Інформаційні технології

УДК 004.051

Ковальов Євген Валерійович

студент

Харківського національного університету радіоелектроніки

Лєсна Наталя Советівна

кандидат технічних наук

професор кафедри програмної інженерії

Харківський національний університет радіоелектроніки

ПОБУДОВА ШВИДКОДІЮЧИХ КЛІЄНТСЬКИХ ВЕБ-ЗАСТОСУВАНЬ ЗА ДОПОМОГОЮ СУЧАСНИХ ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

Анотація. *У роботі були розглянуті існуючі проблеми продуктивності веб-додатків, розглянуті засоби їх усунення через використання засобів, що зменшують час компіляції та виконання мови програмування JavaScript*

Ключові слова: *продуктивність, веб-додаток, JavaScript, WebAssembly, ReactJS, Fiber, V8.*

Проблема і актуальність дослідження. Актуальність дослідження методів побудови швидкодіючих веб-застосувань зумовлена блокувальною властивістю мови програмування JavaScript. Це означає, що під час виконання JavaScript-коду жодна інша дія неможлива. Фактично, більшість браузерів використовують єдиний процес для оновлення інтерфейсу користувача та виконання JavaScript, тому чим довше виконується JavaScript, тим більше триває час, перш ніж браузер може відповісти на

взаємодію користувача. Сьогодні не існує уніфікованого підходу до оптимізації веб-додатків, особливо через наступні фактори:

- найбільш продуктивні програмні рішення не є універсальними;
- частина загальних практик є застарілими;
- обмежений час виділений на реалізацію функціоналу продукту;
- великий вибір мобільних пристроїв;
- різна швидкість інтернет-з'єднань;
- використання фреймворків.

Виходячи з цього слід зазначити, що проблеми продуктивності веб-застосувань віддзеркалюються на зароблених коштах. Тому питання вибору сучасного програмного рішення задля побудови web-застосувань досі є актуальною та є основною метою даної роботи.

Огляд поточного стану об'єкту дослідження. Проблема підвищення продуктивності клієнтських веб-додатків розглядається з моменту появи перших веб-сторінок [1]. З того часу у світі front-end технологій спостерігається зв'язок між функціональністю та обчислювальними можливостями клієнтського пристрою. Ця проблема сприяла появі аналізаторів продуктивності веб-додатків Web.dev, Lighthouse, PageSpeed Insights [2]. Також великий обсяг роботи з покращення інтерфейсу користувача виконують фреймворки та бібліотеки ReactJS, Angular, VueJS через використання ефективних алгоритмів відтворення Document Object Model браузеру. Зараз спостерігається тенденція до використання технологій, що зменшують час роботи процесору та використання пам'яті комп'ютеру, наприклад, WebAssembly, AsmJS.

Мета дослідження. Дана робота містить дослідження проблем продуктивності веб-додатків з метою їх оптимального вирішення у виді програмних підходів. Незважаючи на те, що станом на поточний час проблема продуктивності активно вирішується у сфері front-end технологій та має добре сформовану теоретичну базу, зазвичай її практичне вирішення

обмежено часом, що дається на розробку додатку, складною інфраструктурою проекту, відсутністю професіональної компетенції. Основною новизною є сформульований підхід щодо зменшення часу парсингу, компіляції та виконання коду мови JavaScript, що спричиняє найбільші затримки часу, необхідного до першої можливості взаємодії з веб-додатком, особливо в умовах роботи з мобільного пристрою. Існуючі рішення можна поділити на наступні категорії підвищення продуктивності: зменшення часу парсингу, зменшення часу компіляції, зменшення часу виконання.

