

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**O comparație obiectivă a timpilor de execuție și
costurilor între WebAssembly și JavaScript**

propusă de

Cristian-Iulian Antal

Sesiunea: iunie/iulie, 2023

Coordonator științific

ACS. Drd. Irimia Cosmin

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

**O comparație obiectivă a timpilor de
execuție și costurilor între
WebAssembly și JavaScript**

Cristian-Iulian Antal

Sesiunea: iunie/iulie, 2023

Coordonator științific

ACS. Drd. Irimia Cosmin

Abstract

WebAssembly este o tehnologie nouă, creată pentru Web dar având mai multe scopuri în spate. În această lucrare de cercetare vom analiza cum funcționează WebAssembly, ce beneficii aduce și cum se situează în raport cu JavaScript, un alt limbaj de programare pentru Web. Pentru a analiza cele două tehnologii cu scopul de a vedea în ce situații ar fi mai bine să folosim un limbaj sau altul, am creat o aplicație de tip „Proof of Concept” unde compar anumiți algoritmi aleși de mine, implementați în ambele limbaje de programare. Algoritmii au fost aleși din mai multe arii ale informaticii (criptografie, algoritmică, teoria grafurilor, etc.), majoritatea studiați la facultate, pentru a avea o comparație echilibrată și au fost prezentați în capitolul 3. Prezentarea rezultatelor experimentelor și analiza acestora va fi făcută în capitolul 5, iar în capitolul 5.4 vom sumariza toate informațiile și vom aduce câteva concluzii bazate pe analiza realizată precedent.

Ideea de bază a lucrării a plecat din simpla dorință de a arăta această diferență de performanță între cele două limbaje. Toate experimentele au fost realizate de mine, sub îndrumarea atentă a domnului profesor coordonator. Toate tehnologiile, librăriile, produsele software și elementele grafice sunt „open-source” sau gratuite pentru a fi folosite în proiecte personale. În final, ne propunem să avem o imagine de ansamblu și să putem răspunde unor întrebări cum ar fi: „Va înlocui WebAssembly JavaScript în dezvoltarea aplicațiilor web?”, „În ce condiții WebAssembly va fi o alegere mai bună decât JavaScript?”, „Cu cât este mai rapid WebAssembly față de JavaScript?”.

Cuprins

Abstract

Motivație	2
-----------	---

Introducere	3
-------------	---

1 Tehnologii utilizate	4
------------------------	---

1.1 WebAssembly	4
-----------------	---

1.1.1 AssemblyScript	7
----------------------	---

1.2 JavaScript	8
----------------	---

1.2.1 Procesul de compilare	8
-----------------------------	---

1.2.2 Compilatoare Just-in-time (JIT)	9
---------------------------------------	---

1.2.3 Optimizarea bazată pe profil	10
------------------------------------	----

1.2.4 Motoare de compilare	10
----------------------------	----

1.3 Concluzii	13
---------------	----

2 State of the Art	14
--------------------	----

2.1 Literatură de specialitate	14
--------------------------------	----

2.2 Aplicații similare	16
------------------------	----

2.3 Concluzii	17
---------------	----

3 Algoritmi comparați	20
-----------------------	----

3.1 Calculul factorialului	20
----------------------------	----

3.2 Înmulțirea a două matrici	21
-------------------------------	----

3.3 Căutare binară	22
--------------------	----

3.4 MD5	22
---------	----

3.5 BFS	23
---------	----

3.6 Problema colorării unui graf cu abordarea Greedy	24
--	----

3.7	Problema rucsacului abordarea cu Programare Dinamică	25
3.8	Concluzii	26
4	Procese	27
4.1	Crearea proiectului	27
4.2	Pregătirea testelor	31
4.3	Concluzii	32
5	Comparare și evaluare	34
5.1	Experimente pe sistemul de operare Windows	34
5.1.1	Google Chrome	34
5.1.2	Mozilla Firefox	36
5.1.3	Microsoft Edge	37
5.1.4	Analiza rezultatelor pentru sistemul de operare Windows	39
5.2	Experimente pe sistemul de operare MacOS	41
5.2.1	Google Chrome	41
5.2.2	Mozilla Firefox	42
5.2.3	Safari	43
5.2.4	Analiza rezultatelor pentru sistemul de operare MacOS	44
5.3	Experimente pe sistemul de operare Linux	47
5.3.1	Google Chrome	47
5.3.2	Mozilla Firefox	48
5.3.3	Analiza rezultatelor pentru sistemul de operare Linux	49
5.4	Concluzii	52
	Planuri de viitor și îmbunătățiri	54
	Concluzii	56
	Bibliografie	58

Motivație

În al doilea an, participând la un workshop organizat la facultate de Asociația Studenților Informaticieni din Iași, am aflat despre WebAssembly, un limbaj low-level destul de recent apărut, care promite viteză de procesare mult mai mare decât ce oferă JavaScript în momentul de față. Am rămas fascinat și mi-am propus să învăț despre acest limbaj care pare a fi de viitor datorită evoluției tehnologiei, cantităților mari de date ce trebuie procesate și dorința utilizatorilor de a nu aștepta foarte mult sau chiar deloc atunci când interacționează cu o aplicație de orice tip.

Astfel, am aflat despre WebAssembly că are anumite use-case-uri în care chiar se remarcă viteza acestuia cum ar fi: algoritmi în care se fac calcule complicate, cu numere mari, în aplicațiile web care prelucrează poze, videoclipuri, jocuri de tip FPS sau jocuri ce au componente foarte mari în dimensiuni, aplicații peer-to-peer, aplicații de streaming, aplicații ce folosesc AR/VR, aplicații enterprise cu baze de date extrem de mari, dar și în aplicații ce nu rulează în browser: aplicații native de mobile, calcule simetrice peste mai multe noduri, server pentru o aplicație sau pentru un joc și multe alte situații în care am putea alege WebAssembly în detrimentul altei tehnologii.

Introducere

Lucrarea de licență este structurată în cinci capitole cu următorul conținut:

Capitolul 1 - Tehnologii utilizate , unde vom explica noțiuni introductive legate de WebAssembly și JavaScript pentru a putea înțelege în esență cum funcționează aceste două limbaje de programare și cum ne putem folosi de ele în experimentele noastre.

Capitolul 2 - State of the art , vom prezenta și analiza alte lucrări și aplicații asemănătoare și vom puncta ce facem noi în plus sau diferit în lucrarea noastră.

Capitolul 3 - Algoritmi comparați , vom discuta despre implementarea algoritmilor și alegerile făcute în această privință.

În ce mediu au fost realizate și toate condițiile de rulare, componentele hardware ale laptopului, sistemul de operare, browser-ul folosit și vom prezenta mai multe rezultate pe baza cărora vom face o analiză detaliată în următorul capitol.

Capitolul 4 - Procese , vom explica cum am creat proiectul de AssemblyScript, ce reprezintă fiecare fișier din proiect, unde scriem codul în AssemblyScript și unde ajunge codul compilat în WebAssembly, și în final vom explica cum am pregătit testele pentru a calcula timpul de execuție.

Capitolul 5 - Comparare și evaluare , pe baza experimentelor realizate, vom analiza viteza de execuție și memoria folosită în mai multe condiții: pe mai multe sisteme de operare și pe mai multe browser-e.

Nu în cele din urmă, lucrarea se va încheia cu partea de concluzii, o perspectivă asupra WebAssembly și idei despre cum se poate analiza, descoperi și folosi în mai multe circumstanțe acest limbaj de programare într-o lucrare științifică viitoare.

Capitolul 1

Tehnologii utilizate

În acest capitol voi prezenta noțiuni introductive legate de WebAssembly, respectiv AssemblyScript și JavaScript pentru a putea înțelege mai bine cum funcționează aceste două limbaje la nivelul browser-ului.

1.1 WebAssembly

WebAssembly¹, abreviat Wasm, este un limbaj de programare low-level, sigur, portabil, creat pentru eficiență. WebAssembly a fost lansat oficial în anul 2017 în urma unei colaborări a mai multor companii mari cum ar fi Mozilla, Google, Microsoft și Apple. Acest limbaj de programare reprezintă un standard deschis creat de grupul comunitar W3C (World Wide Web Consortium). Profitând de avantajul formatului binar și a unui set de instrucțiuni compact, WebAssembly atinge viteze mari de încărcare și executare în comparație cu tehnologiile web tradiționale. Acest limbaj permite programatorilor să compileze codul într-un format binar ce poate fi executat direct în browser, neavând nevoie de interpretare sau compilare „just-in-time”, atingând astfel performanțe foarte aproape de cele native în browser. Scopul său principal este de a sta la baza aplicațiilor Web de înaltă performanță, dar și a altor tipuri de aplicații, nefiind un limbaj specific pentru Web.[1] Acesta dovedește performanța pentru care a fost creat în cadrul aplicațiilor de editare a pozelor, videoclipurilor, jocurilor de tip FPS, simulatoarelor științifice etc.[17]

Aplicațiile create în WebAssembly beneficiază de anumite calități pe care orice programator urmărește să le aibă la programele scrise pentru a fi cât mai complete și

¹<https://webassembly.github.io/spec/core/intro/index.html>

complexe, astfel, acest limbaj aduce aplicațiilor[2]:

- **Rapiditate** - codul se execută cu viteză aproape de performanța codului nativ, luând avantaj de capacitățile comune tuturor componentelor hardware contemporane;
- **Siguranță** - codul este executat și validat într-un mediu de lucru izolat, securizat, ce protejează memoria și care împiedică coruperea datelor;
- **Independență de hardware** - codul poate fi compilat pe toate arhitecturile moderne de hardware, pe desktop sau dispozitive mobile precum și pe sisteme embedded;
- **Independență de limbajul de programare** - nu privilegiează niciun limbaj în mod particular, există multe limbaje ce pot fi compilate în cod Wasm, unele într-un mod mai eficient, altele într-un mod mai puțin eficient;
- **Independență de platformă** - poate fi rulat în browser, să ruleze ca o mașină virtuală de sine stătătoare sau să fie integrat în alte medii;
- **Bine definire** - definește complet și precis programe valide și comportamentul acestora într-un mod ușor de înțeles;
- **Compactitate** - formatul binar folosit este unul cu transmitere rapidă fiind mai mic decât un text obișnuit sau formate de cod native;
- **Modularitate** - orice program poate fi împărțit în părți mai mici ce pot fi transmise, cache-uite și consumate separat în mod independent;
- **Eficiență** - codul poate fi validat, compilat într-un singur pas foarte rapid și decodat în același mod fie cu compilare „just-in-time” (JIT) sau compilare „ahead-of-time” (AOT);
- **Portabilitate** - codul poate fi executat pe componente hardware moderne fără a face presupuneri sau a ține cont de arhitectura acestora;
- **Paralelizare a proceselor** - permite decodarea, validarea și compilarea codului pentru a fi împărțit în mai multe fire de execuție paralele;
- **Compilabil în flux** - permite decodarea, validarea și compilarea codului pentru executare înainte ca datele să fie văzute.

Formatul binar folosit de WebAssembly² este organizat în module ce conțin definiții ale funcțiilor, variabilelor globale, tabelelor etc. Fiecare funcție definită poate fi exportată și importată, iar cele importate pot fi exportate din nou. Când un modul este instanțiat, el devine o reprezentare dinamică a programului scris de noi, cu memorie ce poate fi schimbată și o stivă de execuție. Funcțiile sunt organizate în module, pot fi apelate recursiv dar nu pot fi definite în interiorul altei funcții. WebAssembly are patru tipuri de date: numere întregi pe 32-bit și 64-bit și numere reale pe 32-bit și 64-bit, care sunt toate disponibile pe hardware-ul obișnuit. Instrucțiunile din WebAssembly se folosesc doar de aceste patru tipuri de date și sunt asigurate toate conversiile necesare între ele.[16],[3]

În WebAssembly, corpul unei funcții este un bloc cu semnătură ce mapează o stivă goală la răspunsul funcției. Argumentele funcțiilor sunt stocate în variabile locale. O funcție își poate termina execuția în trei moduri: ajungând la finalul blocului de execuție, executând o instrucțiune „return” sau realizând o bifare către blocul funcției cu valorile obținute pentru rezultat ca operațiuni. Funcțiile pot fi apelate direct folosind instrucțiunea „call” care face „pop” la argumentele funcției și „push” la rezultatul returnat. Apelurile indirecte pot fi facute prin instrucțiunea „call indirect” ce preia un index de runtime într-un tabel global de funcții. Tipul funcției este verificat dinamic cu un tip așteptat pentru a asigura siguranță în execuție. O nepotrivire a tipurilor sau accesarea unui index din tabel ce nu există produce „traps”. Aceste „traps” reprezintă condiții de excepție sau erori ce duc la oprirea executării. Exemple de „traps” ar fi: împărțirea la 0, accesul înafara memoriei, nepotrivire de tipuri, cod ce nu poate fi executat pentru ca nu se ajunge la el etc.[3]

WebAssembly nu reprezintă doar un limbaj de programare ci un ecosistem ce se află într-o continuă creștere și dezvoltare, cu o mulțime de programe, framework-uri, biblioteci, extensii, ce ajută și oferă suport programatorilor. Acest ecosistem are niște caracteristici cheie menite să faciliteze dezvoltarea de aplicații și de a atrage cât mai mulți oameni să descopere acest limbaj. În primul rând, există compilatoare și aplicații ce permit convertirea unui cod scris într-un limbaj de nivel mare în WebAssembly bytecode; există foarte mult suport mai ales în limbaje precum C/C++, Rust, TypeScript, C# prin intermediul anumitor librării și instrumente software dedicate. Manager-ii de pachete (de exemplu, npm, Cargo, etc.) oferă modalități ușoare, convenabile, de a distribui modulele de WebAssembly; aceștia simplifică mult procesul de căutare, instalare

²<https://developer.mozilla.org/en-US/docs/WebAssembly>

și folosire a dependențelor. Multe biblioteci și framework-uri au fost special facute sau îmbunătățite pentru a fi folosit WebAssembly, spre exemplu: Emscripten, AssemblyScript, WebAssembly Studio, Blazor Pages din .NET etc. Și cel mai important lucru din acest ecosistem este comunitatea și resursele oferite cu access pentru oricine: forumuri online, documentații extrem de detaliate, tutoriale, proiecte open-source sau template-uri de proiecte, chiar și servere de Discord unde toata lumea dorește să ajute și să contribuie cu resurse și cunoștințe.[3]

1.1.1 AssemblyScript

AssemblyScript³ este un limbaj de programare de nivel înalt gândit a fi un subset al limbajului TypeScript (care la rândul său este un superset al limbajului JavaScript), dar care compilează codul în WebAssembly. Acest limbaj de programare are sintaxa și semantica foarte apropiate de cele folosite în JavaScript, fiind ușor de folosit fără a fi problematic sau mult mai complex. AssemblyScript compilează codul în WebAssembly prin intermediul Binaryen.[18]

Binaryen este o bibliotecă scrisă în C++ cu scopul de a face compilarea codului din AssemblyScript în WebAssembly cât mai rapidă și eficientă, cu capabilități de multithreading, ceea ce îi permite să creeze reprezentări intermediare și în același timp să optimizeze acest proces. Codul scris în AssemblyScript fiind compilat în WebAssembly permite aplicațiilor scrise în acest limbaj de programare să fie folosite în diferite medii, preponderent Web dar și pe partea de server a unei aplicații sau chiar pentru dispozitive din domeniul Internetul Lucrurilor(IoT).[4]

Avantajul cheie al AssemblyScript este accesul la lucrurile de nivel mic cum ar fi manipularea directă a memoriei, pointeri etc. Acest nivel de control permite programatorilor să optimizeze secțiuni critice din programul lor, să interacționeze cu componentele hardware sau API-uri de sistem. Un alt beneficiu important, datorat faptului că acest limbaj este un subset al TypeScript, este integrarea cu ecosistemul limbajului JavaScript prin faptul că programatorii pot exporta și importa funcții între cele două limbaje. Astfel, se pot folosi de ambele tehnologii în cadrul aceleiași aplicații permițând astfel reutilizarea codului și folosirea variantei cele mai bune pentru o optimizare maximă.[4]

Rezumând cele de mai sus, având sintaxă asemănătoare cu TypeScript-ul, opti-

³<https://www.assemblyscript.org/introduction.html>

mizări pentru performanță și codul scris fiind compilat în WebAssembly, AssemblyScript este un limbaj de programare foarte puternic, versatil, ușor de folosit pentru a crea aplicații de mare performanță ce pot rula pe diferite platforme și într-o multitudine de medii oferind totodată și siguranța și productivitatea unui limbaj de programare de nivel înalt.

