

POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATYKI

PRACA DYPLOMOWA MAGISTERSKA

TEMAT: Wydajność WASM+Rust w porównaniu z językiem JavaScript w kontekście aplikacji przeglądarkowych

WYKONAWCA: Kasper Jan Seweryn

OPIEKUN PRACY DYPLOMOWEJ : dr inż. Marek Tabędzki

BIAŁYSTOK 2024 ROK

Subject of diploma thesis:

Efficiency of WASM+Rust compared to JavaScript in the context of web applications

Summary

In chapter one, I discussed performance issues with Just-In-Time compiled code. Then I presented what WebAssembly technology and the Rust language are and what benefits they bring. I also discussed the purpose and scope of the work and then described the structure of the work.

In the second chapter, I described JavaScript, Rust and WebAssembly technologies in detail. I focused on the differences that most impact browser code execution performance and those that impact code security.

In the third chapter, I analyzed the existing research. I presented the results and conclusions of research that compared the performance of JavaScript to Rust compiled into WebAssembly and discussed the differences between this research and my work.

I devoted the fourth chapter to a description of the research conducted. I explained how I built the test environment, what hardware the tests were run on, presented the testing procedure, determined benchmarking metrics, and finally described the tests performed.

In the fifth chapter, I presented the results of my research and analyzed them.

I devoted the sixth chapter to conclusions and summary. I discussed the performance of WebAssembly and JavaScript technologies, answered the question posed in the thesis and presented prospects for further research.

Spis treści

1 Wstęp	1
1.1 Cel i zakres pracy	2
1.2 Struktura pracy	2
2 Opis technologii	4
2.1 JavaScript	4
2.1.1 Typy zmiennych	4
2.1.2 Charakterystyka typów prymitywnych	6
2.1.3 Kompilacja Just-In-Time	7
2.1.4 Mechanizm „Garbage Collector”	11
2.2 Rust	13
2.2.1 Charakterystyka typów	14
2.2.2 System własności, a mechanizm „Garbage Collector”	22
2.3 WebAssembly	26
2.3.1 Kompilacja do binarnego formatu WebAssembly	26
2.3.2 Wykonywanie kodu WebAssembly	27
2.3.3 Zarządzanie pamięcią	28
2.3.4 Operacje SIMD	30
3 Analiza dotychczasowych badań	32
3.1 Mikrobenchmarki WebAssembly i JavaScript	32
3.2 Porównanie wydajności frameworków frontendowych	33
4 Metodyka badań	36
4.1 Środowisko testowe	36
4.1.1 Opis sprzętu	36
4.1.2 Opis oprogramowania	36
4.2 Procedura testowania	37
4.3 Pomiary i metryki porównawcze	38
4.3.1 Pomiary dla języka Rust	38

4.3.2	Pomiary dla języka JavaScript	39
4.3.3	Metryki porównawcze	39
4.4	Przeprowadzone testy	40
4.4.1	Re-enkodowanie stringów	40
4.4.2	Enkodowanie stringów do Base64	41
4.4.3	k-ta liczba Fibonacciego (rekurencyjnie)	41
4.4.4	Mnożenie macierzy 4x4	42
4.4.5	CRC-32	43
4.4.6	CRC-64	44
4.4.7	API DOM — Tworzenie elementów	45
4.4.8	API DOM — Aktualizacja co drugiego elementu	45
4.4.9	API DOM — Usuwanie elementów	46
5	Wyniki badań	48
5.1	Re-enkodowanie stringów	48
5.2	Enkodowanie stringów do Base64	50
5.3	k-ta liczba Fibonacciego (rekurencyjnie)	53
5.4	Mnożenie macierzy 4x4	58
5.5	CRC-32	60
5.6	CRC-64	63
5.7	API DOM — Tworzenie elementów	66
5.8	API DOM — Aktualizacja co drugiego elementu	69
5.9	API DOM — Usuwanie elementów	72
6	Wnioski	75
6.1	Wydajność funkcji wymagających serializacji danych	76
6.2	Wydajność funkcji niewymagających serializacji danych	76
6.3	Wydajność odwołań do API DOM	77
6.4	Wydajność funkcji SIMD	78
6.5	Wydajność WebAssembly i JavaScriptu	79
6.6	Perspektywy dalszych badań	80
Bibliografia		85

Spis tabel	86
Spis rysunków	88
Spis listingów	89

1. Wstęp

Aplikacje przeglądarcowe stają się coraz bardziej popularne ze względu na swoją uniwersalność i wieloplatformowość. Każda z nich może być w prosty sposób przekształcona w PWA (Progressive Web App), aplikację desktopową czy też aplikację mobilną. Środowiska takie jak *Electron* [1], czy *Tauri* [2] renderują je, wykorzystując istniejące silniki przeglądarcowe dostępne na systemach desktopowych. Tauri, pozwala również (podobnie jak *Ionic* [3], czy *Cordova* [4]) na pisanie aplikacji mobilnych w formie stron internetowych, które wyświetlane są w natywnym WebView na systemach Android i iOS. Ten trend sprawia, że wydajność kodu logiki aplikacji przeglądarcowych jest coraz bardziej istotna.

Na przestrzeni ostatnich 15 lat, silniki uruchamiające JavaScript wyspecjalizowały się w optymalizacji tego języka. Kompilacja „Just-In-Time” (JIT) przyśpieszyła je kilkudziesięciokrotnie i pozwoliła na optymizacje kodu podczas jego wykonywania. Wiele z tych optymizacji polega na przewidywaniach i założeniach wydedukowanych na podstawie wcześniejszych wywołań funkcji, a więc im częściej dana funkcja jest wywoływana, tym bardziej „gorąca” się staje. Im gorętsza funkcja, tym mocniej kompilator JIT próbuje ją zoptymalizować. Niestety istnieje wiele czynników, które mogą wpływać na wydajność kompilacji JIT. Jednym z nich jest dosyć często spotykana praktyka, gdzie typ argumentu funkcji może się różnić między wywołaniami. W takiej sytuacji kompilator JIT musi dokonać tak zwanej „deoptimizacji” zoptymalizowanej funkcji, co może doprowadzić do nieoczekiwanej spadku wydajności. Zmusza to programistów do odpowiedniego dostosowania swojego kodu i dbania o wiele aspektów, które mogą negatywnie wpływać na wydajność kompilacji JIT.

W odpowiedzi na trudno przewidywalne spadki wydajności w 2017 roku pojawiła się nowa technologia, WebAssembly (WASM), która pozwoliła na uruchamianie kompilowanych języków bezpośrednio w środowisku JavaScriptowym. Pozwoliło to na tworzenie aplikacji o wydajności zbliżonej do natywnych aplikacji. WebAssembly jest ustandaryzowanym przez W3C [5] formatem kodu wspieranym przez większość nowoczesnych przeglądarek [6] internetowych. Języki takie jak C/C++, Java, Go, Rust i wiele innych umożliwiają kompilację do jego binarnej formy, umożliwiając tym samym uruchomienie programów napisanych w tych językach bezpośrednio w przeglądarkach. Odkrywa to przed programistami nowe

możliwości optymalizacyjne i pozwala na ujednolicenie języka, w którym pisane są aplikacje po stronie klienta i serwera.

Ze wszystkich języków, które wspierają komplikację do WebAssembly najrozsądniej- szym wyborem wydaje się być Rust. Jest to język pozbawiony Garbage Collectora (GC) z oryginalnym systemem własności zmiennych i silnym systemem typów, co wyklucza większość błędów związanych z pamięcią i gwarantuje bezpieczeństwo oraz poprawność wykonania skompilowanego programu. Jak wykazano w wielu badaniach [7][8][9][10], niezależnie od środowiska wykonywania Rust dorównuje wydajnością do języków takich jak C i C++, a nawet prześciga je w niektórych przypadkach, co czyni go idealnym wyborem dla aplikacji, które wymagają wysokiej wydajności przy użyciu WebAssembly.

1.1 Cel i zakres pracy

Celem tej pracy jest porównanie wydajności języka Rust, skompilowanego do WebAssembly, z językiem JavaScript w środowiskach przeglądarkowych oraz odpowiedzenie na pytanie: „Czy Rust jest lepszym wyborem dla aplikacji przeglądarkowych niż JavaScript?”.

W ramach tego przeanalizuję takie aspekty jak:

- Wydajność wywołań rekurencyjnych
- Wydajność pętli
- Wydajność tworzenia elementów DOM
- Wydajność modyfikowania elementów DOM
- Wydajność usuwania elementów DOM
- Wydajność przerzucania danych między kontekstami
- Wydajność algorytmów wykorzystujących operacje SIMD

Następnie zbiorę wyniki i spróbuję odpowiedzieć na pytanie, która z technologii jest lepszym wyborem dla aplikacji przeglądarkowych wymagających wydajności.

1.2 Struktura pracy

W rozdziale pierwszym omówiłem problemy z wydajnością komplikacji Just-In-Time. Następnie przedstawiłem czym jest technologia WebAssembly oraz język Rust i jakie niosą ze sobą korzyści. Omówiłem również cel i zakres pracy po czym dokonałem opisu struktury pracy.

W drugim rozdziale opisałem dokładnie technologie JavaScript, Rust oraz WebAssembly. Skupiłem się na różnicach, które najbardziej wpływają na wydajność wykonania kodu w przeglądarce oraz na tych, które wpływają na bezpieczeństwo kodu.

W trzecim rozdziale dokonałem analizy dotychczasowych badań. Przedstawiłem wyniki i wnioski z badań, które porównywały wydajność języka JavaScript z językiem Rust skompilowanym do WebAssembly i omówiłem różnice między tymi badaniami, a moją pracę.

Czwarty rozdział poświęciłem na opis przeprowadzonych badań. Wytlumaczyłem, w jaki sposób zbudowałem środowisko testowe, na jakim sprzęcie uruchomione były testy, przedstawiłem procedurę testowania, wyznaczyłem metryki porównawcze i ostatecznie opisałem przeprowadzone testy.

W piątym rozdziale zaprezentowałem wyniki moich badań i dokonałem ich analizy.

Szósty rozdział poświęciłem na wnioski i podsumowanie. Omówiłem wydajność technologii WebAssembly i JavaScript, odpowiedziałem na pytanie postawione w tezie pracy i przedstawiłem perspektywy dalszych badań.

2. Opis technologii

Aby dobrze zrozumieć różnice pomiędzy JavaScriptem, a Rustem skompilowanym do WebAssembly, należy zagłębić się w działanie każdej z tych technologii, zrozumieć ich różnice oraz to, w jaki sposób są one zbudowane.

2.1 JavaScript

JavaScript jest językiem skryptowym pozwalającym użytkownikom przeglądarki na bezpośrednią interakcję z aplikacjami webowymi. Przez 29 lat rozwoju, developerzy takich silników jak SpiderMonkey (Netscape Navigator, Firefox), JavaScriptCore (Safari), czy V8 (Chromium) byli w stanie wypracować różne metody na przyśpieszenie wykonywania tego języka. Z prostych interpreterów, silniki przerodziły się w skomplikowane kompilatory generujące i optymalizujące kod maszynowy podczas wykonywania skryptów. Dokonując komplikacji just-in-time, są one w stanie przyśpieszyć wykonywanie kodu kilkudziesięciokrotnie [11] w porównaniu do naiwnego wręcz interpretowania.

2.1.1 Typy zmiennych

JavaScript jest językiem typowanym dynamicznie, w którym zmienne nie mają przypisanego konkretnego typu i do dowolnej zmiennej można przypisać (oraz nadpisać) wartość dowolnego innego typu. Listing 2.1 pokazuje przykład nadpisania zmiennej „x” różnymi typami.

```
1 let x = 42 // x ma teraz typ number
2 x = "foo" // x ma teraz typ string
3 x = true // x ma teraz typ boolean
4 x = {} // x ma teraz typ object
```

Listing 2.1: Przypisanie różnych typów do zmiennej „x”

W JavaScript wyróżniamy kilka typów zmiennych, które można podzielić na dwa rodzaje: typy prymitywne i typy złożone, czyli inaczej obiekty. Pierwsze z nich są przekazywane poprzez ich wartość, natomiast te drugie — poprzez referencję.

Wśród typów prymitywnych możemy wyróżnić:

- "string"
- "number"
- "bigint"
- "boolean"
- "undefined"
- "symbol"
- null

Wszytkie typy prymitywne (oprócz typu "symbol") zostały szerzej opisane w podrozdziale 2.1.2.

Obiekty są strukturami słownikowymi opisującymi relację klucz-wartość. Każdy obiekt może posiadać wiele kluczy typu "string" lub "symbol", które przypisane są do pojedynczej wartości dowolnego typu. To oznacza, że obiekty mogą zawierać w sobie nie tylko typy prymitywne, ale również inne obiekty. Listy, a nawet funkcje, pomimo swoich syntaktycznych różnic, też są obiektami.

Obiekty, w przeciwieństwie do typów prymitywnych, są modyfikowalne. Oznacza to, że aby zmodyfikować wartość zmiennej prymitywnej, należy nadpisać ją nową wartością. Natomiast aby zmienić wartość obiektu, wystarczy zmodyfikować jedną z jego właściwości. Zostało to przedstawione na listingu 2.2.

```
1 let foo = 42
2 let bar = { baz: 42 }
3
4 foo = 21      // modyfikacja zmiennej prymitywnej:
5                 // odrzucenie starej wartości i nadpisanie zmiennej
6
7 bar.baz = 21 // modyfikacja obiektu:
8                 // nadpisanie jednej właściwości obiektu
```

Listing 2.2: Przykład modyfikowania zmiennej prymitywnej oraz obiektu

2.1.2 Charakterystyka typów prymitywnych

string

Typ "**string**" jest reprezentacją ciągu znaków. Specyfikacja ECMA Script opisuje go jako kodowany do UTF-16 [12].

number

Typ "**number**" to 64-bitowa reprezentacja liczby zmiennoprzecinkowej, która wyraża nie tylko dodatnie i ujemne liczby, ale również pozytywną i negatywną nieskończoność, $+0$ (albo po prostu 0) i -0 oraz *NaN* (Not-a-Number). Ta ostatnia oznacza wartość nienumeryczną bądź niezidentyfikowaną, taką jak na przykład wynik działania $x = \frac{0}{0}$.

Wedle specyfikacji ECMA Script [13], "**number**" przechowywany jest jako 64-bitowa liczba zmiennoprzecinkowa podwójnej precyzji zapisana wedle formatu IEEE 754-2019 z tą różnicą, że 9,007,199,254,740,990 różnych wartości oznaczających *NaN* w standardzie IEEE jest wyrażona jako pojedyncza wartość *NaN* w JavaScript.

Warto zwrócić uwagę na to, że standard ECMA Script opisuje, że niektóre operatory działają jedynie na liczbach całkowitych z zakresu $[-2^{31}, 2^{31}-1]$ lub $[0, 2^{16}-1]$. Przykładem są operacje bitowe [14], które wedle standardu muszą być zamienione z 64-bitowej liczby zmiennoprzecinkowej na 32-bitową liczbę całkowitą. Powoduje to ucięcie dokładności liczb i sprawia, że w przypadku algorytmów polegających na operacjach bitowych na liczbach większych niż 32 bity — takich jak na przykład CRC64 — zmuszeni jesteśmy do skorzystania z typu "**bigint**" opisanego w następnej sekcji.

bigint

Typ "**bigint**" reprezentuje liczbę całkowitą, która może być dowolnego rozmiaru i nie jest ograniczona do konkretnej ilości bitów [15]. Jest on zalecany do stosowania dla liczb całkowitych przyjmujących wartości większe niż 2^{53} [16], a więc takie, które nie mieścią się w zakresie typu "**number**". Operacje na zmiennych typu "**bigint**" są wykonywane z pełną precyzją, co oznacza, że nie ma ryzyka utraty dokładności. Należy jednak pamiętać, że operacje wykonywane na tych liczbach nie mają zagwarantowanego wykonania w czasie stałym, a to stwarza zagrożenie dla algorytmów kryptograficznych i otwiera podatność na ataki typu timing attack.

boolean

Typ "**boolean**" reprezentuje wartość prawda / fałsz.

undefined

Typ "**undefined**" ma dokładnie jedną wartość i oznacza niezdefiniowaną wartość — każda zmienna bez przypisanej wartości będzie miała wartość undefined.

null

Typ null przyjmuje dokładnie jedną wartość wyrażającą intencjonalny brak wartości. W przeciwieństwie do innych typów prymitywnych wynik operacji `typeof null` zwróci "**object**" zamiast nazwy typu [17].

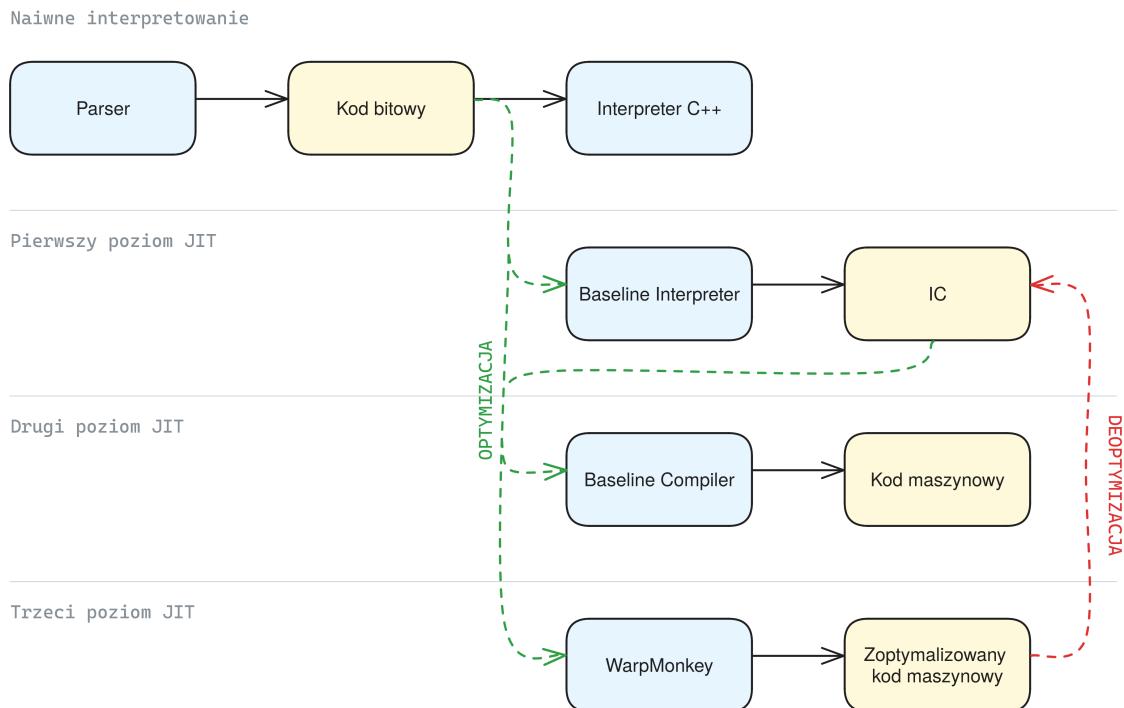
Różnica między null a "**undefined**" jest taka, że "**undefined**" oznacza zmienną, która nie ma przypisanej wartości, natomiast null oznacza zmienną, która ma przypisaną wartość, ale ta wartość jest pusta.

2.1.3 Kompilacja Just-In-Time

Różne silniki w różny sposób wykonują kod JavaScriptowy. W tej pracy skupię się na silniku SpiderMonkey wykorzystywanym w przeglądarce Firefox. SpiderMonkey wpierw generuje drzewo AST przy pomocy parsera, a następnie „BytecodeEmitter” tworzy z niego kod bajtowy (ang. *bytecode*). Ostatecznie kod wykonywany jest przez interpreter JavaScriptowy napisany w języku C++.

Naiwne interpretowanie kodu jest procesem wolnym, dlatego silniki JavaScriptowe wprowadziły mechanizm kompilacji Just-In-Time. Ponieważ wyższe stopnie optymalizacji są bardziej kosztowne, funkcja lub pętla, która wykonuje się wiele razy, jest oznaczona jako gorąca. Im gorętsza się staje, tym większy stopień kompilacji JIT jest zastosowany do zoptymalizowania danego fragmentu kodu. Pierwszym stopniem JIT jest „Baseline Interpreter”, który dokonuje interpretacji kodu bajtowego dołączając do każdej operacji wbudowaną pamięć podręczną (ang. *Inline Cache*). Pozwala na szybsze wykonanie kodu w przyszłości. Gorące fragmenty kodu podane są do „Baseline Compilera”, który wykorzystuje mechanizm IC (wbudowanej pamięci podręcznej) z „Baseline Interpretera” dodatkowo generując kod maszynowy. Jeżeli fragment kodu jest wykonywany bardzo często, to poddawany jest dalszej optymalizacji przy użyciu silnika WarpMonkey, który tłumaczy kod

maszynowy na reprezentację pośrednią. Następnie dokonuje na niej wielu optymalizacji [18] i finalnie generuje kod maszynowy. Cały proces został przedstawiony na rysunku 2.1.



Rysunek 2.1: Proces optymalizacji i deoptymalizacji skryptów JavaScript

Warto zaznaczyć, że optymalizacje dokonane przez WarpMonkey zakładają, że wszystkie przyszłe wykonania zoptymalizowanego fragmentu kodu będą operowały na podobnych danych. Jeżeli skrypt skompilowany przez WarpMonkey natrafi na dane, które nie spełniają założeń optymalizacji (np: wywołanie funkcji z innym typem argumentu), musi wykonać operację ratunkową zwaną inaczej deoptymalizacją kodu. Mechanizm ratunkowy ma za zadanie odbudować ramkę stosu w taki sposób, by możliwe było wykonywanie kodu poprzez „Baseline Interpreter” [19]. Rysunek 2.1 przedstawia to jako czerwoną strzałkę wędrującą z trzeciego poziomu JIT na pierwszy poziom JIT.

Specjalizacja typów

Jednym z mechanizmów optymalizacji kodu JavaScriptowego jest specjalizacja typów. Polega ona na założeniu, że funkcja będzie wywoływana zawsze z tym samym typem argumentów. Jak widać na listingu 2.3, funkcja „add” jest wywoływana wiele razy w pętli z argumentami typu `"number"`. Co więcej, argumenty te nigdy nie przyjmują liczby zmiennoprzecinkowej, więc WarpMonkey może bezpiecznie założyć, że funkcja zawsze

będzie przyjmować liczby całkowite. Na linii 8 znajduje się wywołanie funkcji z dwoma argumentami, które są liczbami całkowitymi, więc SpiderMonkey korzysta ze zoptymalizowanej funkcji. Natomiast na linii 11, funkcja jest wywoływana z argumentami, które są typu `"string"`, więc WarpMonkey musi dokonać deoptimizacji funkcji. Powoduje to, że to oraz następne wywołania będą dokonywane przez „Baseline Interpreter” — znacznie wolniejszego poziomu JIT.

```
1 function add(a, b) {
2     return a + b
3 }
4
5 for (let i = 0; i < 1e6; i++)
6     add(1, i)
7
8 add(41, 1)          // Po wielu wywołaniach, funkcja add jest
9                  // zoptymalizowana dla liczb całkowitych
10
11 add("bar", "qux") // Zmienił się typ argumentów,
12                  // zachodzi zdeoptimizacja
```

Listing 2.3: Przykład kodu ulegającemu specjalizacji typów i deoptimizacji

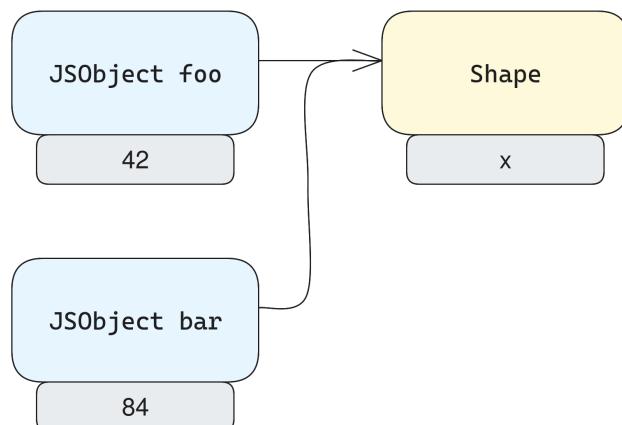
Wywoływanie funkcji z różnym typem argumentów jest dosyć częstą praktyką w projektach JavaScriptowych. Próba optymalizacji takich funkcji może doprowadzić do nieoczekiwanych spadków wydajności związanych z deoptimizacją. W aplikacjach wymagających wysokiej wydajności programiści muszą świadomie dostosować swój kod, aby uniknąć negatywnych skutków optymalizacji JIT.

Ukryte klasy

Częstą optymalizacją w silnikach JavaScriptowych jest tworzenie ukrytych klas obiektów. Aby dobrze zrozumieć to zagadnienie, należy zagłębić się w to, w jaki sposób silniki JavaScriptowe przechowują obiekty w pamięci. Rozważmy więc przypadek przedstawiony w listingu 2.4. Stworzyliśmy dwa obiekty „foo” i „bar” z identycznymi kluczami i różnymi wartościami. Silnik tworzy więc dwa obiekty typu „JSObject” [20] i umieszcza w nich jedynie wartości kluczy w kolejności ich występowania. Następnie tworzy ukrytą klasę „Shape” [20] dla pierwszego z obiektów, w której zapisuje informacje o kolejności kluczy, oraz deskryptory właściwości. Zostało to zilustrowane na rysunku 2.2.

```
1 const foo = { x: 42 }
2 const bar = { x: 84 }
```

Listing 2.4: Przykład dwóch obiektów z identycznymi właściwościami



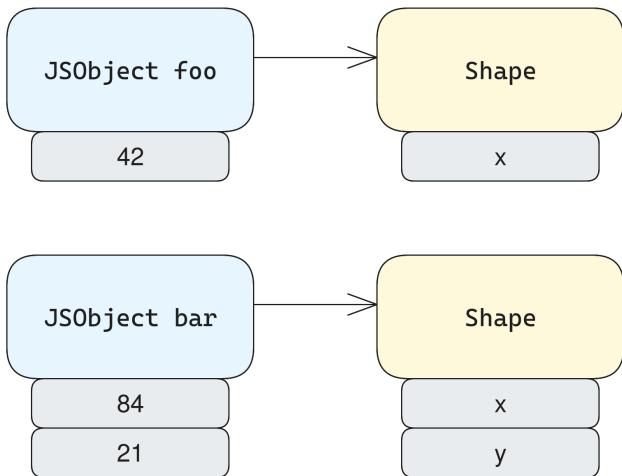
Rysunek 2.2: Przykład wspólnej ukrytej klasy dla dwóch obiektów

Dzięki takiemu zabiegowi, silnik jest w stanie zoptymalizować zapytania o pojedyncze właściwości obiektów, które mają ten sam kształt (tę samą ukrytą klasę) i może dołączyć obiekt do wbudowanej pamięci podręcznej.

Listing 2.5 przedstawia kolejną częstą praktykę w kodzie JavaScriptowym — dynamiczne dodawanie właściwości do obiektów. Z punktu widzenia wydajności, dodawanie nowych właściwości do obiektów prowadzi do stworzenia nowej ukrytej klasy dla obiektu „bar”, co uniemożliwia zastosowanie IC i w wielu przypadkach prowadzi do deoptimizacji. Rysunek 2.3 przedstawia nowy rozkład ukrytych klas dla obiektów „foo” i „bar”.

```
1 const foo = { x: 42 }
2 const bar = { x: 84 }
3
4 bar.y = 21 // Zmienił się kształt obiektu bar
```

Listing 2.5: Przykład dynamicznego dodania właściwości do obiektu



Rysunek 2.3: Przykład różnych ukrytych klas dla dwóch obiektów

2.1.4 Mechanizm „Garbage Collector”

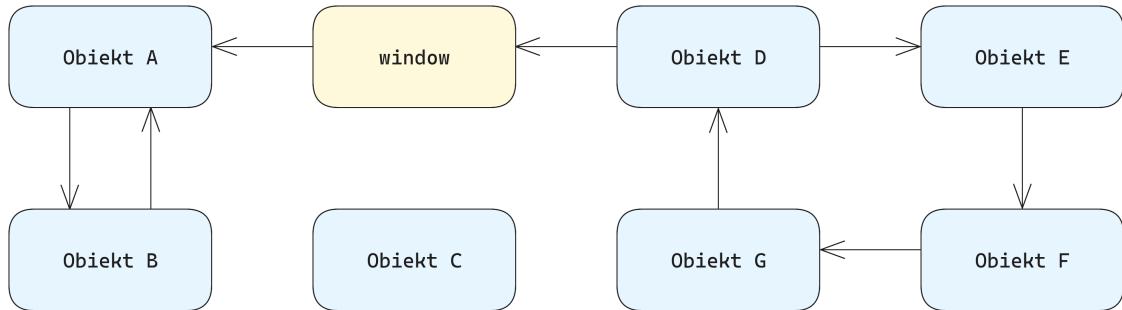
Garbage collector (GC) to mechanizm odpowiedzialny za zarządzanie pamięcią w języku JavaScript. Dzięki niemu nie musimy ręcznie alokować i zwalniać pamięci. Wśród mechanizmów GC można wyróżnić trzy główne strategie: zliczanie referencji, „escape analysis” oraz śledzenie (ang. *tracing*).

Zliczanie referencji jest najprostszym z trzech podejść. Polega ono na podbijaniu licznika referencji, gdy obiekt jest przypisywany do zmiennej i zmniejszaniu go, gdy zmienna wychodzi poza zakres. Największym minusem tego podejścia jest to, że w przypadku odniesień cyklicznych, licznik referencji nigdy nie spadnie do 0. Z tego powodu silniki JavaScriptowe już dawno zrezygnowały z używania tej strategii.

Drugą strategią jest tak zwane „escape analysis”. Jest to optymalizacja stosowana podczas komplikacji, która polega na alokowaniu obiektów na stosie zamiast na stercie. WarpMonkey wykorzystuje ją podczas optymalizacji nazwanej „Scalar Replacement”, która — pod warunkiem że obiekt nigdy nie opuści zakresu — śledzi pojedyncze właściwości obiektu, zamiast alokować cały obiekt.