Методи оптимізації виконання коду через змінення системи типізації V8

Як відомо, веб-додатки мають проблеми виконання коду через невизначену типізацію. Ми визначаємо передбачуваність типів з точки зору здатності компілятора передбачати тип об'єкта на етапі доступу до об'єкта та мінливість типів об'єктів, що спостерігаються під час доступу до об'єкта [3]. Ми називаємо первинну форму передбачуваності типів як швидкість визначення типу, а останню - поліморфізмом. Передбачуваність типів має вирішальне значення для створення високоякісного коду. Основна частина непередбачуваності типу в коді JavaScript на веб-сайтах походить від двох несподіваних джерел а саме, прототипів [4] і прив'язок методів [5]. Прототипи JavaScript мають подібну мету - наслідування класу в статично-типізованих об'єктно-орієнтованих мовах, таких як C ++ та Java. Методи об'єкту служать подібним цілям як класові методи в статично-типізованих мовах. У статично-типізованій мові батьківські класи та прив'язки об'єкта до методу класу задаються під час створення об'єкта і ніколи не змінюються. Компілятори JavaScript оптимізують код в припущенні, що хоча JavaScript є динамічно набраною мовою, його поведінка дуже нагадує статично-типізовану мову.

В даній роботі пропонується відокремити прототипи від типів, змінюючи внутрішні структури даних у компіляторі таким чином, що __proto__ - це покажчик, який переміщений із прихованого класу до самого об'єкта. З цією зміною окремі об'єкти тепер прямо вказують на їх відповідні прототипи. Ця зміна усуває необхідність створювати новий прихований клас кожного разу, коли прототип змінюється.

На рисунку 1 показано цикл, який створює новий об'єкт Obj при кожній ітерації, а потім призначає нові функції для властивостей Foo і Bar.

```
for(var i = 0; i < 100; i++) {
  var Obj = new Object();
  Obj.Foo = function() {};
  Obj.Bar = function() {};
  Obj.Foo();
}
```

Рис. 1. Приклад призначення властивості об'єкту функції, що призводить до невизначеності типів [5]

Виклик функції властивості Foo в рядку 5 виконується за допомогою кеш-пам'яті вхідного ключа. Ми хотіли б, щоб лише один тип досягав вбудованого кешу. Рисунок 2 демонструє стан heap під час виконання простого циклу, проте після виконання змін у компіляторі.

На першій ітерації, перед виконанням тіла Foo, створюється порожній об'єкт Obj (1, але порожній), його початковий тип встановлений для нового прихованого класу 2. На другій ітерації об'єкт Obj (5) створюється знову, його початковий тип встановлюється на прихований клас 2, який був створений на попередній ітерації. Це можливо, незважаючи на різний об'єкт прототипу Foo.prototype (6), оскільки __proto__ більше не є частиною прихованого класу. Аналогічним чином, після виконання тіла Foo, тип об'єкта Obj (5) встановлений на прихований клас 4, створений на попередній ітерації. Отже, тепер Obj має той самий прихований клас в кінці кожної ітерації. В результаті вбудована кеш-пам'ять в рядку 3 завжди бачить той самий тип і залишається в мономорфному стані.

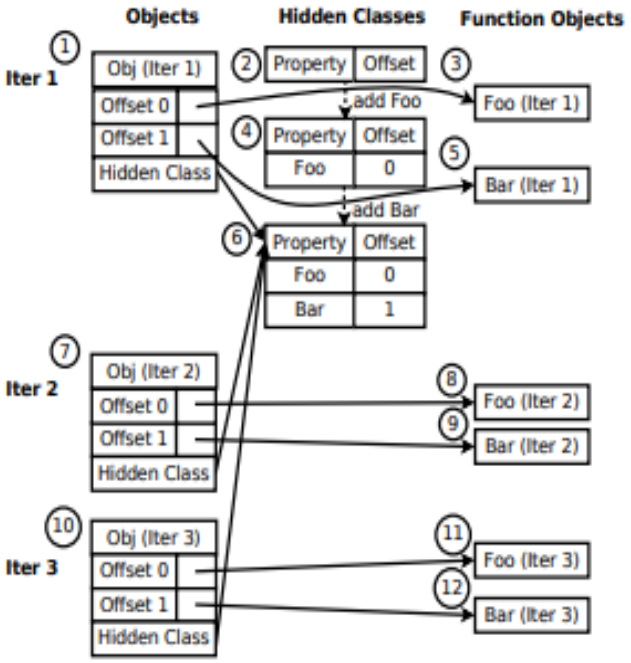

Рис. 2. Стан memory heap після виконання ітерацій циклу з використанням реконструйованої версії V8 [5]