1.2 JavaScript

JavaScript este un limbaj de programare folosit pentru paginile Web cu scopul de a le face mai dinamice. Acest limbaj de programare suportă multiple paradigme de programare cum ar fi programarea orientată obiect, programare procedurală și programare declarativă. Având această natură dinamică, JavaScript utilizează compilarea de tip „just-in-time” (JIT). Procesul de compilare al codului scris în JavaScript în cod mașină gata de executare diferă de la un browser la altul. Pe Firefox, Thunderbird este folosit SpiderMonkey ca motor de compilare, pe Google Chrome, Opera este folosit motorul de compilare V8, JavaScriptCore pe Safari, Webkit, iOS și Chakra pe Internet Explorer și Microsoft Edge Legacy. Oricum, toate aceste tipuri de browser apelează la o tehnică generală pentru a spori performanța și viteza de execuție. Motoarele de compilare pentru JavaScript au adoptat o arhitectură complexă pe mai multe straturi. Aceasta este compusă din mai multe trepte de execuție ce includ și interpretorul și compilatorul. Aceste trepte operează mai apoi pe principiul „funcție cu funcție” pentru a mări viteza de executare a codului.[4]

1.2.1 Procesul de compilare

Pentru a fi executat codul, prima dată trebuie ca acesta să fie transformat din limbajul în care a fost scris într-un limbaj mașină ce poate fi înțeles de calculator și executat. Pentru a se realiza acest lucru există în principal două metode: prin intermediul unui interpretor și prin intermediul unui compilator.

Prima metodă este folosirea unui interpretor care merge prin cod linie cu linie și execută codul în ordinea în care este scris. În această metodă, expresiile sunt evaluate direct iar comenzile sunt executate. Există și posibilitatea ca interpretorul să proceseze aceleași expresii și comenzi de mai multe ori față de un compilator, motiv pentru care se consideră a avea o performanță scăzută. Însă un interpretor are și avantaje: este ușor

de mutat pe o altă mașină și este rapid în procesul de a începe să ruleze programul. Interpretorii sunt rapizi când vine vorba de a porni un program dar destul de înceteți în executarea acestuia.

A doua metodă este folosirea unui compilator ce transformă codul scris de către noi, oamenii, de obicei într-un limbaj de programare de nivel înalt, într-un limbaj mașină, de nivel mic, ce poate fi înțeles și executat de către mașina pe care se rulează programul. În procesul compilării programului, compilatorul poate detecta greșeli, erori de sintaxă ce vor fi afișate ulterior utilizatorului. Noi folosim limbaje de nivel înalt deoarece sunt mult mai apropiate de gândirea logică a omului și prin ușurința oferită de a exprima și rezolva probleme și chiar de a folosi programul scris pe mai multe dispozitive și medii. Avantajul de a folosi un compilator este capabilitatea acestuia de a găsi greșeli și viteza mai mare de executare față de un interpretor.[5]

1.2.2 Compilatoare Just-in-time (JIT)

Compilatoarele JIT au fost create pentru a compila codul pe durata execuției. Acestea reprezintă o încercare de a combina cele două metode de compilare și executare a unui program: prin intermediul interpretorilor și al compilatoarelor.

Acest tip de compilatoare funcționează în trei etape. Prima etapă reprezintă acțiunea unui parser de a traduce codul sursă al unei funcții scrise în JavaScript în cod bytecode. Mai apoi, bytecodu-ul rezultat în urma acestei traduceri este executat pentru a asigura o pornire rapidă. Din acest punct, executarea este repetată de mai multe ori, cu dependență la motorul folosit, mai exact când acesta consideră codul „cald”. Când bytecode-ul este considerat „cald”, compilatorul va începe compilarea bytecode-ului în cod mașină cu optimizări minime. În acel moment, compilatorul JIT începe să colecteze informații de profil prin inserarea unui cod de instrumentare (fragmente extra de cod cu scopul colectării informațiilor de profil). Dacă codul de bază va fi executat de un număr de ori variabil în funcție de motorul browser-ului, se vor putea colecta informații stabile despre profil iar funcția executată va fi considerată „caldă”. Acest lucru va iniția cea de a treia fază și anume recompilarea codului cu aplicarea a numeroase optimizări, cea mai puternică fiind optimizarea bazată pe profil.[5]

1.2.3 Optimizarea bazată pe profil

JavaScript este un limbaj de programare cu tipuri de date dinamice. Acest lucru înseamnă că tipul unor anumite variabile se poate schimba pe durata execuției codului, ba mai mult, se pot adauga și șterge proprietăți în mod dinamic. Datorită acestor modificări dinamice de tip, compilatorul este nevoit să evalueze constant tipul de obiect sau variabilă. După o perioadă de timp compilatorul va încerca să prezică cum se execută anumite secvențe de cod bazat pe anumite modele colectate în execuțiile anterioare. În final, compilatoarele JIT încep să genereze cod potrivit acestor modele și încearcă să prezică ce tip de comportament va fi repetat în viitoarele execuții.

Motoarele de copilare specifice JavaScript au tendința de a folosi clase ascunse, liste de proprietăți și pozițiile lor în cadrul obiectelor. Acest lucru ajută la o reprezentare a structurii unui obiect, asta ducându-ne la concluzia că orice obiect are un pointer către o clasă ascunsă ce este accesată de fiecare dată când ne folosim de o proprietate specifică acelui obiect. Bazat pe codul scris și ce observă compilatorul în încercarea sa de a optimiza codul prin ignorarea claselor ascunse, procedeu numit „inline caching”. Dacă compilatorul observă că o funcție este apelată cu același obiect se va aștepta ca în viitor să primească un obiect de același tip. Compilatorul încearcă să își valideze fiecare asumptie prin inserarea unor bucăți de cod „gardian”.

Codul de tip „gardian” constă în instrucțiuni și verificări suplimentare pentru a confirma corectitudinea asumptiilor și pentru a asigura că execuția continuă în conformitate cu acestea. Adresa clasei ascunse este salvată în memorie și comparată mai apoi cu obiectul curent; dacă este un alt tip de obiect, un semnal se declanșează în cadrul compilatorului cum că un alt tip de obiect a fost trimis într-o funcție unde se aștepta deja un anumit tip. În acest caz, compilatorul va începe un procedeu de redresare numit decompilare, va întrerupe execuția din acel punct și va reveni la codul de bază, fără nicio asumptie și va continua execuția.[5]

1.2.4 Motoare de compilare

În acest subcapitol, voi prezenta două motoare de copilare pentru JavaScript, motorul V8 folosit pe Google Chrome și SpiderMonkey folosit pe Mozilla Firefox, două dintre browser-ele folosite în experimentele realizate pentru această lucrare.

Motorul de compilare V8

Motorul de compilare V8 funcționează în felul următor: preia codul sursă JavaScript și îl trece prin interpretorul propriu pentru a crea o reprezentare intermediară sub forma unui arbore abstract de sintaxă (AST - „Abstract Syntax Tree”). Un AST este o structură reprezentată printr-o formă arborescentă a structurii codului sursă. Fiecare nod în arbore stochează o construcție din codul sursă. Acest arbore este folosit mai apoi de către primul interpretor numit Ignition. Acest interpretor va genera bytecode, un cod mașină neoptimizat. Bytecode-ul generat este rulat mai apoi de câteva ori pentru a fi analizat. Ignition constă într-un set de manipulatori de bytecode, fiecare secvență de bytecode trecând de la un manipulator la altul. Acești manipulatori sunt creați în cod de asamblare, acest lucru permițând interpretorului să fie creat o singură dată iar mai apoi să se bazeze pe următorul compilator, TurboFan, să genereze cod mașină pentru toate arhitecturile suportate de motorul V8.[6]

După ce Ignition își termină treaba, TurboFan, compilatorul pentru optimizare, își începe execuția cu scopul de a optimiza codul sau de a-l deoptimiza în cazul în care anumite condiții din analiza precedentă nu sunt întâlnite și revine la codul de la care a plecat și reia procesul.

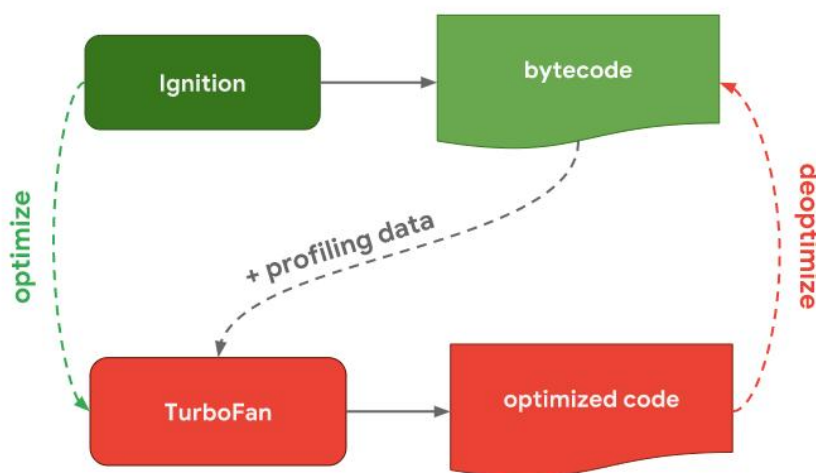


Figura 1.1: Pipeline generic de compilare și executare a codului sursă JavaScript în motorul V8.⁴

⁴<https://mathiasbynens.be/notes/shapes-ics>

Motorul de compilare SpiderMonkey

Motorul de compilare SpiderMonkey folosește o abordare diferită pentru compilatorul său JIT. Codul sursă este trecut prin interpretor care generează AST-ul, care este folosit mai apoi pentru a genera bytecode care este trimis în cele din urmă la compilatorul de optimizare împreună cu informațiile despre profilul creat. Diferența majoră față de motorul de compilare V8 este că SpiderMonkey are două compilatoare de optimizare. Traseul ce îl urmează codul sursă este următorul: prima dată trece prin interpretor pentru a fi trimis la optimizare către primul compilator numit Baseline, care încearcă într-o oarecare măsură să optimizeze și să analizeze codul. După acești doi pași, bytecode-ul și profilul de analiză ajung la compilatorul IonMonkey pentru o optimizare mai eficientă și complexă. În caz de eroare pentru o asumptie, compilatorul IonMonkey se va opri iar compilatorul Baseline va începe o nouă analiză.[4]

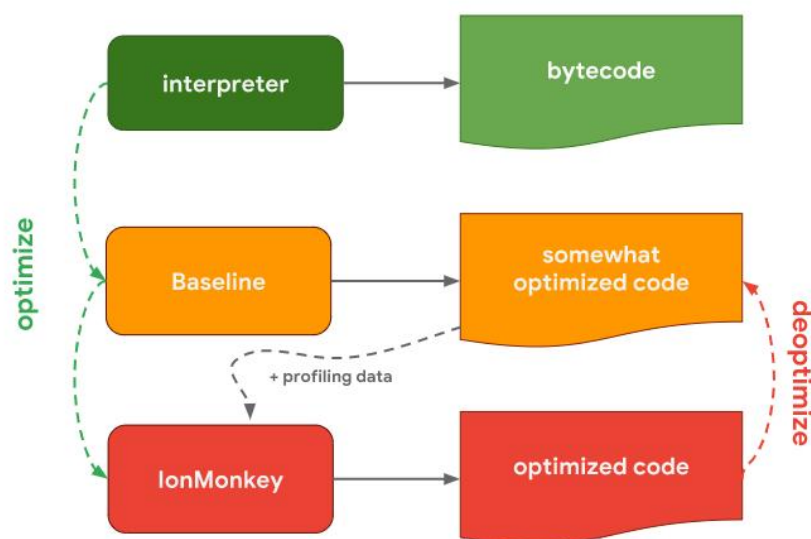


Figura 1.2: Pipeline generic de compilare și executare a codului sursă JavaScript în motorul SpiderMonkey.⁵

⁵<https://mathiasbynens.be/notes/shapes-ics>

1.3 Concluzii

În acest capitol am ales să prezentăm AssemblyScript în detrimentul altui limbaj de programare de nivel înalt ce poate fi compilat în WebAssembly din două motive: sintaxa familiară de TypeScript ce face codul ușor de înțeles și de scris, și faptul că este un limbaj de sine stătător, compilează fără ajutorul unor programe externe în cod WebAssembly fiind un limbaj special creat pentru acest fapt. Pe de altă parte, motivația de a alege JavaScript a fost simplă deoarece singura alternativă ar fi fost TypeScript care este doar un superset al JavaScript.

Noțiunile prezentate au rol introductiv pentru a ne familiariza cu cele două tehnologii și a înțelege cum functionează, astfel cele prezentate în capitolele următoare să fie ușor de înțeles.

Capitolul 2

State of the Art

Această lucrare are drept scop cunoașterea unui nou limbaj de programare, inovativ, concurent cu cel mai folosit limbaj de programare pentru aplicații web, JavaScript. Noțiunile introductive despre WebAssembly și JavaScript au fost colectate și analizate în baza mai multor lucrări științifice găsite pe internet, accesibile oricui. Focusul principal al cercetării este descoperirea acestui limbaj, relativ nou, dar și o analiză detaliată despre performanța acestuia în raport cu JavaScript. Chiar dacă este o tehnologie recent apărută, foarte multe lucrări științifice au fost realizate legat de acest limbaj de programare pentru a răspunde întrebărilor oamenilor sau din proprie curiozitate cu scopul de a învăța ceva nou.

2.1 Literatură de specialitate

Fiind o tehnologie nouă, apărută acum câțiva ani, există foarte multe lucrări de cercetare cu tema WebAssembly. Aflându-se într-o continuă dezvoltare, cercetările făcute la începuturile acestui limbaj de programare nu mai au o acuratețe mare, foarte multe lucruri fiind schimbate, îmbunătățite și multe funcționalități noi adăugate. De asemenea, documentațiile oficiale oferite de creatorii WebAssembly și AssemblyScript au constituit un pilon principal în redactarea lucrării și a experimentelor realizate, fiind actualizate constant. În acest subcapitol ne-am propus să analizăm două lucrări de cercetare cu aceeași temă, diferențele dintre WebAssembly și JavaScript la mai multe categorii: viteză, memorie, consum de energie și optimizare.

Prima lucrare analizată este „WebAssembly and JavaScript Challenge: Numerical

program performance using modern browser technologies and devices”¹ , 2018, scrisă de David Herrera, Hangfeng Chen, Erick Lavoie și Laurie Hendren de la McGill University, School of Computer Science, din Montreal, Canada. Am ales această cercetare scrisă în 2018 pentru a analiza schimbările apărute la WebAssembly în ultimii aproximativ 5 ani. Autorii au plecat de la dorința de a compara tehnologii vechi cu tehnologii noi, rulând experimentele pe telefoane mobile și dispozitive IoT.

În lucrarea lor au făcut experimente pentru a compara și WebAssembly cu JavaScript, capitolul pe care îl vom avea în vedere în analiza noastră. Cercetarea lor a început după ce majoritatea browser-elor au integrat suport pentru Wasm, moment oportun pentru a vedea îmbunătățirile de performanță ce aveau să vină cu aceste actualizări. Experimentele au fost realizate pe mai multe dispozitive: iPhone 10, iPad Pro, Samsung Tab3, Samsung S8, Google Pixel2, Windows Bison, Ubuntu Deer și Raspberry PI, dar și pe diferite browser-e precum: Google Chrome v63, Mozilla Firefox v57, Safari v11, și Microsoft Edge și Samsung Internet, ce erau cele mai noi platforme de tip browser.

După multitudinea de teste realizate au ajuns la concluzia că WebAssembly oferă o performanță excelentă, apropiată de performanța nativă a limbajului C în Mozilla Firefox. Concluzia finală fiind că WebAssembly va fi foarte important în realizarea de calcule numerice complicate într-un mod eficient pe web.

Această lucrare prezintă performanța celor două limbaje pe platforme mobile și pe un Raspberry PI însă nu prezintă o cercetare a performanței pe laptopuri diferite cu sisteme de operare diferite, lucru pe care noi ni-l propunem în lucrarea prezentă. Este destul de relevant mediu în care rulează programele dar și dispozitivul folosit, componentele hardware, sistemul de operare și browser-ul utilizat. Toți acești factori fiind un input foarte bun pentru analiza performanței și a utilizării memoriei.[8]

A doua lucrare analizată este „Understanding the Performance of WebAssembly Applications”² , 2021, realizată de Yutian Yan, Tengfei TU, Lijian Zhao, Yuchen Zhou și Weihang Wang de la University at Buffalo, SUNY și Beijing University of Posts and Telecommunications. Scopul acestei lucrări este de a descoperi WebAssembly și de a-l compara cu JavaScript.

Experimentele realizate de autori au drept scop analiza impactului performanței programelor scrise asupra mai multor browser-e și platforme. Benchmark-urile au

¹<http://www.sable.mcgill.ca/publications/techreports/2018-2/techrep.pdf>

²<https://dl.acm.org/doi/abs/10.1145/3487552.3487827>

fost rulate pentru a determina viteza de rulare și memoria folosită de fiecare limbaj la executare. Pentru realizarea experimentelor s-au folosit de Google Chrome v79 și Mozilla Firefox v71, pe telefon dar și pe calculator, și o serie de algoritmi din diferite arii ale informaticii, spre exemplu: înmulțiri de matrici, algoritmul criptografic AES, simulatoare pentru câmpuri magnetice și electrice etc.