Trzecia strategia, czyli tracing jest najczęściej stosowaną metodą GC w silnikach JavaScriptowych. Najpopularniejszą implementacją tego podejścia jest algorytm „Mark-and-Sweep” [21] polegający na uproszczeniu problemu „obiekt jest już niepotrzebny” do problemu „nie można dotrzeć do obiektu”. Jest to automatyczny algorytm, który sam decyduje, kiedy należy posprzątać, więc trudno jest przewidzieć kiedy zostanie wykonany i jaki wpływ będzie miał na wydajność aplikacji. Aby zrozumieć sposób działania algorytmu

Mark-and-Sweep, rozpatrzmy przykład przedstawiony na rysunku 2.4. Znajduje się na nim siedem obiektów nazwanych od „A” do „G” oraz globalny obiekt „window”. Strzałki między obiektemi oznaczają referencje, np: strzałka od obiektu „D” do obiektu „E” oznacza, że obiekt „D” posiada klucz, który wskazuje na obiekt „E”.



Rysunek 2.4: Przykład referencji obiektów w pamięci JavaScript

Algorytm „Mark-and-Sweep” w swojej pierwszej fazie wchodzi do wszystkich obiektów-korzeni, które są osiągalne z poziomu globalnego (np: obiekt „window”) i oznacza je w swojej pamięci jako osiągalne. Następnie wchodzi do dzieci tych obiektów i je również oznacza jako osiągalne. Zapętlone jest to aż do momentu, gdy nie można znaleźć więcej osiągalnych obiektów.

W drugiej fazie, algorytm „Mark-and-Sweep” przechodzi przez całą pamięć i usuwa obiekty, które nie zostały oznaczone jako osiągalne.

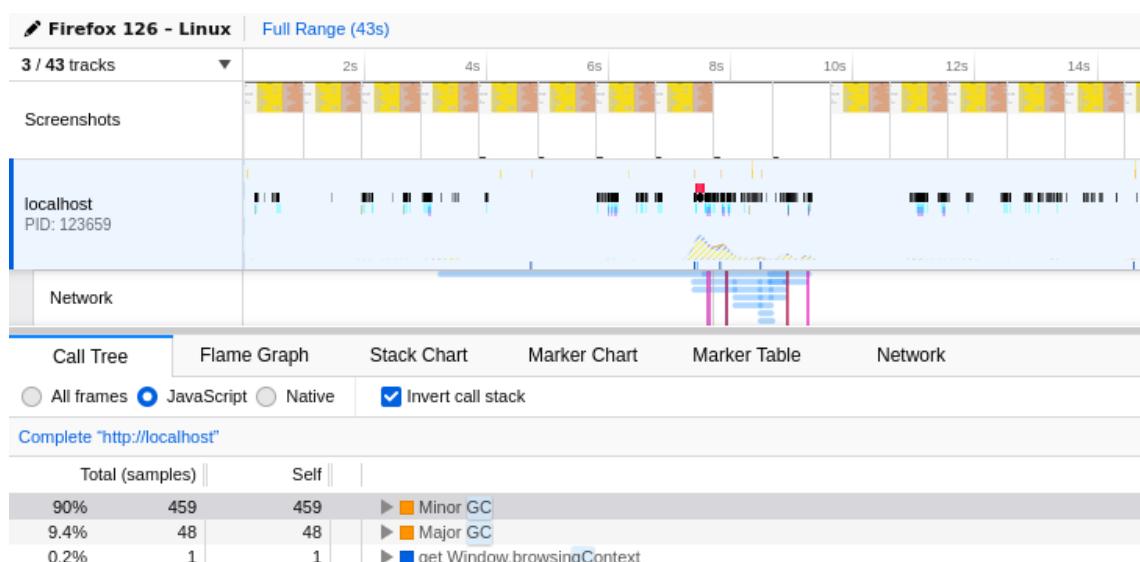
Kierując się przykładem z rysunku 2.4, algorytm oznaczy jako osiągalne jedynie obiekty „A” i „B”. Obiekt „C” jest nieosiągalny, ponieważ nie ma do niego żadnego odniesienia. Obiekty „D”, „E”, „F” i „G” są również nieosiągalne, ponieważ z obiektem globalnym nie da się do nich dotrzeć. Warto zwrócić uwagę na to, że obiekty te tworzą cykl, więc zwykłe zliczanie referencji nie byłoby w stanie sobie z nim poradzić, a „Mark-and-Sweep” bez problemu usuwa je z pamięci.

Incremental Garbage Collection

Algorytm „Mark-and-Sweep” jest stosunkowo prosty, ale ma swoje wady. Działa on w tym samym wątku co wykonanie skryptu, więc aby nie doprowadzić do zatrzymania całej aplikacji (tak zwane „Stop-the-World”), SpiderMonkey uruchamia go inkrementalnie [22]. Oznacza to, że GC dzieli swoje wywołanie na kilka mniejszych kroków tak, aby zminimalizować negatywne skutki dla wydajności aplikacji.

Generational Garbage Collection

W praktyce większość nowo zaalokowanych obiektów staje się nieosiągalne bardzo szybko po alokacji. W związku z tym, SpiderMonkey stosuje strategię „Generational Garbage Collection”, która dzieli obiekty na dwie generacje: młodszą i starszą. Obiekty w młodszej generacji są częściej sprawdzane podczas fazy zwanej „Minor GC”, natomiast obiekty w starszej generacji są sprawdzane rzadziej podczas fazy „Major GC”, która kolekcjonuje obiekty z obu generacji. Jeżeli obiekt znajdujący się w młodszej generacji przetrwa kilka cykli GC, zostaje przeniesiony do starszej generacji. Obie fazy zostały przechwycone w narzędziach przeznaczonych do profilowania i przedstawione wraz z ilością ich wykonania na rysunku 2.5.



Rysunek 2.5: Przykład „Minor GC” i „Major GC” w narzędziach deweloperskich

2.2 Rust

W przeciwieństwie do języka JavaScript, który jest kompilowany Just-In-Time, język Rust jest językiem kompilowanym Ahead-Of-Time (AOT), czyli przed wykonaniem kodu. Jego silny system typów, mechanizm własności i pożyczceń, brak wartości null oraz wiele innych innowacyjnych rozwiązań pozwala mu na dokonanie dużej ilości optymalizacji już na etapie komplikacji. Dzięki temu jest w stanie zapewnić stałą wydajność kodu, bezpieczeństwo pamięci oraz brak nieobsłużonych błędów podczas wykonania.

2.2.1 Charakterystyka typów

Język JavaScript dzieli typy jedynie na prymitywne i złożone. Specyfikacja ECMA Script nie definiuje, czy typy prymitywne powinny znajdować się na stosie, czy być alokowane na stercie. Musimy więc zaufać GC, że będzie w stanie dokonać odpowiednich optymizacji i będzie sprawnie zarządzać pamięcią.

W przypadku języka Rust, typy możemy podzielić na prymitywne, sekwencyjne, funkcyjne, wskaźnikowe, cechowe (ang. *Trait*) oraz zdefiniowane przez użytkownika. Naturalnie, opis każdego typu z każdej z tych grup wymagałby osobnego rozdziału, dlatego skupię się wyłącznie na tych, które są najistotniejsze w porównaniu z językiem JavaScript. Typy, których rozmiar jest z góry znany, będą automatycznie odkładane na stosie w momencie deklaracji wartości i zrzucone z niego w momencie wyjścia poza zakres. Niektóre typy takie jak `Box<T>` pozwalają na ręczną alokację typów o znanym rozmiarze na stercie, w sytuacji gdy użytkownik rzeczywiście tego potrzebuje. Typy, których rozmiar jest dynamiczny, muszą być zaalokowane na stercie ze względu na to, że ich rozmiar nie jest znany podczas komplikacji i może ulec zmianie.

Typy numeryczne

Rust posiada wiele typów numerycznych. Można je podzielić na typy reprezentujące liczby całkowite, przedstawione w tabeli 2.1 oraz na typy reprezentujące liczby zmienno-przecinkowe.

Typy całkowite dzielą się na te ze znakiem (ang. *Signed*) i te bez znaku (ang. *Unsigned*). Wśród każdej z tych grup możemy wyróżnić typy o różnych rozmiarach: 8, 16, 32, 64 i 128 bitów. Typy `isize` i `usize` są zależne od architektury, na której jest komplikowany kod. Na przykład: jeżeli komplikujemy kod na architekturze 64-bitowej, to będą one miały rozmiar 64 bitów.

Typów zmiennoprzecinkowych jest stanowczo mniej. Mianowicie dzielą się one na `f32` i `f64`, które reprezentują odpowiednio 32 i 64-bitowe liczby zmiennoprzecinkowe. Podobnie jak w przypadku typu `"number"` w języku JavaScript, są one zapisane w standardzie IEEE-754-2008 [23].

Jak opisałem w podrozdziale 2.1.2, operacje bitowe w języku JavaScript zamieniają liczby po obu stronach operatora na 32-bitowe liczby całkowite. Rust pozwala na operowanie na liczbach całkowitych większych niż 32 bity nieogranicząc się jedynie do operacji

Rozmiar	Ze znakiem	Bez znaku
8-bitów	<i>i8</i>	<i>u8</i>
16-bitów	<i>i16</i>	<i>u16</i>
32-bity	<i>i32</i>	<i>u32</i>
64-bity	<i>i64</i>	<i>u64</i>
128-bitów	<i>i128</i>	<i>u128</i>
według architektury	<i>isize</i>	<i>usize</i>

Tabela 2.1: Typy reprezentujące liczby całkowite

arytmetycznych. Wszystkie operacje bitowe mogą być wykonywane na liczbach 64-bitowych, co daje nam większe możliwości optymalizacji kodu. Dodatkowo dzięki temu, że rozmiar liczb na których operujemy jest znany, kompilator odkłada liczby na stosie, co w teorii pozwala na szybsze operacje niż w przypadku typu "**bigint**" w języku JavaScript.

Typy tekstowe

Rust posiada wiele typów reprezentujących ciągi znaków. Niektóre z nich są typami specjalistycznymi, które pozwalają na optymalizację kodu, inne pozwalają na lepszą elastyczność, natomiast wszystkie gwarantują nam bezpieczeństwo. Poniżej wymieniłem niektóre z reprezentacji ciągu znaków, które są spotykane w języku Rust:

- `String` i `&String`
- `&str` i `&'static str`
- `Box<str>`
- `Cow<'a, str>`
- `Rc<str>` i `Arc<str>`
- `CString` i `CStr`
- `OsString` i `OsStr`
- `Path` i `PathBuf`

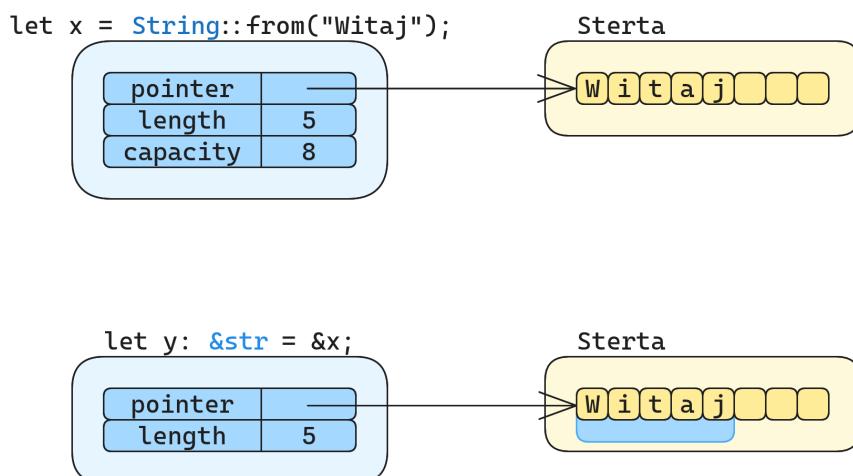
Każdy z wyżej wymienionych typów jest przydatny i ma swoje zastosowanie, lecz spośród wszystkich, pierwsze cztery są najczęściej spotykane. Dlatego skupię się właśnie na nich.

Pierwszym typem jest `String`. Jest to alokowana na stercie, rozszerzalna, enkodowana do poprawnego UTF-8 reprezentacja ciągu znaków, która jest właścicielem wartości, którą przechowuje w sobie [24]. Inaczej mówiąc, jest to struktura zbudowana dokładnie tak samo, jak wektor wartości `u8`, do którego możemy dynamicznie dodawać elementy. `String` jest

więc reprezentacją ciągu znaków, który może być modyfikowalny. Przechowuje on wskaźnik na dane ze sterty, swoją długość oraz maksymalną pojemność.

Typ `&String` jest pożyczaniem typu `String`. Pożyczenia tłumaczę dokładniej w rozdziale 2.2.2, natomiast na chwilę obecną możemy potraktować je jako widoki na konkretne dane ze sterty, które nie są właścicielami tych danych.

Typ `&str` jest dokładnie tym samym co `&String`, z tą różnicą, że `&str` jest niezmienialny. Możemy przejść z typu `&String` na `&str`, natomiast w drugą stronę już nie. Przedstawia on widok na串 znaków, który nie jest modyfikowalny i składa się ze wskaźnika i ilości znaków zapisanych na stercie. Rysunek 2.6 przedstawia różnice pomiędzy typem `String` a `&str` i wizualizuje sposób ich reprezentacji w pamięci.



Rysunek 2.6: Reprezentacja typów `String` i `&str` w pamięci

W przeciwieństwie do typu `String`, który może wskazywać jedynie na dane zaallokowane na stercie, `&str` może również wskazywać na dane znajdujące się na stosie, jak i bezpośrednio w skompilowanym programie. Przykładem tego ostatniego jest typ `&'static str`, który jest pochodną typu `&str`. Oznaczenie „`'static`” mówi o tym, że串 znaków jest dostępny przez cały czas działania programu. Jest to typ, który zostanie automatycznie przypisany zmiennej, gdy użyjemy tak zwanego literala串 znaków, czyli po prostu串 znaków zamkniętego w podwójnym cudzysłowie. Na przykład w linijce `let x = "Witaj";`, zmienna „`x`” będzie miała typ `&'static str`, a串 „Witaj” zostanie zawarty w pliku wykonywalnym.

Rust enkoduje stringi do poprawnego UTF-8, natomiast JavaScript enkoduje je do UTF-16. Wymiana danych pomiędzy kontekstami wymaga więc konwersji, co może negatywnie wpływać na wydajność aplikacji.

Tablice i krotki

Tablice i krotki są zbiorami wielu elementów o z góry określonej długości. Raz zadeklarowane, nie mogą więcej zmieniać swojego rozmiaru. Te pierwsze są deklarowane jako $[T; N]$, gdzie T jest typem elementu, a N jest długością tablicy. Natomiast drugie są deklarowane jako (T_1, T_2, \dots, T_n) , gdzie T_1, T_2, \dots, T_n to typy elementów krotki. Oba zbiory wymagają tego, by ich elementy miały znany rozmiar. Pozwala to na odłożenie ich na stosie [25].

Możliwość stworzenia tablicy lub krotki odłożonej na stosie daje dużą przewagę wydajnościową. W języku JavaScript każda lista alokowana jest na stercie, która (przez swoją naturę) jest wolniejsza od stosu. Dzięki temu, operacje takie jak zwrócenie wielu wartości z funkcji, będą szybsze gdy zwrócimy kilka elementów w krotce niż gdybyśmy musieli zaalokować nowy obiekt w JavaScript.

Wektory

Jeżeli potrzebujemy listy, która może zmienić liczbę swoich elementów w trakcie działania programu, idealnym rozwiązaniem są wektory. Są to listy dynamicznej długości, które rozszerzają swoją pojemność w momencie, gdy kończy się im dotychczas zaalokowane miejsce. Każdy wektor posiada swoją długość, pojemność oraz wskaźnik na dane, które przechowuje. Nie bez powodu przypomina to typ `String` opisany w rozdziale 2.2.1. Język Rust deklaruje go jako strukturę, która przechowuje w sobie wektor wartości `u8`. Kod przedstawiony na listingu 2.6 jest deklaracją tego typu wziętą bezpośrednio z kodu źródłowego języka Rust.

```
1 pub struct String {  
2     vec: Vec<u8>,  
3 }
```

Listing 2.6: Deklaracja typu `String` w języku Rust [26]

Struktury

Podczas gdy obiekty w języku JavaScript przypominają funkcjonalnie dynamiczne słowniki, struktury w Rustie posiadają z góry zdefiniowane pola. Każdy pole ma swój typ, który musi być zadeklarowany w momencie tworzenia struktury.

Jak omówiłem w rozdziale 2.1.3, silniki JavaScriptowe tworzą ukryte klasy dla każdego obiektu. Jest to mechanizm, który pozwala na optymalizacje tych obiektów pomimo ich dynamicznej natury. Z racji na to, że Rust jest językiem kompilowanym AOT, a struktury nie mogą ulec zmianie, nie musi on tworzyć niczego na wzór ukrytych klas. Struktury są reprezentowane jako ciągły blok pamięci, co pozwala to na szybki dostęp do każdego z pól. Jest to możliwe dzięki temu, że kompilator wie, ile miejsca będzie potrzebował na zaalokowanie każdego typu, który może być częścią struktury. [27]

```
1 struct Person {
2     first_name: String,
3     last_name: String,
4 }
5
6 impl Person {
7     fn new(first_name: String, last_name: String) -> Self {
8         Self {
9             first_name,
10            last_name,
11        }
12    }
13
14    fn full_name(&self) -> String {
15        format!("{} {}", self.first_name, self.last_name)
16    }
17 }
```

Listing 2.7: Przykład struktury w języku Rust

Struktury, jak i niżej opisane wyliczenia mogą posiadać implementacje metod. Na listingu 2.7 znajduje się przykład struktury „Person” z metodą „new” zwracającą nową instancję tej struktury oraz „full_name” zwracającą pełne imię i nazwisko. Podanie argumentu „self” w implementacji pozwala na stworzenie metody odnoszącej się do instancji struktury. Funkcje bez tego argumentu będą metodami statycznymi, do których można odnieść się używając

nazwy struktury, a następnie nazwy metody po podwójnym dwukropku. Na przykład:
`Person::new("Kasper", "Seweryn").`

Wyliczenia i wyrażenie match

Szczególnym rodzajem struktur w Rustie są wyliczenia (ang. *enum*). Są to typy definiowane przez użytkownika, które składają się z wariantów. Różnią się one od wyliczeń w innych językach. Większość implementacji pozwala na przechowywanie jedynie jednego konkretnego typu jako wartość wyliczenia. Natomiast Rust pozwala dla każdego wariantu wyliczenia na przechowywanie własnego typu, który może różnić się od typów zdefiniowanych w innych wariantach tego wyliczenia.

```
1 // Własne wyliczenia
2 enum Language {
3     JavaScript,
4     Rust,
5     C,
6 }
7
8 enum FError {
9     RegexError,
10    OtherError(String),
11 }
12
13 // Reimplementacja natywnych wyliczeń
14 enum Option<T> {
15     Some(T),
16     None,
17 }
18
19 enum Result<T, E> {
20     Ok(T),
21     Err(E),
22 }
```

Listing 2.8: Przykłady wyliczeń w Rust

Na listingu 2.8 znajdują się cztery przykłady wyliczeń. Pierwszy z nich, `Language`, zawiera trzy warianty. Aby odnieść się do jednego z tych wariantów należy wpierw odnieść się do nazwy wyliczenia, następnie wstawić podwójny dwukropek i podać pożądany wariant.

Jeżeli używamy wyliczenia bardzo często, możemy zaimportować pełną przestrzeń nazw przy pomocy klauzuli `use Language::*`.

W języku Rust, wartość zawsze musi być zdefiniowana i odpowiadać typowi zmiennej, dlatego gdy potrzebujemy oznaczyć brak wartości możemy zrobić to przy pomocy `Option<T>`. W przeciwieństwie do innych języków, nie ma tu konceptu `nullptr` z języka C (czy `undefined` i `null` z języka JavaScript). `Option<T>` jest jednym z kilku wyliczeń znajdujących się w standardowej bibliotece i jest ono dostępne globalnie wraz ze swoimi wariantami. Jego definicja znajduje się na listingu 2.8, natomiast listing 2.9 reprezentuje przykład jego użycia.

```
1 fn main() {
2     let favorite_language = Some(Language::Rust);
3     check(favorite_language);
4 }
5
6 fn check(language: Option<Language>) -> Result<(), String> {
7     match language {
8         Some(Language::Rust) => {
9             println!("You are hired!");
10            return Ok(());
11        }
12        Some(x) => {
13            println!("We don't need {:?}", x);
14            return Err("Doesn't like rust".to_string());
15        }
16        None => {
17            println!("You have to learn Rust.");
18            return Err("Doesn't like rust".to_string());
19        }
20    }
21 }
```

Listing 2.9: Przykład użycia klauzuli „match”

Jak widać, do zmiennej „`favorite_language`” przypisany jest wariant `Some(Language::Rust)`, który jako wartość przechowuje jeden z wariantów typu `Language`. Jeżeli chcielibyśmy wyrazić brak ulubionego języka, moglibyśmy to zrobić przypisując tej zmiennej wartość `None`. W funkcji „`check`” sprawdzamy wartość zmiennej przy użyciu wyrażenia „`match`”. Wyrażenie to wymaga od nas tego, aby każda wartość —

jaką zmienną może przyjąć — była obsłużona. W przeciwnym wypadku dostaniemy błąd komilacji.

Gdy funkcje zwracają wyliczenia, zwracają one wartość konkretnego typu. Jeżeli ta wartość jest typu `Option<T>`, albo `Result<T, E>` (jak w przykładzie), to należy ją odpowiednio obsługiwać, by dostać się do wartości kryjącej się wewnątrz wariantu. `Result<T, E>` jest również globalnie dostępny i oznacza rezultat wykonania jakiejś funkcji. Funkcja zwraca wariant `Ok()` oznaczający poprawne wykonanie bez zwrotnej wartości, albo `Err(String)` oznaczający błąd przechowujący pewien komunikat. Warto zaznaczyć, że komplator ostrzeże nas o tym, że na linijce 3 nie używamy wartości zwracanej przez funkcję „check”. Zachęca nas do obsługiżenia rezultatu zwracanego przez funkcję.

```
1 const findWordsByPrefix = (text, prefix) => {
2     return text.match(new RegExp(`(?:\\s|^)${prefix}\\S+`, 'gi'))
3 }
```

Listing 2.10: Niebezpieczna funkcja w języku JavaScript

Dzięki takiemu mechanizmowi mamy pewność, że każdy przypadek został obsłużony. Gdy chcemy użyć jakiejś funkcji, ze względu na to, że zwracane dane są opakowane w `Option<T>` lub `Result<T, E>`, musimy je najpierw rozpakować, obsługując wszystkie możliwe przypadki. Język JavaScript nie ma żadnego syntaktycznego zabezpieczenia przed funkcjami, które mogą zwrócić `undefined`, `null` lub wyrzucić błąd. W listingu 2.10 znajduje się prosta i na pozór niegroźna funkcja, która znajduje w tekście słowo rozpoczynające się od zadanego przedrostka. Funkcja ta może zwrócić nam albo listę ze wszystkimi znalezionymi słowami, albo wartość `null` w momencie gdy przedrostek nie istnieje. Jeżeli spróbujemy wykonać `findWordsByPrefix(text, '[')`, to znak „[” będzie potraktowany jako część wyrażenia regularnego i zostanie wyrzucony błąd „Uncaught SyntaxError: unterminated character class”. W powyższym zapisie nic nie wskazuje na to, że funkcja może zwrócić `null` lub wyrzucić błąd. Gdy zestawimy to zachowanie z językiem Rust, okazuje się, że zwrócenie typu `Result<Option<Vec<String>>, FError>` rozwiązuje cały problem. Widzimy dokładnie co może zwrócić funkcja i obsługuje każdy przypadek przy pomocy prostego wyrażenia „match” przedstawionego na listingu 2.11.

```

1 fn find(w: String, p: String) -> Result<Option<Vec<String>>, FError> {
2     // Implementacja funkcji
3 }
4
5 fn main() {
6     let result = find("Witaj świecie!".to_string(), "Wi".to_string());
7
8     // Dokładna obsługa błędów
9     match result {
10         Ok(Some(words)) => println!("Słowa: {:?}", words),
11         Ok(None) => println!("Brak słów."),
12         Err(FError::RegexError) => println!("Błąd wyrażenia."),
13         // Zmienna "_" oznacza, że nie interesuje nas wartość
14         Err(FError::OtherError(_)) => println!("Inny błąd."),
15     }
16 }
```

Listing 2.11: Bezpieczne obsłużenie funkcji w języku Rust

2.2.2 System własności, a mechanizm „Garbage Collector”

Język Rust nie posiada mechanizmu Garbage Collector, który odpowiadałby za automatyczne zarządzanie pamięcią. Nie oznacza to jednak, że musimy ręcznie alokować i zwalniać pamięć. W zamian za to Rust posiada system własności, który jest zbiorem reguł, które są sprawdzane podczas komplikacji. Jeżeli którakolwiek z nich zostanie złamana, program się nie skompiluje. Dzięki temu zarządzanie pamięcią nie wpływa na wydajność kodu podczas jego wykonywania. Zasady, którymi się kieruje, są następujące:

- Każda wartość w Rust ma swojego właściciela.
- Może być tylko jeden właściciel na raz.
- Jeżeli właściciel wyjdzie poza zakres, wartość zostanie zniszczona.

Reguły te, choć na pozór wydają się proste, wymagają od nas nowego spojrzenia na podstawowe aspekty programowania.

Aby lepiej zrozumieć, w jaki sposób system własności alokuje i zwalnia zmienne, rozważmy przykład przedstawiony na listingu 2.12. Widnieje na nim blok kodu definiujący zakres, w którym następuje alokacja zmiennej „s”. Jest ona typu `String`, który z racji na swoją dynamiczną naturę, musi być alokowany na stercie. Zmienna „s” jest dostępna od momentu deklaracji, aż do końca zakresu (czyli końca bloku). Po wyjściu poza zakres,

```

1 {
2     // s nie jest jeszcze zadeklarowana
3
4     let s = String::from("witaj świecie"); // alokacja s
5     // s jest dostępne
6
7 } // s jest zwolnione

```

Listing 2.12: Zakres zmiennej „s”

zmienna „s” jest zwalniana, przez co staje się niedostępna poza zakresem. Każda próba odwołania się do niej zostanie wychwycona przez kompilator, który zwróci odpowiedni błąd.

Zwolnienie zaalokowanej zmiennej przy wyjściu z zakresu może wydawać się niebezpieczną operacją. Co stałoby się, gdybyśmy zrobili przypisanie `let s2 = s;`? W języku JavaScript moglibyśmy nazwać taką operację płytką kopią, gdzie kopujemy wskaźnik na dane ze zmiennej „s” do zmiennej „s2”. Po wyjściu z zakresu oba wskaźniki zostałyby zniszczone, a GC posprzątałby obiekt podczas następnego wywołania. W Rust musielibyśmy zwolnić zarówno „s” jak i „s2”, co mogłoby prowadzić do błędu typu „double free”. Na szczęście, system własności wymaga tego, by właścicielem wartości była tylko jedna zmienna jednocześnie. Oznacza to, że nasze potencjalne przypisanie przypominałoby bardziej przeniesienie pamięci ze zmiennej „s” do zmiennej „s2” i unieważnienie zmiennej źródłowej.

Jeżeli naszym celem byłoby operowanie na dwóch różnych stringach „s” i „s2” oddzielnie, potrzebowalibyśmy mechanizmu głębokiego kopiowania. Rust nigdy nie dokonuje tego rodzaju kopii ze względu na jej niską wydajność. Aby dokonać takiej operacji, musielibyśmy sami wykonać na zmiennej „s” metodę „clone” w następujący sposób: `let s2 = s.clone();`. W tym momencie „s” pokazuje na jedną wersję stringa „witaj świecie”, a „s2” pokazuje na inną wersję znajdującą się w innym miejscu sterты.

Póki co omówiliśmy jedynie zmienne alokowane na stercie. Zmienne o znanych rozmiarach, które są odkładane na stosie, podobnie jak zmienne prymitywne w JavaScript będą po prostu kopiowane na stos.

System własności a funkcje

W wielu językach argumenty przekazywane są na dwa sposoby — po wartości albo po referencji. Rust, ze względu na system własności musiał obrać inną ścieżkę. Przekazując argumenty przekazuje się własność, kopiuje się wartość albo przekazuje się referencje.

```
1 fn foo() {
2     let s = String::from("witaj świecie");
3     bar(s); // Własność wartości „s” jest oddana do funkcji bar.
4         // Nasępne próby odniesienia do „s” będą nieprawidłowe.
5
6     let i = 42;
7     baz(i); // Wartość zmiennej „i” jest skopiowana do funkcji baz.
8         // Można bezpiecznie odnosić się do „i” w funkcji foo.
9 } // „i” wychodzi poza zakres i jest zwalniane, natomiast wartość „s”
10    // została przeniesiona do funkcji bar, gdzie kontynuuje swój żywot
11
12 fn bar(s2: String) {
13     println!("{}", s2);
14 } // „s2” wychodzi poza zakres i jest zwalniane
15
16 fn baz(i2: i32) {
17     println!("{}", i2);
18 } // „i2” wychodzi poza zakres i jest zwalniane
```

Listing 2.13: Przykład przekazania własności zmiennej „s” z funkcji „foo” do „bar”

Dwa pierwsze podejścia zostały przedstawione na listingu 2.13. Różnica między przekazaniem własności, a skopiowaniem wartości jest identyczna jak w przypadku rozważanego wcześniej przypisania. Zmienne, których rozmiar jest znany, a więc są odłożone na stos - będą kopiowane. Dzięki temu można się do nich bezpiecznie odwoływać nawet, gdy przekazaliśmy je do innej funkcji. Zmienne zaalokowane na stercie będą przekazywały swoją własność do wywołanej funkcji, sprawiając, że każde następne odwołanie do tej zmiennej zostanie wychwycone przez kompilator i zostanie wyrzucony błąd.

Ostatni sposób, to przekazanie referencji. Referencję można rozumieć jako wskaźnik, który pokazuje na przekazywane dane, ale odróżnieniu od niego referencja gwarantuje nam to, że zawsze wskazuje na poprawne miejsce w pamięci, w którym znajdują się pożądane dane. Listing 2.14 przedstawia przykładowe przekazanie stringa po referencji. Zapis `&s` tworzy

```

1 fn foo() {
2     let s = String::from("witaj świecie");
3
4     bar(&s); // wartość „s” jest pożyczona do funkcji bar.
5
6     // Funkcja bar zakończyła swoje działanie
7
8     bar(&s); // wartość „s” jest pożyczona do funkcji bar.
9
10    // Funkcja bar zakończyła swoje działanie
11
12 } // „s” wychodzi poza zakres i jej wartość jest zwalniana.
13
14 fn bar(s2: &String) {
15     println!("{}", s2);
16 } // Wartość na którą wskazuje „s2” nie jest zwalniana
17 // ponieważ „s2” jest referencją.