Щоб уможливити повторне використання прихованих класів у кількох динамічних екземплярах функції, початковий прихований клас функції (прихований клас 2 у прикладі) створюється лише один раз для кожної синтаксичної функції. Синтаксична функція - це екземпляр статичної функції в вихідному коді абстрактного дерева синтаксису. Початковий прихований клас кешується у внутрішньому об'єкті синтаксичної функції, що ділиться всіма екземплярами цієї функції. Вилучення __proto__ - покажчика з прихованого класу дозволяє повторно використовувати приховані класи з попереднього глобального контексту виконання. Зокрема, об'єкти, створені за допомогою вбудованих функцій, успадкованих від вбудованих об'єктів прототипу, тепер можуть повторно використовувати приховані класи в контекстах. Це тому, що регенерація об'єктів прототипу не змушує відтворювати приховані класи. Щоб дозволити повторне використання прихованих класів в різних контекстах, початкові приховані класи вбудованих функцій кешуються в контекстах як у попередньому випадку. Коли об'єкти починаються з одного і того ж вихідного прихованого

класу, вони переходять через те саме приховане дерево класів, яке було створено в попередніх виконаннях. Таким чином, практично всі непередбачуваності типів через завантаження сторінок можуть бути усунені.

Найбільш очевидна перевага, яка випливає з модифікацій, полягає в тому, що типи тепер набагато більш передбачувані, що призводить до меншого поліморфізму в онлайнових кешах. Більш того, оскільки оптимізація компіляторів, таких як Crankshaft для V8, залежить від припущень щодо типів, які зберігаються всередині сфери оптимізації, передбачуваність типу може призвести до збільшення швидкості. Роз'єднання прототипів зменшує загальну кількість прихованих класів в купі, зменшуючи потреби в пам'яті. Наприклад, розглянемо рисунок 2 Хай I - число ітерацій, а P - кількість властивостей, призначених за допомогою динамічно виділених функцій. Тоді число прихованих класів до роз'єднання пов'язано з O(min (I, P) * P), в той час як після роз'єднання він пов'язаний лише з O (P). Більше того, зменшений поліморфізм у вбудованих кешах також призводить до зменшення обсягу пам'яті, виділеної для кешу коду. Проте коли __proto__ - покажчик переміщується з прихованого класу на об'єкт для роз'єднання прототипу, це може призвести до збільшення використання пам'яті, оскільки, як правило, в heap більше об'єктів, ніж прихованих класів. Те ж саме можна сказати, коли покажчики функції переміщуються з прихованого класу на об'єкт для від'єднання методу. Також, додаткові додані покажчики в об'єктах можуть вимагати відбору сміття, щоб зробити додаткові переходи, сповільнюючи виконання цієї функції.

Зменшення часу компіляції коду через використання WebAssembly

Сьогодні найефективнішим засобом зменшення витрат часу на компіляцію JavaScript-коду є WebAssembly. Він вирішує проблему

використання безпечного, швидкого, портативного низькорівневого коду в Інтернеті. Попередні спроби вирішити цю проблему, від ActiveX до Native Client і до asm.js, провалилися через нестачу властивостей, що повинен мати низькорівневий скомпільований код, а саме, швидку безпечну та портативну семантику та ефективне представлення. WebAssembly - це перше рішення для низькорівневого коду в Інтернеті, який забезпечує найбільш критичні проектні цілі. Ключовими можливостями WebAssembly є валідація, baseline JIT compiler, optimizing JIT compiler, reference interpreter.

На рисунку 3 бачимо час виконання тестів, узятих із бенчмарк наборів PolyBenchC через використання WebAssembly на V8 та SpiderMonkey, нормалізованих до виконання native коду.

