



Università degli Studi di Salerno
Dipartimento di Informatica

Progetto Reti Geografiche, Struttura, Analisi e Prestazioni

Benchmarking of WebAssembly in and outside the Browser

Professoressa
Delfina Malandrino

Studente
Gerardo Donnarumma

Anno Accademico 2022-2023

Indice

1	Introduzione	1
2	Stato dell'arte	4
2.1	WebAssembly	4
2.1.1	WebAssembly sul Browser	4
2.1.2	Formati WASM e WAT	7
2.1.3	Tipi di dati	7
2.1.4	Gli elementi fondamentali	8
2.2	WasmEdge	8
2.2.1	Casi d'uso	9
2.2.2	Caratteristiche	9
2.3	Wasm3	10
2.3.1	Casi d'uso	10
2.3.2	Caratteristiche	10
2.4	Wasmer	11
2.4.1	Casi d'uso	11
2.4.2	Caratteristiche	11
2.5	Hyperfine	12
2.6	PSPDFKit	12
3	Implementazione	13
3.1	Benchmarking WebAssembly Web	13
3.1.1	Configurazioni	13
3.1.2	Workload	14
3.1.3	Metodologia	15
3.2	Benchmarking Runtime WebAssembly	15
3.2.1	Configurazioni	16
3.2.2	Dataset	16
3.2.3	Workload	17

3.2.4	Strumenti utilizzati	18
3.2.5	Esecuzione dei benchmark	18
4	Risultati	22
4.1	Risultati Benchmarking WebAssembly su Browser	22
4.2	Risultati Benchmarking dei Runtime WebAssembly	24
4.2.1	Fibonacci	24
4.2.2	CoreMark 1.0	37
5	Conclusioni	40
5.0.1	Obiettivi raggiunti	40
5.0.2	Direzioni future	41

Capitolo 1

Introduzione

L'unico linguaggio che i moderni Browser possono eseguire è **JavaScript**. Il linguaggio JavaScript è interpretato: legge il codice e traduce al volo le sue istruzioni in codice macchina. Lo svantaggio dei linguaggi interpretati, a differenza di quelli compilati, è che l'interprete deve convertire il codice in codice macchina ogni volta che viene eseguito.

Per rendere più veloce l'esecuzione del codice, i produttori dei Browser hanno introdotto il concetto di compilazione **JIT** (*Just-In-Time*), in cui l'engine JavaScript monitora il codice mentre lo esegue; se una sezione di codice viene riutilizzata più volte, l'engine tenta di compilare quella sezione in codice macchina, in modo da utilizzare direttamente metodi di basso livello del sistema, molto più veloci. Ma l'engine JavaScript può compilare il codice JavaScript in codice macchina solo dopo che il codice stesso è stato monitorato più volte, essendo la natura del linguaggio dinamica (una variabile può assumere valori che rappresentano tipi di dati differenti durante il ciclo di vita del programma).

I produttori dei moderni Browser hanno trovato un compromesso: **WebAssembly** (abbreviato Wasm), un linguaggio assembly-like di basso livello che può essere eseguito a velocità quasi native da tutti i più recenti Browser desktop e da molti Browser mobile. Di seguito sono elencate le caratteristiche principali di WebAssembly:

- **efficiente e veloce**: è progettato per essere codificato in un formato binario efficiente in termini di dimensioni e tempo di caricamento;
- **sicuro**: WebAssembly viene eseguito in una sandbox sicura per la

memoria, che può anche essere implementata all'interno di macchine virtuali JavaScript esistenti;

- **possibilità di utilizzare linguaggi diversi da JavaScript nel Browser:** è stato progettato proprio per essere un output di un compilatore, quindi gli sviluppatori che desiderano utilizzare un determinato linguaggio per lo sviluppo Web, potranno farlo senza dover trasporre il loro codice in JavaScript.

Potremmo pensare che WebAssembly sia un tipo di linguaggio Assembly, ma anche se il nome contiene tale parola non è proprio così. Quando viene eseguito un programma sul Web, non si sa su quale architettura di destinazione sarà eseguito il codice. Il codice WebAssembly è diverso dai linguaggi Assembly perché è un linguaggio macchina per una macchina concettuale, non una macchina reale, fisica. Per questo motivo, le istruzioni WebAssembly sono talvolta chiamate istruzioni virtuali. Hanno una mappatura molto più diretta al codice macchina rispetto al codice sorgente JavaScript. Rappresentano una sorta di intersezione di ciò che può essere fatto in modo efficiente su hardware comune e popolare. Ma non sono mappature dirette al particolare codice macchina di un hardware specifico. Quindi WebAssembly definisce un nuovo formato binario per una rappresentazione bytecode vicino alla macchina, che può essere facilmente tradotta in istruzioni assembly eseguibili dal motore di esecuzione.