```

Listing 2.14: Przykład pożyczania zmiennej „s” z funkcji „foo” do „bar”

referencję do zmiennej „s”, która jest przekazana do funkcji „bar”. Funkcja ta przyjmuje argument typu `&String`, a więc referencję na stringa. Oznacza to, że nigdy nie będzie ona właścicielem, a jedynie pożyczycy wartość kryjącą się pod zmienną „s”. Mechanizm przekazywania referencji jest w języku Rust nazywany pożyczaniem.

Rust dzieli pożyczania na modyfikowalne i niemodyfikowalne. Mechanizm nazwany „Borrow Checker” sprawdza ilość wystąpień konkretnego rodzaju pożyczzeń. Jednocześnie może występować albo jedno modyfikowalne pożyczanie albo wiele niemodyfikowalnych pożyczzeń. Zapobiega on wyścigowi danych, który jest trudnym do zdiagnozowania problemem. Aby dokonać modyfikowalnego pożyczania należy oznaczyć pożyczanie nie jako `&s`, a jako `&mut s`. Dodatkowo, każda zmienna, która zostanie podana jako modyfikowalny argument, musi sama być oznaczona jako modyfikowalna przez `let mut s = /* ... */;`. Kompilator wymaga od nas świadomości, które zmienne mogą ulec zmianie, a które nie. Pozwala to uniknąć wielu błędów podczas pisania i zapewnia, że każda funkcja, której użyjemy, nie będzie czarną skrzynką pod aspektem modyfikowania argumentów.

Rozróżnienie syntaktyczne pomiędzy przekazaniem własności, a pożyczaniem wartości pozwala kompilatorowi lepiej zrozumieć nasz kod. Wie on kiedy dokładnie ma zwolnić

pamięć, dzięki czemu nie potrzebuje mechanizmu GC, który uruchamia się w losowych (z perspektywy użytkownika) momentach i iteruje się po wszystkich wartościach na stercie.

2.3 WebAssembly

WebAssembly to technologia, która pozwala na uruchamianie niskopoziomowego kodu bezpośrednio w przeglądarce. Posiada ona zarówno format tekstowy (WAT) [28] oraz binarny. Wiele języków, w tym Rust, mogą być skompilowane do binarnego formatu WASM, co pozwala na uruchamianie kodu pisanej w tych językach z prędkością bliską natywnej. Współdzielona pamięć z JavaScriptem pozwala na komunikację pomiędzy obiema technologiami. Umożliwia to odwołania do API DOM oraz innych Web API dostępnych jedynie w kontekście przeglądarkowym.

2.3.1 Kompilacja do binarnego formatu WebAssembly

Kompilacja do WebAssembly jest stosunkowo prosta w przypadku języka Rust. Narzędzie *wasm-bindgen* [29] pozwala na oznaczanie funkcji Rustowych, które mają być eksportowane do modułu WASM oraz funkcji JavaScriptowych, które mają być dostępne z poziomu Rusta. Listing 2.15 przedstawia przykładowy kod, który wykorzystuje funkcję „`alert`” z kontekstu JavaScriptowego i eksportuje do niego funkcję „`przywitanie`”. Kod może być następnie skompilowany przy użyciu polecenia `wasm-pack build`. W pierwszym kroku, polecenie to kompliluje nasz program do binarnego formatu WebAssembly. Następnie generuje pliki `.js`, które są kodem-klejem pozwalającym na prostą komunikację pomiędzy obiema technologiami. [30]

Podstawowo, pliki `.wasm` wygenerowane przez kompilator mogą zajmować nawet do kilkuset kilobajtów. Jest to przyczyną wielu krytycznych opinii na temat WebAssembly. Należy jednak pamiętać, że mediana rozmiaru strony internetowej, według HTTP Archive, na dzień 1 Maja 2024 roku wynosi 2610.9 KB. Z tego 1055.3 KB to zdjęcia, a następne 627.4 KB przeznaczone jest na skrypty w języku JavaScript i dane w formacie JSON. [31] W pełnej perspektywie, krytyka ta wydaje się nieuzasadniona. Jeżeli zależy nam na zminimalizowaniu rozmiaru zasobów, powinniśmy w pierwszej kolejności zatroszczyć się o rozmiar zdjęć, a następnie ograniczyć ilość wykonywanych skryptów do minimum.

```

1 use wasm_bindgen::prelude::*;

2 // Deklaracja funkcji udostępnionych przez kontekst JavaScript
3 #[wasm_bindgen]
4 extern "C" {
5     pub fn alert(s: &str);
6 }

7 // Deklaracja funkcji udostępnionych do kontekstu JavaScript
8 #[wasm_bindgen]
9 pub fn przywitanie(imie: &str) {
10     alert(&format!("Witaj, {}!", imie));
11 }
12
13 }
```

Listing 2.15: Przykład kodu eksportującego funkcję do WASM

Rozmiar plików .wasm można w prosty sposób zmniejszyć, włączając w kompilatorze tak zwane „Link Time Optimizations” (LTO). Pozwala to na lepsze wstawienie i przycinanie funkcji na etapie linkowania, co pozytywnie wpływa zarówno na rozmiar jak i wydajność wykonywanego kodu [32] kosztem czasu komplikacji.

Kolejną techniką, która pozwala na zmniejszenie rozmiaru plików .wasm jest zmiana parametru „opt-level” z wartości oznaczającej optymalizację dla szybkości na optymalizację dla rozmiaru. Rezultaty takiej zmiany nie są gwarantowane. W niektórych przypadkach oryginalna wartość może wygenerować plik mniejszy niż nowa, dlatego warto zbadać obie możliwości.

Następnym sposobem na zmniejszenie rozmiaru plików .wasm jest użycie narzędzia *wasm-opt* [33]. Przetwarza ono plik .wasm, nakładając na niego wiele optymalizacji, w wyniku czego plik wyjściowy może być nawet do 20% mniejszy (nawet po nałożeniu poprzednich optymalizacji). [34]

Ostatnim sposobem jest skompresowanie pliku przy użyciu narzędzia gzip, które jest w stanie zredukować rozmiar binarnego formatu WASM nawet o ponad 50%. [35]

2.3.2 Wykonywanie kodu WebAssembly

Aby wykonać kod napisany w WebAssembly, musimy wpierw pobrać plik .wasm do przeglądarki z poziomu języka JavaScript. To w jaki sposób moduł WASM zostanie pobrany jest w pełni zależne od nas. Pierwotnie byliśmy zmuszeni pobrać cały plik binarny i załadować go do obiektu `ArrayBuffer`, który potem mogliśmy podać do odpowiednich funkcji

inicjalizujących lub kompilujących moduł WebAssembly. W pewnym momencie okazało się, że kompilacja modułu WASM do kodu maszynowego jest szybsza niż prędkość, z jaką pobieramy dane z internetu [36], więc silniki JavaScriptowe dodały możliwość kompilacji i inicjalizacji modułów WASM bezpośrednio ze strumienia bajtów wysyłanego z serwera. Jest to stanowczo szybsze od pobierania całego pliku, a następnie interpretowania i kompilowania go (jak jest to rozwiążane w przypadku JavaScriptu).

Po pobraniu modułu należy go zainicjalizować. Listing 2.16 przedstawia kod tworzący moduł WASM bezpośrednio ze strumienia danych. Funkcja ta podaje strumień danych do kompilatora JIT w przeglądarce, którego zadaniem jest skompilowanie go do kodu maszynowego, który będzie później wykonany. WarpMonkey ma dwa dodatkowe podsilniki: RabaldrMonkey oraz BaldrMonkey. Pierwszy z nich dokonuje szybkiej kompilacji WASM do kodu maszynowego aby zminimalizować czas do pierwszego uruchomienia. Drugi kompiluje WASM do identycznej reprezentacji pośredniej, której używa WarpMonkey, aby wygenerować bardzo szybki kod maszynowy [37].

```
1 const wasmModule = await WebAssembly.instantiateStreaming(
2   fetch('module.wasm')
3 )
4
5 // Zakładamy, że funkcja `main` jest eksportowana z modułu
6 wasmModule.instance.exports.main()
```

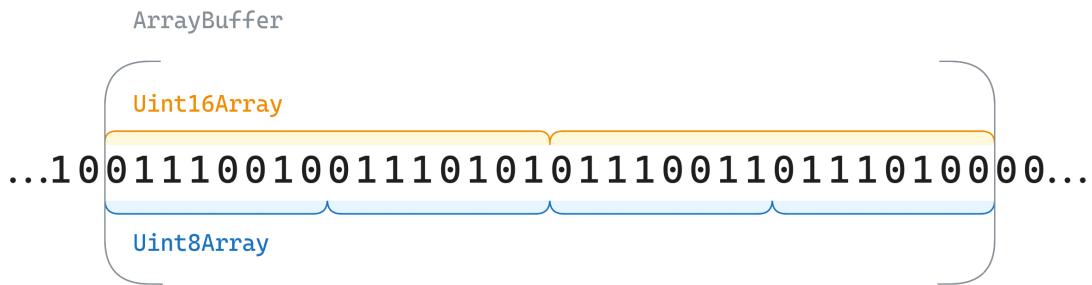
Listing 2.16: Inicjalizacja modułu WebAssembly z pliku binarnego

2.3.3 Zarządzanie pamięcią

WebAssembly jest technologią, która wymaga od użytkownika ręcznego zarządzania pamięcią. Oznacza to, że program skompilowany do WASM jest odpowiedzialny za sprzątanie pamięci, którą zaalokował. Z tego powodu Rust wydaje się wręcz idealnym językiem. Nie posiada on Garbage Collectora, a system własności pozwala na bezpieczne i automatyczne zarządzanie pamięcią.

Aby przekazać pamięć do WASM, JavaScript udostępnia mu instancję obiektu `ArrayBuffer`. Jest on izolowanym wycinkiem liniowej pamięci, który przechowuje jedynki i zera. Z poziomu JavaScriptu nie możemy go bezpośrednio modyfikować, ponieważ same dane binarne nie mówią nam, jakie dane są zapisane w środku. Musimy więc

skorzystać z typowanych tablic takich jak na przykład `Uint8Array`, czy `Uint16Array`. Dzięki temu możemy traktować każdy indeks takiej tablicy jako reprezentację pojedynczej liczby 8-bitowej lub 16-bitowej. Rysunek 2.7 przedstawia sposób, w jaki są reprezentowane te trzy obiekty w pamięci przeglądarki. Natomiast tabela 2.2 przedstawia wszystkie możliwe reprezentacje bufora jako typowane tablice w pamięci.



Rysunek 2.7: Reprezentacja ArrayBuffer, Uint8Array i Uint16Array w pamięci

Rozmiar elementu	Typowane tablice
8-bitów	<code>Int8Array</code> , <code>Uint8Array</code> , <code>Uint8ClampedArray</code>
16-bitów	<code>Int16Array</code> , <code>Uint16Array</code> , <code>Floating32Array</code>
32-bity	<code>Int32Array</code> , <code>Uint32Array</code> , <code>Floating32Array</code>
64-bity	<code>BigInt64Array</code> , <code>BigUint64Array</code> , <code>Floating64Array</code>

Tabela 2.2: Typowane tablice w JavaScript

JavaScript przekazuje wszystkie zmienne do kontekstu WASM właśnie poprzez `ArrayBuffer`. Listing 2.17 przedstawia przykładowy kod pozwalający na przekazanie stringa do modułu WebAssembly. Ponieważ, jak opisałem w podrozdziale 2.1.2, stringi w języku JavaScript są enkodowane do UTF-16, należy użyć obiektu `TextEncoder`, aby reenkodować je do listy bajtów (`Uint8Array`). Następnie należy ją wstrzyknąć do instancji `ArrayBuffer` pełniącej rolę pamięci modułu WASM na odpowiednim indeksie. Przykładowy kod zakłada, że moduł zwraca funkcję „allocate”, która pozwala na zaalokowanie odpowiedniej ilości pamięci i zwróci wskaźnik na zaalokowane miejsce. W prawdziwym zastosowaniu kompilator wygenerowałby nie tylko moduł WASM, ale również JavaScriptowy kod-klej (ang. *glue code*), który pozwala na prostą komunikację między kontekstami bez konieczności ręcznego manipulowania pamięcią.

```

1  const { memory, allocate } = wasmModule.instance.exports
2  const wasmMemory = new Uint8Array(memory.buffer)
3
4  const pushString = (str) => {
5    const bytes = new TextEncoder().encode(str)
6
7    const ptr = allocate(bytes.length)
8    wasmMemory.set(bytes, ptr)
9
10   return ptr
11 }

```

Listing 2.17: Przekazanie stringa do modułu WebAssembly

Podobnie jak zapis, odczyt zmiennych również zachodzi poprzez pamięć reprezentowaną przez `ArrayBuffer`. Jeżeli chcemy odczytać wartość stringa znajdującego się pod wskaźnikiem zwróconym przez funkcję wyeksportowaną z modułu, musimy wpierw znać jego długości. Następnie musimy wyłuskać odpowiednią ilość bajtów z pamięci modułu i podać je do instancji obiektu `TextDecoder`, aby otrzymać wynikowy string.

2.3.4 Operacje SIMD

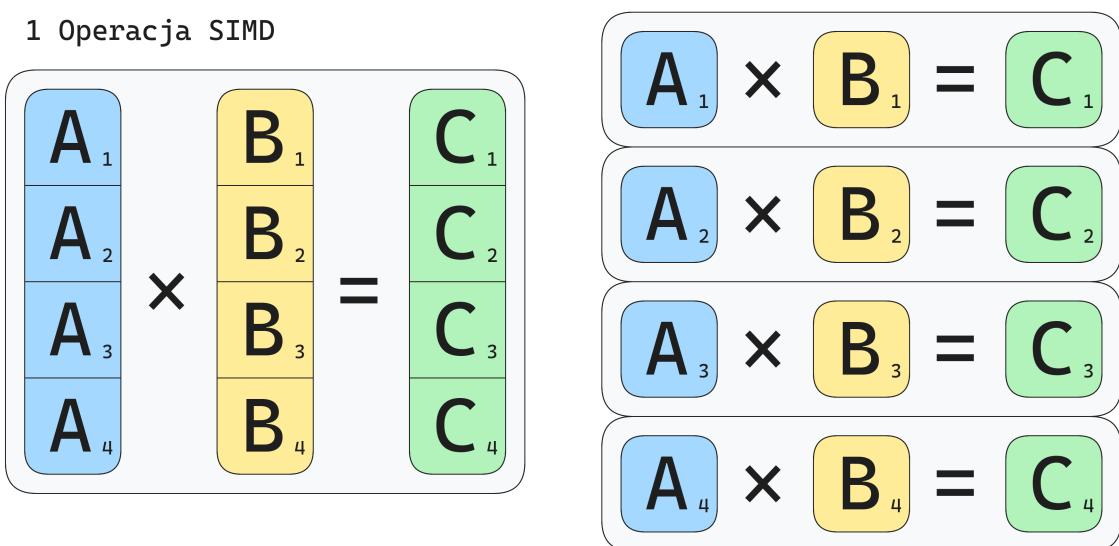
Najatrakcyjniejszą cechą WebAssembly jest jego wsparcie dla operacji SIMD (Single Instruction, Multiple Data). Polegają one na jednoczesnym wykonaniu tej samej instrukcji na wielu danych. W praktyce wygląda to tak, że możemy załadować 16 liczb 8-bitowych, 8 liczb 16-bitowych, 4 liczby 32-bitowe lub 2 liczby 64-bitowe do jednego 128-bitowego wektora. Następnie możemy wykonać operacje arytmetyczne, logiczne, czy bitowe na tych wektorach, co pozwala na znaczące przyspieszenie obliczeń. Rysunek 2.8 przedstawia mnożenie parami ośmiu liczb 32-bitowych zaimplementowane jako operacja SIMD i operacje skalarne.

Operacje SIMD są kilkukrotnie szybsze od operacji skalarnych, co sprawia, że są idealne do przetwarzania grafiki, dźwięku czy sztucznej inteligencji. Zamiast wykonywać 16 dodawań, możemy załadować wszystkie 8-bitowe liczby do jednego 128-bitowego wektora i wykonać jedno dodawanie.

Aplikacje przeglądarkowe były do niedawna ograniczone jedynie do operacji skalarnych. Od 2014 roku Mozilla, Intel oraz Google wspólnie pracowali nad niskopoziomowym API nazwanym SIMD.js [38]. Komisja TC39 zajmująca się standardem ECMAScript przyznała

4 Operacje skalarne

1 Operacja SIMD



Rysunek 2.8: Porównanie operacji SIMD z operacjami skalarnymi

temu API status „Stage 3”, co oznacza, że jest ono rekomendowane do implementacji. W 2017 roku API to zostało jednak porzucone na rzecz WebAssembly, które miało dać możliwość bezpośredniego dostępu do instrukcji SIMD [39]. Od 2023 roku wszystkie najpopularniejsze przeglądarki desktopowe wspierają operacje SIMD w WebAssembly [40] co pozwala na znaczące przyspieszenie niektórych obliczeń w aplikacjach przeglądarkowych.

3. Analiza dotychczasowych badań

WebAssembly staje się coraz atrakcyjniejszą technologią dla stron i aplikacji wymagających wydajności. Powstaje coraz więcej badań wskazujących na przewagę WebAssembly nad JavaScriptem nie tylko pod kątem szybkości, ale również wydajności energetycznej [41].

3.1 Mikrobenchmarki WebAssembly i JavaScript

W pracy magisterskiej zatytułowanej „Analyzing front-end performance using WebAssembly” [42], Jacob Nilsson i Andreas Trattner zbadali wydajność technologii porównywanych w tej pracy. Wykazali oni, że wydajność algorytmów zaimplementowanych w Rust skompilowanym do WebAssembly jest szybsza niż w przypadku czystego JavaScriptu, a serializacja i deserializacja skomplikowanych struktur podczas przenoszenia pamięci między kontekstami generuje dodatkowy narzut, który negatywnie wpływa na wyniki WebAssembly.

Doszli oni do wniosków, że WebAssembly może poprawić czas wykonania kodu w aplikacjach webowych pod warunkiem, że kod spełnia jeden z trzech warunków:

- argumenty i wartości zwracane z funkcji zaimplementowanej w WASM nie wymagają przetwarzania
- argumenty i wartości zwracane z funkcji wymagają przetwarzania, ale czas przetwarzania jest mały w porównaniu z czasem wykonania algorytmu
- dane mogą zostać przesłane pomiędzy kontekstami bezpośrednio poprzez obiekt reprezentujący pamięć WASM

Wszystkie testy przeprowadzone przez autorów skupiły się na wydajności algorytmicznej oraz optymalizacjach, które oferują kompilator języka Rust i kompilator JIT dla JavaScriptu. Aby zbadać wydajność obu technologii, autorzy przetestowali następujące aspekty obu technologii:

- wydajność sortowań quicksort i mergesort
- wydajność mapowania obiektów na liczby i stringi
- wydajność grupowania obiektów względem klucza
- wydajność filtrowania listy stringów
- wydajność algorytmu kNN

- wydajność generowania obrazów
- wydajność generowania kodu QR
- wydajność problemu pakowania pojemników (ang. *bin packing problem*)

Nie sprawdzili oni wpływu operacji SIMD na wydajność, co mogłoby znacząco wpływać na wyniki. Pominęli oni również wykonanie testów związanych z interakcją z API DOM ani nie zbadali przewagi używania typów numerycznych takich jak *u64* czy *i64* nad typami "bigint".

W mojej pracy postaram się sprawdzić, czy wnioski do których doszli autorzy w 2022 roku, wciąż są aktualne oraz uzupełnić ich badania o interakcję z API DOM, operacje SIMD oraz zbadanie różnic wynikających z użycia różnych typów zmiennych.

3.2 Porównanie wydajności frameworków frontendowych

Stefan Krause jest autorem internetowego narzędzia do porównywania wydajności frameworków i bibliotek frontendowych [43]. Wszystkie frameworki implementują tę samą aplikację, która ma za zadanie wchodzić w interakcję z API DOM, a następnie przeprowadzane są testy na różnych wersjach przeglądarek i systemów operacyjnych.

Rysunek 3.1 przedstawia porównanie wydajności najpopularniejszych frameworków frontendowych w językach JavaScript i Rust. Do frameworków w języku JavaScript możemy zaliczyć:

- Vanillajs — Implementacja bazowa w czystym JavaScript, bez użycia żadnego frameworku
- Solid
- Svelte
- Vue
- Vue Vapor
- Angular
- React

Wśród frameworków Rustowych, które zostały porównane na rysunku 3.1, znajdują się:

- wasm-bindgen — Bezpośrednie odniesienia do funkcji JavaScriptowych w kodzie Rust
- Leptos
- Sycamore
- Dioxus

- Yew

Frameworki w Rust obsługuje całą logikę aplikacji po stronie WebAssembly, zaś interakcję z DOM realizują za pomocą JavaScriptu. Analiza tabeli z wynikami pozwala stwierdzić, że wydajność wywołań funkcji JavaScriptowych w wasm-bindgen jest na porównywalnym poziomie do bezpośrednich operacji na API DOM przeprowadzanych z JavaScriptu. Wskazuje to na znikomy narzut wynikający z ciągłej zmiany kontekstu między WebAssembly, a JavaScriptem.

Duration in milliseconds ± 95% confidence interval (Slowdown = Duration / Fastest)												
Name Duration for...	vanillajs	wasm-bindgen-v0.2.84	solid-v1.8.15	svelte-v5.0.0-next.105	vue-vapor-v3.2024.0-761f785	leptos-v0.6.3	sycamore-v0.9.0-beta.2	vue-v3.4.23	dioxus-v0.5.0	angular-ngrx-v17.3.1	react-hooks-v18.2.0	yew-hooks-v0.21.0
Implementation notes	772	772	1139									1139
Implementation link	code	code	code	code	code	code	code	code	code	code	code	code
create rows creating 1,000 rows. (5 warmup runs).	38.4 ± 0.4 (1.03)	41.3 ± 0.5 (1.10)	39.8 ± 0.4 (1.06)	40.2 ± 0.4 (1.07)	42.0 ± 0.3 (1.12)	47.0 ± 0.4 (1.26)	46.4 ± 0.4 (1.24)	46.5 ± 0.9 (1.24)	45.6 ± 0.3 (1.22)	54.3 ± 0.6 (1.45)	49.7 ± 0.4 (1.33)	63.5 ± 0.6 (1.70)
replace all rows updating all 1,000 rows. (5 warmup runs).	42.0 ± 0.3 (1.02)	45.1 ± 0.2 (1.10)	44.7 ± 0.3 (1.09)	45.6 ± 0.3 (1.11)	47.8 ± 0.3 (1.16)	55.8 ± 0.2 (1.36)	55.5 ± 0.3 (1.35)	53.4 ± 0.3 (1.30)	51.0 ± 0.4 (1.24)	65.3 ± 0.3 (1.59)	59.6 ± 0.4 (1.45)	73.6 ± 0.6 (1.79)
partial update updating every 10th row for 1,000 rows. (3 warmup runs), 4 x CPU slowdown.	16.8 ± 0.3 (1.03)	18.2 ± 0.5 (1.12)	17.9 ± 0.4 (1.10)	17.7 ± 0.6 (1.09)	21.4 ± 0.5 (1.31)	19.4 ± 0.7 (1.19)	19.3 ± 0.5 (1.18)	21.9 ± 0.5 (1.34)	20.1 ± 0.5 (1.23)	19.3 ± 0.3 (1.18)	22.2 ± 0.3 (1.36)	31.4 ± 0.6 (1.93)
select row highlighting a selected row. (5 warmup runs), 4 x CPU slowdown.	3.1 ± 0.1 (1.15)	2.7 ± 0.1 (1.00)	3.2 ± 0.1 (1.19)	4.4 ± 0.2 (1.63)	6.8 ± 0.1 (2.52)	3.4 ± 0.1 (1.26)	7.1 ± 0.2 (2.63)	4.3 ± 0.2 (1.59)	6.4 ± 0.2 (2.37)	5.1 ± 0.1 (1.89)	5.7 ± 0.3 (2.11)	15.5 ± 0.2 (5.74)
swap rows swap 2 rows for table with 1,000 rows. (5 warmup runs), 4 x CPU slowdown.	19.1 ± 0.3 (1.00)	19.4 ± 0.4 (1.02)	21.7 ± 1.0 (1.14)	21.1 ± 0.7 (1.10)	23.2 ± 0.3 (1.21)	20.6 ± 0.3 (1.08)	21.2 ± 0.3 (1.11)	22.1 ± 0.4 (1.16)	22.4 ± 0.3 (1.17)	198.0 ± 1.8 (10.37)	172.9 ± 1.3 (9.05)	33.9 ± 0.4 (1.77)
remove row removing one row. (5 warmup runs), 2 x CPU slowdown.	16.4 ± 0.2 (1.11)	16.3 ± 0.2 (1.10)	16.9 ± 0.3 (1.14)	16.9 ± 0.3 (1.14)	17.8 ± 0.1 (1.20)	16.7 ± 0.2 (1.13)	16.8 ± 0.2 (1.14)	19.5 ± 0.1 (1.32)	17.3 ± 0.2 (1.17)	17.5 ± 0.2 (1.18)	19.5 ± 0.1 (1.32)	22.5 ± 0.2 (1.52)
create many rows creating 10,000 rows. (5 warmup runs).	386.8 ± 2.4 (1.05)	424.1 ± 3.5 (1.15)	411.8 ± 1.9 (1.11)	417.8 ± 1.8 (1.13)	437.1 ± 2.9 (1.18)	490.1 ± 2.8 (1.33)	468.5 ± 4.4 (1.27)	469.2 ± 4.4 (1.27)	451.4 ± 2.8 (1.22)	532.1 ± 5.7 (1.44)	646.4 ± 4.0 (1.75)	817.8 ± 9.2 (2.21)
append rows to large table appending 1,000 to a table of 1,000 rows. (5 warmup runs).	44.6 ± 0.5 (1.05)	47.4 ± 0.5 (1.12)	46.2 ± 0.4 (1.09)	46.2 ± 0.3 (1.09)	48.3 ± 0.3 (1.14)	56.4 ± 0.4 (1.33)	53.4 ± 0.5 (1.26)	54.8 ± 0.3 (1.30)	57.4 ± 0.2 (1.36)	58.4 ± 0.4 (1.38)	55.8 ± 0.3 (1.32)	79.2 ± 0.5 (1.87)
clear rows clearing a table with 1,000 rows. (5 warmup runs), 4 x CPU slowdown.	12.9 ± 0.2 (1.03)	13.1 ± 0.2 (1.05)	14.8 ± 0.2 (1.18)	16.0 ± 0.3 (1.28)	16.3 ± 0.3 (1.30)	17.7 ± 0.2 (1.42)	16.9 ± 0.4 (1.35)	19.1 ± 0.2 (1.53)	23.0 ± 0.3 (1.84)	33.4 ± 0.3 (2.67)	27.7 ± 0.6 (2.22)	29.9 ± 0.4 (2.39)
weighted geometric mean of all factors in the table compare: Green means significantly faster, red significantly slower	1.05	1.10	1.11	1.14	1.24	1.27	1.29	1.32	1.33	1.58	1.60	1.98
	compare	compare	compare	compare	compare	compare	compare	compare	compare	compare	compare	compare

Rysunek 3.1: Porównanie wydajności najpopularniejszych frameworków w językach JavaScript i Rust [43]

Inną istotną obserwacją jest to, że frameworki Rustowe, takie jak Leptos czy Sycamore, wykazują lepszą wydajność od trzech najpopularniejszych frameworków JavaScriptowych: Vue, Reacta i Angulara. Warto jednak zauważyć, że frameworki takie jak Solid, Svelte, czy nowo wydany Vue Vapor — które, podobnie jak Leptos i Sycamore, nie korzystają z technologii Virtual DOM — są nadal szybsze od frameworków Rustowych. Dioxus (skupiający się bardziej na aplikacjach desktopowych niż przeglądarkowych i technologicznie bardziej zbliżony do popularniejszych frameworków) ma wydajność porównywalną do Vue.

Tymczasem Yew (będący jednym z pierwszych pełnoprawnych frameworków Rustowych) wypada gorzej niż najwolniejszy uwzględniony framework JavaScriptowy — React.

Frameworki frontendowe są swego rodzaju czarnymi skrzynkami, które wykonują tę samą pracę wolniej lub szybciej. Ich wydajność jest zależna od implementacji znajdującej się w skrzynce a nie od różnic językowych. Ze wszystkich porównanych wyżej rozwiązań w języku Rust, wasm-bindgen jest najniższą warstwą abstrakcji, jaka jest wymagana do uruchomienia języka Rust w przeglądarce w przyjazny dla programisty sposób. Z tego powodu, w mojej pracy postaram się porównać wydajność języka Rust wykorzystującego właśnie wasm-bindgen do czystego JavaScriptu w celu zweryfikowania wyniki Stefana Krause.

4. Metodyka badań

Aby odpowiedzieć na pytanie, która z technologii jest szybsza, przeprowadziłem serię badań, które miały na celu porównanie wydajności WebAssembly i JavaScriptu w kontekście aplikacji przeglądarkowych. W tym rozdziale opisuję zaprojektowane przeze mnie środowisko testowe, dokładną procedurę testowania oraz metryki porównawcze, które zastosowałem w moich testach. Opisuję w nim również sposoby zbierania pomiarów czasu i problemy z nimi związane.

4.1 Środowisko testowe

4.1.1 Opis sprzętu

Testy zostały przeprowadzone przy użyciu przeglądarki Google Chrome w wersji 126.0.6478.62. Przeglądarka działała na komputerze MacBook Air z 2020 roku, wyposażonym w procesor Apple M1, 16 GB pamięci RAM i system operacyjny macOS w wersji Sonoma 14.5.

4.1.2 Opis oprogramowania

Porównanie wydajności JavaScriptu i Rusta skompilowanego do WebAssembly wymaga uruchomienia obu technologii w tym samym środowisku. Możliwe jest uruchomienie obu tych technologii po stronie serwera dzięki środowiskom takim jak Node.js, czy Wasmer [44]. Aczkolwiek z racji, że celem pracy jest porównanie wydajności w kontekście aplikacji przeglądarkowych, zdecydowałem się na przeprowadzenie badań w przeglądarce internetowej.

