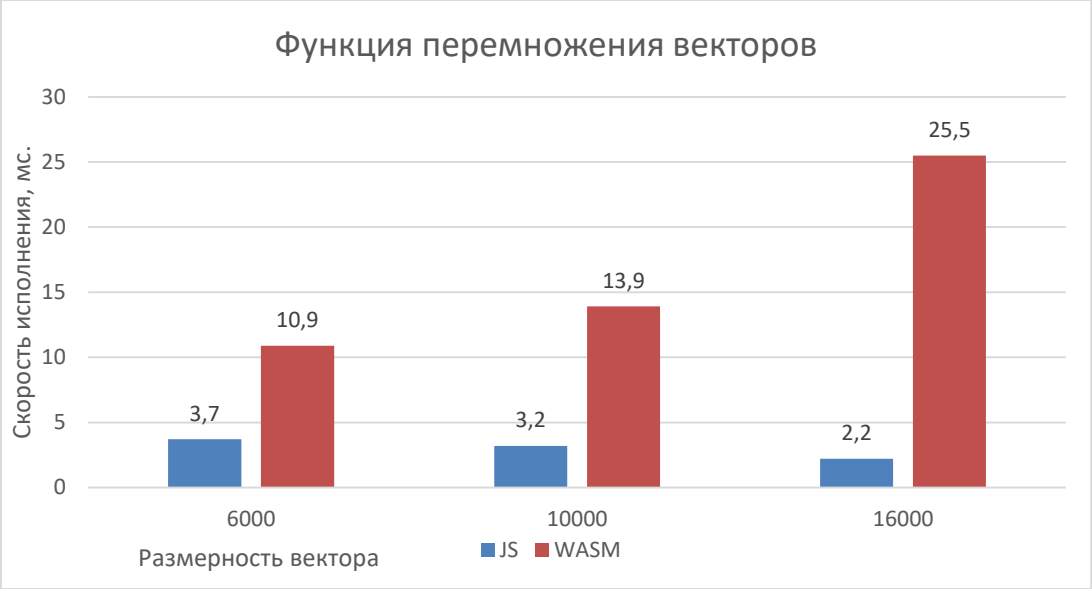


Рисунок 12 – Функция перемножения векторов размерности N, количество запусков 10, размерности 250, 500, 100

Оценим полученные результаты по каждой из протестированных функций.

Рекурсивная функция расчёта n-го числа Фибоначчи. В работе рекурсивной функции WASM прослеживается стабильное ускорение на 25–35 %. В тесте представлен диапазон значений от 40 до 44, однако изменение входных значений функции кардинального влияния на полученное ускорение не оказывает. Таким образом, в случае использования функций с нагруженным стеком вызовов есть смысл отдать предпочтение в сторону использования WASM-версии.

Итеративная функция расчёта n-го числа Фибоначчи. В отличие от рекурсивного варианта, итеративный вариант функции, который сам по себе является гораздо более эффективным, не может гарантировать хотя бы такую же скорость исполнения, как и функция на стандартном JavaScript. Было взято несколько диапазонов входных значений, как те, что были использованы для тестирования рекурсивной функции, так и намного большие. Наблюдалась небольшая тенденция роста ускорения с увеличением входных данных в пределах трёхзначных значений, но так или иначе это все еще хуже исходного JavaScript-варианта без портирования кода. То есть в случае данного теста нет никакой выгоды в использовании WASM.

Функция перемножения целых чисел. Тестирование производилось с одинаковыми аргументами с увеличением количества перемножений. JavaScript показывал лучшие результаты с малым числом перемножений, при 25 перемножений результаты стали одинаковыми, далее с последующим увеличением числа перемножений ускорение в пользу WASM было равно в пределах 10 %.

Функция перемножения векторов. При векторном перемножении WASM показывал хорошие результаты на относительно небольших векторах (длина 250–500). С увеличением длин векторов очень сильно упала производительность и смысл использования технологии пропал вообще.

Полученные результаты дают возможность сделать вывод о том, что использовать WebAssembly не всегда эффективно, а иногда и очень невыгодно с точки зрения производительности. Выделим несколько моментов, влияющих на производительность.

Работа с памятью и вызовы функций. WebAssembly может терять в производительности при многократных вызовах JavaScript-кода. Технология пока ещё теряет в производительности при работе с памятью: обращение выполняет проверку на выход за границы доступного блока памяти.