Il comportamento di esecuzione del codice WebAssembly è definito in termini di una macchina astratta che modella lo stato del programma. Questa macchina astratta è spesso citata come **VM** (Virtual Machine) ed include uno stack, che registra i valori degli operandi e i costrutti di controllo e un archivio astratto contenente lo stato globale. Per ogni istruzione esiste una regola che specifica l'effetto della sua esecuzione sullo stato del programma. La macchina virtuale è conforme alle specifiche WebAssembly descritte nei documenti ufficiali. Un runtime WebAssembly facilita tutte le interazioni necessarie tra lo stack (una macchina virtuale basata su stack) che esegue il formato binario WebAssembly e l'ambiente in cui esiste quella VM. In poche parole i runtime WebAssembly forniscono un modo per chiamare la VM di WebAssembly al di fuori del Browser, su una qualsiasi macchina e su un qualsiasi dispositivo.

Le implementazioni dei runtime variano, ma in generale un runtime consente almeno di istanziare la VM, passare il formato *WASM* o *WAT* alla VM, invocare una funzione del modulo *WASM* passando dei parametri, fornire un meccanismo di restituzione al chiamante del risultato dell'esecuzione e

infine, terminare l'istanza della VM.

Gli obiettivi di questo progetto sono:

- sperimentare la tecnologia WebAssembly sul Web e al di fuori;
- testare le performance di WebAssembly sui vari Browser;
- realizzare dei micro-benchmark per valutare le performance dei runtime WebAssembly più noti.

Capitolo 2

Stato dell'arte

In questo capitolo analizzeremo nel dettaglio WebAssembly per capire come funziona, descriveremo i runtime WebAssembly più noti e mostreremo alcuni benchmark noti per testare WebAssembly all'interno del Browser e al di fuori.

2.1 WebAssembly

Abbiamo definito WebAssembly come un linguaggio assembly-like di basso livello che può essere eseguito a velocità quasi native da tutti i più recenti Browser desktop e mobile. In questa sezione vediamo più nello specifico come funziona WebAssembly e quali sono gli elementi di cui è composto.

2.1.1 WebAssembly sul Browser

Quando un utente apre un sito Web, il Browser scarica i file *JavaScript*, *HTML* e *CSS*. Ogni Browser ha il proprio JavaScript Runtime, dove esegue il codice JavaScript. Anche il **codice WebAssembly** viene scaricato dal Browser e viene eseguito nel motore JavaScript. Proprio per questo motivo tutti i Browser possono eseguirlo.

Per trasformare, ad esempio un codice *C* in WebAssembly, è necessaria un'infrastruttura di compilazione. *Emscripten* è una toolchain di compilazione per WebAssembly, che utilizza *LLVM* (una raccolta di tecnologie di compilazione e toolchain modulari e riutilizzabili ed è nato come progetto di ricerca presso l'Università dell'Illinois), con particolare attenzione alla velocità, alle dimensioni e alla piattaforma Web. Indipendentemente dalla

toolchain utilizzata, il risultato finale è un file con estensione ”*.wasm*” o anche detto formato **WASM**. Nell’immagine 2.1 è riportato un esempio del contenuto di un file *WASM*.

```
00 61 73 6D 0D 00 00 00 01 86 80 80 80 00 01 60
01 7F 01 7F 03 82 80 80 80 00 01 00 04 84 80 80
80 00 01 70 00 00 05 83 80 80 80 00 01 00 01 06
81 80 80 80 00 00 07 96 80 80 80 00 02 06 6D 65
6D 6F 72 79 02 00 09 5F 5A 35 61 64 64 34 32 69
00 00 0A 8D 80 80 80 00 01 87 80 80 80 00 00 20
00 41 2A 6A 0B
```

Figura 2.1: Esempio contenuto di un file .wasm.

Questo è un modulo WebAssembly nella sua rappresentazione in notazione esadecimale. Un altro modo per visualizzare o scrivere il codice WebAssembly è attraverso il formato detto **WAT**, proprio come mostrato nell’immagine 2.2.

```
1 (module
2   (type $type0 (func (param i32)
3     (result i32)))
4   (table 0 anyfunc)
5   (memory 1)
6   (export "memory" memory)
7   (export "_Z7squarer_i" $func0)
8   (func $func0 (param $var0 i32)
9     (result i32)
10    get_local $var0
11    get_local $var0
12    i32.mul
13  )
14 )
```

Figura 2.2: Esempio contenuto di un file .wasm.

Infine è possibile istanziare il modulo *WASM* attraverso il linguaggio JavaScript, che ogni Browser eseguirà sul proprio motore JavaScript. Un esempio di un modulo WebAssembly che effettua la radice quadrata di un numero, è mostrato nelle immagini 2.3 e 2.4.


```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>WASM Squarer Function</title>
5      <meta name="viewport" content="width=device-width, initial-scale=1" />
6  </head>
7  <body>
8
9  <h1>WASM Squarer Function</h1>
10
11 <script src="squarer.js"></script>
12 </body>
13 </html>