În finalul lucrării, aceștia au afirmat că numărul de teste realizate nu este îndeajuns de mare și precis pentru a concretiza o concluzie obiectivă. Cu toate acestea, rezultatele experimentelor au fost următoarele: folosind date de intrare de dimensiuni mici și foarte mici, WebAssembly este de 8.22x respectiv 26.99x mai rapid decât JavaScript, dar aceste rezultate diferă de la o bibliotecă de benchmark la alta.[9]

Lucrarea analizată mai sus, față de prima lucrare, rulează testele atât pe platforme mobil cât și pe calculator, au folosit o varietate mai mare de algoritmi și mai multe biblioteci de benchmark. Noi ne propunem să vedem diferențele ce apar și în funcție de sistemul de operare folosit și componentele hardware ale calculatorului pe care realizăm experimentele. De asemenea, versiunea browser-ului fiind una mult mai nouă în prezent, vom putea vedea dacă rezultatele vor fi cu mult diferite sau nu și să concluzionăm în ce măsură este afectată execuția programelor în funcție de browser.

2.2 Aplicații similare

Există deja o multitudine de experimente realizate fie în lucrări de cercetare, fie din pură curiozitate sau dorința de a descoperi sau promova WebAssembly. Unii oameni chiar au creat și publicat aplicații în acest scop. WasmBoy³ este o bibliotecă pentru un emulator de jocuri scrisă în AssemblyScript pentru WebAssembly. O scurtă descriere a acestui site de benchmark poate fi găsită și pe site-ul oficial al WebAssembly⁴ la secțiunea de aplicații realizate în WebAssembly. Pe WasmBoy avem posibilitatea de a încărca propriile fișiere pe care să se realizeze benchmark-urile sau putem folosi exemplele predefinite. Pentru un test preliminar am rulat același exemplu și același număr de frame-uri din Google Chrome și din Mozilla Firefox pe sistemul de operare Windows 10. Rezultatele se pot observa în Figura 2.1 și Figura 2.2.

Rezultatele diferă destul de mult de la un browser la celălalt, un lucru de luat în calcul și pentru experimentele ce vor fi realizate de noi și analiza de după. O concluzie

³<https://wasmboy.app/benchmark/>

⁴<https://madewithwebassembly.com/showcase/wasmboy/>

la care am ajuns după acest test ar fi că WebAssembly este mai rapid și mai eficient decât JavaScript, cel puțin în cazul unui joc mic, în browser, de tip FPS.

În experimentele noastre, ne propunem să analizăm o altă parte a informaticii, cea de algoritmică. Vom compara nu doar viteza și eficiența programelor, ci și memoria folosită de fiecare în mai multe medii și condiții. Alături de exemplul folosit în acest subcapitol, în final, vom putea analiza două arii diferite în care putem folosi WebAssembly dar și JavaScript, să vedem ce avantaje și ce dezavantaje are fiecare limbaj de programare și pe care îl putem alege în funcție de nevoile noastre.

2.3 Concluzii

În acest capitol am analizat câteva lucrări și aplicații deja existente, gratuite, legate de comparația de performanță între WebAssembly și JavaScript. Există mult mai multe studii pe această temă, unele mai avansate față de celelalte, însă toate au la bază dorința de a explora și exploata capacitățile acestui nou limbaj de programare, WebAssembly. Aceste rezultate ne vor ajuta în realizarea propriilor experimente și aducerea unui plus de valoare față de lucrările de referință.

WasmBoy Benchmarking

[Fork Me on Github](#)
[In-Depth Article and Results](#)

WasmBoy is configured with: audioBatchProcessing, audioAccumulateSamples, tileCaching
 Source is not minified, to allow easy analysis of the bundle.

Current Environment	
Browser	Chrome 113.0.0
Operating System	Windows 10
WasmBoy Lib Version	0.7.1

Setup

Enable Other Online GameBoy Emulators ([Binjb](#), [GameBoy Online](#)): ☐

Current ROM: Back to Color


Runner

Frames to run:


WasmBoy (Web Assembly, Assemblyscript)
 Frames Run: 2500
 Current FPS Average: 394



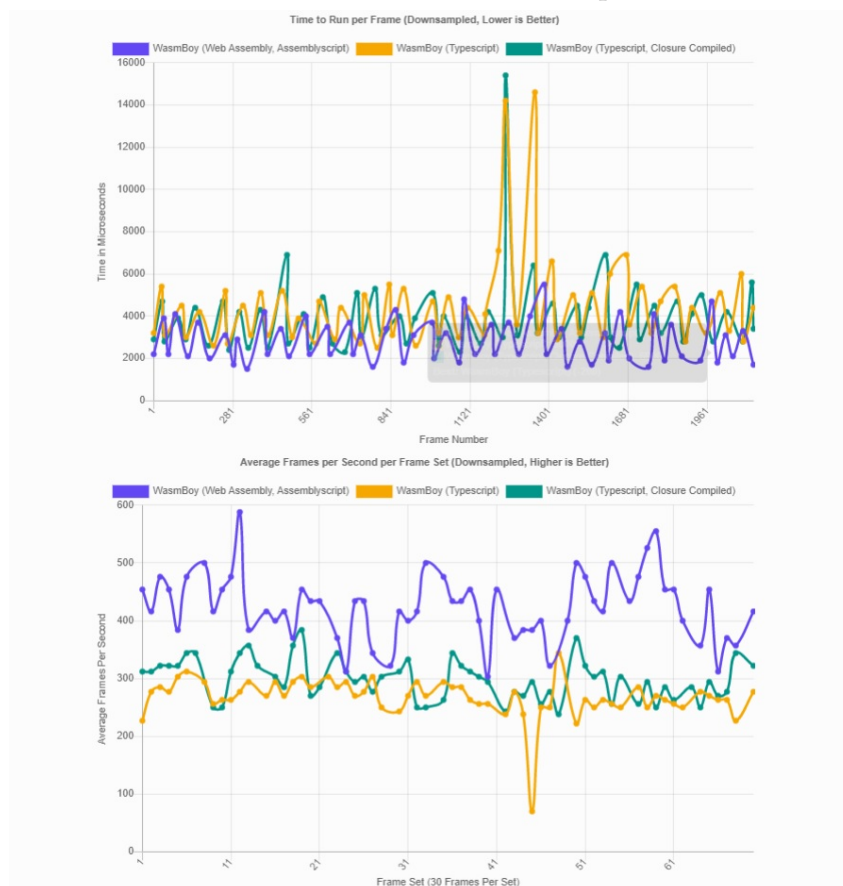
WasmBoy (Typescript)
 Frames Run: 2500
 Current FPS Average: 257



WasmBoy (Typescript, Closure Compiled)
 Frames Run: 2500
 Current FPS Average: 290



(a) Mediul de rulare și rezultate după rulare



(b) Grafice pentru timp și viteză

Figura 2.1: Rezultate WasmBoy pe Google Chrome

WasmBoy Benchmarking

[Fork Me on Github](#)
[In-Depth Article and Results](#)

WasmBoy is configured with: audioBatchProcessing, audioAccumulateSamples, tileCaching
 Source is not minified, to allow easy analysis of the bundle.

Current Enviroment	
Browser	Firefox 113.0.0
Operating System	Windows 10
WasmBoy Lib Version	0.7.1

Setup

Enable Other Online GameBoy Emulators ([Binjb](#), [GameBoy Online](#)): ☐

Current ROM: Back to Color

Runner

Frames to run:

WasmBoy (Web Assembly, Assemblyscript)

Frames Run: 2500
Current FPS Average: 296



WasmBoy (Typescript)

Frames Run: 2500
Current FPS Average: 96

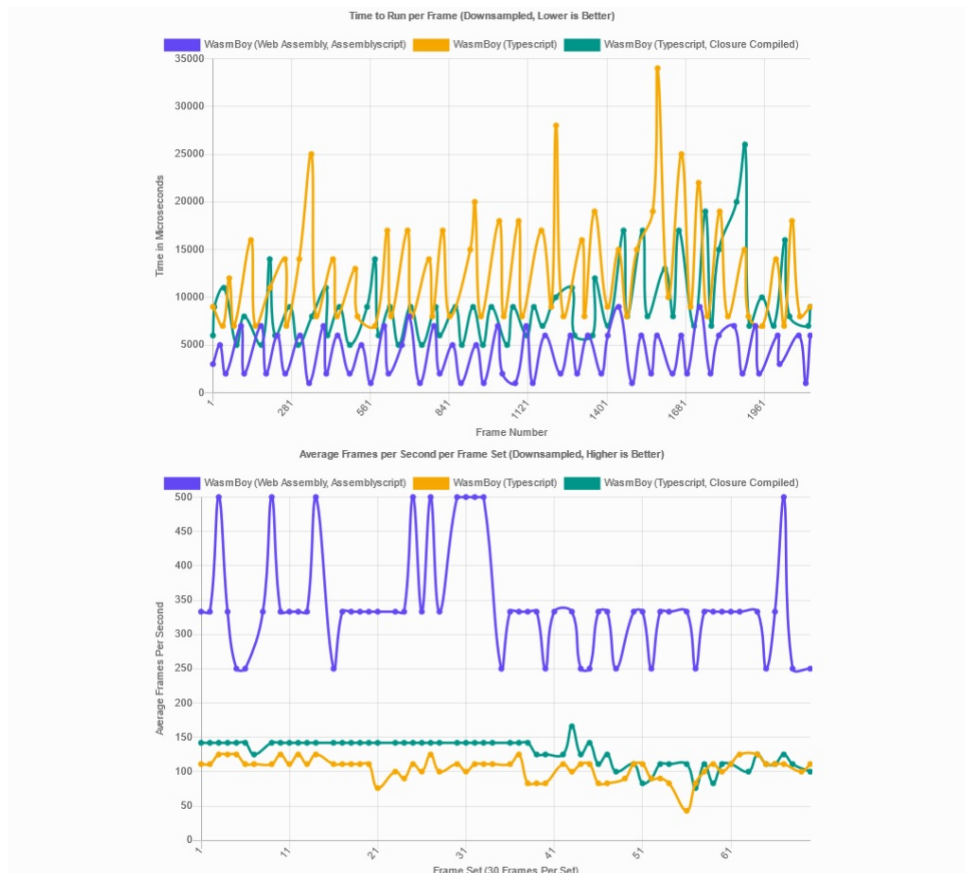


WasmBoy (Typescript, Closure Compiled)

Frames Run: 2500
Current FPS Average: 122



(a) Mediul de rulare și rezultate după rulare



(b) Grafice pentru timp și viteză

Figura 2.2: Rezultate WasmBoy pe Mozilla Firefox

Capitolul 3

Algoritmi comparați

În acest capitol vom descrie succint algoritmi folosiți în experimentele noastre. Am ales algoritmi din mai multe arii ale informaticii, toți studiați în timpul facultății. Vom explica ce reprezintă, cum funcționează, utilitatea lor în aplicații reale și vom implementa un pseudocod pentru a prezenta pașii algoritmului. Nu au existat criterii speciale de alegere a algoritmilor. Ei nu se află în prim planul lucrării prezente, ci reprezintă doar un parametru pentru testele noastre. Am ales să folosim algoritmi pentru comparații deoarece facilitează realizarea acestora, sunt ușor de folosit, înțeleși și utilizat și pentru că aceștia stau, de fapt, la baza oricărei aplicații.

3.1 Calculul factorialului

Problema calculului factorialului unui număr este extrem de importantă în combinatorică și probabilistică. Este folosită pentru a găsi coeficientul binomial¹ al unei ecuații, în formula de calcul a combinațiilor, alte formule matematice cum ar fi teorema lui Taylor². Din punct de vedere matematic, factorialul se calculează după formula:

$$N! = \prod_{i=1}^N i$$

Acest algoritm are complexitate liniară $O(n)$ și realizează înmulțiri cu numere mari care nu sunt ușor de făcut, motiv pentru care am considerat că acest algoritm este de ajutor pentru experimentele ce le vom realiza. Implementarea acestui algoritm este destul de simplă dacă nu aducem optimizări calculelor realizate. Mai jos am ales să prezentăm algoritmul printr-un pseudocod, pentru a fi independent de limbaj.[13]

¹Fowler David, The Binomial Coefficient Function, 1996

²Teorema lui Taylor


```

1 INPUT: N
2 OUTPUT: N!
3 function Factorial(N)
4     result <- 1
5     for i <- 1 to N do:
6         result <- result * i
7     end for
8     return result
9 end function

```

3.2 Înmulțirea a două matrici

Înmulțirea a două matrici este o operație matematică de algebră liniară destul de ușoară, și totuși, în informatică a reprezentat o tematică de cercetare interesantă, mai ales în privința eficienței acestui calcul. Complexitatea algoritmului clasic este de $O(n^3)$, însă există și metode mai eficiente cum ar fi algoritmul lui Strassen³ de înmulțire a două matrici. Am ales acest algoritm deoarece reprezintă o problemă destul de simplă din punct de vedere matematic, calculându-se după formula:

$$C_{ij} = \sum_{k=1}^n A_{ik} * B_{kj}, \forall i, j \in [1, n]$$

Implementarea algoritmului este simplistă, cu o complexitate mare, ceea ce poate evidenția cum se descurcă fiecare limbaj când are de executat un număr mare de operații. Un exemplu de pseudocod este prezentat mai jos.

```

1 INPUT: A, B, n % 2 matrici de dimensiune n x n
2 OUTPUT: C % matricea rezultat
3 function Inmultire_Matricii(A, B, n)
4     for i <- 0 to n do:
5         for j <- 0 to n do:
6             C[i, j] <- 0
7             for k <- 0 to n do:
8                 C[i, j] <- C[i, j] + A[i, k] * B[k, j]
9             end for
10        end for
11    end for
12    return C
13 end function

```

³Cache-Oblivious Algorithms

3.3 Căutare binară

Algoritmul căutării binare poate fi privit ca un joc în care scopul este să ghicești un număr dintr-o listă ordonată, punând o singură întrebare: „X este mai mic/mare decât numărul căutat?”. Știind dimensiunea listei, la fiecare răspuns se înjumătățesc variantele posibile rămase.[11]

Astfel, acest algoritm are o complexitate de $O(\log_2 n)$. Căutarea binară este un algoritm simplu, eficient, fiind foarte folosit în practică. Acestea au fost motivele la baza alegerii pentru experimentele noastre. O implementare posibilă poate fi văzută în pseudocodul următor:

```
1 INPUT: array , x, n % array - lista sortata , x - numarul c utat , n -  
    dimensiunea listei  
2 OUTPUT: True/False  
3 function Cautare_Binara(array , x, n)  
4     low <- 0  
5     high <- n  
6     while low <= high do:  
7         middle <- (low + high) / 2  
8         if x < array[middle] then:  
9             low <- middle + 1  
10        else if array[mid] > x then:  
11            high <- middle - 1  
12        else:  
13            return True  
14        end if  
15    return False  
16    end while  
17 end function
```

3.4 MD5

Criptografia este o arie a informaticii extrem de importantă și dezvoltată. Ea este primordială în absolut orice aplicație pentru a asigura autentificarea, autorizarea, integritatea datelor, dar și în securitatea protocoale de transfer de date, a rețelilor de calculatoare, a dispozitivelor folosite de noi zilnic etc. Algoritmul MD5 are rolul de a primi un mesaj de lungime variabilă și de a returna un șir de caractere de lungime fixă, unic aproape mereu (a se vedea posibilitatea coliziunilor pentru un algoritm de

hashing ⁴).

Acest algoritm complicat se poate rezuma în trei etape simple: etapa de pre-procesare unde se prelucrează mesajul de intrare și se transformă într-un șir de biți, partea de aplicare a unor operații matematice și logice pentru a obține un alt șir de biți ce reprezintă de fapt valoarea mesajului criptat și în final, convertirea șirului de biți obținut într-o reprezentare ușor de utilizat. Am ales acest algoritm deoarece este un algoritm ușor de implementat și utilizat față de ceilalți algoritmi criptografici care sunt mai complecși din natura lor de a oferi o securitate mai bună. MD5 a fost folosit foarte mult în perioada anilor 1990-2000 când tehnologia nu era așa dezvoltată pentru a putea găsi și exploata vulnerabilitățile acestuia.[15] Cu toate acestea, având o implementare destul de lungă și complicată, mai jos vom prezenta un pseudocod explicat în cuvinte pentru acest algoritm:

```
1 INPUT: message % mesajul ce urmeaza a fi criptat
2 OUTPUT: hash % mesajul hash-uit
3
4 function MD5(message)
5     Preprocess the input message
6     Initialize the MD5 state variables (A, B, C, and D)
7     Process each 512-bit block of the input message
8     Concatenate the state variables (A, B, C, and D) to form the
9         final hash
10    Return the hash as a 32-character hexadecimal string
11 end function
```

3.5 BFS

Breadth-First Search, abreviat BFS, este un algoritm extrem de important în teoria grafurilor, ce stă la baza multor alți algoritmi importanți precum: A*, Best-First Search, Bidirectional Search, Flood Fill, Minimum Spanning Tree etc. De asemenea, teoria grafurilor își are o utilitate foarte mare în aplicațiile reale din diferite domenii, folosite de noi zilnic: rețele sociale, aplicații de navigație, rețele de comunicații etc.