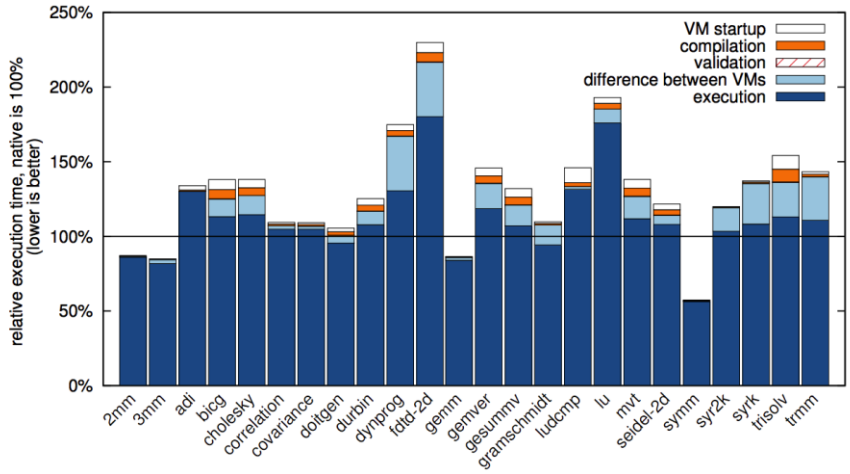

Рис. 3. Відносний час виконання бенчмарків PolyBenchC через представлення WebAssembly до native коду [6]

Час, представлений у вигляді stacked bar, та результати виконання тестів демонструють, що існують різниці в часі через використання різних генераторів коду. Установлено, що час запуску для V8 – 18мс та 30мс – SpiderMonkey. Ці виміри часу зображені разом із часом, що був витрачений на компіляцію, на вершинах колонок для кожного тесту бенчмарку. Зауважимо, що статичні витрати на запуск віртуальної машини вказують на бенчмарки, що виконуються найшвидше. В цілому, результати показують, що WebAssembly дуже конкурентоспроможний тому, що переважна

більшість усіх тестів показують перевагу в 2 рази від native коду. Також був виміряний час виконання asm.js. В середньому WebAssembly на 33,7% швидший за нього, особлива різниця мала місце у валідації коду. Виходячи із затрат пам'яті, маємо, що WebAssembly використав на валідацію коду 1% часу, що був затрачений asm.js [6]. На рисунку 4 маємо співвідношення розмірів похідного коду WebAssebly до нативного коду та asm.js.

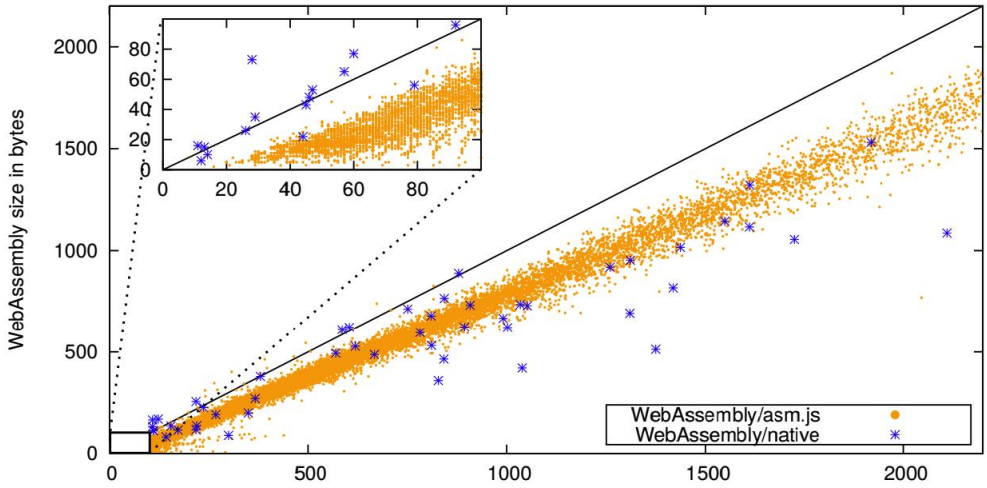

Рис. 4. Бінарний розмір WebAssembly в порівнянні з asmjs та JavaScript кодом [6]