```

Figura 2.3: Esempio file HTML per caricare uno script JavaScript contenente il codice per istanziare un modulo WASM.

```

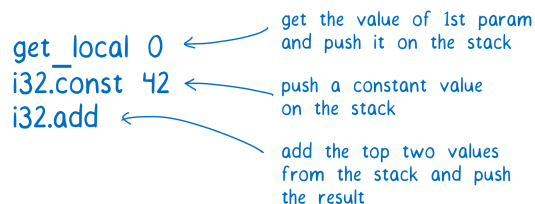
1  let importObject = {
2      imports: {
3          imported_func: function(arg) {
4              console.log(arg);
5          }
6      }
7  }
8
9  async function loadwebAssembly() {
10      let response = await fetch('squarer.wasm');
11      let arrayBuffer = await response.arrayBuffer();
12      let wasmModule = await WebAssembly.instantiate(arrayBuffer, importObject);
13      wasmModule.instance.exports.Z7squarer;
14  }
15
16  loadwebAssembly();

```

Figura 2.4: Script JavaScript contenente il codice per istanziare un modulo WASM.

2.1.2 Formati WASM e WAT

Descriviamo come funziona WebAssembly al suo interno. WebAssembly è anche detto "**Stack Machine**" perché tutti i valori richiesti da un'operazione, vengono accodati nello stack prima che l'operazione venga eseguita. Di seguito è riportato il codice in formato testuale (*WAT*) di una funzione "*add*" che somma due numeri interi.



```
get_local 0  
i32.const 42  
i32.add
```

Annotations:

- get the value of 1st param and push it on the stack (points to `get_local 0`)
- push a constant value on the stack (points to `i32.const 42`)
- add the top two values from the stack and push the result (points to `i32.add`)

Figura 2.5: Funzione *add* in formato *WAT*.

All'operazione di "*add*" servono due valori e WebAssembly prenderà questi valori dalla cima dello stack. Questo significa che l'istruzione "*add*" può essere breve (un singolo *byte*), perché non ha bisogno di specificare i registri di origine o di destinazione. Ciò riduce le dimensioni del file *WASM*, ciò significa che il download richiede meno tempo. Però quando il Browser traduce WebAssembly nel codice macchina su cui è in esecuzione, utilizza i registri. Poiché il codice WebAssembly non specifica i registri, esso offre al Browser una maggiore flessibilità per utilizzare la migliore allocazione dei registri per quella determinata macchina su cui viene eseguito il Browser stesso.

2.1.3 Tipi di dati

Altro aspetto importanti di WebAssembly è la gestione dei tipi. Il WebAssembly ha solo **quattro tipi primitivi** e sono tutti i **numeri interi e float** (*i32*, *i64*, *f32* e *f64*). Il passaggio di tipi di dati complessi tra JavaScript e WebAssembly non è semplice ed immediato. Se volessimo passare una stringa, dovremmo codificarla in un array di numeri e passargli poi un puntatore. Questo perché WebAssembly ha solo accesso alla sua memoria lineare e non ha accesso diretto alle variabili JavaScript.

2.1.4 Gli elementi fondamentali

Vediamo nello specifico i componenti fondamentali di cui è composto un modulo WebAssembly:

- **Import**: le importazioni sono gli elementi a cui è possibile accedere all'interno di un modulo (*Function*, *Global*, *Linear Memory*, *Table*);
- **Function**: contiene le definizioni delle firme di ciascuna funzione definita nel modulo e il corpo di ciascuna delle funzioni;
- **Global**: contiene la definizione delle variabili globali del modulo;
- **Linear Memory**: contiene la definizione della memoria lineare, a partire da una dimensione iniziale e una dimensione massima (facoltativa);
- **Table**: è una memoria lineare in cui gli elementi sono valori opachi di un particolare tipo di elemento della tabella. Nell'*MVP* (*Minimum Viable Product*), il suo scopo principale è quello di implementare chiamate di funzione indirette in *C/C++*;
- **Export**: le esportazioni sono elementi a cui si può accedere dalle *API* di consumo (cioè gli elementi WebAssembly invocati da una funzione JavaScript).

2.2 WasmEdge

È così che definiscono i creatori il runtime **WasmEdge**:

WasmEdge è un runtime WebAssembly leggero, ad alte prestazioni ed estensibile. È la Wasm VM più veloce oggi. WasmEdge è un progetto sandbox ufficiale ospitato dal CNCF. I suoi casi d'uso includono moderne architetture di applicazioni Web (applicazioni Isomorphic e Jamstack), microservizi sull'edge cloud, API SaaS serverless, Smart Contract e dispositivi AI.

WasmEdge è un progetto ospitato ufficialmente dalla *Cloud Native Computing Foundation* (CNCF) e il suo obiettivo è diventare un'implementazione *open source* di riferimento della tecnologia WebAssembly.

2.2.1 Casi d'uso

WasmEdge è ampiamente utilizzato in edge computing, automotive, Jamstack, serverless, SaaS, service mesh e persino applicazioni blockchain. Di seguito riportiamo alcuni casi d'uso più comuni:

- *Edge Computing*;
- *Automotive*;
- *Jamstack* (una moderna architettura di sviluppo Web);
- *Serverless Function-As-A-Service* in *Public Clouds* (*AWS*, *Vercel*, *Netlify*, *Second State*);
- *Service Mesh*;
- Applicazioni *Blockchain*;
- *Smart Device* (*Raspberry Pi*, dispositivi *Android* ed altri).