Considerăm ca acest algoritm este potrivit pentru experimentele noastre datorită utilității sale. Complexitatea algoritmului este variată în funcție de modul de reprezentare, de exemplu, pentru grafuri reprezentate prin liste de adiacență complexitatea

⁴Cryptanalysis of md5

timp este $O(E + V)$, unde E reprezintă numărul de muchii și V reprezintă numărul de noduri, iar pentru grafuri reprezentate prin matrici de adiacență, complexitatea timp este $O(V^2)$. [10] Varianta clasică a algoritmului BFS am prezentat-o prin pseudocodul următor:

```
1 INPUT: vertex , edges , start % vertex – lista nodurilor , edges – listele de
    adiancenta , start – pozitia nodului de start
2 OUTPUT: queue % ordinea de parcurgere a nodurilor din nodul de start
3 function BFS(vertex , edges , start)
4     visited <- empty list
5     queue <- empty list
6     first <- vertex[start]
7     visited[first] = True
8     queue.add(first)
9
10    for i <- 0 to vertex.size() do:
11        if visited[i] == False then:
12            for j in edges[i] do:
13                visited[j] = True
14                queue.add(j)
15            end for
16        end if
17    end for
18    return queue
19 end function
```

3.6 Problema colorării unui graf cu abordarea Greedy

Problema colorării unui graf este un subiect foarte cercetat chiar și în prezent. În linii mari, această problemă răspunde la întrebarea: „Care este numărul minim de culori necesare pentru a colora nodurile unui graf, astfel încât oricare două noduri vecine nu au culori asemănătoare.”

Această problemă, aparent simplă, face parte din categoria problemelor NP-hard, ce se rezolvă în timp polinomial, printr-o aproximare a unei soluții folosind diferite euristici. Abordarea Greedy a algoritmului este cea mai fundamentală metodă pentru rezolvarea acestei probleme, care poate oferi mai multe soluții optime într-un timp destul de scurt. [12] Pe lângă cele enumerate mai sus, Greedy este o cunoscută paradigmă de programare, folosită pentru rezolvarea mai multor probleme, un motiv în

plus pentru a alege o astfel de abordare în experimentele ce le vom realiza.

Mai jos, vom prezenta implementarea clasică a algoritmului de colorare a unui graf cu abordarea Greedy, fără nicio optimizare:

```
1 INPUT: vertex , edges % vertex – lista nodurilor , edges – listele de
    adiacenta
2 OUTPUT: coloring % lista culorile corespunzatoare fiecarui nod
3 function Graph_Coloring(vertex , edges)
4     result <- empty list % initializata cu -1, cu semnificatia ca niciun
    nod nu are asignata o culoare
5     available <- empty list % initializata cu True, cu semnificatia ca
    toate culorile sunt disponibile
6     result[0] <- 0 % asignam prima culoare primului nod
7     for i <- 0 to vertex.size() do:
8         for j <- 0 to edges[i] do:
9             if result[j] = -1 then:
10                 available[result[j]] <- False
11             end if
12         end for
13
14         for c <- 0 to vertex.size() do:
15             if available[c] = True then:
16                 stop
17             end if
18         end for
19         result[i] <- c
20         reset available
21     end for
22     return result
23 end function
```

3.7 Problema rucsacului abordarea cu Programare Dinamică

Ideea de la care a plecat problema rucsacului își are de fapt baza în activitățile comerciale și industriale zilnice: dorința de a fi eficienți cu spațiul de stocare și resursele folosite. Pentru niște cantități mici, noi oamenii ne descurcăm fără probleme, dar când vorbim de cantități foarte mari, puterea de calcul a calculatorului vine în ajutor.

Pe scurt, în enunțul problemei se oferă următoarele informații: avem un rucsac de o anumită capacitate și mai multe obiecte de o anumită greutate și valoare. Noi trebuie să determinăm ce obiecte putem pune în rucsac pentru a maximiza valoarea acestora.

Această problemă, alături de alte 20 de probleme, este catalogată ca fiind NP-Complete, de către Richard Karp, în anul 1972⁵. Prin abordarea cu programare dinamică, problema rucsacului are complexitatea $O(nC)$, unde C este vectorul de capacități.[14]

Astfel, această problemă stă de fapt la baza multor aplicații ce ajută cu gestionarea stocului, obiectelor, aranjarea lor pe rafturi, în depozite etc., motiv pentru care o eficiență mai mare ar fi de folos. Din această cauză, am considerat ca algoritmul acesta, deși unul nu foarte complicat, ar fi un subiect bun pentru test în experimentele noastre. Pseudocodul pentru această problemă este prezentat mai jos:

```
1 INPUT: C, N, V % N – lista de capacitati , V – lista de valori , C –  
   capacitatea rucsacului  
2 OUTPUT: result % valoarea maxima pentru capacitatea rucsacului  
3 function Unbounded_Knapsack(C, N, V)  
4     result[0] <- 0  
5     for i <- 1 to C do:  
6         result[i] <- result[i - 1]  
7         for j <- 0 to N.size() do:  
8             if N[j] <= c then:  
9                 result[c] <- max(result[c], result[c - N[i]] + V[i])  
10            end if  
11        end for  
12    end for  
13    return result[C]  
14 end function
```

3.8 Concluzii

Algoritmii aleși acoperă câteva arii ale informaticii și câteva paradigme de programare. Toți algoritmii efectuează calcule mai complicate sau mai simple, au complexități de timp și de memorie variate, lucru care ne ajută în analiza rezultatelor și luarea unei concluzii obiective.

⁵Richard M. Karp. Reducibility among combinatorial problems, 1972

Capitolul 4

Procese

În acest capitol vom descrie în detaliu cum am creat proiectul pentru experimentele pe care le vom realiza, cum am împărțit proiectul în două module, cum rulează și compilează codul scris și cum vom calcula timpul de execuție al funcțiilor scrise în cele două limbaje de programare, AssemblyScript și JavaScript.

4.1 Crearea proiectului

Pentru partea de dezvoltare și creare a proiectului am folosit IDE-ul Visual Studio Code¹ ca mediu de lucru. Pentru a crea un proiect nou de AssemblyScript am urmărit pașii de pe pagina oficială AssemblyScript². După rularea comenzilor din primii pași din tutorial, proiectul arată ca în figura 4.1.

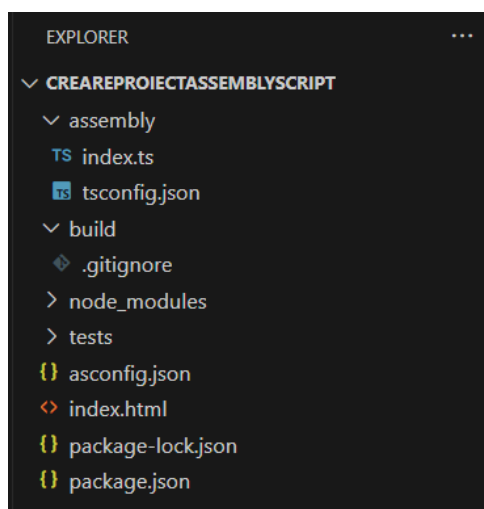


Figura 4.1: Structura inițială a unui proiect de AssemblyScript.

¹<https://code.visualstudio.com/>

²<https://www.assemblyscript.org/getting-started.html#setting-up-a-new-project>

Folder-ul „assembly” conține fișierele cu extensia .ts, fișiere de TypeScript, unde noi putem scrie cod în limbajul AssemblyScript ce va fi compilat mai apoi în limbajul WebAssembly. De asemenea, avem și un fișier de configurări pentru TypeScript, „tsconfig.json”, moștenite de AssemblyScript.

Folder-ul „build” este destinat fișierelor de rezultă în urma compilării codului scris în fișierele din folder-ul „assembly”. Acesta este inițial gol, însă după ce compilăm pentru prima dată, folderul va fi populat cu fișierele ce le putem observa în figura 4.2.

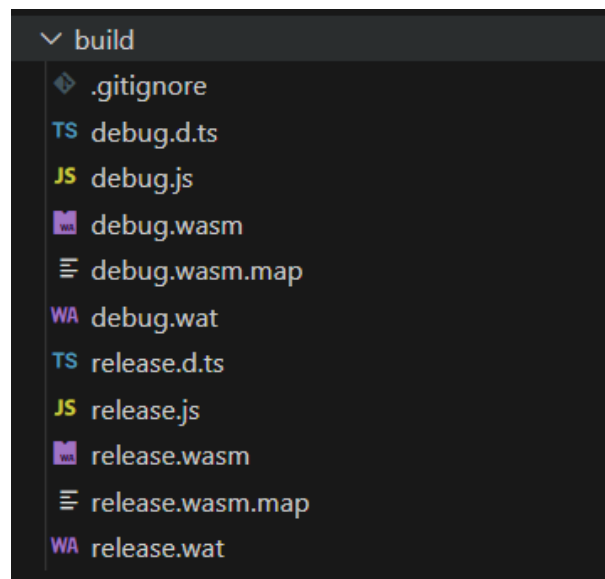


Figura 4.2: Fișierele din folder-ul „build”.

Atunci când creăm proiectul avem în fișierul „index.ts” o funcție de adunare a două numere întregi scrisă în AssemblyScript drept exemplu. Am păstrat această funcție pentru a descrie ce se întâmplă cu ea atunci când compilăm codul. În prima fază, observăm ca avem zece fișiere, dintre care cinci destinate pentru debug, iar altele cinci destinate pentru a fi folosite. Singura diferență între ele este că „debug.wat” are niște inițializări de variabile globale legate de folosirea memoriei. După cum se poate observa în figura 4.3, variabila globală „**__data_end**” are rolul de a stabili o limită a zonelor de memorie. Variabila „**__stack_pointer**” este utilizată pentru a marca adresa vârfului stivei de execuție. Iar variabila „**__heap_base**” reprezintă adresa de bază a regiunii de memorie heap, care este utilizată în alocarea dinamică a memoriei în timpul execuției programului.


```

build > WA debug.wat
1 (module
2   (type $i32_i32 => i32 (func (param i32 i32) (result i32)))
3   (global $~lib/memory/_data_end i32 (i32.const 8))
4   (global $~lib/memory/_stack_pointer (mut i32) (i32.const 32776))
5   (global $~lib/memory/_heap_base i32 (i32.const 32776))
6   (memory $0 0)
7   (table $0 1 1 funcref)
8   (elem $0 (i32.const 1))
9   (export "add" (func $assembly/index/add))
10  (export "memory" (memory $0))
11  (func $assembly/index/add (param $a i32) (param $b i32) (result i32)
12    local.get $a
13    local.get $b
14    i32.add
15    return
16  )
17 )
18

build > WA release.wat
1 (module
2   (type $i32_i32 => i32 (func (param i32 i32) (result i32)))
3   (memory $0 0)
4   (export "add" (func $assembly/index/add))
5   (export "memory" (memory $0))
6   (func $assembly/index/add (param $0 i32) (param $1 i32) (result i32)
7     local.get $0
8     local.get $1
9     i32.add
10    )
11 )
12

```

Figura 4.3: Diferența între „release.wat” și „debug.wat”.

Fișierele „debug.d.ts” și „release.d.ts” sunt fișiere de tipuri (TypeScript declaration file), unde se regăsesc declarațiile funcțiilor, claselor, tipurilor, modulelor etc. scrise în AssemblyScript, oferind astfel posibilitatea unui editor de cod pentru TypeScript să le înțeleagă și să le folosească. Prima declarație exportată în acest fișier este cea a variabilei „Memory” din WebAssembly care ne permite să accesăm memoria folosită. Așa cum se poate vedea în figura 4.4, avem în fișier și interpretarea pentru TypeScript a funcției scrisă în AssemblyScript, cu tipurile de date specifice acestui limbaj.

```

build > TS debug.d.ts > ...
1  /** Exported memory */
2  export declare const memory: WebAssembly.Memory;
3  /**
4   * assembly/index/add
5   * @param a `i32`
6   * @param b `i32`
7   * @returns `i32`
8   */
9  export declare function add(a: number, b: number): number;

```

Figura 4.4: Interpretarea funcției „add” din AssemblyScript în TypeScript.

Cele mai importante fișiere pentru aplicația noastră sunt „debug.js” și „release.js”, deoarece în aceste două fișiere se instanțiază modulul de WebAssembly și se exportă toate declarațiile din „debug.d.ts” și „release.d.ts” pentru a fi folosite în aplicația noastră. După cum vedem în figura 4.5, instanțierea modulului de WebAssembly are loc în mod asincron, apelând un URL ce este creat cu ajutorul fișierului „release.wasm” pe care îl vom analiza în curând. Această instanțiere poate dura mai mult sau mai puțin timp, fiind asincronă, acest fapt este un lucru destul de important de luat în calcul la rezultatele experimentelor noastre: în timpul execuției intră și timpul instanțierii modulului;

dacă modulul este încărcat dinainte, execuția codului este mult mai rapidă.

```
build > JS release.js > ...
1  async function instantiate(module, imports = {}) {
2    const { exports } = await WebAssembly.instantiate(module, imports);
3    return exports;
4  }
5  export const {
6    memory,
7    add,
8  } = await (async url => instantiate(
9    Complexity is 6 It's time to do something...
10   Complexity is 5 Everything is cool!
11   await (async () => {
12     try { return await globalThis.WebAssembly.compileStreaming(globalThis.fetch(url)); }
13     catch { return globalThis.WebAssembly.compile(await (await import("node:fs/promises")).readFile(url)); }
14   })(), {
15   })(new URL("release.wasm", import.meta.url));
```

Figura 4.5: Instantierea modului de WebAssembly.

Fișierele rămase, „release.wasm”, „release.wasm.map” și respectiv „debug.wasm” și „debug.wasm.map” sunt strâns legate între ele. Fișierele cu extensia .wasm reprezintă de fapt codul binar generat în urma compilării programului nostru scris în AssemblyScript. Acesta poate fi folosit în medii ce suportă WebAssembly, cum ar fi majoritatea browser-elor folosite zi de zi. Conținutul acestui fișier nu poate fi vizualizat dintr-un IDE cum este Visual Studio Code, nefiind nevoie de a ști conținutul său. Pentru a face debug pentru fișierul .wasm, avem fișierele .wasm.map, care au rolul de a ajuta anumite instrumente specializate de a face debug pe codul binar pentru WebAssembly, de a-l optimiza și chiar pentru a-l testa. Aceste fișiere nu au niciun rol în execuția codului propriu zisă, pot să nu existe în proiect, dar fără un fișier cu extensia .wasm nu au nici un sens sau utilitate. Ele pot fi deschise într-un IDE, însă sunt greu de citit și înțeles fără ajutor care este conținutul și ce face.

Ultimul fișier important din proiect este „asconfig.json”, un fișier de tip JSON³ ce conține diferite opțiuni de compilare, pentru fișierele de debug și pentru fișierele de release separat, fiind definite două „target”-uri. Cele mai importante opțiuni ce pot fi modificate în cadrul acestui fișier sunt: *--optimizeLevel*, această opțiune poate primi o valoare între [0-3] și reprezintă nivelul de optimizare aplicat la compilare; *--shrinkLevel*, reprezintă nivelul de micșorare a dimensiunii codului, poate primi valori între [0-2], și *--converge*, aceasta specifică compilatorului să reoptimizeze codul până cand nu mai există nicio îmbunătățire de făcut.⁴ De asemenea, putem specifica și stilul de folosire al importurilor și exporturilor, în figura 4.6 putem observa că la „bindings” avem

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

⁴<https://www.assemblyscript.org/runtime.html#variants>

opțiunea „esm” care specifică utilizarea modului ESM(ECMA Script Modules)⁵.

```
{ asconfig.json > ...
1  {
2    "targets": {
3      "debug": {
4        "outFile": "build/debug.wasm",
5        "textFile": "build/debug.wat",
6        "sourceMap": true,
7        "debug": true
8      },
9      "release": {
10       "outFile": "build/release.wasm",
11       "textFile": "build/release.wat",
12       "sourceMap": true,
13       "optimizeLevel": 3,
14       "shrinkLevel": 2,
15       "converge": true,
16       "noAssert": false
17     }
18   },
19   "options": {
20     "bindings": "esm"
21   }
22 }
```

Figura 4.6: Configurările de compilare pentru AssemblyScript.

4.2 Pregătirea testelor

Scopul acestei lucrări este de a compara aceste două limbaje de programare, WebAssembly și JavaScript, așadar avem nevoie să scriem aceleași funcții în ambele limbaje. Pentru funcțiile în JavaScript, am creat un nou modul pentru a le scrie și mai apoi pentru a le exporta și folosi pentru experimente. Am decis ca pentru a rula experimentele să nu folosim nicio bibliotecă specializată, acestea fiind inconsistente în analiza lor. Modul de testare al funcțiilor este destul de simplu, dar are o acuratețe mult mai mare, putem observa în figura 4.7. Folosind obiectul predefinit din JavaScript, „console”⁶, putem să folosim două metode pereche ale acestui obiect, „console.time()” și „console.timeEnd()”, pentru a vedea diferența de timp precisă a executării codului aflat între cele două apeluri de funcție. Pentru a vedea mai precis eficiența fiecărui limbaj, vom apela într-o instrucțiune de tip „for” funcția supusă la test pentru a o executa de un număr mai mare de ori, nu doar o singură dată. Astfel vom putea calcula precis timpul de execuție pentru fiecare funcție apelată de un număr mare de ori.