В середньому маємо, що розмір вихідного коду складає 62,5% від asm.js та 85.3% від коду, написаного на мові C. Лише декілька тестів показують, що WebAssembly більший за native код через те, що вказівники мови C на stack locals потребують shadow stack.
З тих пір як V8 та SpiderMonkey застосували техніку AOT, це стало поштовхом для виконання паралельної компіляції модулей WebAssembly, направляючи окремі функції різним потокам. Обидва движки отримали прискорення швидкості компіляції в 5-6 разів під час використання 8 потоків виконання. Хоча розробники витратили багато ресурсів на покращення швидкості компіляції JIT, щоб скоротити холодний час запуску WebAssembly, ми очікуємо, що теплий час запуску стане важливим, оскільки користувачі зможуть повторно відвідувати ці самі веб-сторінки. API JavaScript для IndexedDB тепер дозволяє JavaScript маніпулювати та компілювати модулі WebAssembly і зберігати їх скомпільоване подання як

opaque blob в IndexedDB. Це дозволяє JavaScript спершу виконувати запит до IndexedDB для кешованої версії свого модуля WebAssembly перед завантаженням та компіляцією. Цей механізм вже був впроваджений у V8 та SpiderMonkey і відповідає за порядок поліпшення часу запуску.

Використання Fiber заради швидкого оновлення DOM

Fiber – це нова архітектура, що покладена в основу React. Основною метою було створення можливості для пріоритизації оновлень контенту. React для забезпечення високої швидкості роботи використовує технологію Vitrual DOM. В пам'яті підтримується спрощена копія DOM, де за вузлами закріплені конкретні екземпляри (instance) компонентів, що ними керують. Алгоритм полягає у розбитті процесу оновлення на дві фази:

- фаза узгодження (reconciliation) – фаза коли виконуються перерахунки компонентів і відбувається оновлення DOM у пам'яті;
- фаза внеску (commit) – фаза коли виконується безпосереднє оновлення DOM.

Варто зазначити, що фазу узгодження (reconciliation) можна переривати. Fiber за допомогою requestIdleCallback просить у браузера виділити час, коли той не буде завантажений роботою. При зворотньому виклику браузер вказує, скільки власне в нього є вільного часу. Це дає змогу fiber-у планувати частину оновлень на цей проміжок. Якщо браузер не підтримує requestIdleCallback, то React робить поліфіл. Алгоритм fiber у свою чергу названий на честь найменшого об'єкта, що лежить в його основі. За кожним екземпляром (компонента чи елемента) закріплений такий об'єкт, що контролює його стан та зв'язок з іншими компонентами. На рисунку 5 бачимо процес оновлення дерева компонентів.

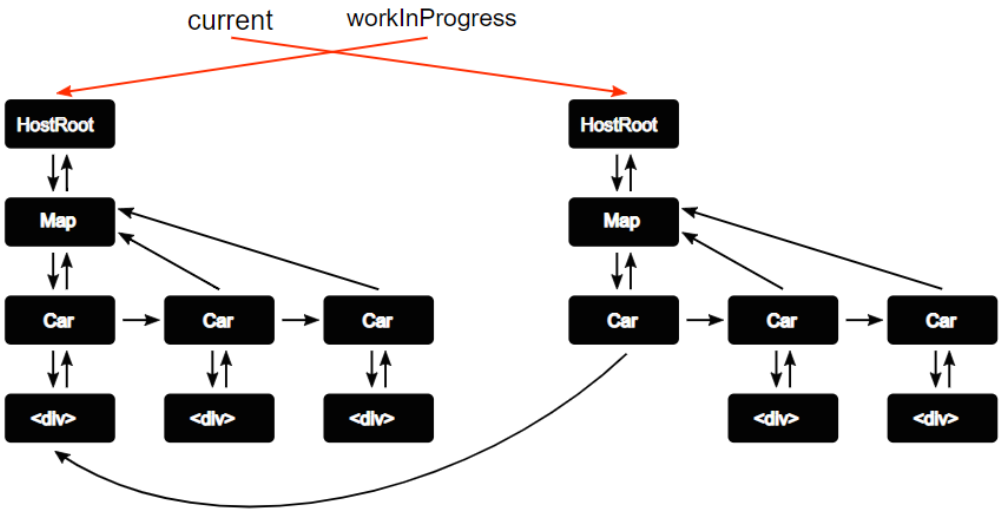

Рис. 5. Процес оновлення дерева компонентів [7]