2.2.2 Caratteristiche

- Supporta un'ampia gamma di sistemi operativi e piattaforme hardware: *Linux*, *MacOS*, *Windows*, *Android*, *seL4*, *OpenWrt*, *OpenHarmony*, *Raspberry Pi*.
- Implementa alcune proposte **WASI** (*WebAssembly System Interface*, un'interfaccia di sistema modulare di API standardizzate per WebAssembly).
- Fornisce entrambe le modalità **JIT** (*Just in Time*) e **AOT** (*Ahead Of Time* basate su *LLVM*). La modalità *AOT* ha generalmente tempi di caricamento iniziale più rapidi e prestazioni più coerenti rispetto alla modalità *JIT*.
- Gli utenti del runtime possono scrivere i propri *plug-in* per estendere le funzionalità (ad esempio, *plug-in* per la connessione ad un database).
- Sono stati realizzati *SDK* nel linguaggio *C*, *GoLang*, *Rust*, *Python* e per la piattaforma *Node JS*.

2.3 Wasm3

I creatori di Wasm3 definiscono il runtime in questo modo:

Un veloce interprete WebAssembly e il runtime WASM più universale.

Wasm3 nasce come un progetto di ricerca che vuole portare il WebAssembly al di fuori dei Browser. A differenza del runtime visto in precedenza, *Wasm3* non usa una compilazione *JIT*, ma utilizza l'approccio dell'**interprete**. La motivazione che ha portato i creatori del runtime ad utilizzare questo approccio, è dovuto al fatto che in molte situazioni la velocità non è la preoccupazione principale, l'utilizzo della **memoria** e la **latenza di avvio** sono altrettanto importanti e possono essere migliorate. Infine, su alcune piattaforme (ad esempio *iOS* e *WebAssembly* stesso) non è possibile generare eseguibili in fase di esecuzione, quindi *JIT* non è disponibile.

2.3.1 Casi d'uso

Ecco alcuni casi d'uso del runtime *Wasm3*:

- *Edge Computing*;
- *Scripting*;
- *Plugin systems*;
- *IoT rules*;
- *Smart Contracts*.

2.3.2 Caratteristiche

- Supera la suite di test delle specifiche WebAssembly e supporta molte interfacce **WASI**.
- Gira su una vasta gamma di architetture (*x86*, *ARM*, *RISC-V*, *PowerPC*, *MIPS*, *ARC32*, ...)
- Supporta varie piattaforme: *Linux*, *Windows*, *OS X*, *FreeBSD*, *Android*, *iOS*, *OpenWrt*, *Yocto*, *Buildroot* (router, modem, ecc.), *Raspberry Pi*, *Orange Pi* e altri.

- L'utilizzo della **memoria** e la **latenza di avvio** sono migliorate grazie all'approccio dell'interprete. Anche la **portabilità** e la **sicurezza** sono molto più facili da raggiungere e mantenere.
- Può essere utilizzato anche come libreria per: *Python3, Rust, C/C++, GoLang, Zig, Perl, Swift, .Net, Nim, Arduino, PlatformIO, Particle, QuickJS*.

2.4 Wasmer

Wasmer è definito dai fondatori in questo modo:

Wasmer è un runtime WebAssembly veloce e sicuro che consente l'esecuzione di container ultraleggeri ovunque: dal desktop ai dispositivi Cloud , Edge e IoT .

In pratica *Wasmer* è un runtime open source per l'esecuzione di WebAssembly sul Server.

2.4.1 Casi d'uso

Ecco alcuni casi d'uso del runtime Wasm3:

- *Blockchain Infrastructure;*
- *Function-as-a-Service platform;*
- *Portable ML/AI application and model;*

2.4.2 Caratteristiche

- Sicuro per impostazione predefinita.
- Nessun accesso a file, rete o ambiente, a meno che non sia esplicitamente abilitato.
- Supporta **WASI** ed **Emscripten**.
- Incorporabile in più linguaggi di programmazione: *C, C++, Rust, Java, PHP, JavaScript* e tanti altri.
- Conforme alle ultime proposte WebAssembly (*SIMD*, tipi di riferimento, *thread*, ...).
- **WAPM** viene fornito come *bundle* insieme a *Wasmer*. *WAPM* è un gestore di pacchetti per moduli WebAssembly.

2.5 Hyperfine

Hyperfine è uno **strumento di benchmarking** a riga di comando, che consente di misurare il tempo di esecuzione di comandi o programmi sulla propria macchina. È progettato per essere **preciso** e **flessibile**, offrendo un'ampia gamma di opzioni per personalizzare le misurazioni e ottenere risultati affidabili.

Hyperfine utilizza la libreria *chrono* di *C++ 11* per misurare il tempo di esecuzione con una precisione molto elevata. Inoltre, include molte funzionalità avanzate, come la media su più esecuzioni, la rimozione di picchi di caricamento del sistema e la stampa di statistiche dettagliate dei risultati.