Также WebAssembly может выигрывать за счёт типа целых переменных. В JS есть только тип Number, фактически это всегда 64-разрядный тип с плавающей точкой и целые числа здесь – это плавающее число без дробной части. При JS-компиляции для целочисленных в движке используется 64-разрядный целый тип [11]. В WASM же предоставляется возможность самостоятельно выбрать разрядность типа, и если мы используем 32-разрядный целый тип, то есть возможность получить преимущество в скорости вычислений, так как операции над 32-разрядными числами немного быстрее, чем над 64-разрядными целыми в JavaScript [10].

Как показала практика, для каждого алгоритма стоит индивидуально определять, будет ли получен прирост скорости с применением WebAssembly. Произведенные опыты позволяют сделать основной вывод, что для обильных вычислений, скорее всего, будет получена более или менее ощутимая выгода.

Выводы. В сравнении с JavaScript, получается, что в среднем WASM быстрее, но при этом возможны индивидуальные ситуации, в которых необходимо разбираться детальнее, потому что

возможен случай получения производительности не только в несколько раз лучше, но так и значительно хуже. Это может зависеть и от используемого браузера, а также его версии.

В каком-то смысле максимально допустимая производительность JS и WASM одинакова, так как оба в итоге преобразуются к низкоуровнему коду процессора. WASM показывает хорошие результаты на объёмных вычислениях, но там, где много операций с памятью WASM, уступает, поскольку основная проблема здесь – это медленный канал общения между JS и WASM.

В июле 2019 года вышла статья «Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code» [12]. Авторами была реализована возможность запуска консольных утилит Linux с использованием WASM, а также ряда тестов для определения производительности в сравнении с этими же тестами, запущенными на asm.js и стандартном коде.

Результаты были получены следующие:

- WASM в среднем на 30 % быстрее, чем JavaScript (на рассмотренных тестах);
- WASM в среднем на 50 % медленнее, чем родной код (на рассмотренных тестах).

Авторы статьи также произвели анализ того, что именно не позволяет WASM работать быстрее:

- примерно вдвое больше операций загрузки/сохранения данных по сравнению с родным кодом;
- больше ветвлений – необходимость дополнительных проверок при обращении к памяти;
- больше «кеш-промахов» уровня L1 [12].

В результате проделанного исследования можно выделить основные причины применения рассмотренной технологии:

- увеличение скорости ряда вычислений;
- использование кода на сторонних языках – портирование (C/C++);
- уменьшение времени загрузки приложения.

Уже сейчас WebAssembly активно применяется в следующем:

- графические редакторы, CAD-системы – например, Figma, AutoCAD;
- эмуляторы, виртуальные машины – например, DOSBox;
- веб-клиенты, активно использующие шифрование;
- кодеки и фильтры для аудио/видео – например, ffmpeg;
- базы данных– например, sqlite;
- игры, игровые движки, движки физики, VR/AR – например, Godot, Doom 3.

В будущем среди новых возможностей технологии ожидаются блоковые операции над памятью, поддержка SIMD-инструкций, поддержка и использование потоков [10].

Так или иначе WASM – это новая развивающаяся технология, которая, как и любая другая поддерживаемая, предоставляет полезную и интересную альтернативу в мире веб-технологий для решения ряда различных задач. Её применение в определённой обстановке может положительно повлиять на скорость выполнения кода в браузере, следовательно, следить за развитием данной технологии и по возможности внедрять достаточно целесообразно.

Библиографический список

1. Yan, Y. Understanding the Performance of WebAssembly Applications / Y. Yan, T. Tu, L. Zhao, Y. Zhou, W. Wang // IMC'21, November 2–4, 2021. – Virtual Event, USA, 2021.

2. Yee, B. Native Client: A Sandbox for Portable, Untrusted x86 Native Code / B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, N. Fullagar // Communications of the ACM. – January 2010. – Vol. 53, № 1. – P. 91–99.

3. Technical Overview // Chrome Developers. – Режим доступа: http://developer.chrome.com/docs/native-client/overview/, свободный. – Заглавие с экрана. – Яз. англ.

4. Van, E. V. A Performant Scheme Interpreter in asm.js / E. V. Van, J. Nicolay, Q. Stiévenart, Th. D'Hondt // Proceedings of the 31st ACM Symposium on Applied Computing, Programming Languages Track (SAC 2016). – 2016. – P. 1941–1951.

