DOI 10.34185/1562-9945-3-158-2025-19
УДК 004.43: 004.054

О.А. Солуян, Л.А. Люшенко

ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ ВИКОРИСТАННЯ WEBASSEMBLY ДЛЯ РОЗРОБКИ ВИСОКОПРОДУКТИВНОГО КОДУ У ВЕБДОДАТКАХ

Анотація: Однією з основних проблем інтерпретованих мов, таких як JavaScript, є їх обмежена швидкодія у порівнянні з компільованими мовами. Інтерпретований код виконується повільніше, що може спричиняти затримки у роботі застосунків, особливо у випадках обробки великих обсягів даних чи виконання складних обчислень. Для подолання цієї проблеми використовуються різноманітні методи оптимізації JavaScript коду. Одним із найбільш перспективних рішень є технологія WebAssembly. WebAssembly дозволяє виконувати код із продуктивністю, наближеною до швидкодії низькорівневих мов, таких як C, C++ або Rust. Серед головних переваг WebAssembly є можливість компіляції коду з різних мов програмування у швидкий і компактний формат, який виконується безпосередньо у браузері. Це значно покращує продуктивність вебзастосунків, забезпечуючи користувачів більш плавним та швидким досвідом взаємодії. Проте потенціал WebAssembly ще не повністю розкритий. Існує необхідність у подальшому розвитку та вдосконаленні методів ефективного кодування, що компілюється у WebAssembly, з метою досягнення максимальної продуктивності сучасних вебзастосунків.
Ключові слова: JavaScript, WebAssembly, оптимізація, продуктивність, вебзастосунки.

Постановка проблеми. У сучасному світі, де зростають вимоги до швидкодії та ефективності вебзастосунків, проблема забезпечення високопродуктивного виконання коду набуває особливої актуальності. Традиційні програмні засоби такі, як JavaScript, хоча й добре адаптовані для розробки інтерфейсів користувача, стикаються з обмеженнями при виконанні складних обчислювальних задач, що призводить до зниження загальної продуктивності застосунків, особливо при роботі з великими обсягами даних чи інтенсивними алгоритмічними операціями. В цьому контексті застосування WebAssembly є перспективним рішенням, яке обіцяє майже нативну швидкість виконання коду завдяки компіляції в низькорівневий байткод, що може виконуватися безпосередньо у браузері. Проте, незважаючи на очевидні переваги, використання WebAssembly для розробки високопродуктивного коду в вебзастосунках стикається з рядом викликів, а саме: необхідно забезпечити ефективну інтеграцію з існуючими вебтехнологіями, вирішити питання сумісності між різними платформами та гарантувати безпеку виконання коду в умовах інтернет середовища. Окрім того дослідження в області оптимізації управління пам'яттю та енергоспоживання, а також аналізуванню

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

можливостей комбінування WebAssembly з іншими мовами програмування, є критично важливими для повноцінного розкриття потенціалу цієї технології. Таким чином, необхідність комплексного дослідження та розробки нових методологій інтеграції WebAssembly у веброзробку є актуальною задачею, вирішення якої дозволить забезпечити створення продуктивних щвидкодіючих вебзастосунків з мінімізацією енерго та комп'ютерних ресурсів.

Аналіз останніх досліджень і публікацій. В останні роки було проведено безліч досліджень у галузі оптимізації вебзастосунків. Серед методів, що сприяють підвищенню продуктивності вебзастосунків, можна виділити мінімізацію розміру JavaScript-файлів, оптимізацію запитів до серверу (зокрема, через використання GraphQL), використання кешування, застосування прогресивних вебзастосунків (PWA), а також впровадження асинхронного завантаження компонентів. WebAssembly як метод оптимізації з'явився відносно нещодавно (2017 рік), і кількість наукових публікацій про неї на даний момент не дуже велика.

У роботі "WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations" [1] автори досліджують продуктивність WebAssembly-модулів, згенерованих із програм, написаних на C++ та Rust. Метою є порівняння ефективності виконання завдань, таких як сортування та множення матриць. Результати показали, що модулі, створені з коду на Rust, демонструють вищу швидкість та ефективність порівняно з модулями на C++.

У статті "WebAssembly versus JavaScript: Energy and Runtime Performance" [2] дослідники проводять систематичне вивчення енергоспоживання та часу виконання WebAssembly у порівнянні з JavaScript. Використовуючи мікро-бенчмарки та реальні застосунки, вони виявили, що WebAssembly перевершує JavaScript за швидкістю та енергоефективністю, хоча різниця залежить від конкретних задач та браузерів.