2.6 PSPDFKit

PSPDFKit è un *kit* per fornire il modo migliore di **visualizzare**, **annotare** e **compilare** moduli nei **documenti PDF** in qualsiasi piattaforma. Alla fine del 2016, è stata rilasciata la prima versione del *PSPDFKit Web SDK*, che si basava su un componente server per il *rendering* dei documenti *PDF*. Successivamente è stata rilasciata una versione aggiornata di *PSPDFKit* per Web, che non richiedesse un componente server e utilizzasse WebAssembly per eseguire il *rendering* dei documenti *PDF* direttamente nel Browser.

Il benchmark PSPDFKit è una semplice applicazione JavaScript che viene eseguita sul Browser. Utilizzando l'*API Web Performance*, viene misurato il tempo di varie azioni *standard* su un documento *PDF*. Infine vengono raggruppati i risultati in due categorie: **tempo di caricamento** (compilazione e inizializzazione) e **tempo di esecuzione**. La separazione dei tempi è stata fatta perché si è notato che i tempi di compilazione sono i maggiori fattori di differenziazione quando vengono usati Browser differenti.

Quello che si sta cercando di realizzare con questo benchmark è avere un **punteggio** utilizzabile per migliorare le prestazioni di *PSPDFKit* per il Web invece di creare un *micro-benchmark*. Pertanto, il benchmark non chiama direttamente il codice WebAssembly, ma testa il *bridge* che viene utilizzato per comunicare con esso, una parte importante di *PSPDFKit* per il Web. Inoltre si vuole un punteggio che possa essere facilmente confrontato su un'ampia varietà di Browser e, per renderlo possibile, si è dovuto rimuovere dall'equazione il rumore relativo alla rete.

Capitolo 3

Implementazione

In questo capitolo descriveremo le metodologie e l'implementazione utilizzate per effettuare i benchmarking di WebAssembly nel Web e dei runtime WebAssembly.

3.1 Benchmarking WebAssembly Web

Nella seguente sezione mostreremo la metodologia, gli strumenti e il workload utilizzati per effettuare il benchmarking della tecnologia WebAssembly sul Web.

3.1.1 Configurazioni

Abbiamo confrontato i seguenti Browser Web:

- **Chrome:** versione *108.0.5359.124*;
- **Safari:** versione *15.4*;
- **Firefox:** versione *108.0.1*.

I vari Browser Web sono stati installati su una macchina **MacBook Air** (*M1, 2020*) con le seguenti configurazioni:

- **Chip:** Apple M1;
- **Numero totale di Core:** 8 (prestazioni 4 ed efficienza 4);
- **RAM:** 16 GB;
- **Sistema Operativo:** *macOS Monterey* versione *12.3*.

3.1.2 Workload

- **Workload:** utilizzo di *PSPDFKit* in ambiente WebAssembly.
- **Input:** documenti *PDF* con varie dimensioni e complessità (i fondatori del benchmark non specificano quanti documenti *PDF* vengono analizzati e la loro dimensione).
- **Descrizione:** il workload consiste nell'utilizzare *PSPDFKit*, un framework per visualizzare e annotare documenti *PDF* in un ambiente WebAssembly. Per ogni Browser, sono stati testati diversi documenti *PDF* con varie dimensioni e complessità, e sono state registrate le metriche delle **prestazioni** per valutare la **velocità di caricamento** e di **rendering dei documenti**.

Nello specifico, nel benchmarking *PSPDFKit* vengono eseguite e analizzate le seguenti operazioni, esattamente nell'ordine in cui sono elencate:

1. **Rendering all pages:** misurazione del tempo di rendering per tutte le pagine utilizzando l'API "*renderPageAsArrayBuffer*" (dati in input una dimensione e un indice di pagina, esegue il *rendering* di una pagina di un documento *PDF* e restituisce il risultato come *ArrayBuffer* di JavaScript).
2. **Search term:** viene utilizzata l'API di ricerca per cercare del testo in modo programmatico.
3. **Export:** viene esportato un documento PDF su file: *InstantJSON* e *XFDF*. I fondatori di *PSPDFKit* hanno sviluppato una specifica *JSON* per annotazioni *PDF* chiamata *InstantJSON*. Quest'ultima è una specifica *JSON* moderna e pulita, che tutti i prodotti *PSPDFKit* utilizzano per importare ed esportare *metadati*.
4. **Create 100 annotations:** viene utilizzata l'API per le annotazioni allo scopo di creare in modo programmatico 100 annotazioni ed esportarle in un documento *PDF*.
5. **Initialization (compilation and instantiation) of the Wasm module:** il processo di inizializzazione consiste in tre passaggi: *download*, *compilazione* e *creazione* di un'istanza (non si tiene traccia del tempo di *download*).

3.1.3 Metodologia

Il benchmark può essere effettuato utilizzando il *client Web PSPDF-Kit*. È possibile anche utilizzare l'applicazione di *PSPDFKit* disponibile sul repository *GitHub* ufficiale. L'applicazione è utilizzabile sia con le licenze cliente che con quelle di prova. Inoltre, si possono contattare i fondatori di *PSPDFKit* per ottenere una chiave di licenza più permissiva, in modo tale da eseguire il benchmark su macchine diverse.