Narzędzie wasm-bindgen generuje kod-klej, który pozwala na wywoływanie funkcji napisanych w Rust z poziomu JavaScriptu. Dzięki temu można testować obie technologie w przeglądarce internetowej bez większych trudności. Pomiar czasu przed i po wywołaniu testowanej funkcji sprawdza się tylko dla funkcji JavaScriptowych. W przypadku funkcji napisanych w Rust taki pomiar nie jest wystarczający, ponieważ mierzy on nie tylko czas wykonania samego algorytmu, ale również czas wykonania kodu-kleju. Oznacza to, że serializacja i deserializacja danych również wliczają się w czas wykonania funkcji.

Z tego powodu zdecydowałem, że zamiast korzystać z gotowych bibliotek, zbuduję własne rozwiązanie do testowania i mierzenia czasu wywołań funkcji oraz serializacji i deserializacji danych. Dokładny opis dokonywanych pomiarów zawarłem w rozdziale 4.3.

Sposób pomiarów czasu

W przeglądarce internetowej istnieje wiele sposobów na to, aby dokonać pomiaru czasu. Jednym z najprostszych jest odowłanie się do funkcji `Date.now()`, która zwraca ilość milisekund od 1 stycznia 1970 roku [45]. Nie jest to jednak zalecany sposób, ponieważ dokładność do 1 milisekundy nie jest zbyt bardzo dokładna. Gdy funkcje wykonują się w mniej niż 1 milisekundę, może dojść do sytuacji, że średnia wszystkich pomiarów będzie równa zero co uniemożliwiłoby porównanie technologii.

Znacznie lepszym sposobem jest skorzystanie z Performance API, które pozwala na pomiary czasu z dokładnością do 5 μ s [46]. Niestety, przeglądarki wdrożyły zabezpieczenia przeciwko atakom timingowym i fingerprintowaniu w postaci drżenia wartości (ang. *jitter*) [47], co sprawia, że pomiary czasu mogą być obarczone dodatkowym błędem. Niemniej jednak, jest to najdokładniejszy sposób na pomiar czasu w przeglądarce internetowej.

Po wywołaniu funkcji `performance.now()` w przeglądarkach Google Chrome i Firefox okazało się, że Google Chrome zwracała dokładniejsze wyniki niż Firefox. Zwracała ona pomiary z dokładnością do 5 μ s, tak jak mówi specyfikacja. Firefox zwracał pomiary z dokładnością do 20 μ s, co spowodowało, że postanowiłem przeprowadzić wszystkie testy na przeglądarce Google Chrome.

4.2 Procedura testowania

- Każdy test składał się z zera lub więcej funkcji JavaScriptowych oraz jednej lub więcej funkcji napisanych w języku Rust.
- Każdy test został podzielony na 3 podtesty, które różniły się danymi wejściowymi (chyba że, w rozdziale 4.4 opisano inaczej).
- Dane wejściowe były losowo generowane przed rozpoczęciem pomiaru czasu wykonania funkcji (chyba że, w rozdziale 4.4 opisano inaczej).
- Każdy podtest był wykonany dla trzech różnych ilości powtórzeń oznaczonych symbolem n .

- Badane w każdym teście funkcje były wykonywane sekwencyjnie, a przed pierwszym wywołaniem każdej funkcji (dla każdego podtestu oraz każdego n), karta przeglądarki była odświeżana.
- Wszystkie funkcje w Języku Rust zostały przetestowane na trzech profilach optymalizacyjnych kompilatora, które zostały opisane w tabeli 4.1.

Oznaczenie	Konfiguracja profilu	Opis konfiguracji
RS_3	<code>lto = true opt_level = 3</code>	LTO + optymalizacje skupiające się na szybkości skompilowanego kodu
RS_s	<code>lto = true opt_level = 's'</code>	LTO + optymalizacje skupiające się na rozmiarze skompilowanego kodu
RS_z	<code>lto = true opt_level = 'z'</code>	LTO + optymalizacje skupiające się na rozmiarze skompilowanego kodu (bez wektoryzacji pętli)

Tabela 4.1: Opis poziomów optymalizacji kompilatora Rusta

4.3 Pomiary i metryki porównawcze

4.3.1 Pomiary dla języka Rust

Dane przekazywane do funkcji skompilowanej w WebAssembly mogą wymagać serializacji. Jest to proces polegający na zamianie formatu danych na taki, który można przenieść między kontekstem JavaScriptowym, a WebAssembly. Dla każdej funkcji dokonałem więc czterech pomiarów:

- T_{RS_1} — przed wywołaniem testowanej funkcji; z poziomu JavaScriptu
- T_{RS_2} — na początku wywoływanej funkcji; z poziomu Rusta
- T_{RS_3} — tuż przed zwróceniem wyniku z funkcji; z poziomu Rusta
- T_{RS_4} — po zakończeniu wywołania testowanej funkcji; z poziomu JavaScriptu

Pozwoliło mi to na zbadanie czasu deserializacji danych, czasu wykonania algorytmu oraz czasu serializacji danych oznaczonych jako T_D , T_A oraz T_S odpowiednio:

$$T_{D_RS} = T_{RS_2} - T_{RS_1}$$

$$T_{A_{RS}} = T_{RS_3} - T_{RS_2}$$

$$T_{S_{RS}} = T_{RS_4} - T_{RS_3}$$

Czasy serializacji i deserializacji są nazwane z perspektywy kontekstu WebAssembly.

Czas $T_{D_{RS}}$ jest on liczyony od momentu tuż przed wywołaniem funkcji kodu kleju do momentu wejścia do funkcji w języku Rust lub do zakończenia ręcznej deserializacji argumentów w tej funkcji. Wlicza on więc serializację danych po stronie JavaScriptu i deserializację danych po stronie modułu WASM.

Czas $T_{S_{RS}}$ jest liczyony od momentu tuż przed zwróceniem wyniku z funkcji w języku Rust do momentu zakończenia wywołania funkcji kodu-kleju. Wlicza on więc serializacje danych po stronie modułu WASM i deserializację w kodzie-kleju.

4.3.2 Pomiary dla języka JavaScript

Aby wywołać funkcję w języku JavaScript, wystarczy podać dane jako argument funkcji. Nie zachodzi więc potrzeba serializacji i deserializacji danych. Oznacza to, że w przypadku tej technologii wystarczy przeprowadzić dwa pomiary:

- T_{JS_1} — przed wywołaniem testowanej funkcji
- T_{JS_2} — po zakończeniu wywołania testowanej funkcji

Oba pomiary pozwalają na zbadanie czasu wykonania algorytmu, który jest równy różnicy tych dwóch pomiarów:

$$T_{A_{JS}} = T_{JS_2} - T_{JS_1}$$

4.3.3 Metryki porównawcze

Pierwszą metryką jest średni czas wykonania algorytmu oznaczony jako $\bar{T}_{A_{RS}}$ i $\bar{T}_{A_{JS}}$ odpowiednio dla funkcji Rustowych i JavaScriptowych. Pozwala on porównać z jaką szybkością wykonuje się sam algorytm zaimplementowany w obu technologiach.

Drugą metryką jest średni całkowity czas wykonania funkcji zdefiniowany jako suma wszystkich składowych czasów dla każdej z technologii. W przypadku funkcji napisanych w języku Rust jest to suma czasu deserializacji, algorytmu i serializacji, natomiast dla JavaScriptu jest to po prostu czas wykonania algorytmu.

$$T_{C_{RS}} = T_{D_{RS}} + T_{A_{RS}} + T_{S_{RS}}$$

$$T_{CJS} = T_{AJS}$$

Trzecią metryką jest całkowite przyśpieszenie. Jest to metryka, która pokazuje ile razy szybciej wykonuje się funkcja w języku Rust w odniesieniu do funkcji w języku JavaScript uwzględniając kod-klej oraz przenoszenie danych pomiędzy kontekstami. Inaczej mówiąc, pozwala ona porównać obie technologie w sytuacji, gdy wywołujemy funkcje WebAssembly bezpośrednio z JavaScriptu i oczekujemy na zwrot danych. Jest ona zdefiniowana jako średni całkowity czas wykonania w języku Rust podzielony przez średni całkowity czas wykonania w języku JavaScript:

$$S_C = \frac{\bar{T}_{CJS}}{\bar{T}_{CRS}}$$

Czwartą metryką jest przyśpieszenie algorytmu. Jest to metryka, która pokazuje ile razy szybciej wykonuje się algorytm zaimplementowany w języku Rust w odniesieniu do algorytmu w języku JavaScript. Inaczej mówiąc, pozwala ona porównać obie technologie w sytuacji, gdy aplikacja przeglądarkowa jest w pełni zaimplementowana w Rust skompilowanym do WebAssembly i nie musimy przekazywać danych między kontekstami. Jest ona zdefiniowana jako średni czas wykonania algorytmu w języku Rust podzielony przez średni czas wykonania algorytmu w języku JavaScript:

$$S_A = \frac{\bar{T}_{AJS}}{\bar{T}_{ARS}}$$

Zbadanie metryk S_C i S_A dla wszystkich par testowanych funkcji byłoby zbyt czasochłonne, a także zbędne. Postanowiłem więc zbadać je jedynie dla najistotniejszych w mojej opinii par funkcji JavaScriptowych i Rustowych.

4.4 Przeprowadzone testy

Sumarycznie przeprowadziłem 9 testów składających się z różnej liczby funkcji, z czego każda była wykonana dla 3 różnych danych wejściowych i 3 różnych wartości n , co sumarycznie dało 291 indywidualnych wyników.

4.4.1 Re-enkodowanie stringów

Ten test ma na celu zbadanie ile czasu zajmuje przenoszenie stringów z języka JavaScript do kontekstu WebAssembly i z powrotem. Proces ten wymaga re-enkodowania danych

między UTF-16 a UTF-8, co w generuje dodatkowy narzut przy wykonywaniu funkcji wyeksportowanych z modułów WASM.

Wszystkie funkcje opisane w tabeli 4.2 były wywołane dla trzech różnych rozmiarów danych: 1KB, 512KB i 1MB. Każda funkcja została wywołana n razy, dla $n \in \{100, 1000, 10000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji.

Język	F_i	Funkcja	Opis funkcji
RS_3	F_1	reencode_strings	Funkcja zwracająca dane wejściowe. Przyjmuje i zwraca typ String .
RS_s	F_2	reencode_strings	
RS_z	F_3	reencode_strings	

Tabela 4.2: Opis funkcji dla testu „Re-enkodowanie stringów”

4.4.2 Enkodowanie stringów do Base64

Ten test ma na celu zbadanie ile czasu zajmie enkodowanie ciągu znaków do Base64 dla różnych wielkości danych.

Wszystkie funkcje opisane w tabeli 4.3 były wywołane dla trzech różnych rozmiarów danych: 1KB, 512KB i 1MB. Każda funkcja została wywołana n razy, dla $n \in \{100, 1000, 10000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji.

Funkcja „base64” w języku JavaScript i „base64” w języku Rust zostały zaimplementowane według identycznego algorytmu. Funkcje F_3 , F_5 oraz F_7 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Funkcja „base64_simd” wykorzystuje zewnętrzną bibliotekę „base64-simd”, która implementuje algorytm obliczający wynik enkodowania ciągu znaków do Base64 wykorzystując do tego operacje SIMD. Funkcje F_4 , F_6 oraz F_8 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

4.4.3 k-ta liczba Fibonacciego (rekurencyjnie)

Ten test ma na celu zbadanie wydajności rekurencji w obu technologiach.

Wszystkie funkcje opisane w tabeli 4.4 były wywołane dla trzech różnych argumentów k gdzie $k \in \{20, 30, 40\}$. Każda funkcja została wywołana n razy, dla $n \in \{100, 500, 1000\}$ w

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	<i>F₁</i>	base64	Własna implementacja algorytmu enkodera
	<i>F₂</i>	btoa	Natywna funkcja dostępna w środowisku przeglądarkowym
<i>RS₃</i>	<i>F₃</i>	base64	Własna implementacja algorytmu enkodera
	<i>F₄</i>	base64_simd	Algorytm wspierający operacje SIMD z pakietu „base64_simd”
<i>RS_s</i>	<i>F₅</i>	base64	Własna implementacja algorytmu enkodera
	<i>F₆</i>	base64_simd	Algorytm wspierający operacje SIMD z pakietu „base64_simd”
<i>RS_z</i>	<i>F₇</i>	base64	Własna implementacja algorytmu enkodera
	<i>F₈</i>	base64_simd	Algorytm wspierający operacje SIMD z pakietu „base64_simd”

Tabela 4.3: Opis funkcji dla testu „Enkodowanie stringów do Base64”

celu zbadania wpływu kompilatora JIT na czas wykonania funkcji. W przypadku tego testu dane nie były generowane losowo.

Funkcja „fib” w języku JavaScript i „fib” w języku Rust zostały zaimplementowane według identycznego algorytmu. Funkcje *F₂*, *F₃* oraz *F₄* różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	<i>F₁</i>	fib	Własna implementacja algorytmu rekurencyjnego
<i>RS₃</i>	<i>F₂</i>	fib	
<i>RS_s</i>	<i>F₃</i>	fib	Własna implementacja algorytmu rekurencyjnego
<i>RS_z</i>	<i>F₄</i>	fib	

Tabela 4.4: Opis funkcji dla testu „k-ta liczba Fibonacciego (rekurencyjnie)”

4.4.4 Mnożenie macierzy 4x4

Ten test ma na celu zbadanie wydajności iteracji w obu technologiach.

Jako jedyny test nie był on rozbity na trzy podtesty i był przeprowadzony jedynie na macierzach cztery na cztery. Każda funkcja opisana w tabeli 4.5 została wywołana n razy, dla $n \in \{100, 1000, 10000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji.

Funkcja „multiplyMatrices” w języku JavaScript i „matrix_mult” w języku Rust zostały zaimplementowane według identycznego algorytmu. Funkcje F_2 , F_4 oraz F_6 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Funkcja „matrix_mult_simd” wykorzystuje zewnętrzną bibliotekę „glam”, która implementuje algorytm obliczający wynik mnożenia macierzy 4x4 wykorzystując do tego operacje SIMD. Reprezentuje on macierze 4x4 jako tablicę 16 wartości. Wobec tego czas deserializacji dla funkcji „matrix_mult_simd” uwzględnia również przemapowanie tablic dwuwymiarowych na tablice 16 wartości. Funkcje F_3 , F_5 oraz F_7 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	F_1	multiplyMatrices	Własna implementacja algorytmu
<i>RS₃</i>	F_2	matrix_mult	Własna implementacja algorytmu
	F_3	matrix_mult_simd	Implementacja wspierająca operacje SIMD z pakietu „glam”
<i>RS_s</i>	F_4	matrix_mult	Własna implementacja algorytmu
	F_5	matrix_mult_simd	Implementacja wspierająca operacje SIMD z pakietu „glam”
<i>RS_z</i>	F_6	matrix_mult	Własna implementacja algorytmu
	F_7	matrix_mult_simd	Implementacja wspierająca operacje SIMD z pakietu „glam”

Tabela 4.5: Opis funkcji dla testu „Mnożenie macierzy 4x4”

4.4.5 CRC-32

Ten test ma na celu zbadanie ile czasu zajmie wykonanie algorytmu CRC-32 dla różnych wielkości danych binarnych. W odróżnieniu od testów przeprowadzonych na ciągach znaków, podczas przesyłania tablicy bajtów nie zachodzi serializacja danych i są one przekazywane bezpośrednio do pamięci modułu WebAssembly.

Wszystkie funkcje opisane w tabeli 4.6 były wywołane dla trzech różnych rozmiarów danych: 1KB, 512KB i 1MB. Każda funkcja została wywołana n razy, dla $n \in \{100, 1000, 10000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji.

Funkcja „crc32” w języku JavaScript i „crc32” w języku Rust zostały zaimplementowane według algorytmu CRC-32 dla odwróconego wielomianu 0xEDB88320. Algorytm nie stosował optymalizacji w postaci wcześniej wygenerowanych tablic LUT. Funkcje F_2 , F_3 oraz F_4 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji
JS	F_1	crc32	Własna implementacja algorytmu
RS_3	F_2	crc32	
RS_s	F_3	crc32	Własna implementacja algorytmu
RS_z	F_4	crc32	

Tabela 4.6: Opis funkcji dla testu „CRC-32”

4.4.6 CRC-64

Ten test ma na celu zbadanie wydajności typu `"bigint"` w porównaniu z natywnymi 64-bitowymi liczbami całkowitymi w WebAssembly.

Wszystkie funkcje opisane w tabeli 4.7 były wywołane dla trzech różnych rozmiarów danych: 1KB, 512KB i 1MB. Każda funkcja została wywołana n razy, dla $n \in \{100, 1000, 10000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji.

Funkcja „crc64” w języku JavaScript i „crc64” w języku Rust zostały zaimplementowane według algorytmu CRC-64-ECMA dla odwróconego wielomianu 0xC96C5795D7870F42. Algorytm nie stosował optymalizacji w postaci wcześniej wygenerowanych tablic LUT. Funkcje F_2 , F_4 oraz F_6 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Funkcja „crc64_simd” wykorzystuje zewnętrzną bibliotekę „crc64fast”, która implementuje algorytm obliczający wynik algorytmu CRC-64-ECMA wykorzystując do tego operacje SIMD. Funkcje F_3 , F_5 oraz F_7 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	<i>F₁</i>	crc64	Własna implementacja algorytmu
<i>RS₃</i>	<i>F₂</i>	crc64	Własna implementacja algorytmu
	<i>F₃</i>	crc64_simd	Implementacja wspierająca operacje SIMD z pakietu „crc64fast”
<i>RS_s</i>	<i>F₄</i>	crc64	Własna implementacja algorytmu
	<i>F₅</i>	crc64_simd	Implementacja wspierająca operacje SIMD z pakietu „crc64fast”
<i>RS_z</i>	<i>F₆</i>	crc64	Własna implementacja algorytmu
	<i>F₇</i>	crc64_simd	Implementacja wspierająca operacje SIMD z pakietu „crc64fast”

Tabela 4.7: Opis funkcji dla testu „CRC-64”

4.4.7 API DOM — Tworzenie elementów

Ten test ma na celu zbadanie wydajności dynamicznego tworzenia elementów i dodawania ich do dokumentu.

Wszystkie funkcje opisane w tabeli 4.8 były wywołane dla trzech różnych ilości elementów k , gdzie $k \in \{100, 1000, 10000\}$. Każda funkcja została wywołana n razy, dla $n \in \{10, 100, 500\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji. W przypadku tego testu dane nie były generowane losowo.

Funkcja „createElements” w języku JavaScript i „create_elements” w języku Rust zostały zaimplementowane według identycznego algorytmu. Algorytm ten tworzy i dodaje do kontenera nowy, pusty element „div” przy każdej z k iteracji. Funkcje F_2 , F_3 oraz F_4 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

4.4.8 API DOM — Aktualizacja co drugiego elementu

Ten test ma na celu zbadanie wydajności modyfikowania treści dokumentu.

Wszystkie funkcje opisane w tabeli 4.9 były wywołane dla trzech różnych ilości elementów k , gdzie $k \in \{100, 1000, 10000\}$. Każda funkcja została wywołana n razy, dla $n \in \{20, 200, 1000\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji. W przypadku tego testu dane nie były generowane losowo.

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	<i>F₁</i>	createElements	Funkcja tworząca i dodająca k elementów do kontenera
<i>RS₃</i>	<i>F₂</i>	create_elements	
<i>RS_s</i>	<i>F₃</i>	create_elements	Funkcja tworząca i dodająca k elementów do kontenera
<i>RS_z</i>	<i>F₄</i>	create_elements	

Tabela 4.8: Opis funkcji dla testu „API DOM — Tworzenie elementów”

Funkcja „updateEvery2ndElement” w języku JavaScript i „update_every_2nd_element” w języku Rust zostały zaimplementowane według identycznego algorytmu. Funkcje F_2 , F_3 oraz F_4 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji
<i>JS</i>	<i>F₁</i>	updateEvery2ndElement	Funkcja ustawiająca pole „innerText” co drugiego elementu na jego indeks
<i>RS₃</i>	<i>F₂</i>	update_every_2nd_element	
<i>RS_s</i>	<i>F₃</i>	update_every_2nd_element	Funkcja ustawiająca pole „innerText” co drugiego elementu na jego indeks
<i>RS_z</i>	<i>F₄</i>	update_every_2nd_element	

Tabela 4.9: Opis funkcji dla testu „API DOM — Aktualizacja co drugiego elementu”

4.4.9 API DOM — Usuwanie elementów

Ten test ma na celu zbadanie wydajności usuwania poddrzewa elementów z dokumentu.

Wszystkie funkcje opisane w tabeli 4.10 były wywołane dla trzech różnych ilości elementów k , gdzie $k \in \{100, 1000, 10000\}$. Każda funkcja została wywołana n razy, dla $n \in \{10, 100, 500\}$ w celu zbadania wpływu kompilatora JIT na czas wykonania funkcji. W przypadku tego testu dane nie były generowane losowo.

Funkcja „clearElements” w języku JavaScript i „clear_elements” w języku Rust zostały zaimplementowane według identycznego algorytmu. Funkcje F_2 , F_3 oraz F_4 różnią się od siebie jedynie nałożonymi przez kompilator optymalizacjami.

Język	F_i	Funkcja	Opis funkcji	
<i>JS</i>	<i>F₁</i>	clearElements	Funkcja ustawiająca pole „innerHTML” kontenera na pusty string	
<i>RS₃</i>	<i>F₂</i>	clear_elements	Funkcja ustawiająca pole „innerHTML” kontenera na pusty string	
<i>RS_s</i>	<i>F₃</i>	clear_elements		
<i>RS_z</i>	<i>F₄</i>	clear_elements		

Tabela 4.10: Opis funkcji dla testu „API DOM — Usuwanie elementów”

Po przedstawieniu metody badawczej i opisaniu wszystkich przeprowadzonych przeze mnie testów, w kolejnym rozdziale prezentuję wyniki moich badań zarówno w formie tabel, jak i wykresów.

5. Wyniki badań

Wyniki badań postanowiłem zaprezentować w postaci tabel oraz wykresów. Tabele przedstawiają pomiary czasów T_D , T_A i T_S dla każdej funkcji, rozmiaru danych i ilości powtórzeń. Wykresy przedstawiają pomiary czasów pojedynczej funkcji jako poziomy słupki złożony z czasów T_D , T_A i T_S . Wszystkie słupki reprezentujące średni czas wywołania funkcji są zgrupowane według ilości powtórzeń.

5.1 Re-enkodowanie stringów

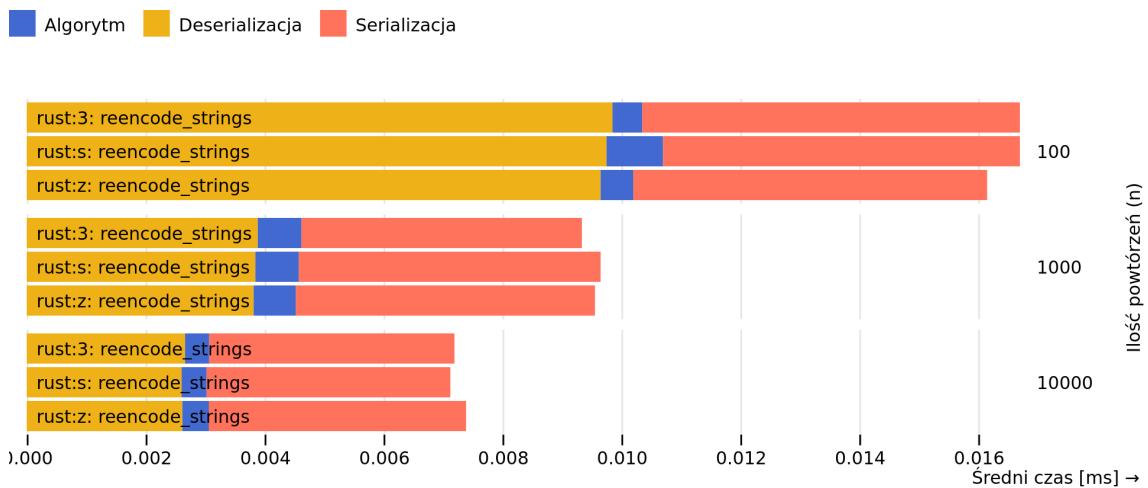
Tabela 5.1 przedstawia wyniki przeprowadzonego testu.

W przypadku 1KB danych, czasy \bar{T}_D i \bar{T}_S przyjmują wartości mniejsze niż $10\mu s$ co może wydawać się subtelną różnicą. Jednakże przeskok z $9.8\mu s$ do $3.9\mu s$, a następnie do $2.7\mu s$ dla każdego n w przypadku \bar{T}_D dla funkcji F_1 jest dosyć zauważalny na rysunku 5.1. Identyczny trend można zauważyć dla pozostałych funkcji zarówno dla \bar{T}_D i \bar{T}_S dla 1KB danych. Oznacza to, że re-enkodowanie jest zrobione po stronie JavaScriptu i jest optymalizowane przez kompilator JIT.

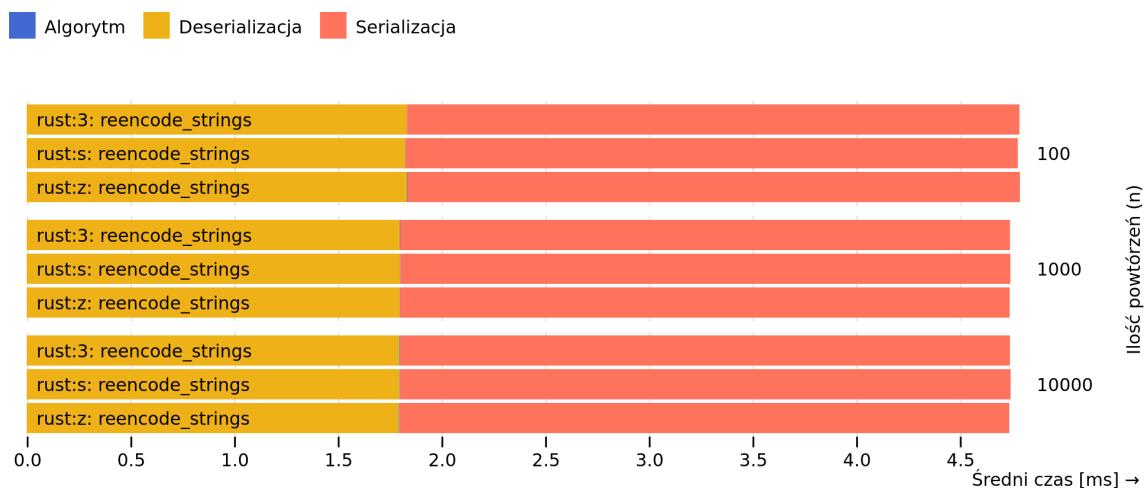
Co ciekawe, dla 512KB danych czas \bar{T}_D również maleje wraz ze wzrostem n dla każdej z funkcji, natomiast czas \bar{T}_S maleje jedynie przy przejściu z $n = 100$ do $n = 10^3$. Przy przejściu z $n = 10^3$ do $n = 10^4$ zaczyna on wzrastać.

Dla 1MB danych \bar{T}_D rośnie wraz z ilością powtórzeń każdej z funkcji, natomiast \bar{T}_S spada dla F_2 i F_3 .

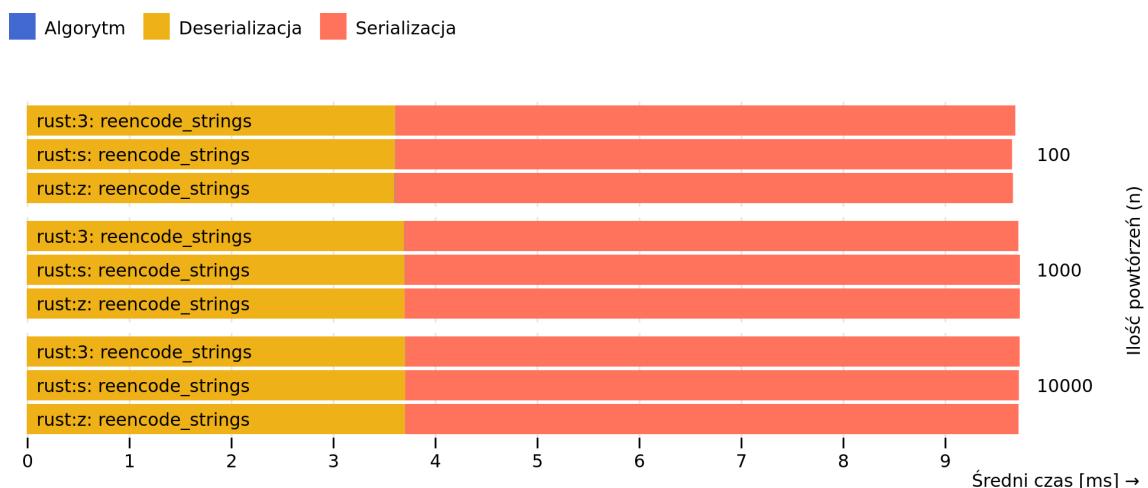
Różnice spowodowaną optymalizacjami JIT dla większej ilości danych trudno jest zobaczyć na wykresach 5.2 i 5.3 ze względu na ich rząd wielkości względem czasu wykonania samej serializacji i deserializacji. Oznacza to, że przesyłanie dużych stringów jest czasochłonne, a komplikacja JIT procesu re-enkodowania nie robi większej różnicy w czasie wykonania funkcji.