У дослідженні "Empowering Web Applications with WebAssembly: Are We There Yet?" [3] аналізується, як браузерні рушії оптимізують виконання WebAssembly у порівнянні з JavaScript. Результати показують, що WebAssembly може бути швидшим за JavaScript для невеликих обсягів даних, але при збільшенні розміру вхідних даних JavaScript може перевершувати WebAssembly через агресивніші JIT-оптимізації. Крім того, WebAssembly споживає більше пам'яті через свою лінійну модель пам'яті.

У статті "Bringing the Web Up to Speed with WebAssembly" [4] представлено архітектуру та можливості WebAssembly як портативного низькорівневого байт-коду, що забезпечує компактне представлення, ефективну валідацію та компіляцію, а також безпечне виконання з низькими накладними витратами. Автори підкреслюють потенціал WebAssembly для значного підвищення продуктивності веб-застосунків.

Таким чином, незважаючи на відносно невелику кількість наукових публікацій на тему WebAssembly, проведені дослідження показують значний потенціал цієї технології для підвищення продуктивності вебзастосунків. Однак, вибір мови програмування, з якої створюється WebAssembly-модуль, а також тип задачі можуть суттєво впливати на кінцеві показники продуктивності.

Мета дослідження - визначення ключових можливостей інтеграції WebAssembly при створенні високопродуктивних вебзастосунків для виявлення переваг застосування WebAssembly та визначити стратегії використання цієї технології в сучасній веброзробці. Визначається продуктивність WebAssembly-коду в порівнянні з аналогічними реалізаціями на JavaScript із застосуванням компілятора Cheerp [5].

Виклад основного змісту. Особливості організації пам'яті та способи її використання мають безпосередній вплив на продуктивність програм. WebAssembly та JavaScript виконуються у середовищі рушіїв JavaScript, проте їх підходи до виконання коду та управління пам'яттю суттєво різняться [6].

Віртуальна машина WebAssembly оперує кількома регіонами пам'яті (рис.1). Пам'ять керованого коду містить безпосередньо код програми WebAssembly та доступна виключно віртуальній машині. В зв'язку з цим WebAssembly код не може її зчитувати чи змінювати. Керований стек викликів зберігає адреси повернення, представлені типом i32, який також використовується для позначення вказівників та адрес у пам'яті. Він забезпечує відстеження активних викликів функцій і захищає від атак, що базуються на підміні адрес повернення. Керований стек оцінки використовується для передачі параметрів інструкціям і збереження їх результатів. Він може містити чотири базові типи WebAssembly — i32, i64, f32 та f64, що відповідають цілим числам і числам з плаваючою крапкою, закодованим на 32 або 64 біти [7].

Лінійна пам'ять призначена для зберігання нескалярних типів даних, таких як рядки, масиви чи списки. Вона являє собою єдиний неперервний сегмент, у якому відсутні поняття прав доступу, тому усі дані можна як читати, так і записувати. Крім того, лінійна пам'ять не використовує механізми рандомізації адрес, як-от ASLR або PIE, які підтримуються всіма основними операційними системами. Управління цією пам'яттю покладається на програму, але у більшості мов програмування і відповідних компіляторів структура організації пам'яті аналогічна нативним виконуваним файлам і включає стек, купу та зону для статичних або заздалегідь визначених значень. Ці зони зберігають основну частину даних, розподілених відповідно до вихідного коду та використовуваного компілятора.

Локальні та глобальні змінні WebAssembly є додатковим механізмом керування пам'яттю. Як і стек оцінки, вони обмежені чотирма базовими типами. Глобальні змінні доступні у межах усього модуля, а локальні — лише в межах виконуваної функції. Значення цих змінних обробляються за допомогою спеціальних інструкцій і зберігаються у визначеній таблиці, недоступній з лінійної пам'яті. Водночас варто зазначити, що сучасні інструментарії зазвичай не відображають локальні та глобальні змінні з мов програмування безпосередньо на змінні WebAssembly.

Компілятори WebAssembly використовують лінійну пам'ять для створення схеми розташування пам'яті, що включає три зони, а саме: стек, купу та зону для статичних даних. Ці зони можуть бути організовані різними способами, і на практиці різні компілятори приймають різні рішення, що призводить до різних схем. У цій статті буде

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

зосереджено увагу на двох варіантах, доступних у LLVM [8] toolchain: «stack-first» та «no-stack-first» (рис. 1).