Nel nostro caso avendo realizzando il benchmarking su un'unica macchina, abbiamo utilizzato il *client Web*. Abbiamo eseguito il benchmark **10 volte** per ogni Browser Web utilizzato (*Chrome*, *Safari* e *Firefox*) . Inoltre, abbiamo calcolato il "**90th-percentile**" per effettuare questo *Performance Benchmarking*. Ci siamo fermati a 10 esecuzioni del benchmark per ogni Browser Web perché abbiamo notato che andando avanti con le esecuzioni, si ripetevano gli stessi risultati.

Perché utilizzare il 90° percentile? In generale, se lo *SLA (Service Level Agreement)* specificato ha un *NFR (Nonfunctional Requirement)* al *90° percentile* e si verifica durante il test, allora dimostra che il 90% degli utenti ha un'esperienza che corrisponde agli obiettivi di prestazione, dando fiducia all'utente. Il percentile è spesso considerato un obiettivo di prestazione. Inoltre, a volte il tempo medio di risposta appare estremamente elevato e i singoli set di dati sembrano normali. Anche un paio di picchi nei tempi di risposta, distorcono i numeri del tempo di risposta medio e incidono sul test. In tali scenari, il *90° percentile* (o altri valori percentili) elimina i dati sui picchi insoliti dal risultato. In realtà, la maggior parte delle applicazioni presenta pochissimi picchi elevati. Un grafico delle *performance* che presenta una "coda lunga" non implica molte transazioni lente, ma poche che sono molto più lente della norma. In tal caso, il *90° percentile* è utile perché ignora il 10% delle richieste con il picco.

3.2 Benchmarking Runtime WebAssembly

In questa sezione mostreremo tutto il procedimento di benchmarking dei runtime WebAssembly.

Abbiamo deciso di effettuare alcuni benchmarking della funzione di *Fibonacci* e il *CoreMark Benchmark*. Nelle sezioni a seguire saranno descritti nel dettaglio i vari benchmarking.

3.2.1 Configurazioni

Hardware utilizzato:

- *MacBook Air* (anno 2020);
- *Chip Apple M1*;
- *8 core* (prestazioni 4 ed efficienza 4);
- *16 GB RAM*;
- Sistema Operativo *macOS Monterey* versione 12.3.

Runtime WebAssembly utilizzati:

- **WasmEdge** versione 0.11.2;
- **Wasm3** versione 0.5.0;
- **Wasmer** versione 3.1.0.

Strumenti utilizzati:

- **Hyperfine** versione 1.15.0.

3.2.2 Dataset

Come primo *step* abbiamo dovuto preparare i vari *dataset*. I *dataset* necessari sono i moduli **WASM** da passare in input ai vari runtime.

Dataset Fibonacci

Per testare la funzione di **Fibonacci ricorsiva**, abbiamo utilizzato il codice C 3.1 convertito in WebAssembly utilizzando il **Web Tool WasmFiddle**.

```

1 #include <stdio.h>
2
3 long fibonacci(long i)
4 {
5     if (i < 0) return -1;
6
7     if (i == 0) return 0;
8     else if (i == 1) return 1;
9     else return fibonacci(i-1) + fibonacci(i-2);
10 }
11
12 long recursive_fibonacci(long n)
13 {
14     return fibonacci(n);
15 }

```

Listing 3.1: Fibonacci ricorsivo in C

Dataset CoreMark 1.0

Per effettuare il benchmark **CoreMark 1.0** abbiamo utilizzato il modulo **WASM** realizzato dai fondatori del runtime **Wasm3**. Precisamente abbiamo utilizzato il file "coremark.wasm" reperibile sul repository *GitHub* della libreria *Wasm3*.

3.2.3 Workload

Workload Fibonacci

- **Workload:** calcolo della sequenza di Fibonacci ricorsiva.
- **Input:** 20, 30 e 40.
- **Descrizione:** per ogni input, la funzione di Fibonacci ricorsiva è stata eseguita per calcolare i primi 20, 30 o 40 numeri della sequenza di Fibonacci.

Workload CoreMark 1.0

Come prima cosa descriviamo in breve che cos'è il benchmark **CoreMark 1.0**. Il benchmark è progettato specificamente per testare la funzionalità del core di un processore. L'esecuzione di *CoreMark* produce un **punteggio** di un solo numero che consente agli utenti di fare rapidi confronti tra i processori.

Il workload è costituito da un insieme di funzioni di test, che includono:

- operazioni aritmetiche su numeri interi e floating point;
- copia e gestione delle stringhe;
- manipolazione dei bit e operazioni di confronto;
- funzioni di hashing e compressione.

Tutte queste funzioni vengono eseguite ripetutamente per un determinato numero di volte, e il tempo di esecuzione viene registrato e utilizzato per calcolare un **punteggio di prestazioni**. Il punteggio *CoreMark* è una misura della **velocità di elaborazione** delle operazioni incluse nel workload.