⁵<https://nodejs.org/api/esm.html>

⁶<https://developer.mozilla.org/en-US/docs/Web/API/console>

```

51 // Factorial benchmark
52 export function factorial_AssemblyScript(n: i32, p: i32): void {
53     const numberOfTests = 100;
54     console.time(`Time to run ${numberOfTests} factorial function call: `);
55     for (let i = 0; i < numberOfTests; i++) {
56         factorial(n, p);
57     }
58     console.timeEnd(`Time to run ${numberOfTests} factorial function call: `);
59 }

```

Figura 4.7: Modul de măsurare al timpului de execuție pentru o funcție.

Pentru a măsura memoria folosită la fiecare test, putem să folosim instrumentele special create pentru programatori integrate în fiecare browser. Vom lua exemplu Google Chrome, dar toate browser-ele folosite în experimentele noastre au acest instrument de măsurare a memoriei folosite. În Google Chrome, apăsând tasta F12 sau click dreapta la mouse și apoi „Inspect”, vom deschide DevTools⁷, unde avem mai multe ferestre, fiecare reprezentând un instrument ajutător în procesul de programare a unui WebSite. Printre aceste ferestre regăsim și „Memory”, un instrument ce ne arată exact câtă memorie este folosită pentru fereastra curentă a browser-ului, în cazul nostru, fereastra unde vom rula testele. De sigur, putem vedea foarte multe detalii legate de folosirea memoriei, însă cum noi rulăm doar niște teste, neavând alte elemente în pagină, memoria folosită este strict pentru execuția experimentului curent. Idem și pentru Mozilla Firefox, Microsoft Edge și Safari. Putem vedea în figura 4.8 cum arată exact această fereastră unde putem vedea memoria folosită.

4.3 Concluzii

În acest capitol am aflat cum se poate crea un proiect nou de AssemblyScript și ce reprezintă fiecare fișier specific din acest proiect. De asemenea, am prezentat și modul de testare al funcțiilor folosite în experimentele noastre, cu ajutorul metodelor predefinite din JavaScript dar și cu ajutorul instrumentelor ajutătoare din fiecare browser pe care vom realiza testele noastre.

⁷<https://developer.chrome.com/docs/devtools/>

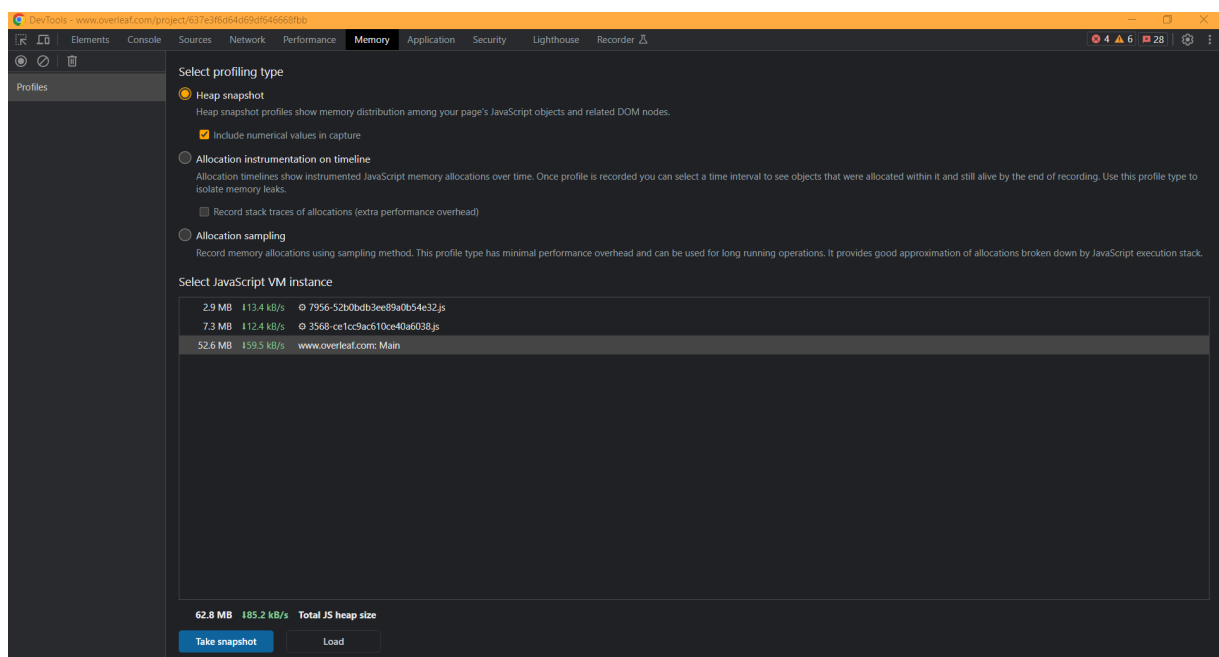


Figura 4.8: Modul de măsurare a memoriei folosite de execuția funcției testate.

Capitolul 5

Comparare și evaluare

În acest capitol vom aplica toate noțiunile prezentate anterior pentru a implementa algoritmi studiați, a realiza testele propuse și a analiza rezultatele, având mai apoi o discuție pe baza rezultatelor și a concluziona acest studiu. O condiție comună pentru rularea testelor este rularea fără o conexiune la internet, pe rețeaua locală, pentru a nu fi influențate de o conexiune mai slabă la internet. Vom rula câte un test pe rând, iar între teste vom reseta pagina pentru ca testul următor să fie făcut în exact aceleași condiții. De asemenea, toți timpii de execuție afișați în grafice pe axa verticală sunt în milisecunde.

5.1 Experimente pe sistemul de operare Windows

Aceste experimente vor fi realizat pe un laptop marca Dell, având un procesor Intel Core i5-8365U, cu 16GB memorie RAM și sistemul de operare Windows 10 Enterprise.

5.1.1 Google Chrome

Primul browser pe care vom rula testele noastre este Google Chrome, versiunea 114. Google Chrome are la bază proiectul Chromium¹, care este un motor de căutare open-source, supervizat de compania Google. Vom rula testele într-o fereastră Incognito pentru a fi mai eficient și a nu exista impedimente externe.

¹<https://www.chromium.org/Home/>

Măsurarea timpului de execuție

Am început prin a colecta date despre timpul de execuție al funcțiilor. Acești timpi de execuție au fost pentru 100 de rulări a fiecărei funcții. Între fiecare test am dat ferestrei deschise o reîmprospătare. Timpul de execuție rezultat este o medie a 4-5 timpi de execuție pentru fiecare funcție, fiind mai precis decât o singură executare a testului, timpii mereu fiind variabili de la o rulare la alta. Astfel, în figura 5.1 putem observa rezultatele de timpi de executare pentru cele 7 teste.

Windows - Google Chrome

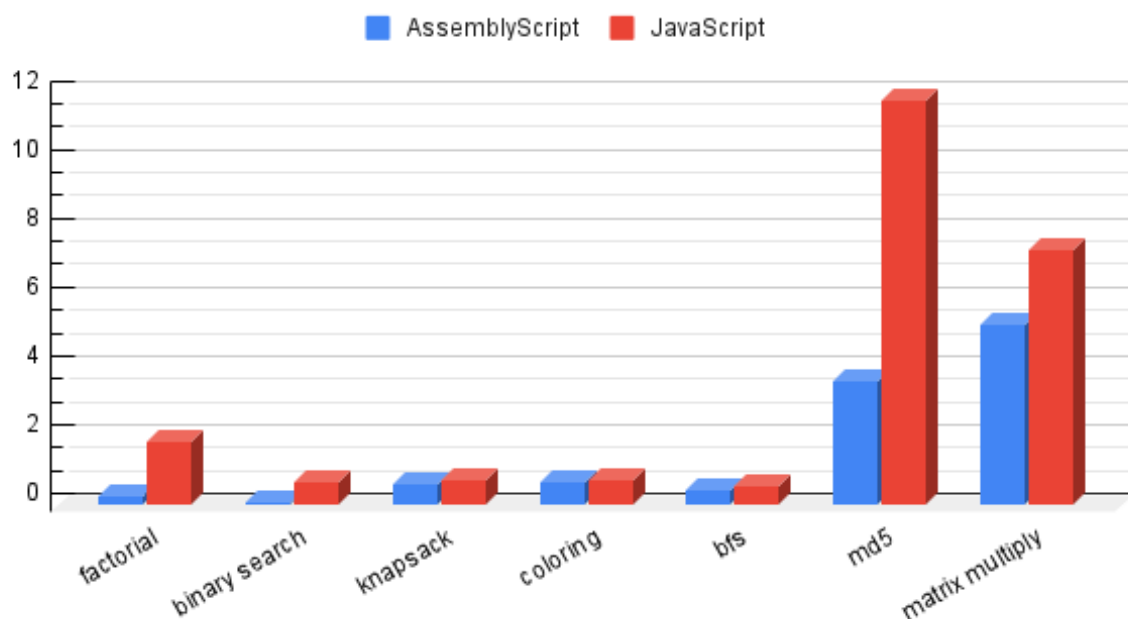


Figura 5.1: Grafic pentru timpii de execuție a testelor pe Google Chrome.

Măsurarea memoriei folosite

După colectarea datelor legate de timpii de execuție a testelor, ne-am folosit de instrumentul de măsurare al memoriei inclus în DevTools din Google Chrome. Acest instrument ne permite să dam start la o înregistrare în timp real a utilizării memoriei iar atunci când apelăm o funcție, putem vedea în timp real ce memorie a fost folosită pentru a se executa funcția. Acest proces se poate observa în figura 5.2.

Fiecare funcție a fost apelată o singură dată, mai multe apeluri ale aceleiași funcții nu influențează memoria folosită pentru executare. Acest lucru se întâmplă deoarece la o a doua apelare spațiul necesar rulării scade la doar câțiva bites datorită optimizării

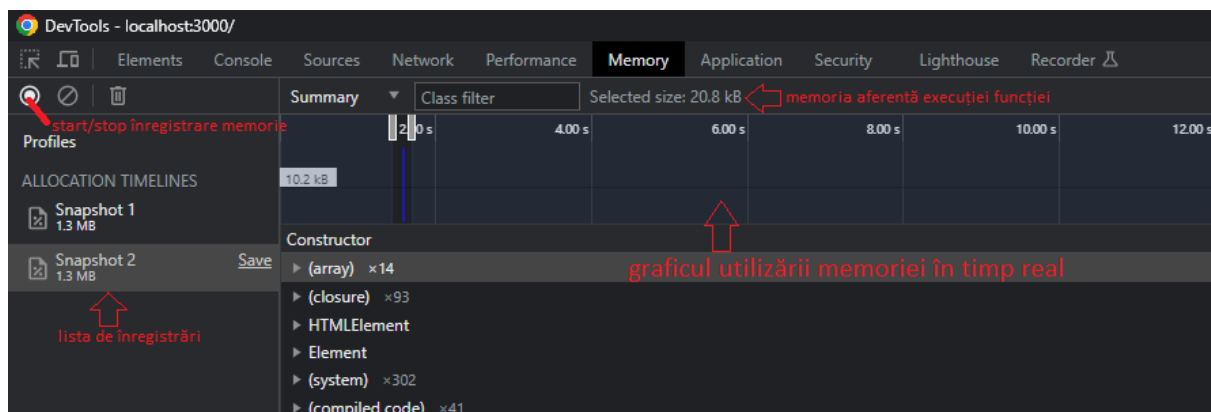


Figura 5.2: Instrumentul de măsurare al memoriei în timp real din DevTools.

motorului browser-ului de reutilizare și economisire a memoriei. Rezultatele obținute după măsurarea memoriei necesare fiecărei funcții pot fi văzute în tabelul 5.1.

Tabela 5.1: Utilizarea Memoriei pe browser-ul Google Chrome.

Funcția	AssemblyScript	JavaScript
Factorial	20.6kB	22.1kB
Căutare binară	21.0kB	26.7kB
Problema rucsacului	20.6kB	32.9kB
Colorarea unui graf	20.4kB	30.5kB
BFS	20.8kB	29.6kB
Înmulțire de matrici	20.9kB	29.9kB
MD5	22.1kB	38.7kB

5.1.2 Mozilla Firefox

În continuare, vom rula testele pe browser-ul Mozilla Firefox, versiunea 114. Acest browser are la bază motorul de navigare Gecko² care este, la fel ca și Chromium, un proiect open-source, doar că acesta este supervizat de Mozilla Corporation.[19] Și aici vom rula testele într-o fereastră privată pentru eficiență și pentru a nu exista vreun factor extern care să influențeze rezultatele.

²<https://wiki.mozilla.org/Gecko:Overview>

Măsurarea timpului de execuție

Datorită diferenței proiectului de bază, pentru realizarea experimentelor am modificat numărul de apeluri pentru fiecare funcție pentru a putea vedea un timp puțin mai detaliat. În consola browser-ului Mozilla Firefox, timpul de execuție este un număr întreg, față de Google Chrome unde timpul este număr real. Astfel, pentru anumite funcții, cum ar fi funcția de calcul a factorialului sau căutarea binară, am crescut numărul de teste pentru a putea avea o valoare puțin mai mare, diferită de zero. Însă, pentru funcțiile MD5 și înmulțirea a două matrici, numărul de apeluri de funcție a rămas același. În figura 5.3 putem vedea un grafic cu diferențele de timp și cu numărul modificat de apeluri de funcție, care se regăsește în dreptul numelui testului, de sub grafic.

Pentru Mozilla Firefox nu am mai analizat memoria folosită de fiecare funcție deoarece în acest moment nu există niciun instrument pentru a vedea în timp real cum este alocată memoria.

Windows - Mozilla Firefox

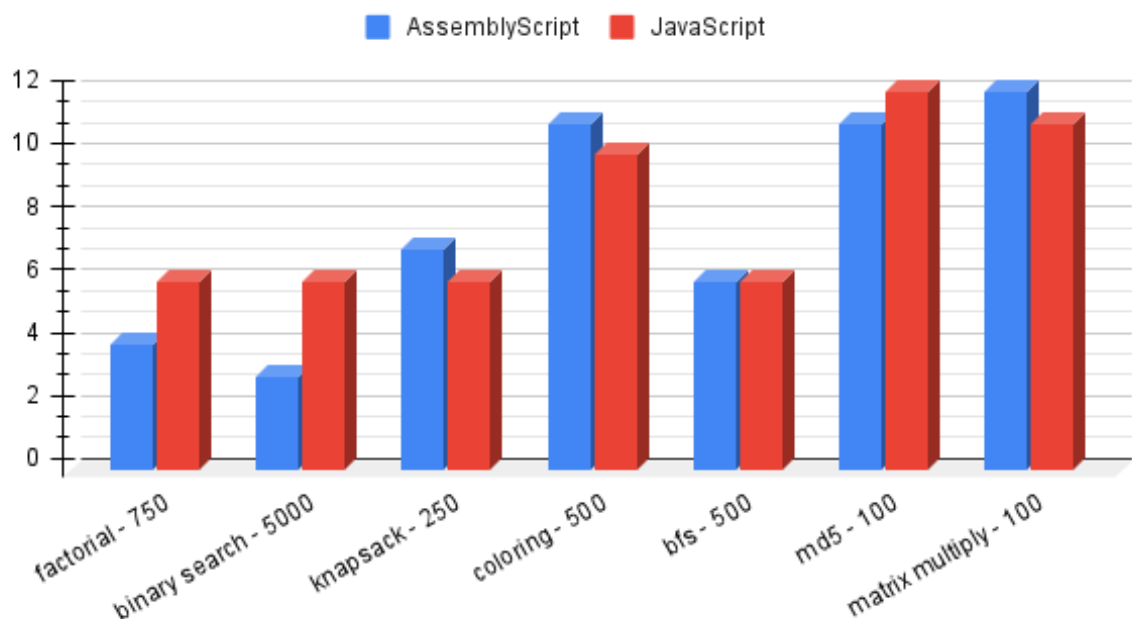


Figura 5.3: Grafic pentru timpii de execuție a testelor pe Mozilla Firefox.

5.1.3 Microsoft Edge

Ultimul browser pe care vom rula testele pentru Windows este Microsoft Edge, un browser specific Windows, fiind creat de Microsoft, însă poate fi folosit pe mai

multe sisteme de operare, și are la bază tot proiectul open-source Chromium, la fel ca și Google Chrome. Vom rula testele și pe acest browser, într-o fereastră privată, să vedem dacă există diferențe de optimizare între acestea.

Măsurarea timpului de execuție

În figura 5.4 putem observa rezultatele testelor pentru timpul de execuție în cadrul acestui browser. Valorile pentru aceste teste sunt de asemenea o medie aproximativă a 4-5 rulări pentru fiecare test, pentru a elimina un posibil timp de excepție, spre exemplu un timp mai mare de 0.1ms pentru testul căutării binare. Numărul de apeluri pentru fiecare funcție este același ca la secțiunea 5.1.1.