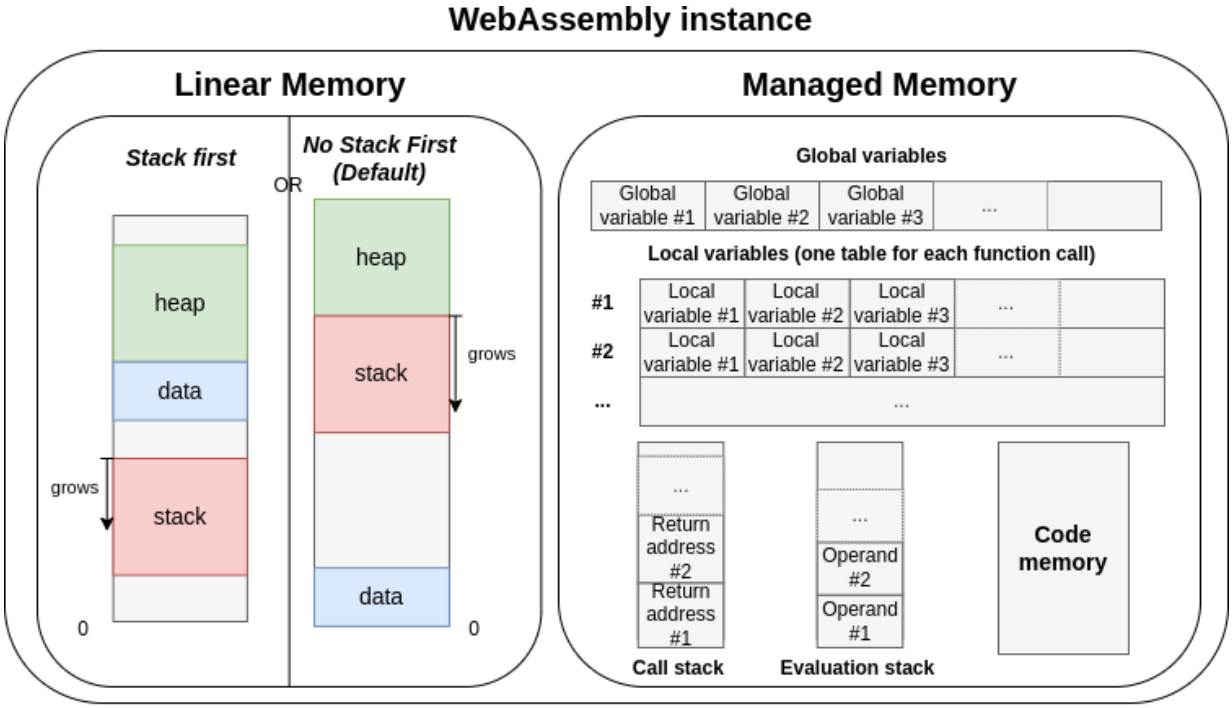

Рисунок 1 - Макет пам'яті віртуальної машини WebAssembly

За замовчуванням LLVM використовує схему «no-stack-first», де зона даних фіксованого розміру розташована за найнижчими адресами. Далі йде стек, який зростає у напрямку до менших адрес, а купа — у напрямку до більших. Відсутність чіткого розмежування пам’яті в WebAssembly означає, що переповнення стека (тобто ситуація, коли його розширення доходить до зіткнення з іншою зоною) може безслідно пошкодити дані у зоні даних. Через цей недолік розробники Rust запропонували схему «stack-first», де стек розташований на нижчих адресах, а зона даних і купа — на вищих. Такий підхід призводить до аварійного завершення роботи WebAssembly runtime у разі переповнення стека, оскільки стек буде зростати до недопустимих (негативних) адрес, не перезаписуючи інші дані, що дає змогу однозначно виявити переповнення та уникнути непередбачуваної поведінки. Станом на сьогодні він був прийнятий за замовчуванням у Rust, у Zig, а в LLVM обговорюють можливість зробити його стандартним.

Однак, для досягнення високої продуктивності недостатньо лише враховувати модель пам’яті. Не менш важливим є аналіз операцій передачі даних між JavaScript і WebAssembly, що супроводжується серіалізацією та десеріалізацією складних структур.

Загалом, правильне вимірювання продуктивності WebAssembly має охоплювати не лише загальний час виконання алгоритму, а й такі аспекти, як ініціалізація WebAssembly, передача даних між JavaScript і WebAssembly, а також витрати часу на перемикання контексту виконання. WebAssembly алгоритм має наступні фази виконання:

1. запуск WebAssembly (час від ініціалізації модуля);
2. вхід у функцію з JavaScript;
3. десеріалізація параметрів;
4. виконання алгоритму;
5. серіалізація результатів;
6. повернення до контексту JavaScript.

WebAssembly у своїй основі підтримує лише чотири примітивні типи (i32, i64, f32 та f64). Це означає, що складні типи, такі як рядки, масиви, об'єкти або структуровані дані, не є частиною його внутрішнього представлення. Робота з такими складними структурами залежить від механізмів серіалізації та десеріалізації, які використовують компілятори мов програмування, що транслюються в WebAssembly. Так у C/C++ зазвичай працюють з покажчиками та виділенням пам’яті, використовуючи malloc та free, а самі складні структури передаються через буфери в пам’яті WebAssembly. Аналогічно в Rust управління пам’яттю відбувається через систему володіння (ownership), а при компіляції в WebAssembly використовується певний механізм для взаємодії з JavaScript, наприклад wasm-bindgen, який може автоматично перетворювати Rust-об'єкти в серіалізовані формати [9].