У нас є поточне (current) дерево компонентів та елементів, сформоване за допомогою об'єктів fiber. Стрілочки вниз це child, вгору - parent, вправо - sibling. Створюється паралельне робоче (workInProgress) дерево, що частково складається зі старого дерева, далі визначаються компоненти що мають зміни, дерево поступово розгортається, і на його основі відбудовується нове дерево. Там де є оновлення – клонуються елементи і вносяться зміни. Там де оновлень немає – використовуються наявні елементи. В результаті формується внесок (pending commit). Що для застосування потребує вже більшого проміжку часу, тому що фазу внеску переривати не можна. Після того, як відбувається внесок (commit), поточне (current) дерево не знищується. Для економії часу дерева просто міняються місцями [7].

На рисунку 6 бачимо результат роботи архітектури Fiber. Зверху позначена частота оновлення кадрів (fps), знизу - оновлення virtual DOM та виконання анімації. Як бачимо, обробка CSS анімацій не зупиняється навіть при високій завантаженості оновленнями DOM.

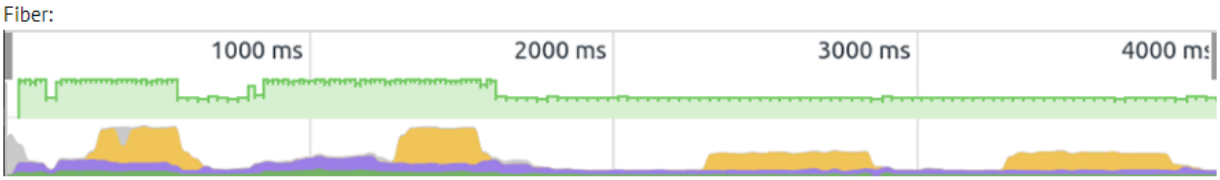

Рис. 6. Порівняння Fiber та Stack

Таким чином, маємо ефективний алгоритм, що дозволяє уникнути важливих проблем, пов'язаних із провисанням веб сторінки під час виконання анімації.

Висновки. Від самого початку роботи над проектом, розробники мають розглядати проблеми продуктивності. З часом з'являються все більш універсальні підходи до їх вирішення, проте під час розв'язання тривіальних проблем, таких як оптимізація зображень, веб-шрифтів та мережевих взаємодій, маємо не вирішену проблему витрати великого обсягу часу на компіляцію та виконання коду. В цій роботі було розглянуто три можливі підходи до вирішення цієї проблеми, а саме: змінення системи типізації, що використовується сучасними браузерами, використання WebAssembly заради компіляції коду мови програмування C++ у бінарні файли wasm, задля уникнення кроку компіляції та парсингу безпосередньо браузером, та використання ефективного алгоритму оновлення сторінки Fiber, оскільки WebAssembly не підтримує роботу з DOM. Таким чином, використання зазначених підходів має на меті прискорення роботи веб-додатку та можливість їх застосування має бути розглянута розробниками на первинних кроках роботи над проектом.

Література

1. Jeremy Wagner. Why performance matters / Google Developers [Електронний ресурс]. – Режим доступу: https://developers.google.com/web/fundamentals/performance/why-performance-matters

2. Google developers blog. Lighthouse / Google Developers [Електронний ресурс]. – Режим доступу: https://developers.google.com/web/tools/lighthouse/
3. Mathias Bynens JavaScript engine fundamentals: Shapes and Inline Caches / Mathias Bynens blog [Електронний ресурс]. – Режим доступу: https://mathiasbynens.be/notes/shapes-ics
4. Mathias Bynens JavaScript engine fundamentals: optimizing prototypes // Mathias Bynens blog [Електронний ресурс]. – Режим доступу: https://mathiasbynens.be/notes/prototypes#tradeoffs
5. Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving JavaScript Performance by Deconstructing the Type System. In ICSE, 2014. — 12 с.

A. Haas, A. Rossberg. Bringing the Web up to Speed with WebAssembly In ICSE, 2017. — 16 с.

6. Lin Clark. A cartoon intro into Fiber. React Conf 2017 [Електронний ресурс]. – Режим доступу: https://www.youtube.com/watch?v=ZCuYPiUIONs