Windows - Microsoft Edge

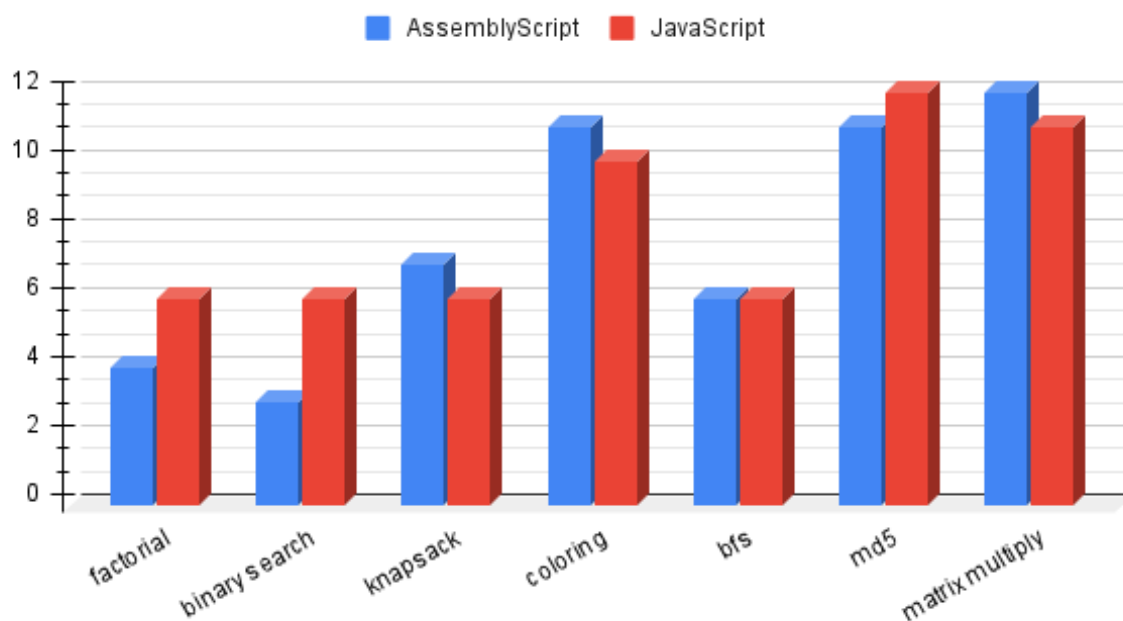


Figura 5.4: Grafic pentru timpii de execuție a testelor pe Microsoft Edge.

Măsurarea memoriei folosite

Procedeul de măsurare al memoriei folosite de fiecare apel de funcție este identic cu cel descris în secțiunea 5.1.1, iar în tabelul 5.2 putem observa valorile înregistrate pentru acest browser.

Tabela 5.2: Utilizarea Memoriei pe browser-ul Microsoft Edge.

Funcția	AssemblyScript	JavaScript
Factorial	21.1kB	23.6kB
Căutare binară	21.3kB	28.5kB
Problema rucsacului	20.7kB	33.4kB
Colorarea unui graf	21.2kB	30.7kB
BFS	20.7kB	29.8kB
Înmulțire de matrici	21.1kB	30.2kB
MD5	21.3kB	38.6kB

5.1.4 Analiza rezultatelor pentru sistemul de operare Windows

Având adunate toate datele necesare, putem face o analiză detaliată a rezultatelor pentru acest sistem de operare. Dorim să vedem cu cât este mai bună performanța funcțiilor scrise în AssemblyScript față de cele scrise în JavaScript, acolo unde este cazul, ce use-case-uri fac AssemblyScript-ul să iasă în evidență și o analiză sumară a memoriei utilizate în cadrul browser-ului pentru fiecare test.

Pentru timpii de execuție rezultați pe Google Chrome, AssemblyScript reușește să aibă o performanță de **6.59x** mai mare față de JavaScript pentru funcția de calcul a factorialului unui număr, de **5.37x** mai mare pentru funcția de căutare binară și de **2.96x** mai mare pentru algoritmul criptografic MD5. Din aceste date putem concluziona că AssemblyScript, în cazul calculelor complexe, matematice, logice, sau pe biți, are o performanță foarte bună.

În cazul lucrului cu vectori sau matrici, performanța scade iar acest lucru se poate observa la problema rucsacului, unde AssemblyScript este de **1.12x** mai rapid, la problema colorării unui graf unde este de **1.03x** mai rapid, pentru funcția BFS este de **1.02x** mai încet și funcția de înmulțire a două matrici unde performanța sa este de doar **1.26x** mai rapidă. Din aceste date putem concluziona că AssemblyScript nu excelează așa cum ne-am dori în cazul lucrului cu vectori mari, cu mai mulți vectori sau matrici.

Rezultatele obținute pe Microsoft Edge nu diferă cu mult față de cele obținute pe Google Chrome. Astfel, cazurile în care AssemblyScript are o performanță considerabil mai bună sunt: calculul factorialului unui număr, fiind de **7.43x** mai rapid, funcția de căutare binară, fiind de **7.22x** mai rapid și pentru algoritmul criptografic MD5 unde AssemblyScript este de **3.27x** mai rapid față de implementarea în JavaScript. În cazul

celorlalte implementări, performanța nu este extraordinară având următoarele valori: pentru problema rucsacului este de doar **1.17x** mai rapid, pentru problema colorării unui graf este de **1.09x** mai rapid, pentru BFS este de **1.21x** mai rapid și în final, pentru înmulțirea a două matrici, AssemblyScript este de **1.41x** mai rapid față de implementarea funcției în JavaScript.

Chiar dacă Google Chrome și Microsoft Edge au la bază Chromium, putem vedea că pe Microsoft Edge performanța este puțin mai bună, cu o medie de **1.186x** mai rapid.

Pentru analiza rezultatelor obținute în urma rulării testelor pe Mozilla Firefox putem trage o singură concluzie: cu cât mărim numărul de apeluri de funcție executate, performanța AssemblyScript scade sub nivelul de performanță al JavaScript sau se apropie de acesta. În acest caz, pentru funcția de calcul a factorialului, crescând numărul de apeluri de la 100 la 750, performanța AssemblyScript a ajuns la același nivel cu cea a JavaScript. Căutarea binară a scăzut de la o eficiență de **7.22x** pentru 100 de apeluri pe Microsoft Edge, la o eficiență de doar **2x** pe Mozilla Firefox pentru 5000 de apeluri de funcție. Pentru problema rucsacului, a colorării unui graf și pentru algoritmul BFS, timpii de execuție odată cu creșterea numărului de apeluri au ajuns să fie aproximativ egali, cu o variație de o milisecundă.

Rezultatele neașteptate apar în cazul funcției de înmulțire a două matrici și a algoritmului MD5, unde chiar dacă numărul de apeluri a rămas identic, performanța pentru AssemblyScript este aproximativ egală cu cea a funcțiilor scrise în JavaScript dar cu mult mai mică față de Google Chrome și Microsoft Edge, de până la **3.24x** mai mică, pentru același test realizat în aceleași condiții. Din aceste date, putem afirma că Mozilla Firefox, respectiv proiectul de bază, Gecko, nu sunt foarte optimizate pentru WebAssembly pe acest sistem de operare.

Pentru memoria utilizată în cadrul browser-ului, contrar așteptărilor inițiale, WebAssembly are nevoie de mult mai puțin spațiu de memorie față de JavaScript, așa cum putem vedea în tabelul 5.1. Deoarece valorile rezultate pe Google Chrome și pe Microsoft Edge sunt aproape identice, vom analiza doar un set de date.

Astfel, observăm că AssemblyScript, pentru fiecare funcție din setul nostru de date are nevoie între 20.4kB și 22.1kB, în această cantitate de memorie, nu intră și inițializarea modului de WebAssembly ce are loc la încărcarea paginii, o singură dată. JavaScript în schimb folosește între 22.1kB și 38.6kb memorie pentru funcțiile din setul nostru de teste. Un exemplu concret poate fi rularea algoritmului MD5, unde pentru implementarea în AssemblyScript este nevoie doar de 22.1kB memorie la ape-

lare iar pentru implementarea în JavaScript este nevoie de 38.7kB, care este de **1.75x** mai mare. În cazul acesta, această diferență de memorie este neglijabilă. însă într-un sistem mult mai mare, ar putea conta foarte mult pentru performanța aplicației noastre.

5.2 Experimente pe sistemul de operare MacOS

Aceste seturi de experimente vor fi rulat pe un laptop marca Apple, model MacBook Pro, având un procesor Apple M1 Max, cu 64GB memorie RAM și sistemul de operare MacOS Ventura 13.4.

5.2.1 Google Chrome

Primul browser pe care vom rula testele pe acest sistem de operare este Google Chrome, versiunea 114. Vom executa experimentele folosind o fereastră de tip Incognito pentru izolarea sesiunii curente.

Măsurarea timpului de execuție

Primele rezultate pe acest sistem de operare legate de timpul de execuție se pot observa în figura 5.5. Fiecare test are același număr de apeluri, la fel ca și pe sistemul de operare Windows, și anume 100 de apeluri de funcție.

Măsurarea memoriei folosite

Pentru a măsura memoria alocată de browser fiecărei funcții ne vom folosi de același instrument predefinit în DevTools, cu care vom putea vedea în timp real exact câtă memorie necesită fiecare apel de funcție. Astfel, rezultatele obținute se pot vedea în tabelul 5.3.

MacOS - Google Chrome

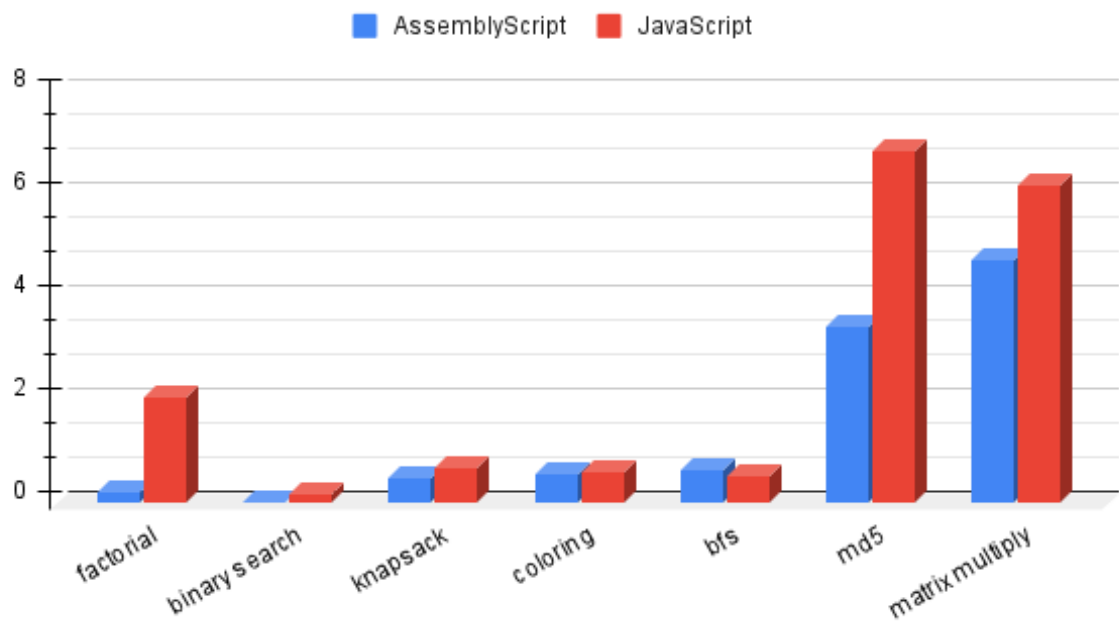


Figura 5.5: Grafic pentru timpii de execuție a testelor pe Google Chrome.

Tabela 5.3: Utilizarea Memoriei pe browser-ul Google Chrome.

Funcția	AssemblyScript	JavaScript
Factorial	20.6kB	22.2kB
Căutare binară	21.0kB	25.8kB
Problema rucsacului	20.6kB	24.6kB
Colorarea unui graf	20.5kB	27.2kB
BFS	20.6kB	27.1kB
Înmulțire de matrici	21.2kB	29.1kB
MD5	21.1kB	29.8kB

5.2.2 Mozilla Firefox

În continuare, vom rula testele pe browser-ul Mozilla Firefox, versiunea 114. Testele vor fi rulate într-o fereastră privată pentru a izola sesiunea curentă și pentru a avea rezultate cât mai precise.

Măsurarea timpului de execuție

Și pe acest sistem de operare, testele realizate pe Mozilla Firefox au fost adaptate cu un număr mai mare de apeluri de funcție pentru a putea vedea un timp de execuție mai exact, acesta fiind un număr întreg. Numărul de apeluri pentru fiecare funcție este scris în figura 5.6 în dreptul numelui funcției de sub grafic.

MacOS - Mozilla Firefox

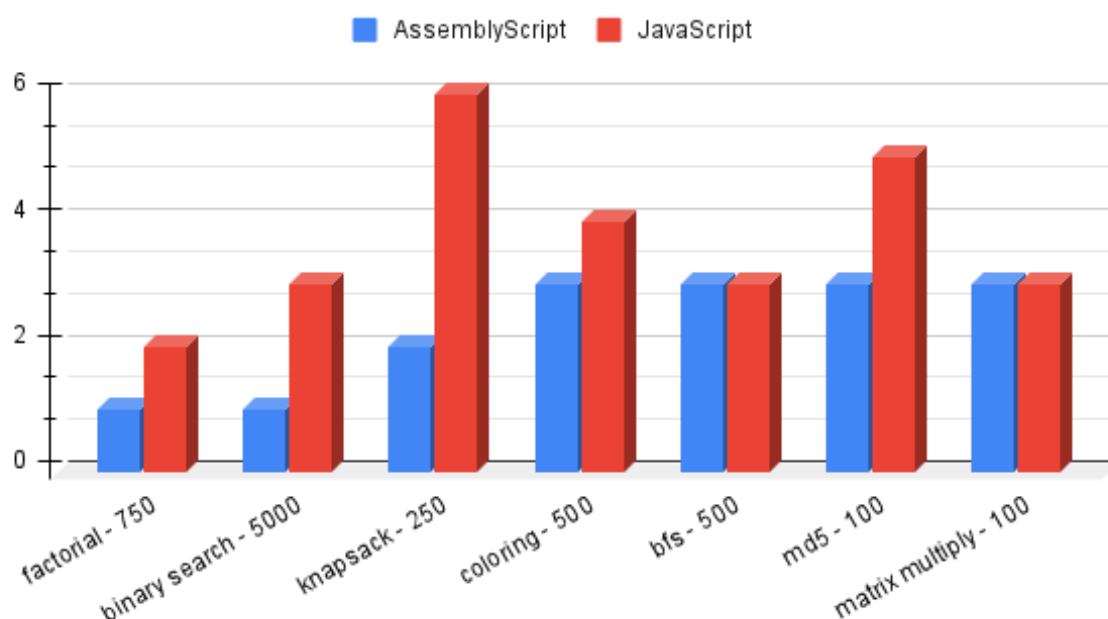


Figura 5.6: Grafic pentru timpii de execuție a testelor pe Mozilla Firefox.

5.2.3 Safari

Vom încheia setul de teste pe sistemul de operare MacOS pe browser-ul Safari, versiunea 16. Safari este un browser dezvoltat de compania Apple, specific pentru iOS și MacOS. Acest browser se remarcă față de celelalte browser-e prin mai multe caracteristici specifice cum ar fi: performanța remarcabilă și viteza oferită utilizatorilor, integrarea perfectă în ecosistemul Apple și securitatea implicită în cadrul browser-ului. Acesta utilizează o tehnologie remarcabilă numită Sandbox³ care are rolul de a limita accesul aplicațiilor și paginilor web la resursele sistemului, astfel performanța dispozitivului rămâne la un nivel mare când avem deschis și Safari.

³<https://developer.apple.com/documentation/xcode/configuring-the-macos-app-sandbox/>

Măsurarea timpului de execuție

Vom rula testele de măsurare a timpului de execuție într-o fereastră de navigare privată, pentru a izola sesiunea curentă și a nu exista nicio influență asupra testelor noastre. Rezultatele obținute se pot observa în figura 5.7.

MacOS - Safari

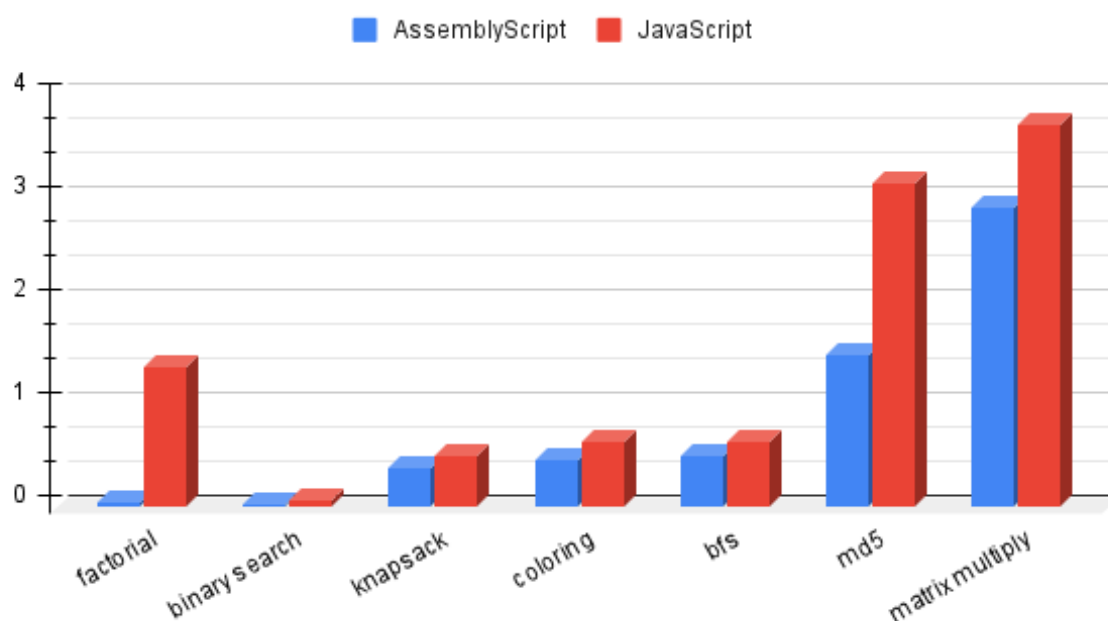


Figura 5.7: Grafic pentru timpii de execuție a testelor pe Safari.