У JavaScript взаємодія з WebAssembly зазвичай відбувається через WebAssembly.Memory, де велика область пам’яті використовується як масив байтів (ArrayBuffer). Складні структури перетворюються у відповідні масиви байтів перед передачею в WebAssembly, а зворотне перетворення здійснюється при зчитуванні даних. Для цього використовуються бібліотеки та компіляторні утиліти, такі як AssemblyScript [10], який адаптує TypeScript для WebAssembly, або Emscripten для C/C++.

Таким чином, ефективність роботи зі складними структурами у WebAssembly значною мірою залежить від обраної мови програмування та її механізмів обробки пам’яті. Оптимізація серіалізації та десеріалізації, а також мінімізація копіювання даних є ключовими аспектами для підвищення продуктивності.

Це має суттєве значення при дослідженні швидкодії WebAssembly. Адже сама лише серіалізація складних JavaScript-об’єктів в байтовий ArrayBuffer і зворотня десеріалізація можуть зайняти більше часу ніж виконання самого алгоритму у wasm-фукнції. На лістингах 1 і 2 наведені вбудовані функції компілятора AssemblyScript, які десеріалізують(опускають) складні структури данних до зрозумілого для WebAssembly формату та серіалізують їх назад після завершення виконнання функції у контекст JavaScript.

Лістинг 1 - Фукнція __lowerArray

```
function __liftArray(liftElement, align, pointer) {
    if (!pointer) return null;
    const dataStart = __getU32(pointer + 4),
        length = __dataview.getUint32(pointer + 12, true),
        values = new Array(length);
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
        for (let i = 0; i < length; ++i) values[i] = liftEle-
ment(dataStart + ((i << align) >>> 0));
    return values;}
```

Лістинг 2 - Фукнція __liftArray

```
    function __lowerArray(lowerElement, id, align, values) {
        if (values == null) return 0;
        const length = values.length,
            buffer = exports.__pin(exports.__new(length << align,
1)) >>> 0,
            header = exports.__pin(exports.__new(16, id)) >>> 0;
        __setU32(header + 0, buffer);
        __dataview.setUint32(header + 4, buffer, true);
        __dataview.setUint32(header + 8, length << align, true);
__dataview.setUint32(header + 12, length, true);
        for (let i = 0; i < length; ++i) lowerElement(buffer + ((i
<< align) >>> 0), values[i]);
        exports.__unpin(buffer); exports.__unpin(header);
        return header;}
```

Коли wasm-функція потребує перетворення великої кількості даних, представлених в javascript-об'єктах, в зрозумілий формат byte-масиву, wasm буде як правило показувати гірший перформанс ніж JavaScript. Це свідчить про те, що відповідним варіантом використання WebAssembly є випадки, коли вхідні дані прості, передача даних обмежена, але вони використовуються для складних обчислень, які дають багато вихідних результатів.

Враховуючи це, основну увагу даного дослідження зосереджено на аналізі продуктивності WebAssembly у порівнянні з JavaScript в умовах ізольованого виконання алгоритмів без передачі контексту.

Гіпотеза дослідження полягає у припущенні, що WebAssembly здатен забезпечити вищу продуктивність у виконанні обчислювально-інтенсивних алгоритмів у порівнянні з JavaScript за рахунок попередньої компіляції у байткод та ефективнішого управління пам'яттю. Очікується, що саме у сценаріях із обмеженою передачею складних структур даних між JavaScript і WebAssembly, останній продемонструє значну перевагу в швидкодії.

У якості тестових прикладів обрано алгоритми лінійної алгебри з набору бенчмарків PolyBenchC [11]. Це дозволяє провести об'єктивне порівняння сирої обчислювальної продуктивності обох технологій без впливу зовнішніх факторів. У таблиці 1 представлено 20 програм на C з набору бенчмарків PolyBenchC.

Таблиця 1

Бенчмарки PolyBenchC

№	Бенчмарк	Опис
1	2mm	Two matrix multiplications
2	3mm	Three matrix multiplications
3	atax	Matrix transpose vector multiplication
4	bicg	BiConjugate gradient
5	cholesky	Matrix decomposition
6	deriche	Smoothing filter (Deriche)
7	durbin	Toeplitz systems solver (Durbin's algorithm)
8	floyd-warshall	All-pairs shortest paths
9	gemm	General Matrix Multiplication
10	gemver	Multiple Matrix-Vector Multiplication
11	gesummv	Summed Matrix-Vector Multiplication
12	gramschmidt	Orthogonalization (Gram-Schmidt process)
13	lu	LU decomposition
14	ludcmp	LU decomposition (compact)
15	mvt	Matrix vector multiplication
16	symm	Symmetric matrix multiplication
17	syr2k	Symmetric rank-2k update
18	syrk	Symmetric rank-k update
19	trisolv	Triangular matrix solver
20	trmm	Triangular matrix multiplication

На рис.2 наведено процедуру вимірювання продуктивності WebAssembly та JavaScript, яка складається з чотирьох етапів:

1. Трансформація вихідного коду.
2. Компіляція у Wasm/JS.
3. Розгортання.
4. Збір даних.