Rysunek 5.1: Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 1KB)”



Rysunek 5.2: Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 512KB)”



Rysunek 5.3: Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 1MB)”

		Średni czas wykonania [ms]								
		n = 100			n = 10 ³			n = 10 ⁴		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
1KB	F_1	0.0097	0.0010	0.0060	0.0038	0.0007	0.0051	0.0026	0.0004	0.0041
	F_2	0.0098	0.0005	0.0063	0.0039	0.0007	0.0047	0.0027	0.0004	0.0041
	F_3	0.0097	0.0005	0.0059	0.0038	0.0007	0.0050	0.0026	0.0004	0.0043
512KB	F_1	1.8260	0.0013	2.9525	1.8003	0.0025	2.9409	1.7978	0.0030	2.9439
	F_2	1.8326	0.0013	2.9534	1.7987	0.0027	2.9408	1.7970	0.0037	2.9410
	F_3	1.8334	0.0027	2.9533	1.7997	0.0017	2.9390	1.7949	0.0030	2.9404
1MB	F_1	3.6102	0.0010	6.0504	3.7065	0.0007	6.0309	3.7098	0.0006	6.0182
	F_2	3.6130	0.0007	6.0800	3.7007	0.0009	6.0220	3.7119	0.0008	6.0233
	F_3	3.6068	0.0030	6.0606	3.7043	0.0008	6.0326	3.7092	0.0008	6.0150

Tabela 5.1: Wyniki dla testu „Re-enkodowanie stringów”

5.2 Enkodowanie stringów do Base64

Patrząc na wykresy 5.4, 5.5 i 5.6 można zaobserwować, że najszybszą funkcją jest zdecydowanie F_2 („btoa”) będąca natywną funkcją dostępną w środowisku przeglądarkowym. Najszybszą funkcją w języku Rust jest F_4 , która jest implementacją wykorzystującą operacje SIMD.

Dla 1KB danych S_C dla najszybszych funkcji obu technologii wynosi odpowiednio:

$$\text{dla } n = 100, S_C = \frac{0.0036}{0.0152} = 0.2368$$

$$\text{dla } n = 10^3, S_C = \frac{0.0033}{0.0091} = 0.3626$$

$$\text{dla } n = 10^4, S_C = \frac{0.0031}{0.0074} = 0.4189$$

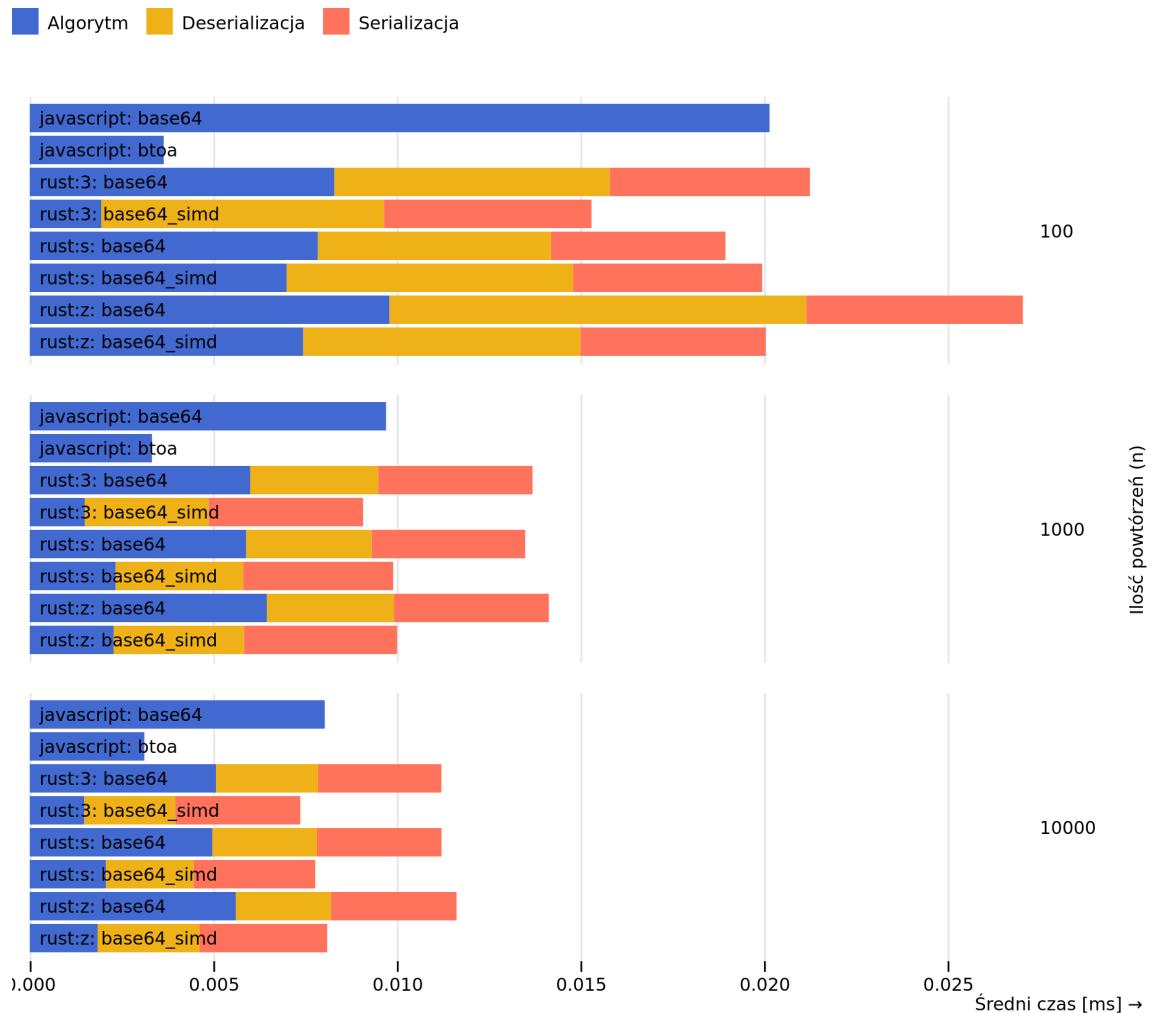
Oznacza to, że najszybsza funkcja w języku Rust, jest od 4 do 2.4 raza wolniejsza od JavaScriptu w zależności od poziomu optymalizacji JIT. Jeżeli porównamy natomiast samo przyśpieszenie algorytmu S_A dla tych samych danych, okaże się, że sam czas wykonania algorytmu jest dla funkcji Rustowych średnio 2 razy szybszy niż dla algorytmów zaimplementowanych w języku JavaScript:

$$\text{dla } n = 100, S_A = \frac{0.0036}{0.0019} = 1.8947$$

$$\text{dla } n = 10^3, S_A = \frac{0.0033}{0.0015} = 2.2000$$

$$\text{dla } n = 10^4, S_A = \frac{0.0031}{0.0015} = 2.0667$$

Trend ten można zaobserwować dla wszystkich testowanych rozmiarów danych. Oznacza to, że WebAssembly traci wydajność na serializacji i deserializacji danych, natomiast sam algorytm jest w stanie przebić wydajność natywnego algorytmu zaimplementowanego przez przeglądarkę.



Rysunek 5.4: Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 1KB)”

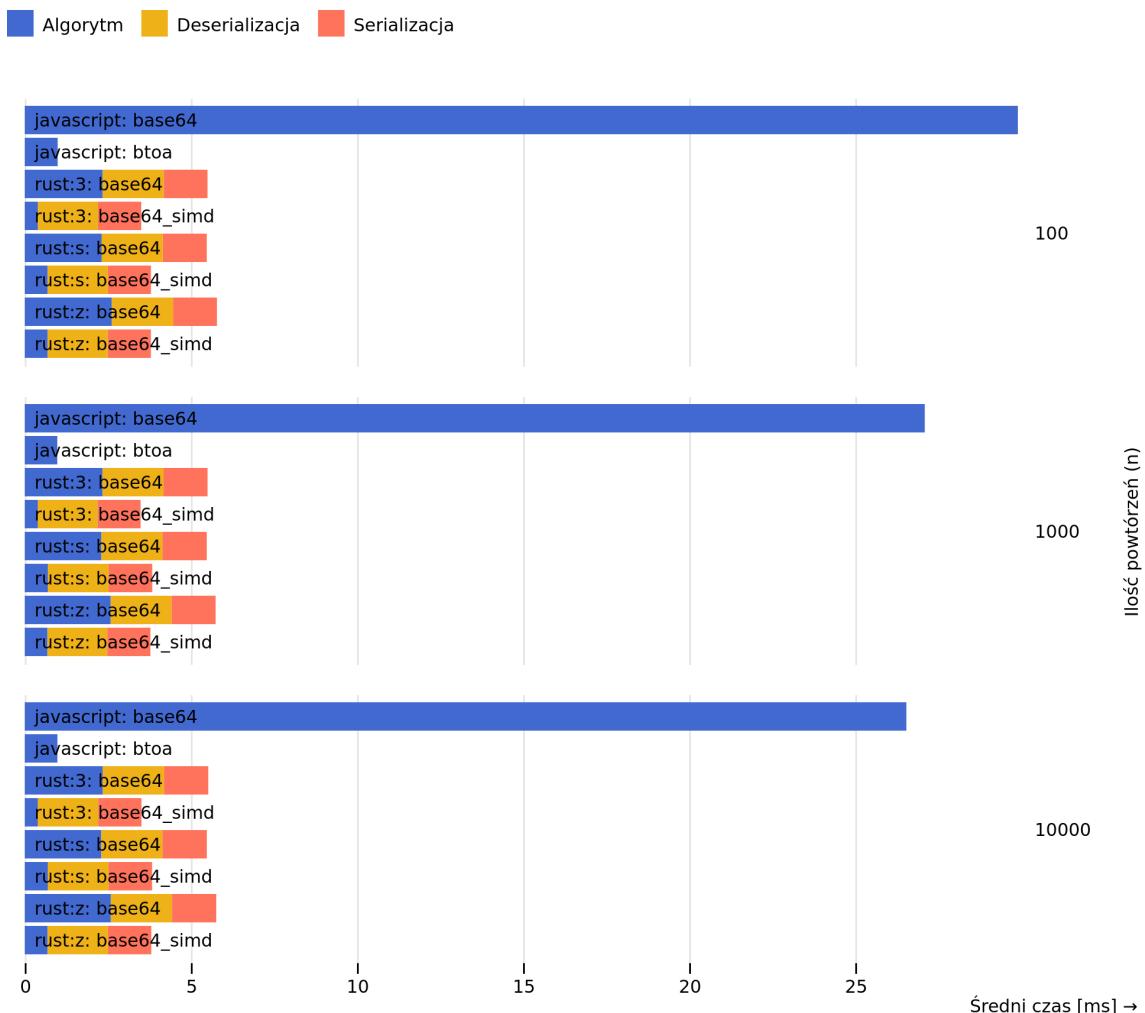
Kolejnym wartym omówienia aspektem są funkcje F_1 oraz F_3 , F_5 i F_7 . Pierwsza jest zaimplementowana według identycznego algorytmu co pozostałe. Funkcje F_3 , F_5 i F_7 są dokładnie tą samą implementacją skompilowaną na różnych poziomach optymalizacji. Najszybszą z nich jest F_3 , która jest skompilowana z optymalizacjami celującymi w szybkość wykonania kodu.

Jak widzimy na wykresie 5.4, F_1 prześciga F_3 dla każdej ilości powtórzeń, lecz różnica nie jest aż tak diametralna jak w przypadku pary najszybszych funkcji. Dla największego badanego poziomu optymalizacji JIT, S_C i S_A wynoszą odpowiednio:

$$\text{dla } n = 10^4, S_C = \frac{0.0080}{0.0113} = 0.7080$$

$$\text{dla } n = 10^4, S_A = \frac{0.0080}{0.0051} = 1.5686$$

Oznacza to, że wydajność algorytmu enkodującego w WebAssembly jest 1.6 razy szybsza od JavaScriptu dla testowanych danych, a czas serializacji i deserializacji (który sumarycznie wynosi 6.4μs) można uznać za znikomy.



Rysunek 5.5: Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 512KB)”

Wyniki tych funkcji wyglądają ciekawiej dla danych o rozmiarze 512KB i 1MB. Na wykresach 5.5 i 5.6 możemy zauważyc, że funkcja F_1 zajmuje niepodważalnie więcej czasu

niż funkcje F_3 , F_5 i F_7 . Już dla 512KB danych i najwyższego badanego poziomu optymalizacji JIT przyśpieszenia S_C i S_A między F_1 a F_3 wynoszą odpowiednio:

$$\begin{aligned} \text{dla } n = 10^4, S_C &= \frac{26.5381}{5.5284} = 4.8003 \\ \text{dla } n = 10^4, S_A &= \frac{26.5381}{2.3467} = 11.3087 \end{aligned}$$

Oznacza to, że ten sam algorytm wywołany w WebAssembly dla dużej ilości danych wykonuje się ponad 11 razy szybciej od kodu JavaScriptowego optymalizowanego przez kompilator JIT w tym samym środowisku.

Warto również zbadać przyśpieszenie funkcji F_4 dla większej ilości danych niż 1KB. Dla 512KB danych i najwyższego stopnia optymalizacji JIT przyśpieszenia S_C i S_A względem własnej implementacji F_1 wynoszą odpowiednio:

$$\begin{aligned} \text{dla } n = 10^4, S_C &= \frac{26.5381}{3.5133} = 7.5536 \\ \text{dla } n = 10^4, S_A &= \frac{26.5381}{0.3927} = 67.5786 \end{aligned}$$

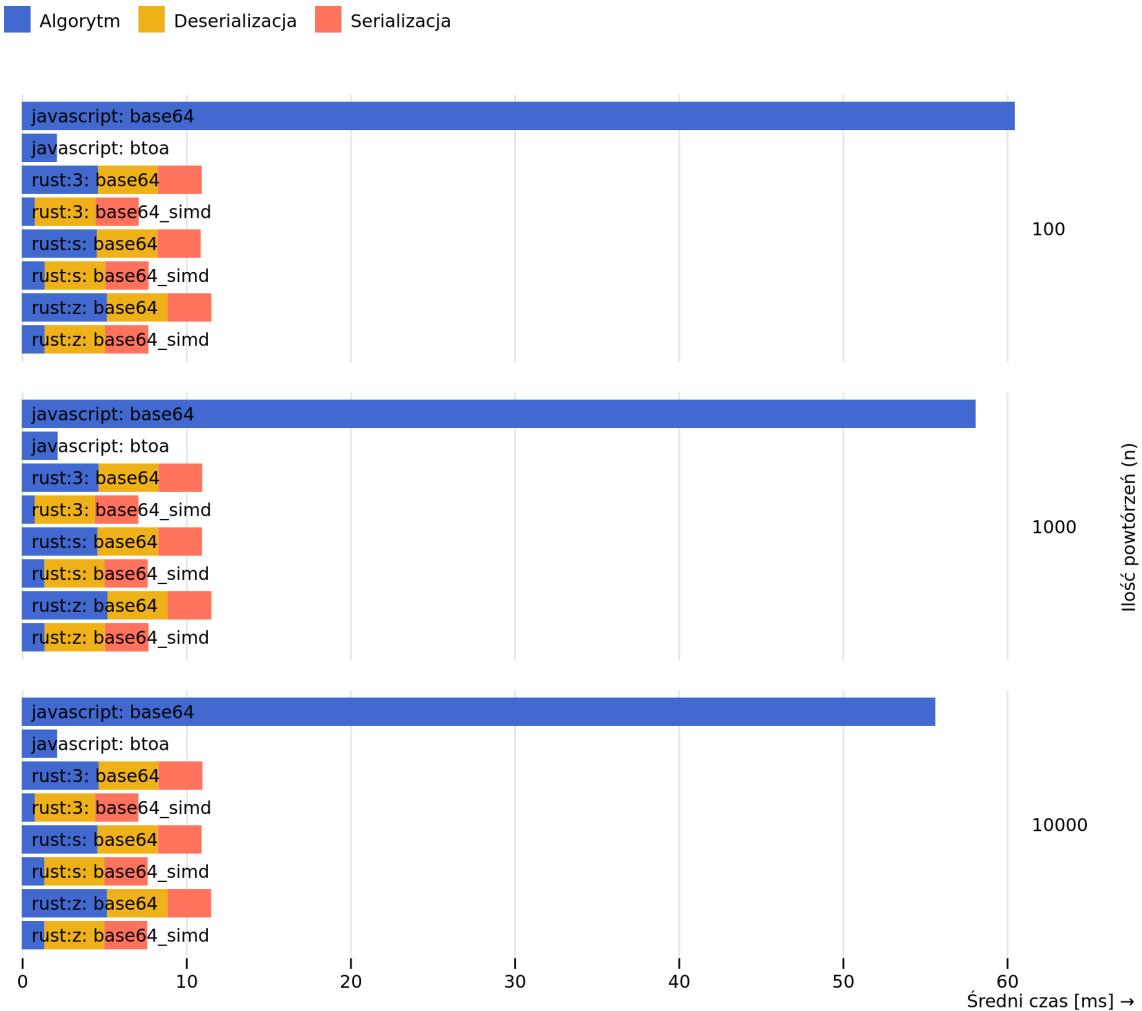
Oznacza to, że algorytm wykorzystujący operacje SIMD jest o 67.6 raza szybszy niż algorytm skalarny zaimplementowany w języku JavaScript. Jest to zaskakująco dobry wynik, który wskazuje na to, że w sytuacji w której jesteśmy skazani na własną implementację pewnego algorytmu, warto rozważyć wektoryzację przy użyciu operacji SIMD.

5.3 k-ta liczba Fibonacciego (rekurencyjnie)

Na każdym z wykresów 5.7, 5.8 i 5.9 dla każdej ilości powtórzeń widać dokładnie tą samą zależność: funkcja F_1 będąca implementacją w języku JavaScript jest kilka razy wolniejsza od każdego z 3 zoptymalizowanych wariantów algorytmu w języku Rust.

Funkcje F_2 i F_3 były zbliżone wydajnością. Możemy więc z tego wywnioskować, że kompilator Rust nie nakłada podobny zestaw optymalizacji na proste rekurencyjne wywołania zarówno w przypadku optymalizacji szybkościowej jak i rozmiarowej. Funkcja F_4 odstaje nieznacznie wydajnością od innych wariantów w przypadku każdego z testów. Jest to spowodowane tym, że stopień optymalizacji „z” jest jednakowy ze stopniem „s” za wyjątkiem wektoryzacji pętli. W przypadku stopnia „z” wektoryzacja pętli nie występuje. Stąd też funkcja F_4 jest wolniejsza od pozostałych dwóch.

Przyśpieszenia S_C i S_A dla funkcji F_1 i F_2 przy największym badanym stopniu optymalizacji JIT i 40 liczbie Fibonacciego wynoszą odpowiednio:



Rysunek 5.6: Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 1MB)”

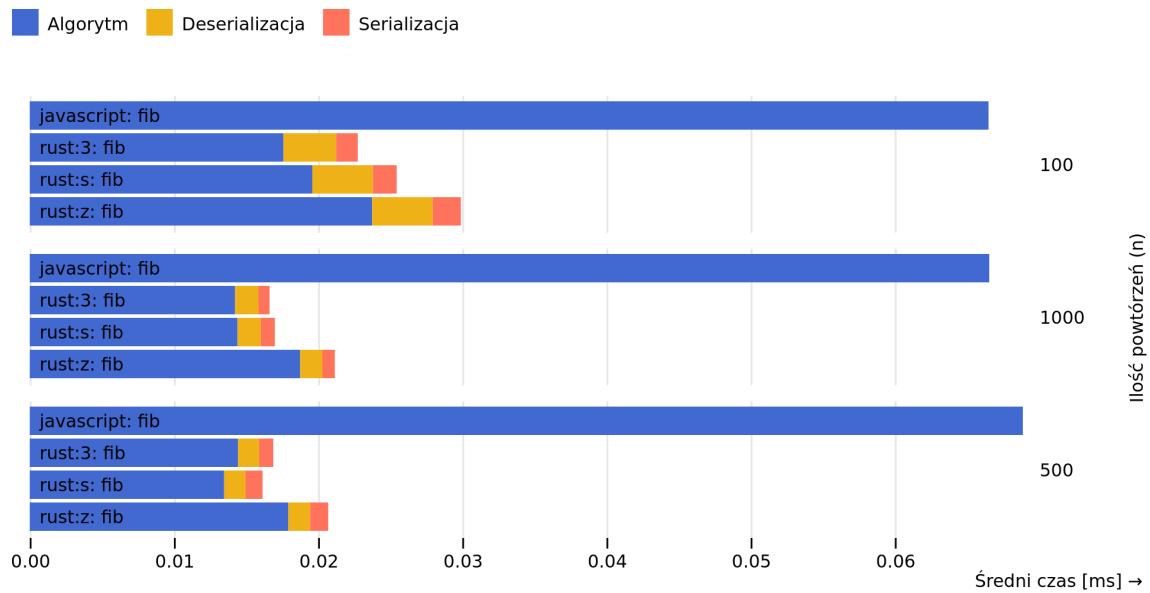
$$\text{dla } n = 10^4, S_C = \frac{950.4028}{194.9189} = 4.8759$$

$$\text{dla } n = 10^4, S_A = \frac{950.4028}{194.9125} = 4.8760$$

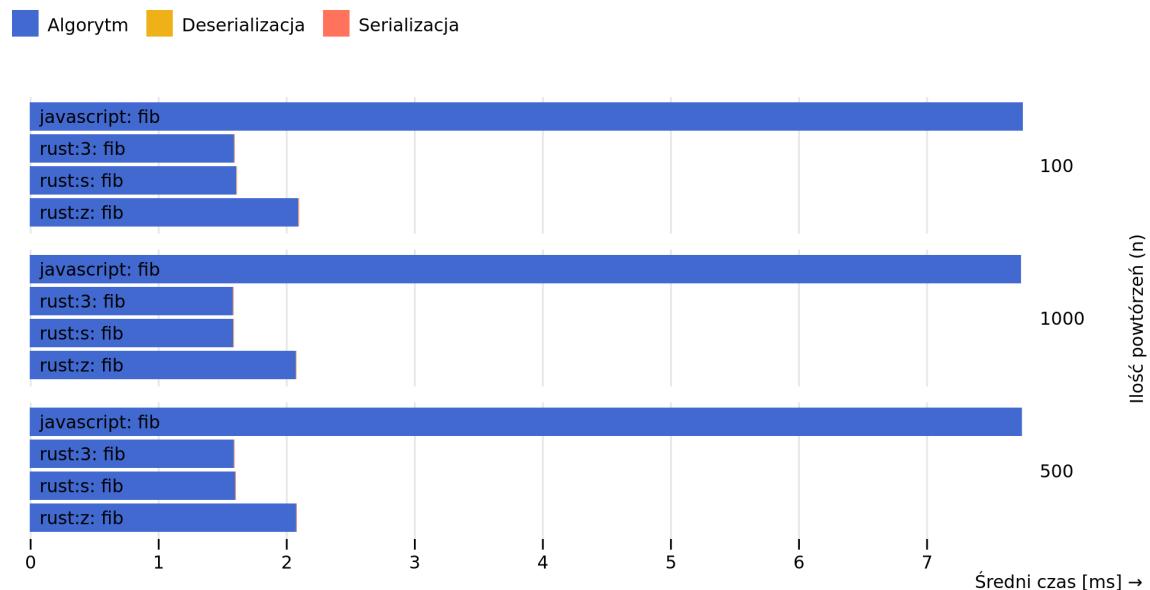
Algorytm zaimplementowany w Rust wywołuje się więc prawie 5 razy szybciej niż jego odpowiednik w języku JavaScript. Zbliżone wartości obu przyśpieszeń są spowodowane znikomym wpływem serializacji i deserializacji danych, na całkowity czas wykonania funkcji T_{CRS} .

		Średni czas wykonania [ms]								
		n = 100			n = 10 ³			n = 10 ⁴		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
1KB	F_1	—	0.0201	—	—	0.0097	—	—	0.0080	—
	F_2	—	0.0036	—	—	0.0033	—	—	0.0031	—
	F_3	0.0075	0.0083	0.0055	0.0035	0.0060	0.0042	0.0028	0.0051	0.0034
	F_4	0.0077	0.0019	0.0056	0.0034	0.0015	0.0042	0.0025	0.0015	0.0034
	F_5	0.0064	0.0078	0.0048	0.0034	0.0059	0.0042	0.0029	0.0050	0.0034
	F_6	0.0078	0.0070	0.0051	0.0035	0.0023	0.0041	0.0024	0.0021	0.0033
	F_7	0.0113	0.0098	0.0059	0.0035	0.0065	0.0042	0.0026	0.0056	0.0034
	F_8	0.0076	0.0074	0.0050	0.0036	0.0023	0.0042	0.0028	0.0018	0.0035
512KB	F_1	—	29.8940	—	—	27.0921	—	—	26.5381	—
	F_2	—	0.9912	—	—	0.9804	—	—	0.9839	—
	F_3	1.8428	2.3447	1.3143	1.8382	2.3430	1.3221	1.8487	2.3467	1.3294
	F_4	1.8126	0.3938	1.3009	1.8017	0.3925	1.2939	1.8183	0.3927	1.3023
	F_5	1.8442	2.3131	1.3214	1.8413	2.3074	1.3307	1.8440	2.3053	1.3315
	F_6	1.8134	0.6898	1.2962	1.8271	0.6963	1.3144	1.8287	0.6932	1.3119
	F_7	1.8470	2.6208	1.3177	1.8308	2.5884	1.3260	1.8406	2.5930	1.3309
	F_8	1.8106	0.6897	1.2974	1.7992	0.6862	1.2979	1.8178	0.6890	1.3036
1MB	F_1	—	60.4989	—	—	58.1103	—	—	55.6559	—
	F_2	—	2.1446	—	—	2.1905	—	—	2.1573	—
	F_3	3.6605	4.6478	2.6596	3.6679	4.6899	2.6363	3.6714	4.6840	2.6516
	F_4	3.6885	0.8019	2.6391	3.6723	0.7948	2.6415	3.6717	0.7979	2.6375
	F_5	3.6734	4.5885	2.6403	3.7044	4.6225	2.6513	3.6872	4.6211	2.6475
	F_6	3.6891	1.3999	2.6407	3.6430	1.3855	2.6474	3.6530	1.3881	2.6274
	F_7	3.6900	5.2013	2.6616	3.6702	5.2215	2.6608	3.6837	5.2051	2.6526
	F_8	3.6796	1.3974	2.6365	3.6695	1.3961	2.6582	3.6428	1.3840	2.6195

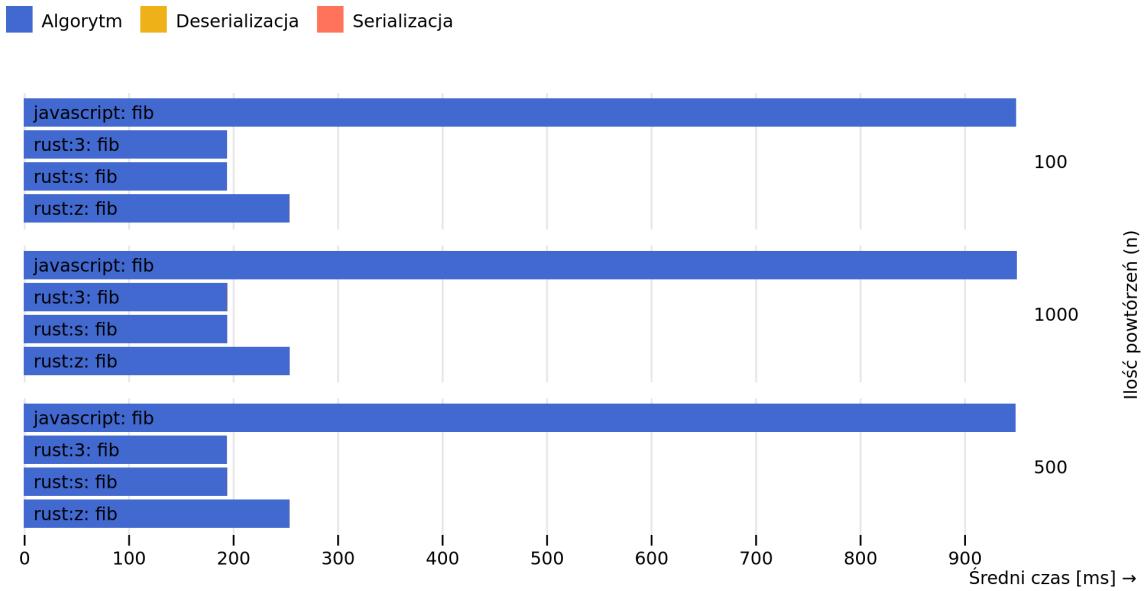
Tabela 5.2: Wyniki dla testu „Enkodowanie do Base64”



Rysunek 5.7: Wykres średniego czasu wykonania testu „20 liczba Fibonacciego (rekurencyjnie)”



Rysunek 5.8: Wykres średniego czasu wykonania testu „30 liczba Fibonacciego (rekurencyjnie)”



Rysunek 5.9: Wykres średniego czasu wykonania testu „40 liczba Fibonacciego (rekurencyjnie)”

Średni czas wykonania [ms]										
		$n = 100$			$n = 500$			$n = 10^3$		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
$k = 20$	F_1	—	0.0665	—	—	0.0689	—	—	0.0666	—
	F_2	0.0037	0.0176	0.0015	0.0015	0.0144	0.0010	0.0016	0.0142	0.0008
	F_3	0.0042	0.0196	0.0016	0.0015	0.0135	0.0012	0.0016	0.0144	0.0010
	F_4	0.0042	0.0238	0.0020	0.0015	0.0179	0.0012	0.0015	0.0188	0.0009
$k = 30$	F_1	—	7.7549	—	—	7.7478	—	—	7.7405	—
	F_2	0.0027	1.5962	0.0022	0.0017	1.5949	0.0028	0.0020	1.5868	0.0036
	F_3	0.0025	1.6135	0.0019	0.0018	1.6051	0.0025	0.0019	1.5894	0.0027
	F_4	0.0028	2.0976	0.0032	0.0021	2.0812	0.0029	0.0019	2.0787	0.0025
$k = 40$	F_1	—	949.7064	—	—	949.2112	—	—	950.4028	—
	F_2	0.0030	194.6100	0.0077	0.0020	194.5433	0.0056	0.0024	194.9125	0.0040
	F_3	0.0032	194.5790	0.0076	0.0022	194.7224	0.0053	0.0025	194.6945	0.0050
	F_4	0.0032	254.4528	0.0076	0.0026	254.4928	0.0064	0.0026	254.5499	0.0074

Tabela 5.3: Wyniki dla testu „ k -ta liczba Fibonacciego (rekurencyjnie)”

5.4 Mnożenie macierzy 4x4

W przypadku mnożenia macierzy wielkość danych wejściowych jest stała, natomiast nie można ich bezpośrednio przekazać do kontekstu WebAssembly. Wymaga to więc serializacji i deserializacji danych, co wpływa na czas wykonania funkcji. Wykres 5.10 wskazuje na to, że serializacja i deserializacja tych danych jest poddawana optymalizacji JIT. W takim wypadku warto przebadać przyśpieszenia dla największego poziomu optymalizacji.