Per maggiori informazioni è possibile visitare il sito *Eembc.org*.

3.2.4 Strumenti utilizzati

Come strumento di benchmarking abbiamo utilizzato il tool a linea di comando **Hyperfine** (spiegato nella sezione 2.5). Grazie ad *Hyperfine* siamo riusciti a produrre dei benchmarking affidabili e riproducibili. Inoltre, abbiamo ottenuto anche dei risultati senza influenza della cache.

3.2.5 Esecuzione dei benchmark

Per eseguire i benchmark con *Hyperfine* abbiamo utilizzato il seguente comando:

```
1 hyperfine --prepare "purge" --export-json file.json --run x
```

Listing 3.2: Comando Hyperfine

Di seguito la spiegazione delle opzioni:

- l'opzione " **-prepare 'purge'** " significa che si sta utilizzando *Hyperfine* per preparare l'ambiente di benchmarking e garantire che i risultati siano il più precisi possibile, eliminando eventuali **effetti negativi** causati da altri processi. Precisamente ad ogni esecuzione del benchmark, con il comando " **purge** ", *Hyperfine* elimina i file dalla *cache* e libera lo spazio occupato. Il sistema potrebbe accedere alla *cache* per prendere i risultati di precedenti esecuzioni, senza dover eseguire nuovamente il codice. Ciò può influire sulla precisione dei risultati del

benchmarking, poiché i tempi di esecuzione risultanti potrebbero non essere rappresentativi delle prestazioni effettive del sistema.

- l'opzione "***export-json***" indica ad *Hyperfine* di salvare i risultati in formato *JSON* (secondo un *JSON* definito da *Hyperfine* stesso) sul file indicato.
- l'opzione "***run x***" indica ad *Hyperfine* di eseguire ogni benchmark *x* volte, dove *x* è un valore da noi definita, rispetto alla metodologia e al workload stabiliti.

Esecuzione Fibonacci

La funzione di Fibonacci è stata eseguita dando in input i numeri: **20**, **30** e **40**. Per ogni input sono state effettuate **100 esecuzioni**, per avere dei risultati quanto più precisi possibile.

```
1 hyperfine
2 --prepare "purge"
3 --export-json fibonacci20_results.json
4 --runs 100
5 'wasmedge --reactor recursive_fib_aot.wasm recursive_fib 20'
6 'wasmedge --reactor recursive_fib.wasm recursive_fib 20'
7 'wasmer run recursive_fib.wasm -i recursive_fib 20'
8 'wasm3 --func recursive_fib recursive_fib.wasm 20'
9 -i
```

Listing 3.3: Comando Hyperfine esecuzione benchmark Fibonacci(20)

È stato mostrato solo il comando per eseguire il file *WASM* con l'invocazione della funzione di Fibonacci con input **20**, ma il comando è stato ripetuto anche per gli input **30** e **40**.

Spiegazione del comando:

- Il comando "***hyperfine***" con con le opzioni (spiegate in precedenza) permette di eseguire i benchmarking con il tool ***Hyperfine***.
- L'opzione "***-i***" di *Hyperfine* (non spiegata in precedenza), ignora i codici di uscita non nulli dei programmi sottoposti al benchmark. È stato utilizzato perché l'esecuzione della funzione di **Fibonacci** sui vari runtime, produce un output che crea problemi ad *Hyperfine*, che lo

considera un errore. Quindi diciamo ad *Hyperfine* di ignorare qualsiasi output.

- Il comando "***wasmedge -reactor***" permette di eseguire il runtime *WasmEdge* definendo il path del file *WASM*, la *funzione* da invocare e l'*input* da passare alla funzione.
- Il comando "***wasmer run***" esegue il runtime *Wasmer* definendo il path del file *WASM*, la *funzione* da invocare e l'*input* da passare alla funzione.
- Il comando "***wasm3 -func***" consente di eseguire il runtime *Wasm3* definendo il path del file *WASM*, la *funzione* da invocare e l'*input* da passare alla funzione.

Si può notare che il comando "***wasmedge -reactor***" è stato eseguito **due volte** con due file ***WASM*** differenti. Questo perché il runtime *WasmEdge* consente la **compilazione AOT** (spiegata nella sezione 2.2.2 delle caratteristiche di *WasmEdge*). Essendo l'unico runtime tra i tre a consentire la compilazione *AOT*, abbiamo effettuato la compilazione *AOT* del file *WASM* chiamato "*recursive_fibonacci.wasm*", il quale ha prodotto un nuovo file chiamato "*recursive_fibonacci_aot.wasm*". Successivamente abbiamo eseguito i vari benchmarking anche con il nuovo file prodotto attraverso la compilazione *AOT*. Il comando per compilare il file *WASM* con la modalità *AOT* è il seguente:

```
1 wasmedgec recursive_fibonacci.wasm recursive_fibonacci_aot.wasm
```

Listing 3.4: Comando per compilazione AOT di WasmEdge

Esecuzione CoreMark 1.0

Anche per il benchmark ***CoreMark 1.0*** abbiamo utilizzato il tool *Hyperfine*. Questa volta al posto di eseguire 100 volte il benchmark per ogni runtime, abbiamo eseguito il benchmark **10 volte** per ogni runtime perché il CoreMark 1.0 esegue già un determinato numero di iterazioni per valutare le prestazioni del sistema ed eseguire questo benchmark troppe volte potrebbe essere eccessivo e comportare tempi di esecuzione molto lunghi.