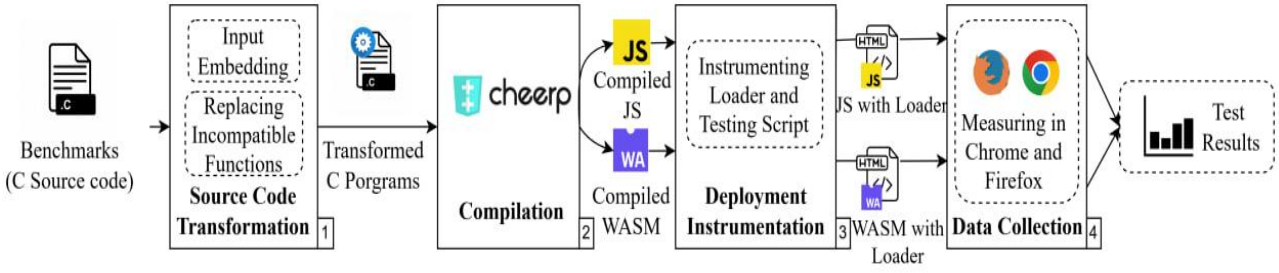

Рисунок 2 - Вимірювання продуктивності WebAssembly та JavaScript

Оскільки деякі реалізації алгоритмів мали помилки компіляцій, виправляємо їх на першому етапі, застосовуючи трансформацію вихідного коду. Трансформація вихідного коду передбачає заміну несумісних конструкцій мови С, які не підтримуються

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Cheerp. На другому етапі компілюємо 20 програм на C за допомогою Cheerp для генерації WebAssembly та JavaScript. Попередньо до компіляції додаємо код для вимірювання часу виконання алгоритмів. На третьому етапі створюємо мінімальну HTML-сторінку для завантаження програм на WebAssembly/JavaScript. На четвертому етапі запускаємо згенеровані програми WebAssembly/JavaScript на HTML-сторінках і збираємо дані про час виконання та використання пам'яті за допомогою вбудованих інструментів розробника у браузерах. В таблиці 2 наведені результати проведеного дослідження.

Таблиця 2

Часові показники виконання WebAssembly і JavaScript

Алгоритм	Розмір датасету	JavaScript Сер. час (с)	WebAssembly Сер. час (с)	Performance Ratio (JS/WASM)
3mm	SMALL	0.016500	0.003060	5.392157
3mm	MEDIUM	0.056000	0.051440	1.088647
3mm	LARGE	5.266580	8.851160	0.595016
atax	SMALL	0.003960	0.000640	6.187500
atax	MEDIUM	0.008160	0.002780	2.935252
atax	LARGE	0.047140	0.027160	1.735641
bicg	SMALL	0.005160	0.000820	6.292683
bicg	MEDIUM	0.009360	0.002380	3.932773
bicg	LARGE	0.049800	0.024320	2.047697
cholesky	SMALL	0.011280	0.006940	1.625360
cholesky	MEDIUM	0.088400	0.132020	0.669596
cholesky	LARGE	14.906300	21.574020	0.690938
deriche	SMALL	0.024220	0.001520	15.934211
deriche	MEDIUM	0.069100	0.013540	5.103397
deriche	LARGE	0.901520	0.273100	3.301062
durbin	SMALL	0.001900	0.000580	3.275862
durbin	MEDIUM	0.002820	0.000880	3.204545
durbin	LARGE	0.011560	0.013400	0.862687
floyd	SMALL	0.022100	0.013540	1.632201
floyd	MEDIUM	0.282500	0.219700	1.285844
floyd	LARGE	46.080300	38.256420	1.204512
gemm	SMALL	0.005780	0.001580	3.658228
gemm	MEDIUM	0.025000	0.018420	1.357220
gemm	LARGE	1.300060	2.171760	0.598620
gemver	SMALL	0.005220	0.000740	7.054054
gemver	MEDIUM	0.012180	0.002700	4.511111
gemver	LARGE	0.065880	0.043280	1.522181
gesummv	SMALL	0.003480	0.000700	4.971429
gesummv	MEDIUM	0.006260	0.001100	5.690909

gesummv	LARGE	0.045220	0.019140	2.362591
gramschmidt	SMALL	0.008900	0.001180	7.542373
gramschmidt	MEDIUM	0.025680	0.027900	0.920430
gramschmidt	LARGE	3.681840	4.257600	0.864769
lu	SMALL	0.012500	0.006600	1.893939
lu	MEDIUM	0.099860	0.151680	0.658360
lu	LARGE	18.965720	27.973800	0.677982
ludcmp	SMALL	0.012920	0.006280	2.057325
ludcmp	MEDIUM	0.111520	0.143000	0.779860
ludcmp	LARGE	19.636700	24.696860	0.795109
mvt	SMALL	0.005020	0.000620	8.096774