W przypadku języka Rust, zbadałem dwie implementacje: identyczną jak w JavaScript i SIMD. Przyśpieszenia dla najszybszych wywołań identycznych implementacji wynoszą:

$$\text{dla } n = 10^4, S_C = \frac{0.0014}{0.0074} = 0.1892$$

$$\text{dla } n = 10^4, S_A = \frac{0.0014}{0.0008} = 1.7500$$

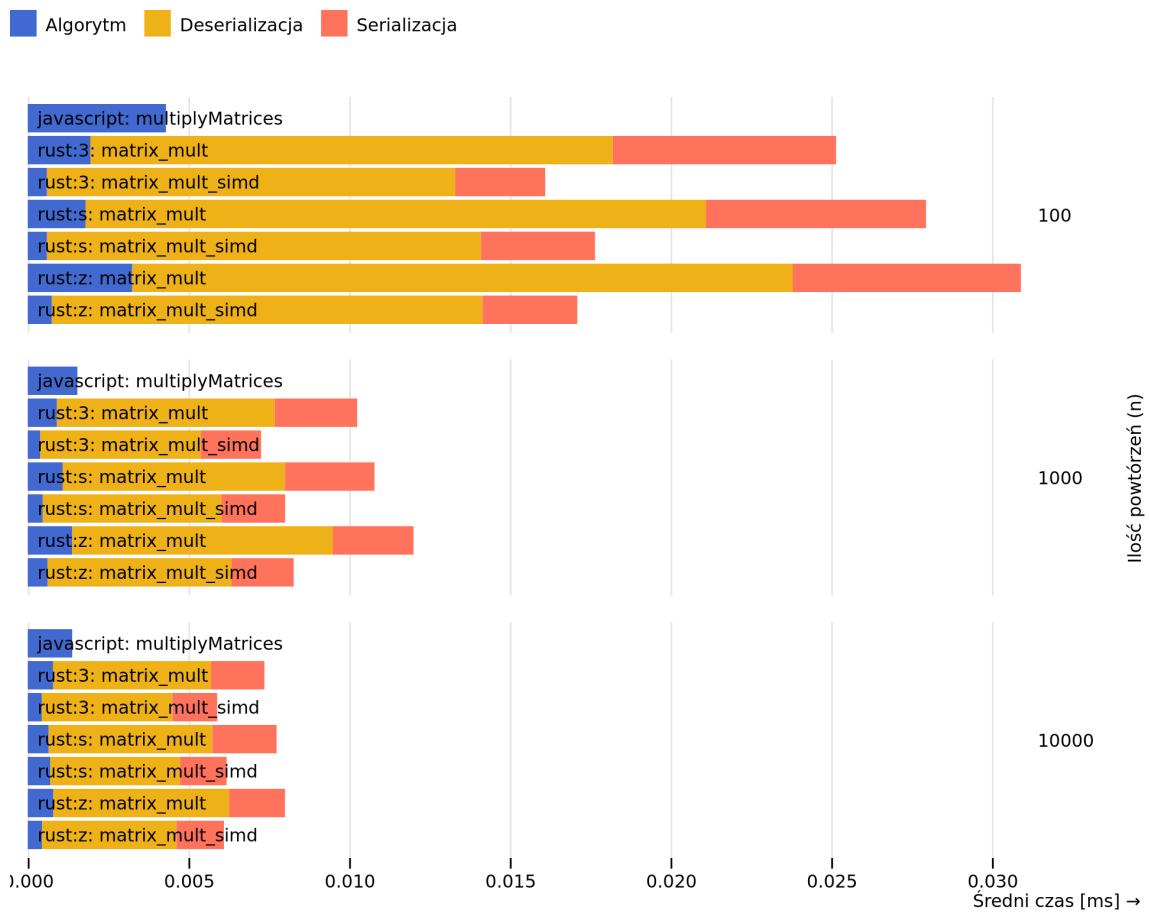
Jak widać, identyczny algorytm zaimplementowany w języku Rust jest ograniczony jedynie przez szybkość serializacji i deserializacji danych w kontekście JavaScript. Algorytm w języku Rust jest 1.75 raza szybszy niż w języku JavaScript, natomiast po uwzględnieniu czasu serializacji i deserializacji, jest on ponad 5 razy wolniejszy.

Przyśpieszenia dla implementacji iteracyjnej w JavaScript i implementacji SIMD w przypadku Rust wynoszą:

$$\text{dla } n = 10^4, S_C = \frac{0.0014}{0.0059} = 0.2373$$

$$\text{dla } n = 10^4, S_A = \frac{0.0014}{0.0004} = 3.5000$$

Przyśpieszenie całkowite, S_C , mówi o tym, że funkcja F_3 wykonuje się ponad 4 razy wolniej niż jej JavaScriptowy odpowiednik — F_1 . Jednakże, przyśpieszenie S_A pokazuje, że algorytm mnożący macierze wykorzystując operacje SIMD jest 3.5 raza szybszy niż podejście iteracyjne w języku JavaScript.



Rysunek 5.10: Wykres średniego czasu wykonania testu „Mnożenie macierzy 4x4”

Średni czas wykonania [ms]									
	$n = 100$			$n = 10^3$			$n = 10^4$		
	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
F_1	—	0.0043	—	—	0.0015	—	—	0.0014	—
F_2	0.0163	0.0020	0.0069	0.0068	0.0009	0.0026	0.0049	0.0008	0.0017
F_3	0.0127	0.0006	0.0028	0.0050	0.0004	0.0019	0.0041	0.0004	0.0014
F_4	0.0193	0.0018	0.0069	0.0069	0.0011	0.0028	0.0051	0.0007	0.0020
F_5	0.0135	0.0006	0.0036	0.0056	0.0005	0.0020	0.0040	0.0007	0.0014
F_6	0.0206	0.0033	0.0071	0.0081	0.0014	0.0025	0.0055	0.0008	0.0017
F_7	0.0134	0.0008	0.0030	0.0057	0.0006	0.0019	0.0042	0.0004	0.0015

Tabela 5.4: Wyniki dla testu „Mnożenie macierzy 4x4”

5.5 CRC-32

Każdy z wykresów 5.11, 5.12 i 5.13 przedstawia dwie identyczne zależności. Pierwszą z nich jest to, że funkcja F_1 będąca implementacją algorytmu CRC-32 w języku JavaScript jest wolniejsza od implementacji w języku Rust nawet po uwzględnieniu czasu serializacji i deserializacji.

Po przeanalizowaniu tabeli 5.5, można zauważyć, że czas deserializacji nie przekracza 55 μ s nawet dla 1MB danych. W przypadku poprzednich testów, które operowały na ciągach znaków, deserializacja takiej ilości danych potrafił zająć nawet 3.7ms. Niski czas deserializacji dla testów CRC-32 i CRC-64 jest spowodowany tym, że przekazywana jest tablica 8-bitowych wartości, która może być przekazana bezpośrednio do pamięci modułu WebAssembly.

Drugą zależnością, jaką można zauważyć na wykresach, jest to, że przy coraz większej ilości powtórzeń, funkcja JavaScriptowa wykonuje się coraz szybciej, natomiast funkcje uruchomione w kontekście WebAssembly wydają się mieć stałą wydajność. Średnia szybkość 100 wykonań funkcji F_1 jest, ponad dwukrotnie większa od średniej szybkości 10000 wykonań niezależnie od rozmiaru przesyłanych danych. Dla 1MB danych, różnica ta wynosi 47ms, natomiast dla funkcji F_2 , która uruchomiona jest w kontekście WebAssembly, różnica ta wynosi 35.1 μ s. Jest to przykład tego, że języki kompilowane JIT, w odróżnieniu od języków kompilowanych AOT, mają nieprzewidywalną wydajność.

Przyśpieszenia S_C i S_A dla najszybszych funkcji w obu technologiach (F_1 i F_2) wywołanych dla 1MB i największego poziomu optymalizacji JIT, wynoszą odpowiednio:

$$\text{dla } n = 10^4, S_C = \frac{28.1723}{0.0059} = 3.3466$$

$$\text{dla } n = 10^4, S_A = \frac{28.1723}{8.3668} = 3.3672$$

Dla 1KB danych przyśpieszenia te są równe:

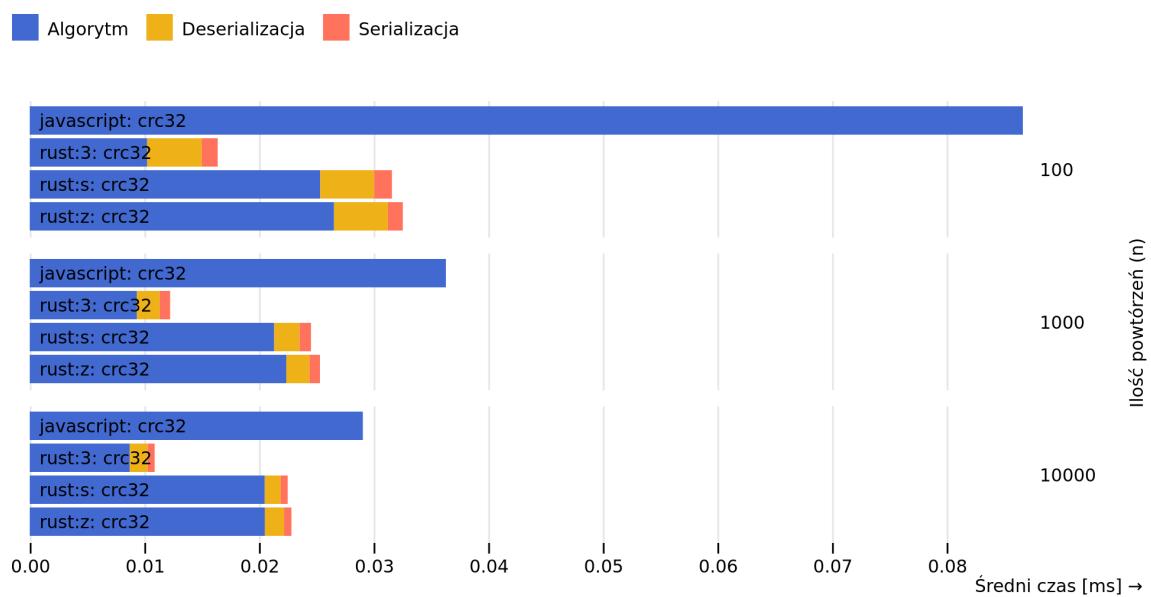
$$\text{dla } n = 10^4, S_C = \frac{0.0291}{0.0109} = 2.6697$$

$$\text{dla } n = 10^4, S_A = \frac{0.0291}{0.0087} = 3.4483$$

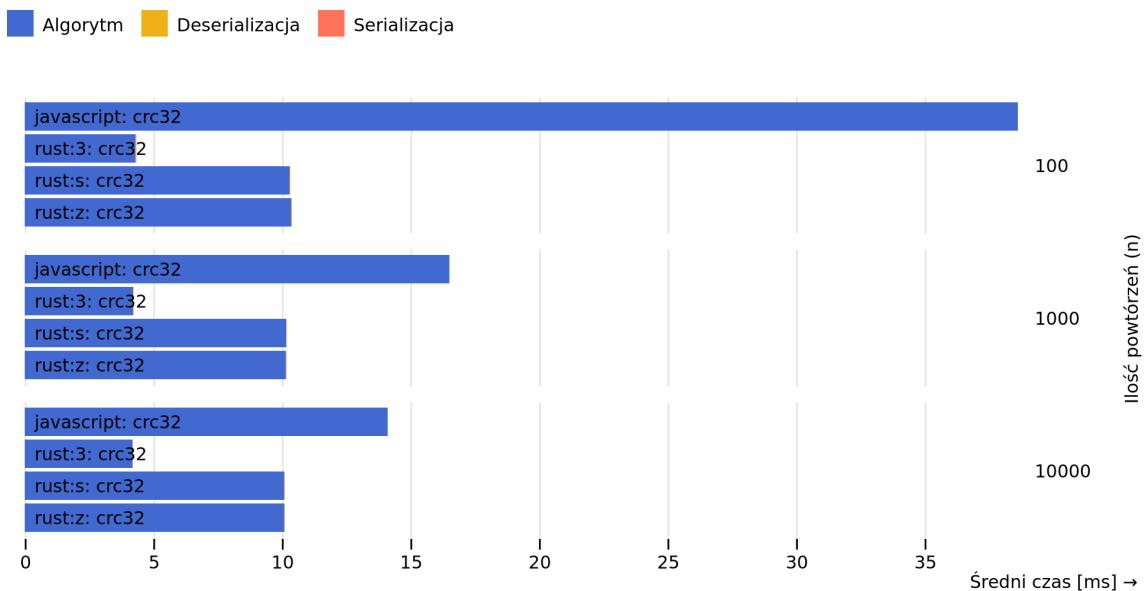
Wartości przyśpieszeń S_C i S_A wskazują na silną przewagę funkcji Rustowych uruchamianych w WebAssembly nad funkcjami JavaScriptowymi niezależnie od rozmiaru danych. Przekazanie wcześniej enkodowanych danych pozbywa się problemu długiego czasu deserializacji stringów w kontekście modułu WASM.

		Średni czas wykonania [ms]								
		$n = 100$			$n = 10^3$			$n = 10^4$		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
1KB	F_1	—	0.0867	—	—	0.0363	—	—	0.0291	—
	F_2	0.0047	0.0102	0.0014	0.0020	0.0094	0.0009	0.0016	0.0087	0.0006
	F_3	0.0047	0.0254	0.0016	0.0022	0.0213	0.0010	0.0014	0.0205	0.0006
	F_4	0.0047	0.0266	0.0013	0.0020	0.0224	0.0009	0.0017	0.0205	0.0006
512KB	F_1	—	38.6261	—	—	16.5178	—	—	14.1169	—
	F_2	0.0221	4.3107	0.0032	0.0216	4.2120	0.0035	0.0207	4.1864	0.0030
	F_3	0.0210	10.3036	0.0023	0.0197	10.1684	0.0035	0.0219	10.0899	0.0040
	F_4	0.0239	10.3649	0.0045	0.0238	10.1569	0.0028	0.0224	10.0910	0.0039
1MB	F_1	—	75.6474	—	—	32.5490	—	—	28.1723	—
	F_2	0.0441	8.4019	0.0039	0.0469	8.3756	0.0034	0.0472	8.3668	0.0042
	F_3	0.0454	20.2028	0.0051	0.0508	20.1859	0.0048	0.0544	20.1608	0.0032
	F_4	0.0483	20.2032	0.0050	0.0449	20.1863	0.0072	0.0525	20.1653	0.0067

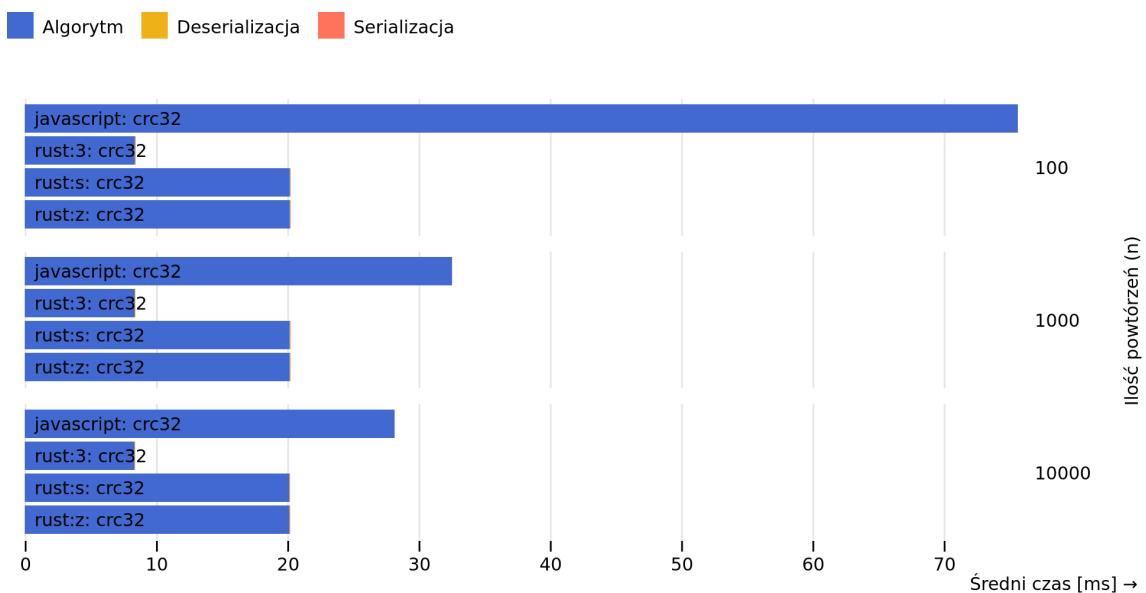
Tabela 5.5: Wyniki dla testu „CRC-32”



Rysunek 5.11: Wykres średniego czasu wykonania testu „CRC-32 (Plik 1KB)”



Rysunek 5.12: Wykres średniego czasu wykonania testu „CRC-32 (Plik 512KB)”



Rysunek 5.13: Wykres średniego czasu wykonania testu „CRC-32 (Plik 1MB)”

5.6 CRC-64

Wykresy 5.14, 5.15 i 5.16 przedstawiają wyniki wykonania funkcji obliczającej CRC-64. Funkcja F_1 będąca JavaScriptową implemnentacją algorytmu CRC-64 jest zdecydowanie wolniejsza od każdej z implementacji w języku Rust dla każdego rozmiaru danych i ilości powtórzeń. Wskazuje to na niską wydajność typu "**bigint**" w porównaniu do typu **u64** w języku Rust.

Przyśpieszenia S_C i S_A dla funkcji F_1 i F_2 , które są identycznymi implementacjami w obu technologiach przyjmują następujące wartości dla 1MB danych i największego stopnia optymalizacji JIT:

$$\begin{aligned} \text{dla } n = 10^4, S_C &= \frac{403.5189}{12.3553} = 32.6596 \\ \text{dla } n = 10^4, S_A &= \frac{403.5189}{12.3038} = 32.7962 \end{aligned}$$

32-krotne przyśpieszenie dla identycznego algorytmu, wykorzystującego 64-bitowe liczby, pokazuje jak wolne są operacje bitowe na dużych liczbach całkowitych, które nie mają zagwarantowanego rozmiaru.

Drugą ważną do omówienia funkcją jest funkcja F_3 , która jest zaimplementowana z wykorzystaniem operacji SIMD. W odniesieniu do JavaScriptowej implementacji, czyli funkcji F_1 , pozwala ona na następujące przyśpieszenia dla 1KB danych:

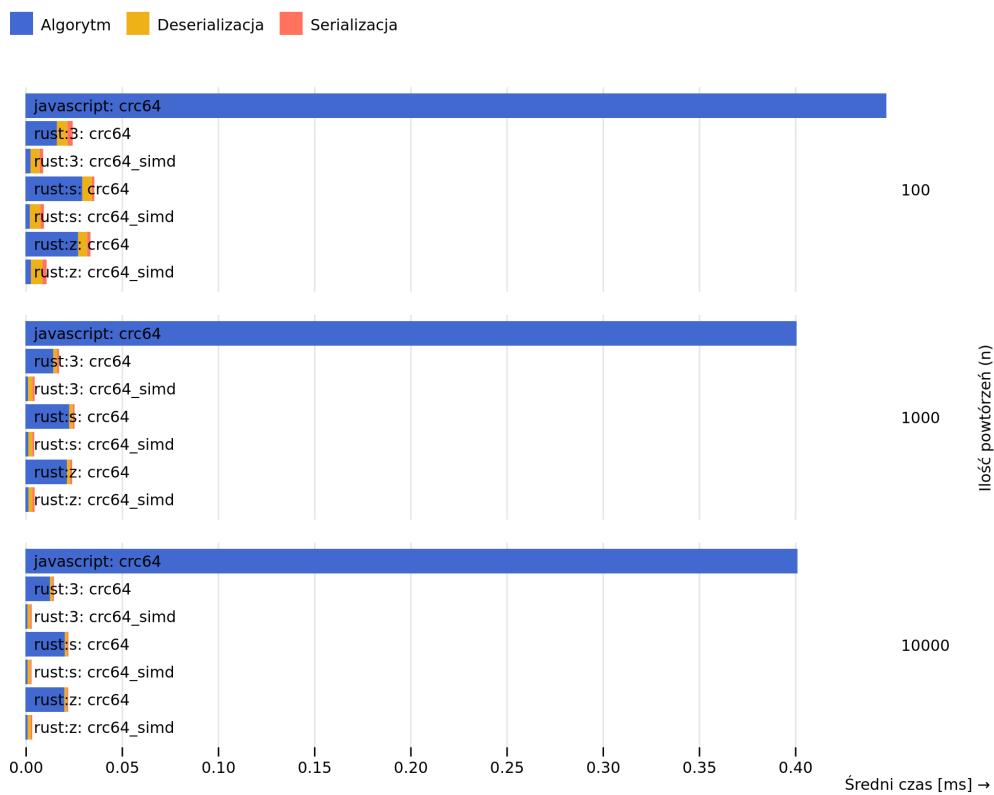
$$\begin{aligned} \text{dla } n = 10^4, S_C &= \frac{0.4014}{0.0033} = 121.6364 \\ \text{dla } n = 10^4, S_A &= \frac{0.4014}{0.0012} = 334.5000 \end{aligned}$$

Jak widać, dla małej ilości danych koszt przerzucenia pamięci do kontekstu WebAssembly znaczco wpływa na wynik przyśpieszenia. Wartość przyśpieszenia S_A jest o 2.76 raza większa od S_C .

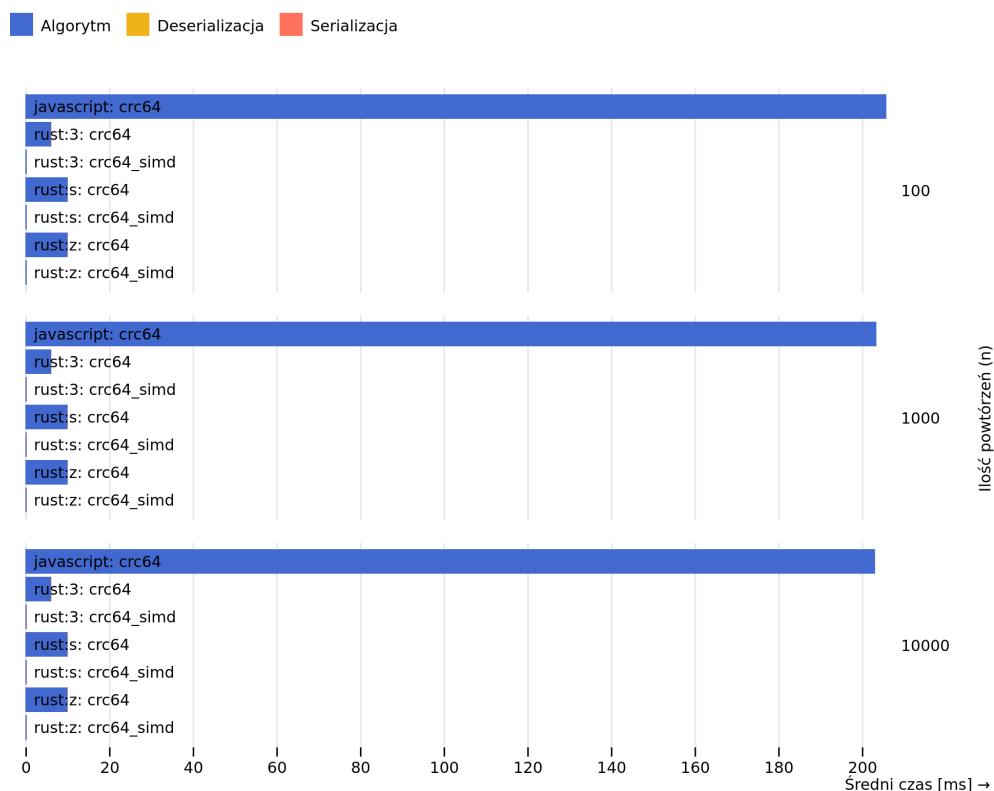
Warto jednak zaznaczyć, że przy większej ilości danych (np: 1MB), czasy serializacji przestają aż tak wpływać na przyśpieszenie:

$$\begin{aligned} \text{dla } n = 10^4, S_C &= \frac{403.5189}{0.6631} = 608.5340 \\ \text{dla } n = 10^4, S_A &= \frac{403.5189}{0.6288} = 641.7285 \end{aligned}$$

Przy 1MB danych, zarówno przyśpieszenie S_A jak i S_C pokazują, że funkcja F_3 wykona się ponad 600 razy szybciej niż F_1 . Jest to ogromna różnica, która pokazuje przewagę operacji SIMD, nad operacjami skalarnymi.



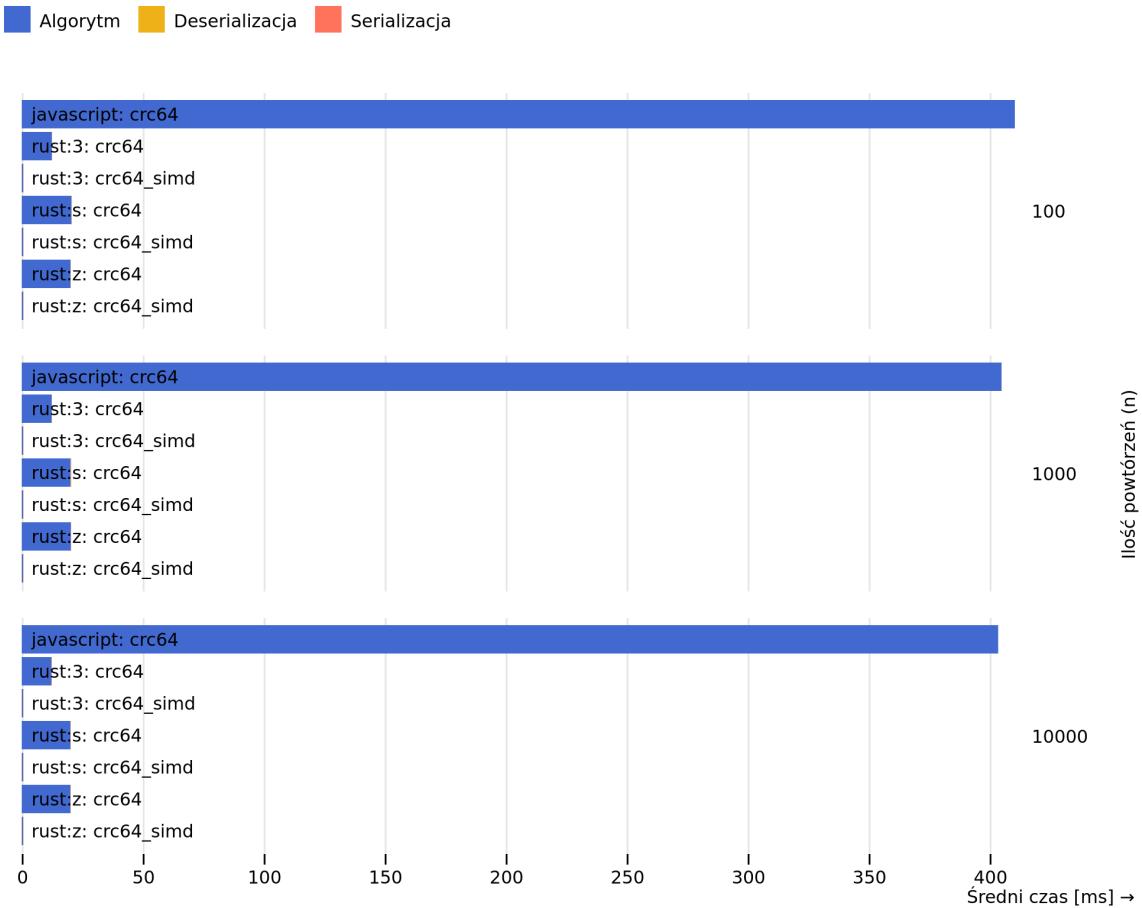
Rysunek 5.14: Wykres średniego czasu wykonania testu „CRC64 (Plik 1KB)”



Rysunek 5.15: Wykres średniego czasu wykonania testu „CRC64 (Plik 512KB)”

		Średni czas wykonania [ms]								
		n = 100			n = 10 ³			n = 10 ⁴		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
1KB	F_1	—	0.4475	—	—	0.4009	—	—	0.4014	—
	F_2	0.0055	0.0164	0.0027	0.0019	0.0146	0.0010	0.0013	0.0128	0.0007
	F_3	0.0047	0.0027	0.0019	0.0022	0.0014	0.0012	0.0014	0.0012	0.0007
	F_4	0.0050	0.0296	0.0013	0.0018	0.0228	0.0009	0.0013	0.0205	0.0006
	F_5	0.0055	0.0024	0.0018	0.0020	0.0016	0.0010	0.0014	0.0012	0.0006
	F_6	0.0047	0.0274	0.0018	0.0017	0.0217	0.0009	0.0013	0.0203	0.0006
	F_7	0.0060	0.0029	0.0022	0.0020	0.0017	0.0011	0.0017	0.0012	0.0007
512KB	F_1	—	205.8687	—	—	203.4827	—	—	203.1603	—
	F_2	0.0229	6.1596	0.0048	0.0238	6.1839	0.0027	0.0222	6.1617	0.0054
	F_3	0.0199	0.3430	0.0013	0.0157	0.3175	0.0008	0.0169	0.3147	0.0008
	F_4	0.0250	10.1454	0.0050	0.0227	10.1014	0.0052	0.0229	10.1002	0.0055
	F_5	0.0193	0.3441	0.0017	0.0155	0.3189	0.0009	0.0168	0.3160	0.0008
	F_6	0.0216	10.1316	0.0045	0.0233	10.1332	0.0048	0.0258	10.0888	0.0041
	F_7	0.0192	0.3453	0.0016	0.0167	0.3204	0.0010	0.0166	0.3173	0.0010
1MB	F_1	—	410.4072	—	—	404.9163	—	—	403.5189	—
	F_2	0.0418	12.4447	0.0031	0.0499	12.4119	0.0047	0.0483	12.3038	0.0032
	F_3	0.0359	0.6555	0.0014	0.0334	0.6320	0.0010	0.0335	0.6288	0.0008
	F_4	0.0462	20.5309	0.0047	0.1547	20.2071	0.0048	0.0491	20.1668	0.0050
	F_5	0.0360	0.6557	0.0017	0.0331	0.6342	0.0010	0.0330	0.6317	0.0010
	F_6	0.0498	20.2047	0.0058	0.0494	20.2820	0.0054	0.0477	20.1663	0.0044
	F_7	0.0357	0.6561	0.0014	0.0336	0.6376	0.0009	0.0342	0.6270	0.0008

Tabela 5.6: Wyniki dla testu „CRC-64”



Rysunek 5.16: Wykres średniego czasu wykonania testu „CRC64 (Plik 1MB)”

5.7 API DOM — Tworzenie elementów

Na wykresach 5.17, 5.18 i 5.19 widać, że funkcje F_2 , F_3 i F_4 są zdecydowanie wolniejsze od ich JavaScriptowego odpowiednika. Co ciekawe, na wykresach 5.17 i 5.18 możemy zauważyć, że kompilator JIT optymalizuje nie tylko funkcje F_1 , ale również funkcje zaimplementowane w języku Rust. Wskazuje to na to, że odwoływanie się do API DOM polega na ciągłym przełączaniu kontekstów pomiędzy WebAssembly, a JavaScriptem.