5. Resig, J. Asm.js: The JavaScript Compile Target / J. Resig // JavaScript Programming. – Режим доступа: https://john-resig.com/blog/asmjs-javascript-compile-target/, свободный. – Заглавие с экрана. – Яз. англ.

6. Why WebAssembly is Faster Than asm.js // The Web developer blog. – Режим доступа: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js, свободный. – Заглавие с экрана. – Яз. англ.

7. Бородин, О. В. Многопоточная обработка изображений с использованием APIWEB-WORKERS / О. В. Бородин, В. А. Егунов // Прикаспийский журнал: управление и высокие технологии. – 2021. – № 3 (55). – С. 33–46.

8. Haas, A. Bringing the Web up to Speed with WebAssembly / A. Haas, A. Rossberg, D. Schuff, B. Titzer // Communications of the ACM. – December 2018. – Vol. 61, № 12. – P. 107–115.

9. Hilbig, A. An Empirical Study of Real-World WebAssembly Binaries / A. Hilbig, D. Lehmann, M. Pradel // WWW'21 : Proceedings of the Web Conference, April 2021. – 2021. – P. 2696–2708.

10. Знакомство с WebAssembly // Хабр. – Режим доступа: https://habr.com/ru/post/342180/, свободный. – Заглавие с экрана. – Яз. англ.

11. Sletten, Brian. WebAssembly: The Definitive Guide: Safe, Fast, and Portable Code / Sletten Brian. – O'Reilly Media, Inc., 2021. – P. 334.

12. Jangda, Abhinav. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code / Jangda Abhinav, Powers Bobby, Berger Emery D., Guha Arjun // Proceedings of the 2019 USENIX Annual Technical Conference. July 10–12, 2019. – Renton, WA, USA, 2019.

References

1. Yan, Y., Tu, T., Zhao, L., Zhou, Y., Wang, W. Understanding the Performance of WebAssembly Applications. *IMC'21, November 2–4, 2021*. Virtual Event, USA, 2021.

2. Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Communications of the ACM, January 2010*, 2010, vol. 53, no. 1, pp. 91–99.

3. Technical Overview. *Chrome Developers*. Available at: http://developer.chrome.com/docs/native-client/overview/.

4. Van, E. V., Nicolay, J., Stiévenart, Q., D'Hondt, Th. A Performant Scheme Interpreter in asm.js. *Proceedings of the 31st ACM Symposium on Applied Computing, Programming Languages Track (SAC 2016)*, 2016, pp. 1941–1951.

5. Resig, J. Asm.js: The JavaScript Compile Target. *JavaScript Programming*. Available at: https://john-resig.com/blog/asmjs-javascript-compile-target/.

6. Why WebAssembly is Faster Than asm.js. *The Web developer blog.* Available at: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js.

7. Borodin, O. V., Egunov, V. A. Mnogopotochnaya obrabotka izobrazheniy s ispolzovaniem APIWEB-WORKERS [Multithreaded image processing using APIWEB-WORKERS]. *Prikaspiyskiy zhurnal: upravlenie i vysokie tekhnologii* [Caspian Journal: Control and High Technologies], 2021, no. 3 (55), pp. 33–46.

8. Haas, A., Rossberg, A., Schuff, D., Titzer, B. Bringing the Web up to Speed with WebAssembly. *Communications of the ACM, December 2018*, 2018, vol. 61, no. 12, pp. 107–115.

9. Hilbig, A., Lehmann, D., Pradel, M. An Empirical Study of Real-World WebAssembly Binaries. *WWW '21: Proceedings of the Web Conference, April 2021,* 2021, pp. 2696–270810.

10. Znakomstvo s WebAssembly [Getting to know WebAssembly]. *Habr.* Available at: https://habr.com/ru /post/342180/.

11. Sletten, Brian. *WebAssembly: The Definitive Guide: Safe, Fast, and Portable Code*. O'Reilly Media, Inc., 2021, p. 334.

12. Jangda, Abhinav, Powers, Bobby, Berger, Emery D., Guha, Arjun. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. *Proceedings of the 2019 USENIX Annual Technical Conference. July 10–12, 2019*. Renton, WA, USA, 2019.