На рис. 3 наведено результати експериментів, у яких WebAssembly продемонстрував кращу швидкодію на певних алгоритмах з визначеним розміром вхідних даних. У свою чергу, на рис. 4 показані експерименти, де JavaScript показав себе краще на інших алгоритмах з іншими розмірами вхідних даних.

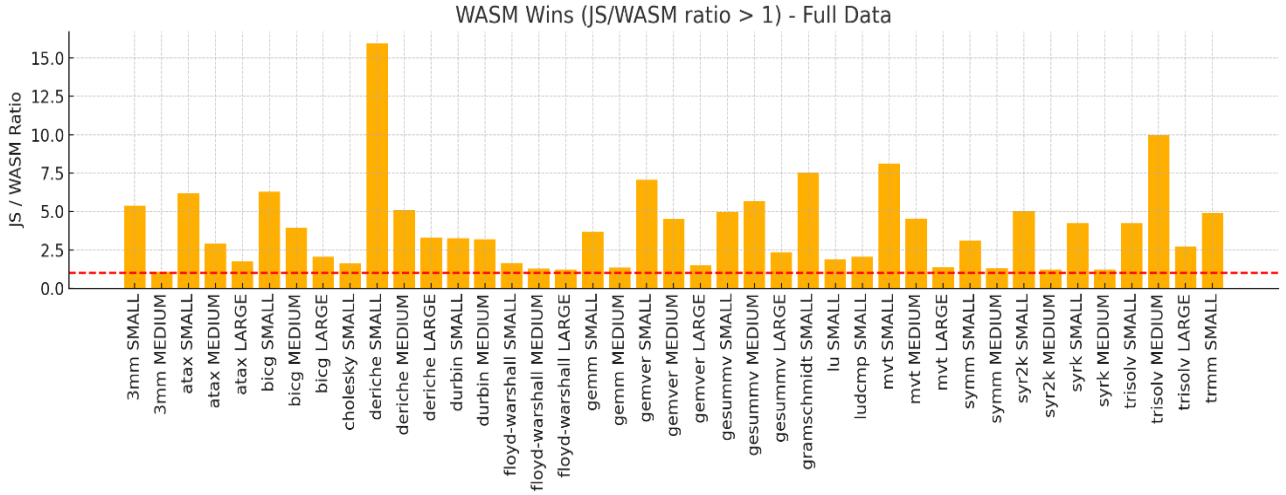

Рисунок 3 - Експерименти, де WebAssembly швидше за JS

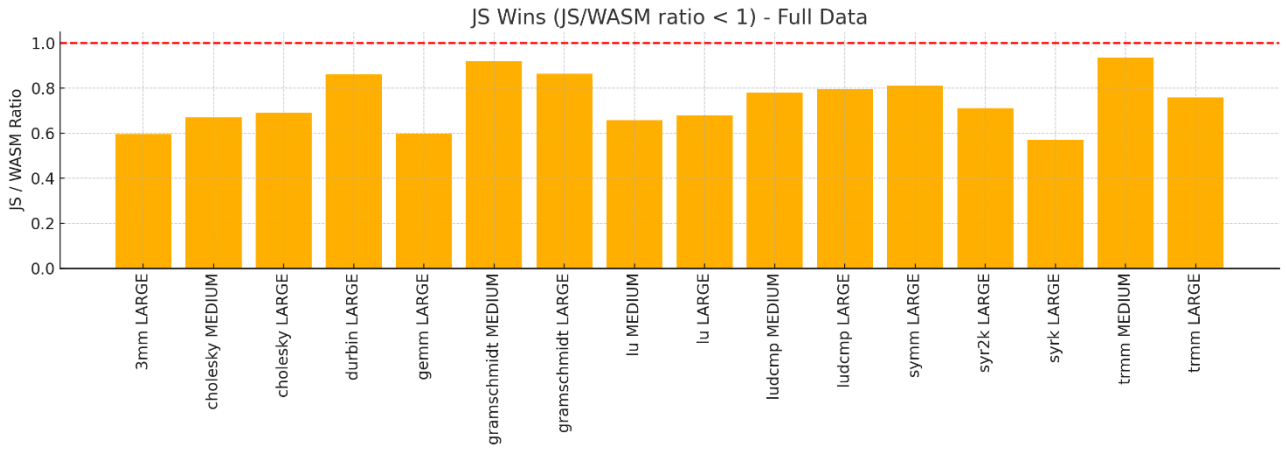

Рисунок 4 - Швидкодія JavaScript вища за WebAssembly

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Таким чином, дослідження продуктивності WebAssembly і JavaScript дозволило отримати низку важливих результатів. По-перше, WebAssembly продемонстрував істотну перевагу у виконанні обчислювально інтенсивних алгоритмів на малих та середніх обсягах даних. У багатьох випадках фіксувалося прискорення в 5–15 разів порівняно з JavaScript. Найвищу продуктивність WebAssembly показав у задачах, що пов'язані з лінійною алгеброю, фільтрацією сигналів та симуляціями — зокрема, у таких алгоритмах як deriche, atax, trisolv, gemm, bicg.

По-друге, у випадку великих наборів даних спостерігалася змішана динаміка: частина алгоритмів (cholesky, lu, syr2k, trmm) працювала швидше у JavaScript або з мінімальною різницею. Це пояснюється тим, що великі об'єми даних збільшують витрати на серіалізацію/десеріалізацію, ініціалізацію пам'яті та завантаження WebAssembly-модулів, тоді як рушії JavaScript (наприклад, V8) активно оптимізують довготривалі обчислення. У таких сценаріях традиційний JavaScript, завдяки прямому доступу до об'єктної моделі та оптимізації рушіїв, може виявитися більш продуктивним.

Таким чином, результати підтверджують, що WebAssembly є найбільш доцільним у контекстах, де:

- структура даних є простою;
- обмін між WebAssembly та JavaScript мінімальний;
- обчислення зосереджені у самій wasm-функції;
- критично важлива продуктивність.

На основі проведеного дослідження доцільно рекомендувати використання WebAssembly у вебзастосунках, що потребують високої обчислювальної продуктивності за умов мінімального обміну даними з JavaScript. Найефективніше ця технологія проявляє себе у виконанні математичних та логічних обчислень, зокрема в алгоритмах лінійної алгебри, фільтрації, обробці сигналів, симуляціях, коли структура вхідних даних є простою, а сама обробка зосереджена всередині WebAssembly-модуля. У таких випадках WebAssembly дозволяє досягти значного прискорення порівняно з JavaScript, особливо на малих і середніх наборах даних.