5.2.4 Analiza rezultatelor pentru sistemul de operare MacOS

După adunarea tuturor informațiilor necesare din urma testelor realizate pe acest sistem de operare, putem realiza o analiză detaliată a diferențelor de performanță dintre AssemblyScript și JavaScript. Ne dorim să vedem diferențele dintre cele două limbaje în cadrul aceluiași browser, diferențele rezultatelor între browser-e diferite dar și o comparație față de rezultatele de la secțiunea 5.1.

Primul set de teste a fost realizat pe Google Chrome, iar rezultatele pot fi observate în figura 5.5. AssemblyScript are și pe acest sistem de operare o performanță foarte bună în cazul următoarelor funcții: pentru calculul factorialului unui număr este de **9.02x** mai mare față de JavaScript, funcția de căutare binară este de **6.12x** mai rapidă decât implementarea în JavaScript, iar algoritmul criptografic MD5 reușește să calculeze mesajul criptat, cu implementarea în AssemblyScript, de **2.85x** mai repede decât

implementarea în JavaScript.

De asemenea, performanța nu este cu mult ridicată în cazul funcțiilor unde se folosesc vectori și matrici, astfel, după o analiză a timpilor de execuție observăm că pentru problema rucsacului, AssemblyScript este de **1.35x** mai rapid, la problema colorării unui graf timpii sunt aproape identici, fiind diferență insesizabilă de doar 0,011ms. Totuși, pentru algoritmul BFS, AssemblyScript este cu 0.1ms mai încet față JavaScript, iar pentru funcția de înmulțire a două matrici, AssemblyScript este de **1.3x** mai rapid.

Fiind la capitolul de analiză a timpilor obținuți pe Google Chrome, putem face o comparație între timpii obținuți de funcțiile scrise în AssemblyScript între cele două sisteme de operare pentru a vedea cum se descurcă în medii diferite. Pe sistemul de operare MacOS au o performanță medie de **1.32x** mai mare față de implementarea pe Windows următoarele funcții: calculul factorialului unui număr, funcția de căutare binară, problema rucsacului, înmulțirea a două matrici și algoritmul MD5. Funcțiile de colorare a unui graf și BFS sunt mai lente pe MacOS, dar cu o diferență insesizabilă.

Următorul browser pe care am rulat experimentele noastre a fost Mozilla Firefox, unde față de rezultatele obținute pe Windows, rezultatele obținute pe MacOS sunt mult mai bune. Avem următoarele date: AssemblyScript a scos 2ms pentru funcția de calcul a factorialului, iar JavaScript 2ms pentru căutarea binară, avem 1ms față de 3ms; pentru problema rucsacului, AssemblyScript a scos un timp de 2ms față de JavaScript cu 6ms; problema colorării unui graf are o diferență mai mică, de doar 1ms între cele două implementări, iar BFS are același timp de execuție, de 3ms, în ambele limbaje, la fel ca și înmulțirea a două matrici; algoritmul MD5 este cu 2ms mai încet în implementarea în JavaScript.

Aceste rezultate se apropie ca diferențe între cele două limbaje cu testele rulate pe Windows pe browser-ul Mozilla Firefox, însă pe MacOS timpul de execuție pentru ambele limbaje este mult mai mic. Vom analiza doar implementarea în AssemblyScript iar pentru acest limbaj avem următoarele date: implementarea funcției de calcul a factorialului unui număr este de **5x** mai rapidă pe MacOS, algoritmul de căutare binară este de **3x** mai rapid, problema rucsacului este de **3.5x** mai rapidă pe MacOS față de Windows, și BFS este tot doar de **2x** mai rapidă. Rezultatele incredibile se remarcă la problema colorării unde pe MacOS avem un timp de 3ms iar pe Windows un timp de 11ms; înmulțirea a două matrici are un timp de 3ms pe MacOS și 12ms pe Windows și ultima funcție, MD5, are 3ms pe MacOS și 11ms pe Windows. Niste diferențe remarcabile ce scot în evidență faptul că MozillaFirefox este mai optimizat pe MacOS față de

Windows.

Ultimul browser pe care am realizat experimentele noastre, este Safari, browser-ul specific MacOS și iOS. Pentru acest browser nu vom avea posibilitatea de comparare cu rezultatele de pe alte sisteme de operare, dar putem compara cu rezultatele obținute pe Google Chrome, pe MacOS, fiind aceleași teste rulate în aceleași condiții și același număr de apelări ale funcțiilor.

Urmărind datele din figura 5.7, putem analiza diferențele dintre cele două limbaje. Astfel, pentru funcția de calcul a factorialului unui număr, implementarea în AssemblyScript este de **33.9x** mai rapidă, cea mai mare diferență de până acum în testele noastre, funcția de căutare binară are o performanță de doar **2.375x** mai rapidă. Idem analizelor anterioare, funcțiile de colorare, BFS și problema rucsacului nu excelează așa de bine în AssemblyScript față de JavaScript, fiind doar de **1.41x**, **1.29x**, respectiv **1.35x** mai rapide. În același caz se află și funcția de înmulțire a două matrici, fiind doar de **1.27x** mai rapidă implementarea în AssemblyScript față de cea în JavaScript, iar pentru algoritmul MD5, este de **2.12x** mai rapidă.

Cu toate că diferențele între AssemblyScript și JavaScript sunt normale pentru Safari, cu excepția funcției de căutare binară, performanța per total este cu mult mai bună față de cea de pe Google Chrome, pentru ambele limbaje. Și în acest caz vom analiza doar diferențele între rezultatele obținute de funcțiile scrise în AssemblyScript. Calculul factorialului se realizează de **5.7x** mai rapid, având o viteză de 0.04ms pe Safari și 0.228ms pe Google Chrome; timpul pentru execuția căutării binare este același, cu o diferență de 0.001ms. Problema rucsacului pe Safari este de **1.33x** mai rapid, problema colorării unui graf se rezolvă cu un timp de **1.27x** mai rapid, iar BFS este de **1.27x** mai rapid în timpul de execuție. Același caz se aplică și pentru funcția de înmulțire a două matrici, care este de **1.61x** mai rapidă pe Safari, și algoritmul MD5 care este de **2.33x** mai rapid.

Aceste date reflectă încă odată faptul că Safari este un browser foarte optimizat, cu performanțe deosebite ce se integrează perfect cu sistemul de operare pentru care a fost creat. AssemblyScript de asemenea profită de această optimizare a browser-ului și ne oferă niște rezultate neașteptat de bune pentru testele realizate de noi.

Nu în ultimul rând, pentru a finaliza analiza pentru sistemul de operare MacOS, putem compara memoria folosită de fiecare funcție pe Google Chrome, și să comparăm rezultatele de pe MacOS cu cele de pe Windows.

AssemblyScript are nevoie pentru a executa funcțiile din teste între 20.5kB și

21.2kB, iar funcțiile scrise în JavaScript au nevoie între 21.3kB și 29.8kB. Și pe MacOS, ca și pe Windows, funcțiile implementate în JavaScript ce au nevoie de cel mai mult spațiu de memorie sunt căutarea binară, colorarea unui graf, BFS, înmulțirea a două matrici și algoritmul MD5. Diferența majoră apare la problema rucsacului care pe MacOS are nevoie de mai puțină memorie decât pe Windows.

Per total, funcțiile implementate în AssemblyScript au nevoie de aceeași memorie alocată și pe Windows, și pe MacOS, ceea ce arată faptul că WebAssembly este mai optimizat din acest punct de vedere față de JavaScript, iar implementările funcțiilor pe MacOS necesită o memorie alocată mai mică sau egală față de memoria necesară pe Windows, ceea ce ne spune că și JavaScript este mai optim pe MacOS, nu doar la timpii de execuție dar și la memoria necesară executării programelor.

5.3 Experimente pe sistemul de operare Linux

Vom finaliza experimentele noastre pe sistemul de operare Linux, mai precis Kali Linux, release 2023.2. Laptopul pe care rulează acest sistem de operare are marca HP, un procesor i7-8750H, cu 16GB de memorie RAM.

5.3.1 Google Chrome

Primul browser pe care vom rula testele pe acest sistem de operare este Google Chrome, versiunea 114. Vom menține aceleași condiții ca în celelalte teste, rulând testele pe rețeaua locală, într-o fereastră privată pentru izolarea sesiunii curente.

Măsurarea timpului de execuție

În figura 5.8 putem observa rezultatele testelor. Numărul de apeluri pentru fiecare funcție a rămas același și anume 100 de apeluri pentru fiecare test. Astfel, ne vom putea folosi de datele extrase pentru a compara în mod obiectiv performanța limbajului AssemblyScript pe sisteme de operare diferite.

Măsurarea memoriei folosite

Măsurarea memoriei a fost realizată folosind același instrument predefinit în DevTools pe Google Chrome. Datele extrase, măsurând în timp real memoria folosită, se pot regăsi în tabelul 5.4.

Linux - Google Chrome

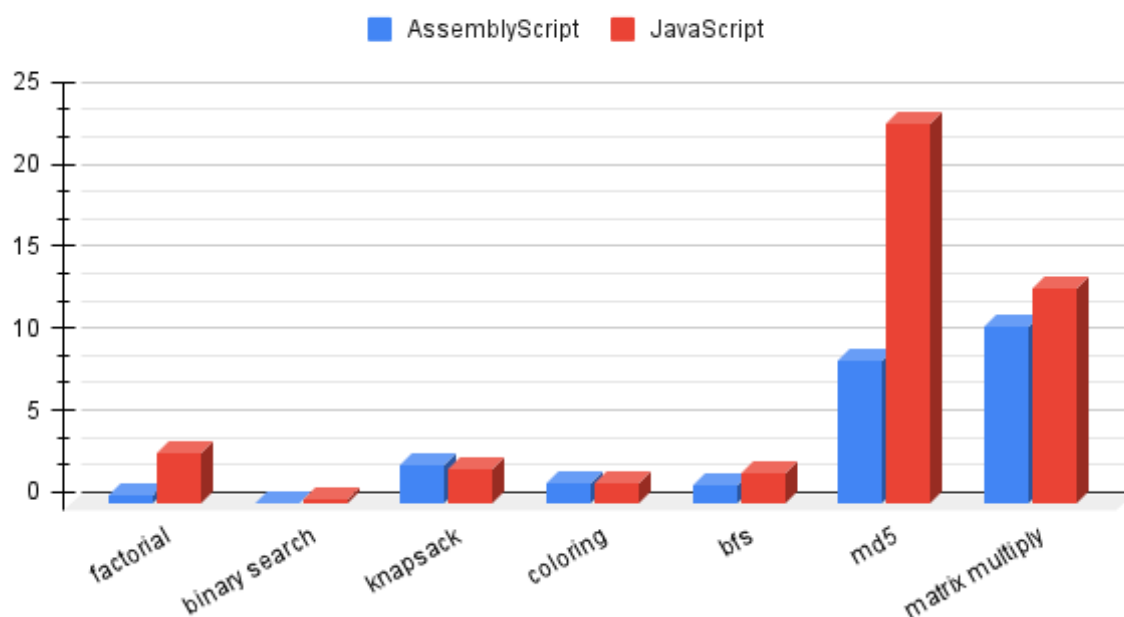


Figura 5.8: Grafic pentru timpii de execuție a testelor pe Google Chrome.

Tabela 5.4: Utilizarea Memoriei pe browser-ul Google Chrome.

Funcția	AssemblyScript	JavaScript
Factorial	21.3kB	23.4kB
Căutare binară	21.3kB	25.2kB
Problema rucsacului	20.6kB	26.2kB
Colorarea unui graf	20.6kB	24.7kB
BFS	20.8kB	26.8kB
Înmulțire de matrici	20.9kB	29.2kB
MD5	21.3kB	29.1kB

5.3.2 Mozilla Firefox

Ultimul browser din suita noastră de teste este Mozilla Firefox, versiunea 102. Acest browser este implicit instalat pe Linux și este optimizat diferit pentru acest sistem de operare. Vom analiza cum influențează acest lucru testele noastre, și dacă există vreo diferență, cât de mare este aceasta.

Măsurarea timpului de execuție

În figura 5.9 putem observa rezultatele testelor. Numărul de apeluri pentru fiecare funcție a rămas același asemenea testelor realizate pe acest browser pe celelalte sisteme de operare, numărul de teste aflându-se lângă numele funcției în grafic.

Linux - Mozilla Firefox

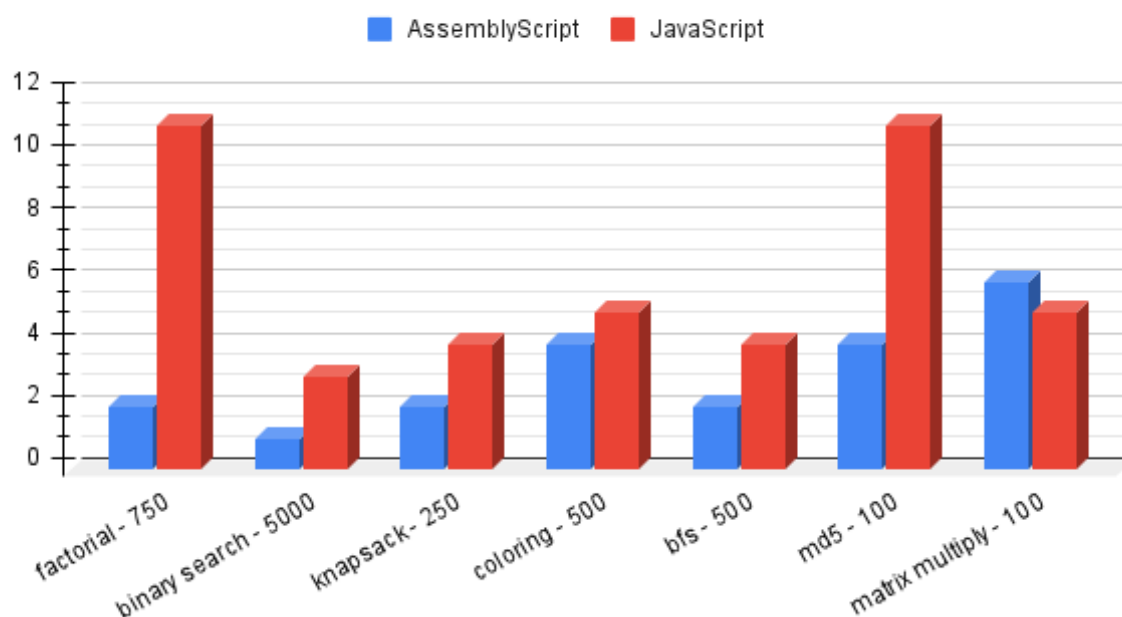


Figura 5.9: Grafic pentru timpii de execuție a testelor pe Mozilla Firefox.

5.3.3 Analiza rezultatelor pentru sistemul de operare Linux

Având toate datele adunate, putem face o analiză extrem de detaliată legată de performanțele celor două limbaje pe acest sistem de operare dar și o comparație obiectivă între performanțele în cadrul aceluiași browser dar pe sisteme de operare diferite. Ne propunem astfel să vedem dacă AssemblyScript își păstrează performanța și în cazul acestui sistem de operare.

Primul set de teste a fost realizat pe browser-ul Google Chrome, iar rezultatele pot fi observate în figura 5.8. Asemănător celorlalte rezultate de la secțiunile 5.1.4 și 5.2.4, AssemblyScript are o performanță remarcabilă în cazul următoarelor teste: calculul factorialului unui număr unde este de **5.62x** mai rapid, funcția de căutare binară este de **7.19x** mai rapidă în AssemblyScript decât implementarea în JavaScript, și nu în cele din urmă, algoritmul criptografic MD5 care este de **2.63x** mai rapid.

Aceste date întăresc ipoteza că AssemblyScript este foarte rapid și optimizat în problemele ce implică calcule matematice, logice sau operații pe biți, reușind să obțină diferențe mari și foarte mari în timpul necesar de execuție a funcțiilor.

Următoarele rezultate ce vor fi prezentate susțin faptul că AssemblyScript are o performanță cu puțin mai slabă, la fel de bună sau cu foarte puțin mai bună decât JavaScript, când ajungem la capitolul de lucru cu vectori sau matrici. Astfel, din datele colectate putem observa că AssemblyScript are o performanță de **1.09x** mai mică pentru problema rucsacului, este de **1.02x** mai încet în implementarea algoritmului de colorare a unui graf, pentru algoritmul BFS reușește să fie de **1.54x** mai rapid în implementarea cu AssemblyScript față de implementarea cu JavaScript, și nu în cele din urmă, pentru funcția de înmulțire a două matrici, AssemblyScript are o performanță de **1.21x** mai bună decât JavaScript.