Z racji na to, że funkcje F_2 , F_3 i F_4 przyjmują jako argument jedynie liczbę elementów do stworzenia i niczego nie zwracają, obliczanie przyśpieszenia S_A wydaje się zbędne. Tak więc, dla F_1 i F_2 , dla 100 tworzonych elementów przyśpieszenie S_C wynosi:

$$\text{dla } n = 10, S_C = \frac{0.0425}{0.1735} = 0.2450$$

$$\text{dla } n = 100, S_C = \frac{0.0236}{0.0620} = 0.3806$$

$$\text{dla } n = 500, S_C = \frac{0.0213}{0.0567} = 0.3743$$

Natomiast dla 10000 tworzonych elementów przyśpieszenie S_C wynosi:

$$\text{dla } n = 10, S_C = \frac{1.7385}{4.7330} = 0.3673$$

$$\text{dla } n = 100, S_C = \frac{1.7960}{4.6199} = 0.3888$$

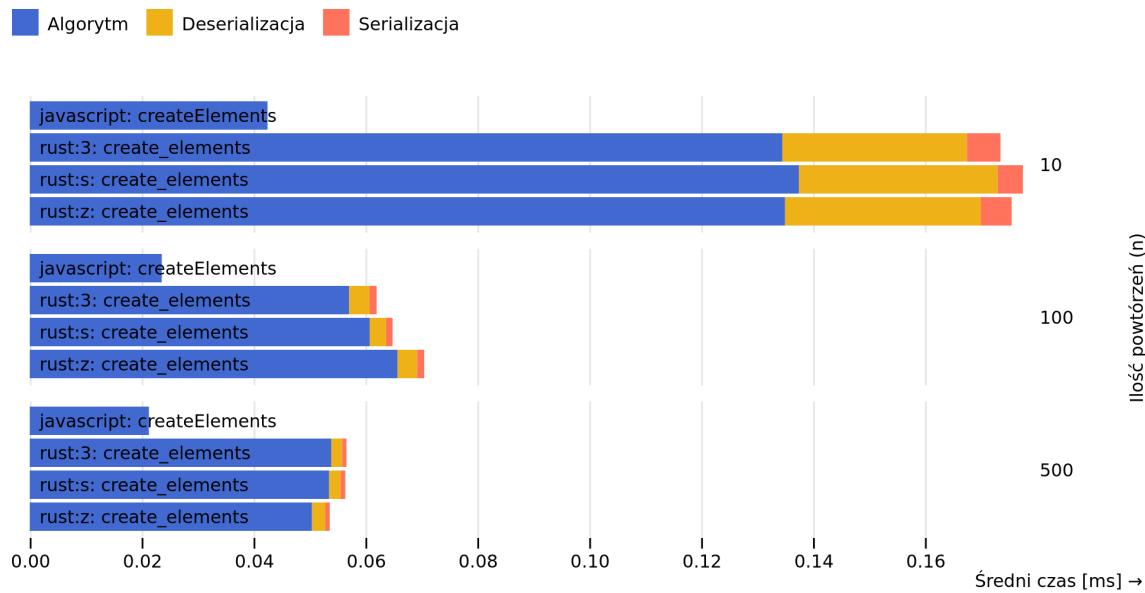
$$\text{dla } n = 500, S_C = \frac{1.9003}{4.7219} = 0.4024$$

Oznacza to, że tworzenie elementów i dodawanie ich do dokumentu w pętli z poziomu WebAssembly jest wolniejsze od 2.5 do 4 razy niż wykonywanie tej samej czynności bezpośrednio z poziomu JavaScriptu.

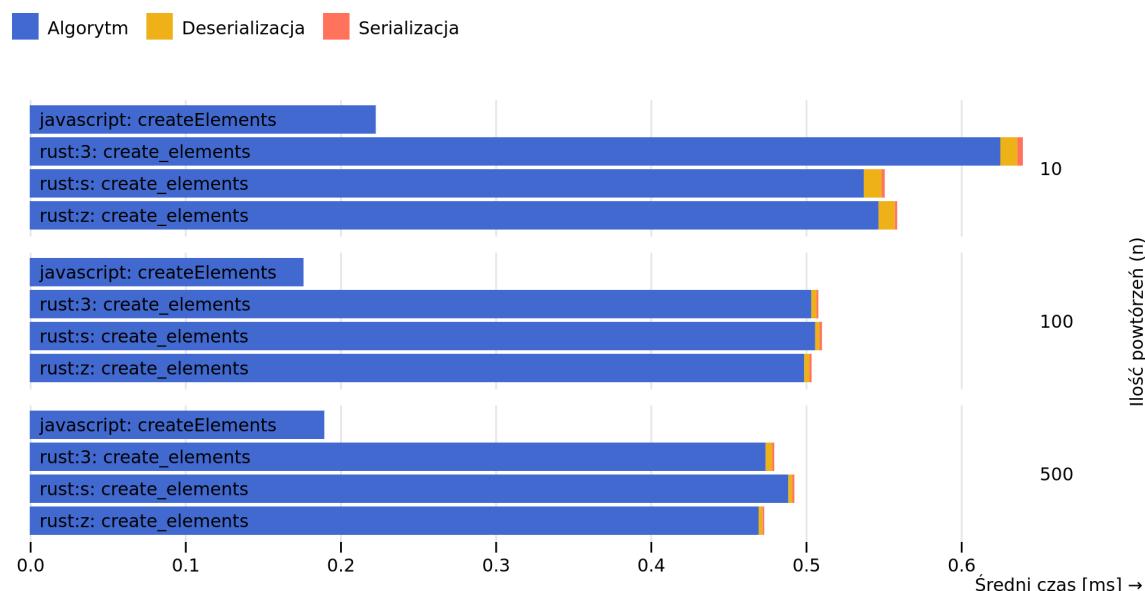
Kolejną wartą zauważenia zależnością jest to, że wśród wszystkich trzech opcji optymalizacyjnych dla języka Rust trudno wskazać najszybszą funkcję. Nie dość, że wydajność WebAssembly w tym teście nie jest stała między ilością powtórzeń, to dodatkowo wydajność tych funkcji wydaje się być losowa.

Średni czas wykonania [ms]										
		$n = 10$			$n = 100$			$n = 500$		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
$k = 100$	F_1	—	0.0425	—	—	0.0236	—	—	0.0213	—
	F_2	0.0330	0.1345	0.0060	0.0037	0.0571	0.0012	0.0020	0.0539	0.0008
	F_3	0.0355	0.1375	0.0045	0.0030	0.0607	0.0011	0.0020	0.0535	0.0008
	F_4	0.0350	0.1350	0.0055	0.0035	0.0658	0.0013	0.0024	0.0505	0.0008
$k = 10^3$	F_1	—	0.2230	—	—	0.1764	—	—	0.1898	—
	F_2	0.0110	0.6255	0.0035	0.0033	0.5037	0.0012	0.0043	0.4742	0.0013
	F_3	0.0115	0.5375	0.0020	0.0028	0.5062	0.0015	0.0025	0.4889	0.0013
	F_4	0.0105	0.5470	0.0015	0.0034	0.4992	0.0014	0.0025	0.4697	0.0011
$k = 10^4$	F_1	—	1.7385	—	—	1.7960	—	—	1.9003	—
	F_2	0.0100	4.7155	0.0075	0.0039	4.6114	0.0046	0.0028	4.7149	0.0042
	F_3	0.0105	4.6175	0.0060	0.0037	4.7487	0.0053	0.0028	4.7760	0.0041
	F_4	0.0115	4.6155	0.0080	0.0032	4.6328	0.0053	0.0027	4.9036	0.0040

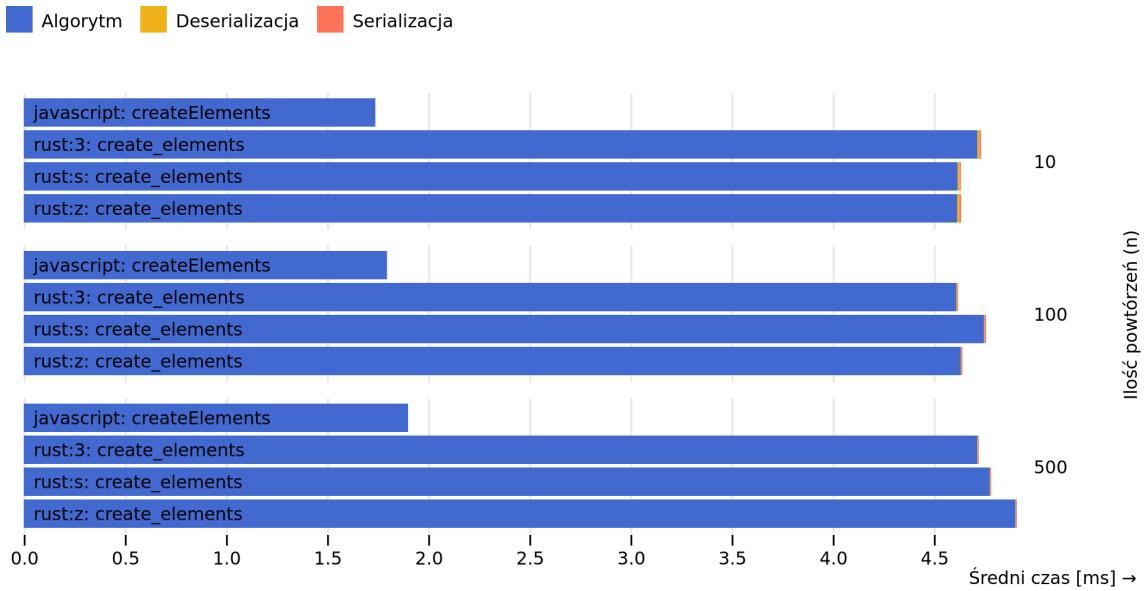
Tabela 5.7: Wyniki dla testu „API DOM — Tworzenie elementów”



Rysunek 5.17: Wykres średniego czasu wykonania testu „API DOM — Tworzenie 100 elementów”



Rysunek 5.18: Wykres średniego czasu wykonania testu „API DOM — Tworzenie 1000 elementów”



Rysunek 5.19: Wykres średniego czasu wykonania testu „API DOM — Tworzenie 10000 elementów”

5.8 API DOM — Aktualizacja co drugiego elementu

Tak samo jak w przypadku poprzedniego testu, na wykresach 5.20, 5.21 i 5.22 możemy zobaczyć, że wydajność funkcji JavaScriptowej jest znacznie większa niż funkcji skompilowanych do WebAssembly. Dodatkowo widać też, że funkcje te nie mają stałej wydajności i podlegają optymalizacjom JIT.

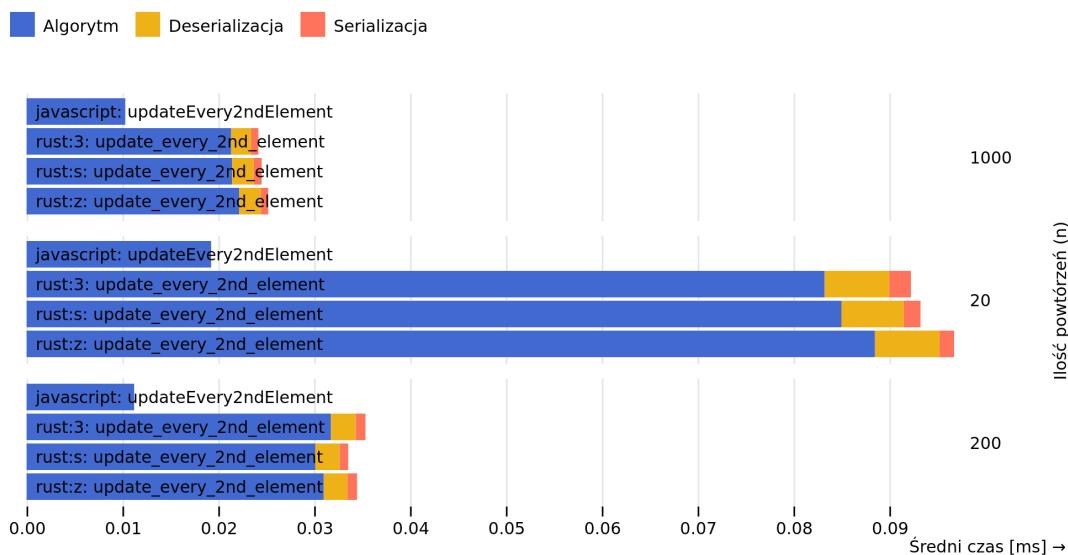
Podobnie jak w poprzednim teście, wydajność funkcji uruchomionych w kontekście WebAssembly wydaje się być losowa. Dopiero przy aktualizacji 10000 elementów wydajność tych funkcji wydaje się stabilizować, co widać na rysunku 5.22.

Z tego powodu zdecydowałem się obliczyć S_C dla 10000 elementów dla najbardziej zoptymalizowanych funkcji F_1 i F_2 .

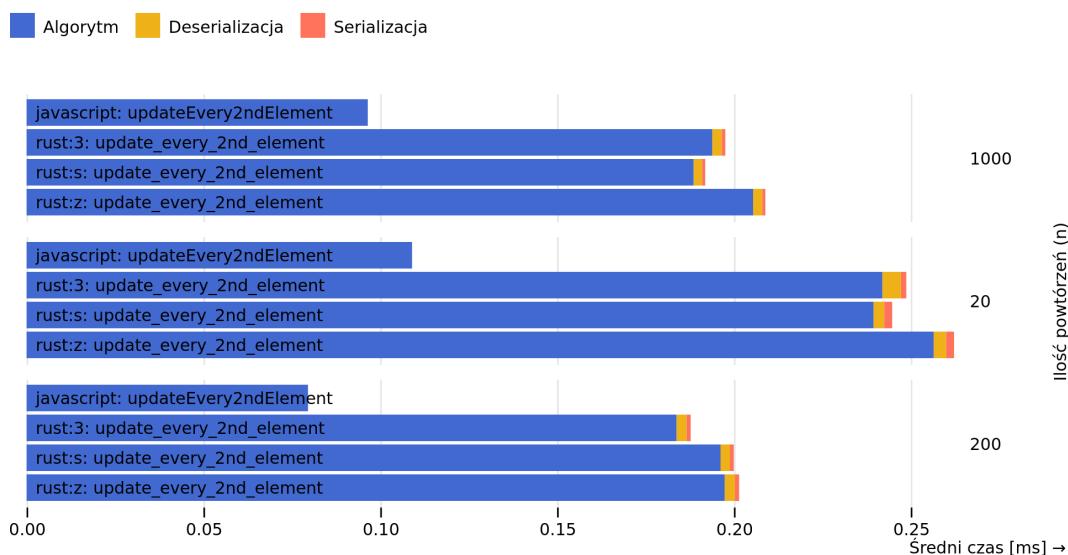
$$\text{dla } n = 1000, S_C = \frac{0.8885}{1.9172} = 0.4634$$

Oznacza to, że funkcja uruchomiona w kontekście WebAssembly jest ponad 2 razy wolniejsza od funkcji w języku JavaScript. W przypadku tego testu jest to wręcz spodziewane. Aktualizacja elementu treści wymaga przesłania ciągów znaków, które muszą być re-enkodowane. Z racji na to, że czas deserializacji i serializacji jest badany jedynie przy rozpoczęciu i zakończeniu funkcji, nie wykazuje on serializacji i deserializacji,

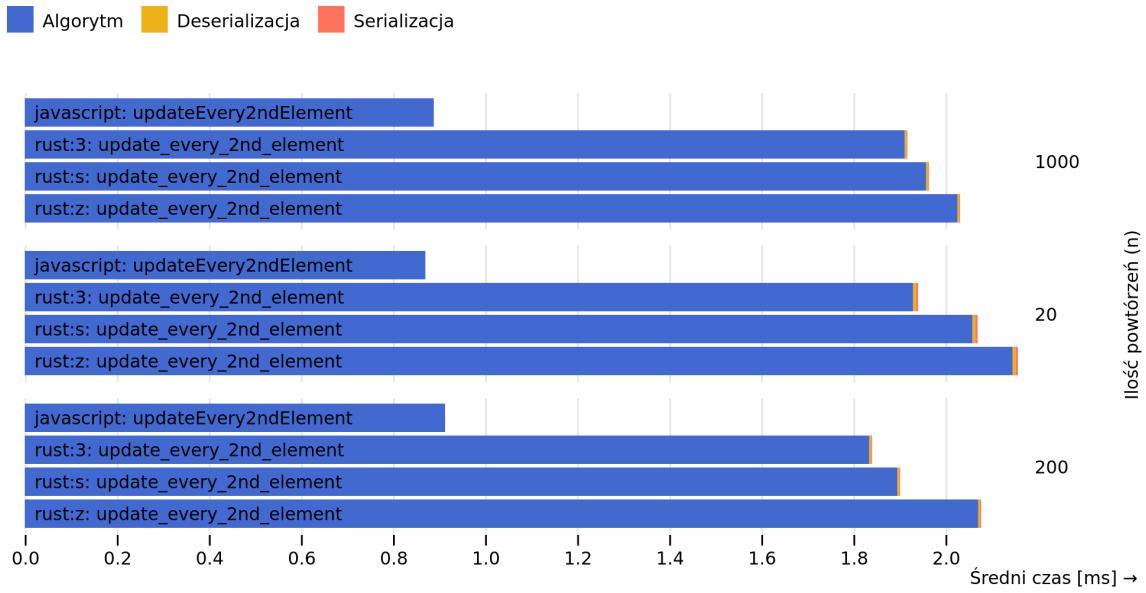
która zachodzi podczas zmiany kontekstu. Stąd też może się brać optymalizacja JIT funkcji skompilowanych do modułu WASM.



Rysunek 5.20: Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego ze 100 elementów”



Rysunek 5.21: Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego z 1000 elementów”



Rysunek 5.22: Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego z 10000 elementów”

Średni czas wykonania [ms]										
		$n = 20$			$n = 200$			$n = 10^3$		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
$k = 100$	F_1	—	0.0193	—	—	0.0112	—	—	0.0103	—
	F_2	0.0067	0.0833	0.0022	0.0026	0.0317	0.0010	0.0021	0.0213	0.0008
	F_3	0.0065	0.0850	0.0017	0.0025	0.0301	0.0009	0.0023	0.0214	0.0008
	F_4	0.0067	0.0885	0.0015	0.0025	0.0310	0.0010	0.0023	0.0222	0.0007
$k = 10^3$	F_1	—	0.1090	—	—	0.0796	—	—	0.0965	—
	F_2	0.0053	0.2420	0.0015	0.0028	0.1838	0.0012	0.0027	0.1939	0.0010
	F_3	0.0030	0.2395	0.0023	0.0027	0.1962	0.0011	0.0024	0.1886	0.0009
	F_4	0.0035	0.2565	0.0022	0.0029	0.1974	0.0012	0.0026	0.2054	0.0009
$k = 10^4$	F_1	—	0.8702	—	—	0.9131	—	—	0.8885	—
	F_2	0.0063	1.9298	0.0045	0.0042	1.8346	0.0017	0.0040	1.9116	0.0016
	F_3	0.0060	2.0590	0.0048	0.0041	1.8956	0.0020	0.0042	1.9581	0.0018
	F_4	0.0067	2.1458	0.0048	0.0042	2.0711	0.0023	0.0039	2.0261	0.0016

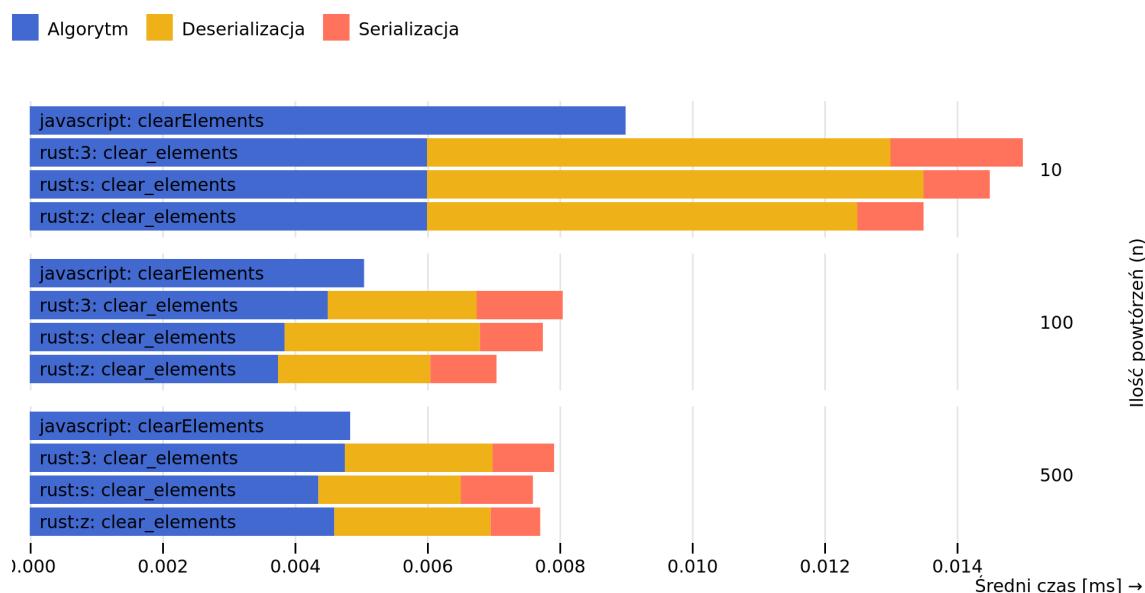
Tabela 5.8: Wyniki dla testu „API DOM — Aktualizacja co drugiego elementu”

5.9 API DOM — Usuwanie elementów

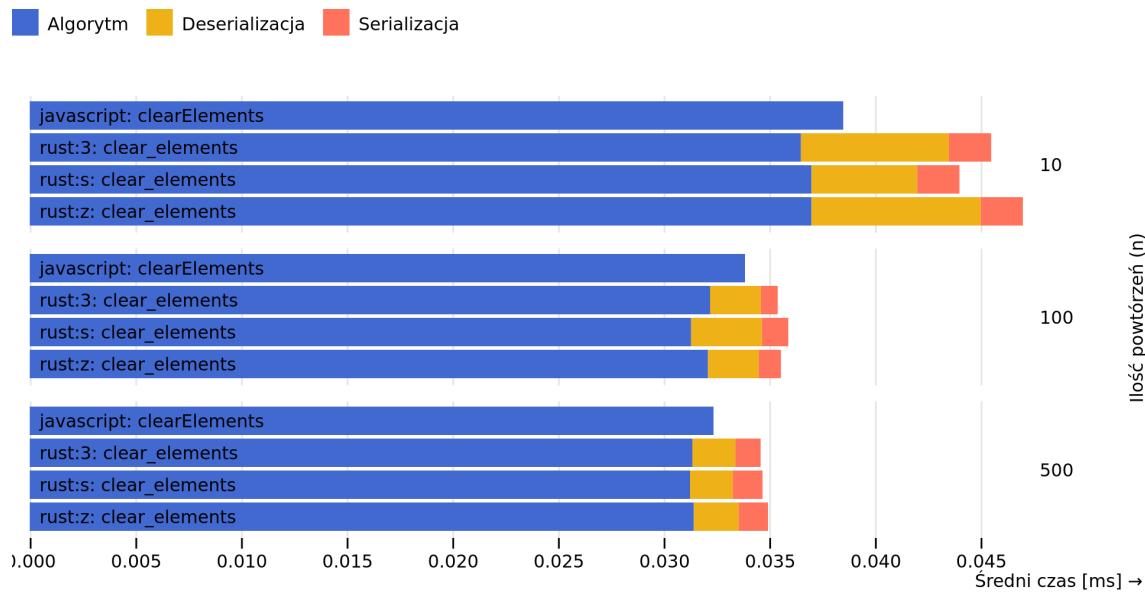
Usuwanie elementów jest banalnie prostym testem. Polega ono na ustawieniu treści kontenera na pusty string. Jak widać na wykresach 5.23, 5.24 i 5.25, wydajność wszystkich badanych funkcji jest do siebie zbliżona niezależnie od ilości powtórzeń, ilości elementów w kontenerze, czy stopnia optymalizacji.

Dla 100 elementów, różnice w czasach wydają się największe pomimo bardzo krótkich czasów wykonania. Z tego powodu postanowiłem obliczyć S_C dla funkcji F_1 i F_2 dla 100 elementów i 500 powtórzeń. Wybór funkcji F_2 zamiast F_4 może wydawać się na pierwszy rzut oka nieuzasadniony, ponieważ F_4 wykonuje się szbciej od F_1 dla wszystkich ilości powtórzeń przy tej liczbie elementów. Należy jednak pamiętać, że różnica w tych czasach jest równa $0.1\mu\text{s}$ co sprawia, że są one praktycznie identyczne. Tak więc dla 100 elementów, przy 500 powtórzeniach, przyśpieszenie S_C wynosi:

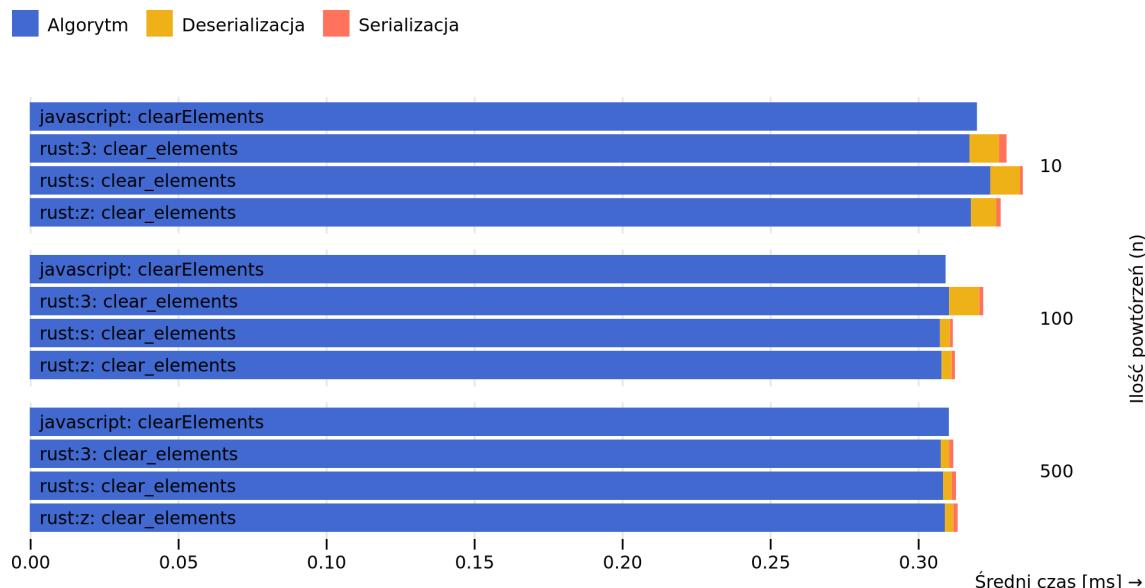
$$\text{dla } n = 500, S_C = \frac{0.0048}{0.0079} = 0.6076$$



Rysunek 5.23: Wykres średniego czasu wykonania testu „API DOM — Usuwanie 100 elementów”



Rysunek 5.24: Wykres średniego czasu wykonania testu „API DOM — Usuwanie 1000 elementów”



Rysunek 5.25: Wykres średniego czasu wykonania testu „API DOM — Usuwanie 10000 elementów”

		Średni czas wykonania [ms]								
		n = 10			n = 100			n = 500		
		\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S	\bar{T}_D	\bar{T}_A	\bar{T}_S
$k = 100$	F_1	—	0.0090	—	—	0.0050	—	—	0.0048	—
	F_2	0.0070	0.0060	0.0020	0.0023	0.0045	0.0013	0.0022	0.0048	0.0009
	F_3	0.0075	0.0060	0.0010	0.0030	0.0038	0.0009	0.0021	0.0044	0.0011
	F_4	0.0065	0.0060	0.0010	0.0023	0.0037	0.0010	0.0024	0.0046	0.0008
$k = 10^3$	F_1	—	0.0385	—	—	0.0338	—	—	0.0324	—
	F_2	0.0070	0.0365	0.0020	0.0024	0.0322	0.0008	0.0020	0.0314	0.0012
	F_3	0.0050	0.0370	0.0020	0.0033	0.0313	0.0012	0.0020	0.0313	0.0014
	F_4	0.0080	0.0370	0.0020	0.0024	0.0321	0.0011	0.0021	0.0314	0.0014
$k = 10^4$	F_1	—	0.3200	—	—	0.3094	—	—	0.3105	—
	F_2	0.0100	0.3175	0.0025	0.0102	0.3106	0.0013	0.0028	0.3078	0.0014
	F_3	0.0100	0.3245	0.0010	0.0033	0.3075	0.0010	0.0029	0.3086	0.0015
	F_4	0.0085	0.3180	0.0015	0.0034	0.3081	0.0011	0.0028	0.3092	0.0015

Tabela 5.9: Wyniki dla testu „API DOM — Usuwanie elementów”

6. Wnioski

Celem tej pracy było porównanie wydajności języka Rust, skompilowanego do WebAssembly, z językiem JavaScript w środowiskach przeglądarkowych oraz odpowiedzenie na pytanie: „Czy Rust jest lepszym wyborem dla aplikacji przeglądarkowych niż JavaScript?”.