Тому доцільним є комбінований підхід: обчислювальні «гарячі точки» реалізовуються за допомогою WebAssembly, а загальна логіка застосунку, обробка подій і взаємодія з DOM залишаються у JavaScript. Крім того, важливо компілювати WebAssembly-модулі з максимальними оптимізаціями (-O3, -Ofast) і обов'язково проводити тестування кожного конкретного випадку, щоб обґрунтовано оцінити доцільність застосування цієї технології.

Висновки. JavaScript залишається основною мовою веброзробки завдяки своїй універсальності, інтеграції з DOM та потужним рушіям виконання. Однак його інтерпретована природа створює обмеження при вирішенні задач, які потребують високої продуктивності.

WebAssembly, як компільована альтернатива, надає суттєві переваги у швидкодії, особливо для обчислювально важких задач, що реалізуються мовами типу C/C++ або

Rust. Його можливості забезпечують продуктивність, близьку до нативної, у браузерному середовищі, при цьому зберігаючи платформну незалежність.

Тема дослідження є актуальною у зв'язку з постійним зростанням потреб у високопродуктивних вебдодатках, зокрема у галузях:

- фінансового моделювання;
- машинного навчання в браузері;
- візуалізації великих даних;
- ігрової індустрії.

Проведений аналіз підтверджує, що WebAssembly є ефективним інструментом для покращення швидкодії вебдодатків. Водночас його використання повинне базуватись на точному розумінні архітектури застосунку, характеру задач і обсягу даних, які передаються. У майбутньому очікується подальше вдосконалення засобів інтеграції WebAssembly у вебсередовище, що ще більше розширить можливості його практичного застосування.

ЛІТЕРАТУРА

1. Rishi Kiran Aiyatham Prabakar. WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations. *Blekinge Institute of Technology, bachelor's thesis work.* URL: https://www.diva-portal.org/smash/get/diva2:1879948/FULLTEXT01.pdf (дата звернення 05.01.2025).

2. Joao De Macedo, Rui Abreu, Rui Pereira, Joao Saraiva. WebAssembly versus JavaScript: Energy and Runtime Performance. *2022 International Conference on ICT for Sustainability (ICT4S).* URL: https://repositorio.inesctec.pt/server/api/core/bitstreams/0870fb76-d463-456b-9e34-5b33bb7c0dd1/content (дата звернення 15.01.2025).