Adunând toate datele necesare, putem compara performanța limbajului AssemblyScript, pe browser-ul Google Chrome, pe cele trei sisteme de operare pe care am realizat testele. Astfel, putem observa că pentru testele: calculul factorialului, căutarea binară și algoritmul MD5, AssemblyScript are o performanță remarcabilă, iar pentru celelalte teste reușește să fie cel puțin la fel de bun, sau puțin mai bun decât implementările în JavaScript. Observăm că pe toate cele trei sisteme de operare, procentajele de performanță între cele două limbaje au aproximativ aceleași valori, ceea ce înseamnă că AssemblyScript nu depinde de noroc, circumstanțe sau mediul în care este executat, pentru a aduce rezultatele dorite.

Însă, analizând strict performanța limbajului AssemblyScript pe Google Chrome de la un sistem de operare la altul, observăm că pentru funcția de calcul a factorialului, pe Windows și MacOS, timpul de execuție este aproximativ același, însă pe Linux este de aproape **2x** mai mare. Același caz de performanță scăzută pe sistemul de operare Linux este întâlnit și la problema rucsacului, unde valorile pentru Windows și MacOS sunt aproximativ egale iar rezultatul de pe Linux este de **4x** mai slab. Pentru celelalte teste realizate, și anume funcția de căutare binară, problema colorării unui graf, funcția BFS, funcția de înmulțire a două matrici și algoritmul criptografic MD5, performanțele de pe Windows și MacOS sunt de **2-3x** mai bune față de cele de pe Linux. Astfel, putem ajunge la concluzia că Google Chrome nu este foarte optimizat pentru acest sistem de operare.

Ultimul browser pe care am realizat experimentele noastre pentru acest sistem de operare este Mozilla Firefox. Fiind un browser optimizat din mai multe puncte de

vedere pentru Linux, dorim să vedem dacă rezultatele testelor noastre susțin acest fapt.

În primul rând, vom analiza diferențele de performanță între AssemblyScript și JavaScript. Urmărind datele din figura 5.9, observăm că AssemblyScript este mai rapid cu **9ms**, de **5.5x**, în timpul de execuție față de implementarea în JavaScript; căutarea binară este de **3x** mai rapidă, cu o diferență de **2ms**, problema rucsacului are un timp de **2ms** pentru implementarea în AssemblyScript și este de **2x** mai rapid decât implementarea în JavaScript, aceeași performanță o regăsim și pentru algoritmul BFS; pentru problema colorării unui graf, diferența nu este așa mare, de doar **1ms**, timpii de execuție fiind de **4ms** pentru AssemblyScript și **5ms** pentru JavaScript, aceeași diferență o regăsim și la funcția de înmulțire a două matrici doar că AssemblyScript este mai încet față de JavaScript, timpii de execuție fiind de **6ms** și respectiv de **5ms**; ultima funcție analizată, MD5, este de **2.75x** mai rapidă în AssemblyScript, cu un timp de **4ms** comparativ cu **11ms** pentru implementarea în JavaScript.

Comparând datele colectate pentru AssemblyScript de pe browser-ul Mozilla Firefox de pe Linux, cu celelalte date extrase de pe același browser de pe Windows și MacOS, observăm că pe Linux performanța este în medie de **2.95x** mai bună față de performanța obținută pe Windows, însă este de **1.44x** mai slabă față de performanța obținută pe MacOS. Astfel, putem afirma că browser-ul Mozilla Firefox este foarte puțin optimizat pentru Windows, dar foarte optimizat pentru MacOS, iar optimizările specifice sistemului de operare Linux, nu sunt la fel de bune ca cele de pe MacOS, dar având o performanță mai bună decât cea de pe Windows.

Nu în ultimul rând, folosindu-ne de datele din tabelul 5.4 putem analiza și necesarul de memorie pentru implementarea fiecărui test în fiecare limbaj. Astfel, pentru implementările în AssemblyScript, funcțiile noastre necesită între 20.6kB și 21.3kB, iar implementările în JavaScript necesită între 23.4kB și 29.2kB de memorie.

Comparând acum memoria folosită de testele noastre pe browser-ul Google Chrome pe toate cele trei sisteme de operare, putem afirma că AssemblyScript are nevoie de aceeași memorie pentru executare, valorile fiind aproximativ în același interval, însă pentru JavaScript putem observa că pentru MacOS și Linux valorile de memorie necesare sunt în același interval, fiecare funcție necesitând aproximativ aceleași valori, însă pentru Windows, valorile diferă destul de mult în cazul următoarelor funcții: problema rucsacului, problema colorării unui graf, o diferență destul de neglijabilă pentru BFS și o diferență notabilă pentru algoritmul MD5, care necesită aproape 10kB mai multă memorie pe Windows față de necesarul de memorie pe MacOS și Linux.

5.4 Concluzii

Am adunat toate datele rezultate în urma experimentelor realizate în tabela 5.5 pentru timpii de execuție și în tabela 5.6 pentru memoria utilizată. În tabela 5.5 toate valorile reprezintă timpii de execuție în milisecunde, iar pentru tabela 5.6 toate valorile reprezintă memoria necesară execuției funcției în kB.

Având centralizate toate informațiile într-un singur loc, putem afirma în mod obiectiv că AssemblyScript este mai bun decât JavaScript sau cel puțin la fel de bun în anumite situații. Acest lucru este susținut de următoarele statistici: 43 de test cu privire la timpii de execuție dintr-un total de 56 de comparații între AssemblyScript și JavaScript au un timp mai bun, în doar 8 teste AssemblyScript are o performanță mai mică decât JavaScript iar în restul de teste au o performanță egală. Din totalul de 43 de teste cu timp de execuție mai mare, 24 teste au o performanță de cel puțin două ori mai bună, iar dintre acestea, 10 teste au o performanță de cel puțin cinci ori mai mare.

Și în cazul testelor pentru memoria folosită de ambele limbaje pentru a executa un apel pentru fiecare funcție AssemblyScript oferă niște rezultate satisfăcătoare, contrar așteptărilor inițiale. Am sumarizat toate datele obținute în urma testelor noastre în tabela 5.6. În toate cazurile este necesară o memorie mai mică pentru implementarea în AssemblyScript decât pentru implementarea în JavaScript, diferențele de memorie fiind de la 1.6kB până la 17.3kB. Diferența nu este așa mare, fiind vorba de o funcție cu puțini parametri și calcule puține, dar transpunând aceste procentaje de diferență într-o aplicație reală, poate conta foarte mult pentru optimizarea aplicației noastre.

Definiții tabela 5.5: SO - sistem de operare, Win - Windows, GC - Google Chrome, MF - Mozilla Firefox, ME - Microsoft Edge, S - Safari, AS - AssemblyScript, JS - JavaScript, Fact - Funcția de calcul a factorialului, CB - Funcția de căutare binară, Rucsac - Problema rucsacului, Color - Problema colorării unui graf, BFS - Breadth First Search, Matrici - Funcția de înmulțire a două matrici.

Definiții tabela 5.6: SO - sistem de operare, Win-GC - Windows, browser Google Chrome, Win-ME - Windows, browser Microsoft Edge, Fact - Funcția de calcul a factorialului, CB - Funcția de căutare binară, Rucsac - Problema rucsacului, Colorare - Problema colorării unui graf, BFS - Breadth First Search, Matrici - Funcția de înmulțire a două matrici.

Tabela 5.5: Timpii de execuție pentru toate experimentele.

SO	Browser	Limbaj	Fact	CB	Rucsac	Color	BFS	Matrici	MD5
Win	GC	AS	0.28	0.03	0.66	0.55	0.50	6.61	3.71
Win	GC	JS	1.85	0.19	0.74	0.57	0.48	8.37	11.02
Win	MF	AS	5	3	7	11	6	12	11
Win	MF	JS	5	6	6	10	6	10	12
Win	ME	AS	0.25	0.03	0.63	0.66	0.45	5.23	3.60
Win	ME	JS	1.90	0.22	0.74	0.72	0.55	7.41	11.79
MacOS	GC	AS	0.22	0.02	0.49	0.57	0.62	4.71	3.44
MacOS	GC	JS	2.05	0.15	0.67	0.58	0.51	6.17	6.83
MacOS	MF	AS	1	1	2	3	3	3	3
MacOS	MF	JS	2	3	6	4	3	3	5
MacOS	S	AS	0.04	0.02	0.37	0.45	0.49	2.91	1.47
MacOS	S	JS	1.35	0.05	0.50	0.63	0.63	3.71	3.13
Linux	GC	AS	0.54	0.4	2.35	1.32	1.20	10.87	8.80
Linux	GC	JS	3.08	0.30	2.15	1.29	1.87	13.17	23.18
Linux	MF	AS	2	1	2	4	2	6	4
Linux	MF	JS	11	3	4	5	4	5	11

Tabela 5.6: Memoria necesară pentru toate experimentele.

SO	Limbaj	Fact	CB	Rucsac	Colorare	BFS	Matrici	MD5
Win-GC	AssemblyScript	20.6	21.0	20.6	20.4	20.8	20.9	22.1
Win-GC	JavaScript	22.1	26.7	32.9	30.5	29.6	29.9	38.7
Win-ME	AssemblyScript	21.1	21.3	20.7	21.2	20.7	21.1	21.3
Win-ME	JavaScript	23.6	28.5	33.4	30.7	29.8	30.2	38.6
MacOS	AssemblyScript	20.6	21.0	20.6	20.5	20.6	21.2	21.1
MacOS	JavaScript	22.2	25.8	21.3	27.2	27.1	29.1	29.8
Linux	AssemblyScript	21.3	21.3	20.6	20.6	20.8	20.9	21.3
Linux	JavaScript	23.4	25.2	26.2	23.7	26.8	29.2	29.1

Planuri de viitor și îmbunătățiri

Am discutat în această lucrare de cercetare despre acest limbaj relativ nou, WebAssembly, despre performanțele acestuia, despre cazurile în care chiar excelează în performanța adusă și versatilitatea ce o oferă programatorilor în crearea unei aplicații. Lucrarea noastră s-a focusat asupra unui singur aspect și anume performanța limbajului în mai multe condiții și medii diferite față de performanța adusă de JavaScript, care este limbajul principal de programare pentru platforme web. Oferind o multitudine de posibilități de a fi folosit, WebAssembly nu se rezumă doar la aplicații web, sau să fie o posibilă alternativă pentru JavaScript, iar în această parte a lucrării vom prezenta anumite idei pentru o cercetare viitoare legate de acest limbaj.

Vorbind de dezvoltarea aplicațiilor web, WebAssembly ar putea aduce o performanță mult mai bună aplicațiilor ce folosesc un framework în acest sens, iar integrarea acestui limbaj cu un framework, cum ar fi Angular⁴, React⁵, VueJs⁶, NextJs⁷ etc., ar putea duce la o creștere de performanță semnificativă. O posibilă cercetare viitoare ar putea avea în prim plan integrarea și folosirea unui modul de WebAssembly sau a unui proiect de AssemblyScript cu un framework, sau mai multe, din cele enumerate mai sus. De asemenea, explorarea și exploatarea posibilității de lucru cu memoria în AssemblyScript ar putea aduce un plus și mai mare de valoare unui astfel de proiect, având posibilitatea de a optimiza maxim codul scris încă de la început.

WebAssembly poate fi și baza unui server pentru o aplicație, scris într-un limbaj de nivel înalt oarecare și compilat în WebAssembly folosind instrumentele și bibliotecile deja specializate pentru așa ceva, și a se face o comparație obiectivă între o implementare cu WebAssembly și o implementare cu un alt limbaj de programare pentru un server al unei aplicații, sau chiar o comparație între două servere ce au la baza WebA-

⁴<https://angular.io/>

⁵<https://react.dev/>

⁶<https://vuejs.org/>

⁷<https://nextjs.org/>

ssembly, dar compilat din două limbaje diferite.

O altă abordare interesantă legată de utilitatea limbajului WebAssembly este în domeniul Internet of Things(IoT)⁸, mai exact o analiză despre cum poate îmbunătăți performanța sistemelor IoT și ce beneficii ar aduce. Această tematică poate fi corelată în mai multe feluri cu ideea precedentă, de a folosi WebAssembly ca limbaj de bază pentru server-ul unei aplicații.

Acestea ar fi doar câteva planuri de viitor realizabile, însă acest limbaj ne oferă o plajă largă de cazuri în care se poate folosi pentru programare, o multitudine de beneficii și posibilitatea de a duce performanța la un alt nivel, fiind o tehnologie nouă și în continuă dezvoltare.

⁸<https://www.oracle.com/internet-of-things/what-is-iot/>

Concluzii

Scopul acestei lucrări este de a descoperi și a testa WebAssembly, un limbaj nou, ce promite performanțe foarte bune pe platformele web și nu numai. Am început această cercetare plecând de la motivație, aflând ce înseamnă WebAssembly și limbajul cu care îl vom compara în experimentele noastre, JavaScript, iar mai apoi ce aplicații și cercetări există deja în domeniu de care ne putem folosi.

Acumulând aceste informații, am reușit să pregătim experimentele noastre prin crearea unui proiect nou de AssemblyScript și scrierea funcțiilor alese și discutate, în cele două limbaje. Având pregătite testele, am început efectiv colectarea de date în urma experimentelor și analiza acestora. Ne-am folosit în total de trei laptop-uri cu sisteme de operare diferite: un laptop marca Dell cu sistemul de operare Windows, un laptop marca Apple cu sistemul de operare MacOS și un laptop marca HP cu sistemul de operare Linux. Pentru a aduna un număr suficient de mare de date în urma testelor, ne-am folosit de Google Chrome și Mozilla Firefox pe toate cele trei laptop-uri, de Microsoft Edge pe laptop-ul cu sistemul de operare Windows și de Safari pe laptopul cu sistemul de operare MacOS.

Odata cu finalizarea adunării și analizei datelor rezultate în urma execuției testelor, am ajuns la concluzia că AssemblyScript, respectiv WebAssembly este cel puțin la fel de rapid ca JavaScript, dar în cele mai multe cazuri este mult mai rapid și mai eficient, și din punct de vedere al timpului de execuție, dar și al memoriei necesare pentru a apela funcțiile create de noi pentru teste. Acest limbaj de programare excelează în performanță în cazul calculelor complicate, matematice și logice, sau în lucrul cu operații pe biți, însă nu se descurcă la fel de bine la lucrul cu vectori și matrici.

Concluzia finală pe care o putem afirma în urma rezultatelor obținute în această lucrare este că AssemblyScript, respectiv WebAssembly, reprezintă o alternativă reală și fiabilă pentru JavaScript pentru dezvoltarea aplicațiilor web. Atât rezultatele timpilor de execuție obținuți cât și memoria necesară de AssemblyScript pentru execuția

codului scris sunt mai bune și oferă o performanță mai optimă. Ne rămâne în viitor să explorăm acest limbaj în profunzimea sa și să aflăm adevăratul său potențial, nu doar în cadrul aplicațiilor web, dar și în alte situații în care poate fi folosit.

Bibliografie

- [1] Marius Musch, Christian Wressnegger, Martin Johns și Konrad Rieck, *New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild*, 2019.
- [2] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai și JF Bastien, *Bringing the Web up to Speed with WebAssembly*, 2017.
- [3] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou și Weihang Wang, *Understanding the Performance of WebAssembly Applications*, 2021.
- [4] Denis Eleskovic, *A closer look at WebAssembly*, 2020.
- [5] B. Meurer și M. Bynens, *JavaScript engine fundamentals: Shapes and Inline Caches*, 2018.
- [6] F. Hinkelmann, *Understanding V8's Bytecode*, 2017.
- [7] Conrad Watt, Andreas Rossberg și Jean Pichon-Pharabod, *Weakening WebAssembly*, 2019.
- [8] Max van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja de Ruijter Tanjina Islam și Ivano Malavolta, *Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices*, 2022.
- [9] David Herrera, Hanfeng Chen, Erick Lavoie și Laurie Hendren, *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices*, 2018.
- [10] Scott Beamer, Krste Asanovic și David Patterson, *Direction-Optimizing Breadth-First Search*, 2012.
- [11] Robert Nowak, *Generalized Binary Search*, 2008.

- [12] R.M.R. Lewis, *A Guide to Graph Colouring - Algorithms and Applications*, Springer International Publishing Switzerland, 2016.
- [13] Rully Soelaiman, Ferdinand Jason Gondowijoyo, M.M. Irfan Subakti și Yudhi Purwananto, *AN EFFICIENT APPROACH TO CALCULATE FACTORIALS IN PRIME FIELD*, 2021.
- [14] Leonardo Fernando Dos Santos Moura, *An Efficient Dynamic Programming Algorithm For The Unbounded Knapsack Problem*, 2012.
- [15] Joseph D. Touch, *Performance Analysis of MD5*, 1995.

Linkuri:

- [16] <https://webassembly.github.io/spec/core/intro/index.html>
- [17] <https://developer.mozilla.org/en-US/docs/WebAssembly>
- [18] <https://www.assemblyscript.org/introduction.html>
- [19] <https://wiki.mozilla.org/Gecko:Overview>