Aby odpowiedzieć na to pytanie, stworzyłem środowisko testowe, w którym byłem w stanie porównać zarówno funkcje JavaScriptowe, jak i funkcje napisane w języku Rust skompilowane do WebAssembly. Pozwoliło mi to na osiągnięcie celu pracy w postaci zbadanie różnic wydajności obu technologii dla różnych danych wejściowych i na różnym poziomie optymalizacji kompilatora JIT. Następnie dokonałem trzech pomiarów dla każdej funkcji:

- Czasu deserializacji danych w kontekście WebAssembly
- Czasu wykonania algorytmu
- Czasu serializacji danych w kontekście WebAssembly

Serializacja i deserializacja danych nie jest wymagana dla funkcji w języku JavaScript, więc w takich przypadkach czasy te nie były brane pod uwagę.

Pomiary te pozwoliły mi na ustalenie dwóch rodzajów przyśpieszeń:

- Przyśpieszenia całkowitego funkcji w języku Rust w odniesieniu do funkcji JavaScriptowej
- Przyśpieszenia algorytmicznego funkcji w języku Rust w odniesieniu do funkcji JavaScriptowej

Przyśpieszenie całkowite różni się od przyśpieszenia algorytmicznego tym, że uwzględnia ono czasy serializacji i deserializacji danych. Reprezentuje ono wydajność funkcji skompilowanej do WebAssembly w przypadku zastosowania jej w aplikacjach przeglądarkowych zaimplementowanych w języku JavaScript, które wymieniają dane z modułami WebAssembly w celu wykonania obliczeń. Przyśpieszenie algorytmiczne reprezentuje natomiast porównanie wydajności pomiędzy dwoma aplikacjami: jedną zaimplementowaną w języku JavaScript i drugą w pełni zaimplementowaną w WebAssembly.

Należy jednak pamiętać, że środowiska przeglądarkowe, w których przeprowadzone były moje testy dokonują pomiarów z maksymalną dokładnością do $5\mu\text{s}$. Niektóre przeglądarki mogą dodatkowo nakładać tak zwane drżenie wartości na pomiary w celu zabezpieczenia

się przed atakami typu „timing attack”. Mogło to wpływać na poprawność moich wyników, jednakże, aby zminimalizować wpływ tych zabezpieczeń, każdy pomiar był wykonywany dla różnych ilości powtórzeń z przedziału od 10 do 10000, a wyniki były uśredniane. Takie podejście pozwoliło zrozumieć nie tylko wydajność obu technologii, ale również pokazało w jaki sposób kompilator JIT optymalizuje kod JavaScriptowy.

Wyliczenie tych przyśpieszeń dla każdej pary funkcji we wszystkich testach byłoby zbyt czasochłonne, jeżeli nie zbędne. Dlatego też wyliczyłem wartości tych przyśpieszeń dla testów, które w moim odczuciu najlepiej reprezentują różnice w wydajności pomiędzy językiem JavaScript a WebAssembly.

6.1 Wydajność funkcji wymagających serializacji danych

Dla funkcji wymagających serializacji małej ilości danych, takich jak Base64 czy mnożenie macierzy 4x4, przyśpieszenie całkowite przybierało wartość mniejszą niż 1, natomiast przyśpieszenie algorytmiczne było większe niż 1. Oznacza to, że serializacja przekazywanych danych (dokonywana zarówno po stronie JavaScriptu, jak i WebAssembly) negatywnie wpływa na wydajność modułów WASM. Przyśpieszenie algorytmiczne dla tych funkcji przewyższało wartość 1, co oznacza, że sam algorytm wykonany z kontekstu WebAssembly był szybszy niż algorytm wykonany w kontekście JavaScriptowym.

Przy większych rozmiarach danych wejściowych, WebAssembly prześcigało implementacje JavaScriptowe. Dla własnej implementacji enkodera Base64 i 512KB danych przyśpieszenie całkowite wynosiło w zaokrągleniu 4.8, natomiast algorytmiczne wynosiło 11.3. Wskazuje to na to, że implementacje w języku JavaScript mogą być lepszym rozwiązaniem w momencie, gdy pracujemy z małą ilością danych. Gdy natomiast wiemy, że nasze dane będą na tyle duże, że ich rozmiar stanie się wąskim gardłem dla wydajności, warto rozważyć implementację w języku Rust skompilowanym do WebAssembly.

6.2 Wydajność funkcji niewymagających serializacji danych

W przypadkach gdy serializacja danych nie była wymagana, zarówno przyśpieszenia całkowite, jak i przyśpieszenia algorytmiczne przybierały wartość większą od 1. Funkcje zaimplementowane w języku Rust, które nie wymagały serializacji, takie jak rekurencyjne obliczenie k-tej liczby Fibonacciego czy obliczenie sumy CRC-32 były od 2.7 do 4.9

raza szybsze od ich JavaScriptowych odpowiedników niezależnie od rozmiaru danych wejściowych. Przyśpieszenia algorytmiczne dla tych funkcji przybierały wartości od 3.4 do 4.9, co wskazuje na to, że algorytmy zaimplementowane w języku Rust były znacznie bardziej wydajne niż ich odpowiedniki w języku JavaScript.

Funkcja obliczająca sumę CRC-64 napisana w języku Rust była w zaokrągleniu 33 razy szybsza niż identyczna implementacja w języku JavaScript. Jest to negatywne zaskoczenie ponieważ świadczy to o ekstremalnie niskiej wydajności typu "**bigint**" w języku JavaScript. Warto więc wziąć WebAssembly pod uwagę w przypadku implementacji aplikacji przeglądarkowych, które wymagają algorytmów operujących na liczbach całkowitych większych niż 64 bity.

6.3 Wydajność odwołań do API DOM

Testy skupiające się na API DOM wykazały, że ciągłe przełączanie się pomiędzy kontekstami w celu wywołania funkcji JavaScriptowych z poziomu WebAssembly negatywnie wpływa na wydajność. Funkcje zaimplementowane w języku Rust były od 2.5 do 4 razy wolniejsze od ich odpowiedników w języku JavaScript. Testy były zaprojektowane w taki sposób, że elementy były dodawane i aktualizowane dynamicznie, w każdej iteracji pętli. Bibliotek wasm-bindgen wykorzystywana do komunikacji z API DOM mogła wysyłać nazwy elementów bądź nową treść elementów przy każdej iteracji, co tłumaczyłoby niską, niestałą wydajność funkcji napisanych w języku Rust. Przenoszenie tych ciągów znaków wiązałoby się z koniecznością serializacji danych, która byłaby uznana za czas algorytmu przez zaprojektowany przeze mnie system pomiarowy. Kolejną potencjalną przyczyną niestałej, niskiej wydajności czasu algorytmu WebAssembly mogło być uruchamianie mechanizmu Garbage Collectora po przełączeniu do kontekstu JavaScriptowego.

Można znaleźć wiele strategii na zoptymalizowanie zapytań do API DOM. Nałożenie wszystkich modyfikacji DOM w jednym kroku, czy też oddelegowanie krytycznych operacji do kodu w JavaScript to tylko kilka z nich. Na chwilę obecną, póki WebAssembly nie ma bezpośredniego dostępu do API DOM, powinniśmy pokusić się o skorzystanie z frameworków pokroju Leptos, czy Sycamore, które są w stanie zoptymalizować te operacje za nas.

Dodatkowo, w przeciwieństwie do innych testów, czasy wykonania funkcji WebAssembly dla testów API DOM ulegały optymalizacji JIT, co potwierdza przypuszczenia, że duża część czasu tych funkcji jest spędzana w kontekście JavaScript.

6.4 Wydajność funkcji SIMD

Operacje SIMD są dostępne jedynie w WebAssembly i mogą drastycznie przyśpieszyć działanie wielu funkcji. Dla enkodera Base64, obliczania CRC-64 oraz mnożenia macierzy 4x4 przebadałem dodatkowe implementacje wykorzystujące operacje SIMD. Jednak przed omówieniem ich wyników, warto zauważyć, że implementacje wykorzystujące operacje SIMD różniły się od porównywanych do nich funkcji działaniem algorytmu. Aby zmienić implementację na taką, która wykorzystuje operacje SIMD do obliczenia wyników należałoby diametralnie zmienić algorytm. To sprawiłoby, że porównanie funkcji SIMD w WebAssembly do funkcji skalarnej w JavaScript porównałoby dwa zupełnie różne algorytmy w dwóch różnych kontekstach wykonujących kod. Co więcej, celem testowanych funkcji skalarnych było porównanie identycznych implementacji, które nie były implementowane z największą wydajnością w zamyśle. Należy więc patrzeć na następujące wyniki z przymrużeniem oka.

Zbadane przyśpieszenie całkowite dla funkcji enkodującej Base64 z wykorzystaniem SIMD w porównaniu do natywnej funkcji dostępnej w Web API wynosiło w przybliżeniu 7.6. Wynik ten jest jednak spowodowany wąskim gardłem serializacji danych. Jeżeli popatrzymy na przyśpieszenie algorytmiczne, wynosi ono 67.6. Jest to bardzo duży skok wydajności względem naiwnej, skalarnej implementacji algorytmu JavaScriptowego.

Implementacja SIMD funkcji mnożącej dwie macierze 4x4 była w przybliżeniu 4 razy wolniejsza od JavaScriptowej implementacji skalarnej. Testowane dane były na tyle małe, że funkcja JavaScriptowa wykonała się bardzo szybko, a jej czas wykonania nie przebił sumarycznego czasu serializacji danych. Przyśpieszenie algorytmiczne wynosiło natomiast 3.5, co oznacza, że algorytm wykorzystujący operacje SIMD był kilka razy szybszy od wywołania funkcji JavaScriptowej.

Największym zaskoczeniem jednak była implementacja SIMD dla algorytmu obliczającego CRC-64. Przyśpieszenia całkowite i algorytmiczne wynosiły odpowiednio 608.5 i 641.7. Oznacza to, że implementacja ta jest prawie 20 razy szybsza niż skalarna implementacja w języku Rust, która była ponad 30 razy szybsza od implementacji JavaScriptowej. Wartości

tych przyśpieszeń były dla mnie tak zaskakujące, że podczas przeprowadzania tego badania wielokrotnie weryfikowałem, czy funkcja SIMD rzeczywiście zwraca poprawne wyniki. Wszystko jednak wskazywało na poprawność wyników funkcji i pomiarów.

6.5 Wydajność WebAssembly i JavaScriptu

W zdecydowanej większości testów można zauważyc, że wydajność WebAssembly jest stała niezależnie od ilości powtórzeń. Jest to bardzo duży atut kodu skompilowanego Ahead-Of-Time, który pozwala na stabilne i przewidywalne zachowanie aplikacji.

Bez rozbicia pomiarów na czas algorytmu oraz czasy serializacji, duża ilość przyśpieszeń wskazywałyby na przewagę JavaScriptu nad WebAssembly. Jednakże, dzięki takiemu rozbiciu jestem w stanie stwierdzić, że odpowiedź na pytanie „Czy Rust jest lepszym wyborem dla aplikacji przeglądarkowych niż JavaScript?” nie jest jednoznaczna.

Dla aplikacji hybrydowych, odwołujących się do WebAssembly jedynie w krytycznych dla wydajności momentach warto pamiętać, by dane były wcześniej przygotowane w taki sposób, by serializacja nie była wymagana, a same dane mogły być bezpośrednio zamieszczone w pamięci modułu WebAssembly. Zapewni to najwyższą wydajność aplikacji i ograniczy wpływ serializacji danych na czas wykonania kodu.

Jeżeli jednak zastanawiamy się, czy napisać całą logikę po stronie JavaScriptu, czy po stronie WebAssembly, a aplikacja wymaga wydajności i stabilności, WebAssembly może okazać się znacznie lepszym wyborem. Większe przyśpieszenie algorytmiczne i możliwość wektoryzacji algorytmów dzięki wsparciu dla operacji SIMD obiecują szybsze działanie aplikacji. Podczas modyfikacji dokumentu należy jednak być ostrożnym i dbać o to, aby nie wysyłać często dużych ilości danych między kontekstami. Alternatywą jest skorzystanie z frameworka, który zoptymalizuje te operacje za nas.

Moje badania potwierdziły wyniki Jacoba Nilssona i Andreasa Trattnera i rzuciły dodatkowe światło na wydajność funkcji SIMD w WebAssembly. Zgadzam się więc z wnioskami autorów dotyczącymi tego, że może poprawić czas wykonania kodu w aplikacjach webowych pod warunkiem, że koszt serializacji danych będzie niski lub zerowy w porównaniu z czasem wykonania algorytmu.

Przeprowadzone testy dotyczące API DOM były jednak niewystarczające, aby potwierdzić wyniki uzyskane przez Stefana Krause'a w jego benchmarkach. W przyszłości należałoby skupić się na bardziej złożonych operacjach na elementach DOM i spróbować różnych

strategii optymalizacji zapytań do API DOM. Moje badania pokazały jednak, że komunikacja między WebAssembly a JavaScriptem nie jest bezkosztowna i w sytuacjach, w których zależy nam na wydajności, należy mieć świadomość, kiedy i w jaki sposób taka komunikacja zachodzi oraz jakie niesie ze sobą implikacje wydajnościowe.

6.6 Perspektywy dalszych badań

Warto przeprowadzić dokładniejsze badania nad strategiami zapytań do API DOM z poziomu WebAssembly aby zrozumieć, skąd dokładnie wynika niska wydajność tych funkcji. Można również przetestować wydajność bibliotek takich jak *egui* [48] jako alternatywę dla zapytań do API DOM. Biblioteka ta, zamiast modyfikować dokument, rysuje wszystkie elementy bezpośrednio do dwuwymiarowego kontekstu elementu `<canvas>`. Oznacza to, że musi ona przesyłać jedynie tablicę 8-bitowych wartości, którą można wstrzyknąć bezpośrednio do pamięci modułu WASM bez konieczności serializacji.

Aby lepiej zrozumieć różnice pomiędzy wydajnością JavaScriptu a WebAssembly, w przyszłości można by było przeprowadzić identyczne badania na więcej niż jednej przeglądarce internetowej. Przeglądarki Firefox i Safari mają swoje własne implementacje WebAssembly, które mogą różnić wydajnością się od implementacji w Chrome.

Dodatkowo środowiska takie jak Node.js czy Wasmer pozwalają na uruchamianie kodu WebAssembly poza przeglądarką. Zbadanie wydajności obu technologii w środowiskach serwerowych rzuciłoby więcej światła na temat różnic w wydajności pomiędzy JavaScriptem a WebAssembly.

Aby lepiej zrozumieć wpływ optymalizacji nakładanych przez kompilator JIT, można by było obliczyć średnią kroczącą czasów wykonania funkcji w zależności od ilości powtórzeń. Pozwoliłoby to zbadać jak agresywne są optymizacje nakładane przez kompilator JIT w czasie i jak wpływają one na wydajność funkcji JavaScriptowych.

W tej pracy skupiłem się na porównaniu wydajności funkcji JavaScriptowych wykonywanych wielokrotnie w celu jak największej optymalizacji poprzez kompilator JIT. W przyszłości warto również przebadać jaki wpływ na wydajność kodu JavaScriptowego będzie miała deoptimizacja zoptymalizowanych funkcji i jak będzie zachowywała się funkcja, która co jakiś czas jest wywoływana z argumentami innego typu.

Bibliografia

- [1] OpenJS Foundation. *Electron*. URL: <https://www.electronjs.org/>. stan z 30.05.2024r.
- [2] Kontrybutorzy Tauri. *Tauri*. URL: <https://tauri.app/>. stan z 30.05.2024r.
- [3] an OutSystems company Ionic. *Ionic*. URL: <https://ionic.io/>. stan z 6.06.2024r.
- [4] Apache Software Foundation. *Apache Cordova*. URL: <https://cordova.apache.org/>. stan z 6.06.2024r.
- [5] World Wide Web Consortium. *WebAssembly | Working Groups | Discover W3C groups | W3C*. URL: <https://www.w3.org/groups/wg/wasm/>. stan z 29.05.2024r.
- [6] Can I use. *WebAssembly | Can I use... Support tables for HTML5, CSS3, etc.* URL: <https://caniuse.com/wasm>. stan z 29.05.2024r.
- [7] Hugo Heyman i Love Brandefelt. *A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++*. 2020.
- [8] Magnus Medin. *Performance comparison between C and Rust compiled to WebAssembly*. 2021.
- [9] William Bugden i Ayman Alahmar. „Rust: The programming language for safety and performance”. W: *arXiv preprint arXiv:2206.05503* (2022).
- [10] Ignas Plauska, Agnus Liutkevičius i Audronė Janavičiū tė. „Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller”. W: *Electronics* 12.1 (2022), s. 143.
- [11] Ryan Paul. *Firefox to get massive JavaScript performance boost*. URL: <https://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/>. stan z 3.06.2024r.
- [12] ECMA. *The String Type - ECMAScript® 2023 Language Specification*. URL: <https://262.ecma-international.org/#sec-ecmascript-language-types-string-type>. stan z 3.06.2024r.

- [13] ECMA. *The Number Type - ECMAScript® 2023 Language Specification*. URL: <https://262.ecma-international.org/#sec-ecmascript-language-types-number-type>. stan z 1.06.2024r.
- [14] ECMA. *NumberBitwiseOp - ECMAScript® 2023 Language Specification*. URL: <https://262.ecma-international.org/#sec-numberbitwiseop>. stan z 3.06.2024r.
- [15] ECMA. *The BigInt Type - ECMAScript® 2023 Language Specification*. URL: <https://262.ecma-international.org/#sec-ecmascript-language-types-bigint-type>. stan z 1.06.2024r.
- [16] Mozilla Developer Network. *BigInt - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt. stan z 3.06.2024r.
- [17] Mozilla Developer Network. *JavaScript data types and data structures - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values. stan z 4.06.2024r.
- [18] Mozilla. *MIR optimizations from a thousand feet — Firefox Source Docs documentation*. URL: <https://firefox-source-docs.mozilla.org/js/MIR-optimizations/index.html>. stan z 5.06.2024r.
- [19] Mozilla. *SpiderMonkey — Firefox Source Docs documentation*. URL: <https://firefox-source-docs.mozilla.org/js/index.html#warpmonkey>. stan z 5.06.2024r.
- [20] Mozilla. *SpiderMonkey — Firefox Source Docs documentation*. URL: <https://firefox-source-docs.mozilla.org/js/index.html#js-value-and-jsobject>. stan z 6.06.2024r.
- [21] Mozilla. *Garbage Collection - mozilla-spidermonkey/spidermonkey-embedding-examples*. URL: <https://github.com.mozilla-spidermonkey/spidermonkey-embedding-examples/blob/00daf235174fd2a25615bb37c64fd046a7b9705c/docs/Garbage%20Collection.md#design-overview>. stan z 6.06.2024r.
- [22] Mozilla. *SpiderMonkey garbage collector — Firefox Source Docs documentation*. URL: <https://firefox-source-docs.mozilla.org/js/gc.html>. stan z 6.06.2024r.

- [23] Rust. *Numeric types - The Rust Reference*. URL: <https://doc.rust-lang.org/reference/types/numeric.html>. stan z 7.06.2024r.
- [24] Rust. *Textual types - The Rust Reference*. URL: <https://doc.rust-lang.org/reference/types/textual.html>. stan z 7.06.2024r.
- [25] Rust. *Common Collections - The Rust Programming Language*. URL: <https://doc.rust-lang.org/book/ch08-00-common-collections.html>. stan z 8.06.2024r.
- [26] Rust. *rust/library/alloc/src/string.rs · rust-lang/rust*. URL: <https://github.com/rust-lang/rust/blob/63491e10125d6bac9da9b80f4969c18afa28bcc1/library/alloc/src/string.rs#L365-L367>. stan z 14.06.2024r.
- [27] Rust. *Type layout - The Rust Reference*. URL: <https://doc.rust-lang.org/reference/type-layout.html>. stan z 12.06.2024r.
- [28] Mozilla Developer Network. *Understanding WebAssembly text format - WebAssembly | MDN*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format. stan z 13.06.2024r.
- [29] Alex Crichton. *wasm-bindgen*. URL: <https://github.com/rustwasm/wasm-bindgen>. stan z 13.06.2024r.
- [30] Mozilla Developer Network. *Compiling from Rust to WebAssembly - WebAssembly | MDN*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_Wasm#building_the_package. stan z 13.06.2024r.
- [31] HTTP Archive. *HTTP Archive: Page Weight*. URL: https://httparchive.org/reports/page-weight?start=2024_04_01&end=latest&view=list. stan z 13.06.2024r.
- [32] Rust. *Shrinking .wasm Size - Rust and WebAssembly*. URL: <https://rustwasm.github.io/docs/book/reference/code-size.html>. stan z 13.06.2024r.
- [33] Społeczność WebAssembly. *Binaryen Optimizations*. URL: <https://github.com/WebAssembly/binaryen?tab=readme-ov-file#binaryen-optimizations>. stan z 13.06.2024r.
- [34] Rust. *Use the wasm-opt Tool*. URL: <https://rustwasm.github.io/docs/book/reference/code-size.html#use-the-wasm-opt-tool>. stan z 13.06.2024r.

- [35] Rust. *Why Care About Code Size?* URL: <https://rustwasm.github.io/docs/book/reference/code-size.html#why-care-about-code-size>. stan z 13.06.2024r.
- [36] Lin Clark. *Making WebAssembly even faster: Firefox's new streaming and tiering compiler - Mozilla Hacks - the Web developer blog*. URL: <https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/>. stan z 13.06.2024r.
- [37] Mozilla. *WebAssembly — SpiderMonkey — Firefox Source Docs documentation*. URL: <https://firefox-source-docs.mozilla.org/js/index.html#id1>. stan z 14.06.2024r.
- [38] Dan Gohman. *Introducing SIMD.js - Mozilla Hacks - the Web developer blog*. URL: <https://hacks.mozilla.org/2014/10/introducing-simd-js/>. stan z 14.06.2024r.
- [39] TC39. *SIMD numeric type for EcmaScript*. URL: https://github.com/tc39/ecmascript_simd. stan z 14.06.2024r.
- [40] Can I use. *WebAssembly SIMD | Can I use... Support tables for HTML5, CSS3, etc.* URL: <https://caniuse.com/wasm-simd>. stan z 14.06.2024r.
- [41] Max Van Hasselt i in. „Comparing the energy efficiency of webassembly and javascript in web applications on android mobile devices”. W: *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 2022, s. 140–149.
- [42] Jacob Nilsson i Andreas Trattner. „Analyzing front-end performance using Webassembly”. Prac. mag. Lund University, 2022.
- [43] Stefan Krause. *A comparison of the performance of a few popular javascript frameworks*. URL: https://krausest.github.io/js-framework-benchmark/2024/table_chrome_125.0.6422.60.html. stan z 29.05.2024r.
- [44] Inc. Wasmer. *wasmerio/wasmer: The leading Wasm Runtime supporting WASIX, WASI and Emscripten*. URL: <https://github.com/wasmerio/wasmer>. stan z 18.06.2024r.

- [45] ECMA. *Time Values and Time Range - ECMAScript® 2023 Language Specification*. URL: <https://tc39.es/ecma262/multipage/numbers-and-dates.html#sec-time-values-and-time-range>. stan z 16.06.2024r.
- [46] Mozilla Developer Network. *High precision timing - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance_API/High_precision_timing. stan z 16.06.2024r.
- [47] W3C. *High Resolution Time - W3C*. URL: <https://w3c.github.io/hr-time/#ref-for-index-term-implementation-defined-3>. stan z 16.06.2024r.
- [48] Emil Ernerfeldt. *emilk/egui: egui: an easy-to-use immediate mode GUI in Rust that runs on both web and native*. URL: <https://github.com/emilk/egui>. stan z 19.06.2024r.
- [49] Greg Johnston. *Leptos*. URL: <https://leptos.dev/>. stan z 29.05.2024r.
- [50] Kontrybutorzy Sycamore. *Sycamore*. URL: <https://sycamore-rs.netlify.app/>. stan z 29.05.2024r.
- [51] Rust. *Boolean type - The Rust Reference*. URL: <https://doc.rust-lang.org/reference/types/boolean.html>. stan z 7.06.2024r.

Spis tabel

Tabela 2.1	Typy reprezentujące liczby całkowite	15
Tabela 2.2	Typowane tablice w JavaScript	29
Tabela 4.1	Opis poziomów optymalizacji kompilatora Rusta	38
Tabela 4.2	Opis funkcji dla testu „Re-enkodowanie stringów”	41
Tabela 4.3	Opis funkcji dla testu „Enkodowanie stringów do Base64”	42
Tabela 4.4	Opis funkcji dla testu „k-ta liczba Fibonacciego (rekurencyjnie)” . .	42
Tabela 4.5	Opis funkcji dla testu „Mnożenie macierzy 4x4”	43
Tabela 4.6	Opis funkcji dla testu „CRC-32”	44
Tabela 4.7	Opis funkcji dla testu „CRC-64”	45
Tabela 4.8	Opis funkcji dla testu „API DOM — Tworzenie elementów”	46
Tabela 4.9	Opis funkcji dla testu „API DOM — Aktualizacja co drugiego elementu”	46
Tabela 4.10	Opis funkcji dla testu „API DOM — Usuwanie elementów”	47
Tabela 5.1	Wyniki dla testu „Re-enkodowanie stringów”	50
Tabela 5.2	Wyniki dla testu „Enkodowanie do Base64”	55
Tabela 5.3	Wyniki dla testu „k-ta liczba Fibonacciego (rekurencyjnie)”	57
Tabela 5.4	Wyniki dla testu „Mnożenie macierzy 4x4”	59
Tabela 5.5	Wyniki dla testu „CRC-32”	61
Tabela 5.6	Wyniki dla testu „CRC-64”	65
Tabela 5.7	Wyniki dla testu „API DOM — Tworzenie elementów”	67
Tabela 5.8	Wyniki dla testu „API DOM — Aktualizacja co drugiego elementu”	71
Tabela 5.9	Wyniki dla testu „API DOM — Usuwanie elementów”	74

Spis rysunków

Rysunek 2.1 Proces optmizacji i deoptymizacji skryptów JavaScript	8
Rysunek 2.2 Przykład wspólnej ukrytej klasy dla dwóch obiektów	10
Rysunek 2.3 Przykład różnych ukrytych klas dla dwóch obiektów	11
Rysunek 2.4 Przykład referencji obiektów w pamięci JavaScript	12
Rysunek 2.5 Przykład „Minor GC” i „Major GC” w narzędziach deweloperskich	13
Rysunek 2.6 Reprezentacja typów String i &str w pamięci	16
Rysunek 2.7 Reprezentacja ArrayBuffer, Uint8Array i Uint16Array w pamięci .	29
Rysunek 2.8 Porównanie operacji SIMD z operacjami skalarnymi	31
Rysunek 3.1 Porównanie wydajności najpopularniejszych frameworków w językach JavaScript i Rust [43]	34
Rysunek 5.1 Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 1KB)”	49
Rysunek 5.2 Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 512KB)”	49
Rysunek 5.3 Wykres średniego czasu wykonania testu „Reenkodowanie stringów (String 1MB)”	49
Rysunek 5.4 Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 1KB)”	51
Rysunek 5.5 Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 512KB)”	52
Rysunek 5.6 Wykres średniego czasu wykonania testu „Enkodowanie do Base64 (String 1MB)”	54
Rysunek 5.7 Wykres średniego czasu wykonania testu „20 liczba Fibonacciego (rekurencyjnie)”	56
Rysunek 5.8 Wykres średniego czasu wykonania testu „30 liczba Fibonacciego (rekurencyjnie)”	56
Rysunek 5.9 Wykres średniego czasu wykonania testu „40 liczba Fibonacciego (rekurencyjnie)”	57
Rysunek 5.10 Wykres średniego czasu wykonania testu „Mnożenie macierzy 4x4”	59

Rysunek 5.11 Wykres średniego czasu wykonania testu „CRC-32 (Plik 1KB)”	61
Rysunek 5.12 Wykres średniego czasu wykonania testu „CRC-32 (Plik 512KB)”	62
Rysunek 5.13 Wykres średniego czasu wykonania testu „CRC-32 (Plik 1MB)”	62
Rysunek 5.14 Wykres średniego czasu wykonania testu „CRC64 (Plik 1KB)”	64
Rysunek 5.15 Wykres średniego czasu wykonania testu „CRC64 (Plik 512KB)”	64
Rysunek 5.16 Wykres średniego czasu wykonania testu „CRC64 (Plik 1MB)”	66
Rysunek 5.17 Wykres średniego czasu wykonania testu „API DOM — Tworzenie 100 elementów”	68
Rysunek 5.18 Wykres średniego czasu wykonania testu „API DOM — Tworzenie 1000 elementów”	68
Rysunek 5.19 Wykres średniego czasu wykonania testu „API DOM — Tworzenie 10000 elementów”	69
Rysunek 5.20 Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego ze 100 elementów”	70
Rysunek 5.21 Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego z 1000 elementów”	70
Rysunek 5.22 Wykres średniego czasu wykonania testu „API DOM — Aktualizacja co drugiego z 10000 elementów”	71
Rysunek 5.23 Wykres średniego czasu wykonania testu „API DOM — Usuwanie 100 elementów”	72
Rysunek 5.24 Wykres średniego czasu wykonania testu „API DOM — Usuwanie 1000 elementów”	73
Rysunek 5.25 Wykres średniego czasu wykonania testu „API DOM — Usuwanie 10000 elementów”	73

Spis listingów

2.1	Przypisanie różnych typów do zmiennej „x”	4
2.2	Przykład modyfikowania zmiennej prymitywnej oraz obiektu	5
2.3	Przykład kodu ulegającemu specjalizacji typów i deoptimizacji	9
2.4	Przykład dwóch obiektów z identycznymi właściwościami	10
2.5	Przykład dynamicznego dodania właściwości do obiektu	10
2.6	Deklaracja typu String w języku Rust [26]	17
2.7	Przykład struktury w języku Rust	18
2.8	Przykłady wyliczeń w Rust	19
2.9	Przykład użycia klauzuli „match”	20
2.10	Niebezpieczna funkcja w języku JavaScript	21
2.11	Bezpieczne obsłużenie funkcji w języku Rust	22
2.12	Zakres zmiennej „s”	23
2.13	Przykład przekazania własności zmiennej „s” z funkcji „foo” do „bar” . . .	24
2.14	Przykład pożyczania zmiennej „s” z funkcji „foo” do „bar”	25
2.15	Przykład kodu eksportującego funkcję do WASM	27
2.16	Inicjalizacja modułu WebAssembly z pliku binarnego	28
2.17	Przekazanie stringa do modułu WebAssembly	30