3. Weihang Wang. Empowering Web Applications with WebAssembly: Are We There Yet? *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).*
*URL:*https://par.nsf.gov/servlets/purl/10312863https://dl.acm.org/doi/pdf/10.1145/3062341.3062363 (дата звернення 04.01.2025).

4. Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer. Bringing the Web up to Speed with WebAssembly. *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. URL: https://dl.acm.org/doi/pdf/10.1145/3062341.3062363 (дата звернення 20.01.2025).

5. Cheerp Documentation Overview. *Documentation.* URL: https://cheerp.io/docs/overview (дата звернення 19.02.2025).

6. WebAssembly Developer Documentation. *Documentation.* URL: https://webassembly.org/ (дата звернення 02.02.2025).

7. Securing Stack Smashing Protection in WebAssembly Applications. *Preprint publication.* URL:
https://www.researchgate.net/publication/385177717_Securing_Stack_Smashing_Protection_in_WebAssembly_Applications (дата звернення 05.02.2025).

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

8. LLVM Reference Manual. *Documentation.*URL: https://llvm.org/docs/CodeGenerator.html (дата звернення 15.02.2025).
9. Jacob Nilsson, Andreas Trattner. Analyzing front-end performance using Webassembly. *Lund University, master's thesis work.*
URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9094683&fileOId=9094684 (дата звернення 07.02.2025).
10. AssemblyScript Runtime Variants. *The AssemblyScript Book.*
URL: https://www.assemblyscript.org/runtime.html (дата звернення 01.03.2025).
11. PolyBench/C -- Homepage. *PolyBench/C the Polyhedral Benchmark suite.* URL: https://www.cs.colostate.edu/~pouchet/software/polybench/ (дата звернення 10.03.2025).

REFERENCES

1. Rishi Kiran Aiyatham Prabakar. WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations. *www.diva-portal.org.* Retrieved from https://www.diva-portal.org/smash/get/diva2:1879948/FULLTEXT01.pdf.
2. Joao De Macedo, Rui Abreu, Rui Pereira, Joao Saraiva. WebAssembly versus JavaScript: Energy and Runtime Performance. *repositorio.inesctec.pt*. Retrieved from https://repositorio.inesctec.pt/server/api/core/bitstreams/0870fb76-d463-456b-9e34-5b33bb7c0dd1/content.
3. Weihang Wang. Empowering Web Applications with WebAssembly: Are We There Yet? *par.nsf.gov*. Retrieved from
https://par.nsf.gov/servlets/purl/10312863https://dl.acm.org/doi/pdf/10.1145/3062341.3062363.
4. Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer. Bringing the Web up to Speed with WebAssembly. *dl.acm.org*. Retrieved from
https://dl.acm.org/doi/pdf/10.1145/3062341.3062363.
5. Cheerp Documentation Overview. *cheerp.io.* Retrieved from
https://cheerp.io/docs/overview.
6. WebAssembly. *webassembly.org.* Retrieved from https://webassembly.org/
7. Securing Stack Smashing Protection in WebAssembly Applications. *www.researchgate.net.* Retrieved from
8. https://www.researchgate.net/publication/385177717_Securing_Stack_Smashing_Protection_in_WebAssembly_Applications
9. LLVM Reference Manual. *llvm.org.* Retrieved from
https://llvm.org/docs/CodeGenerator.html
10. Jacob Nilsson, Andreas Trattner. Analyzing front-end performance using Webassembly. *lup.lub.lu.se.* Retrieved
from https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9094683&fileOId=9094684
11. AssemblyScript Runtime Variants. *www.assemblyscript.org.* Retrieved from https://www.assemblyscript.org/runtime.html#variants
PolyBenchC: the polyhedral benchmark suite. *web.cs.ucla.edu*. Retrieved from http://web.cs.ucla.edu/~pouchet/software/polybench/

Received 22.04.2025.
Accepted 25.04.2025.

Research of WebAssembly usage for high-performance code development in web applications

The paper examines the potential of using WebAssembly in web applications to ensure high performance. It is shown that WebAssembly technology allows achieving nearly native speed compared to JavaScript due to bytecode compilation. The performance of WebAssembly is analyzed using linear algebra algorithms with PolyBenchC benchmarks. The research results demonstrated that WebAssembly shows advantages in executing computationally intensive algorithms with small and medium data sizes. The main factors affecting WebAssembly's efficiency when processing large data volumes are identified.

Люшенко Леся Анатоліївна – кандидат технічних наук, доцент кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського".
Солуян Олександр Анатолійович – магістр кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського".

Liushenko Lesia – candidate of technical sciences, associate professor of the Department of Software for Computer Systems, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute".
Soluian Oleksandr – master of the Department of Software for Computer Systems, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)