Di seguito è riportato il comando utilizzato:

```
1 hyperfine
2 --prepare "purge"
3 --export-json coremark_wasmedge_results.json
4 --runs 10
5 'wasmedge coremark.wasm'
6 --show-output
7 | tee coremark_wasmedge_results.txt
```

Listing 3.5: Comando Hyperfine esecuzione benchmark CoreMark 1.0 con runtime WasmEdge.

In questo caso utilizziamo il comando "***show-output***" perché vogliamo mostrare l'output di *CoreMark 1.0* sul terminale e usiamo il comando "***tee***" per salvare l'output su un file di testo. Facciamo questo perché vogliamo usare *Hyperfine* per rendere i test affidabili, ma non ci interessa tanto il tempo di esecuzione che ci fornisce *Hyperfine*, ci serve il risultato fornito da *CoreMark 1.0*. Nello specifico è utile il parametro "***Iterations/Sec***" (sarà spiegato nel capitolo 4).

Capitolo 4

Risultati

In questo capitolo mostreremo i risultati dei vari benchmarking discussi nel capitolo 3.

4.1 Risultati Benchmarking WebAssembly su Browser

La **Tabella 4.1** riporta lo *score PSPDFKit* per ogni esecuzione dei benchmark effettuati su **Chrome**, la **Tabella 4.2** indica gli *score PSPDFKit* dei benchmark eseguiti su **Safari** mentre la **Tabella 4.3** riporta gli *score PSPDFKit* ottenuti dai benchmark eseguiti su **Firefox**.

#Iteration	PSPDFKit Wasm Score
1	2995
2	3026
3	3047
4	3026
5	3021
6	3025
7	3036
8	3009
9	3026
10	3022

Tabella 4.1: PSPDFKit WebAssembly Benchmarking Chrome version 108.0.5359.124.

#Iteration	PSPDFKit Wasm Score
1	3858
2	3857
3	3621
4	3783
5	3684
6	3685
7	3663
8	3652
9	3711
10	3967

Tabella 4.2: PSPDFKit WebAssembly Benchmarking Safari version 15.4 (17613.1.17.1.6).

#Iteration	PSPDFKit Wasm Score
1	3049
2	3050
3	3109
4	3010
5	3024
6	3017
7	2996
8	3024
9	3006
10	3016

Tabella 4.3: PSPDFKit WebAssembly Benchmarking Firefox version 108.0.1.

Nella figura 4.1 è riportato il *90° percentile* del *Performance Benchmarking* WebAssembly di *PSPDFKit Web*. Consideriamo i risultati dei benchmark ottenuti su *Chrome* (riportati nella Tabella 4.1). Il *90° percentile* è il risultato dell'iterazione numero 9, quindi significa che 90% delle iterazioni totali ha un *PSPDFKit Wasm Score* di **3036** o meno. Più basso è il punteggio e migliori sono le prestazioni.

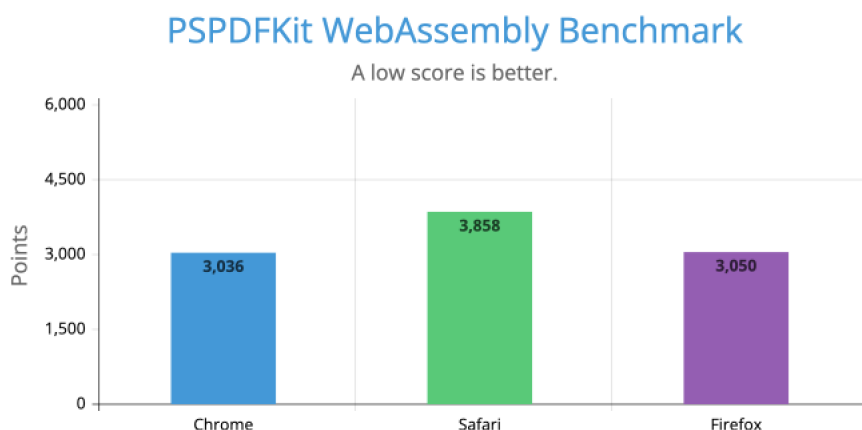


Figura 4.1: Risultati Benchmarking WebAssembly di *PSPDFKit Web*.

4.2 Risultati Benchmarking dei Runtime WebAssembly

In questa sezione mostreremo e descriveremo i risultati ottenuti del *Performance Micro-benchmark* dei runtime WebAssembly, rispetto ai workload stabiliti.

I grafici riportati sono stati realizzati tramite alcuni *script Python* creati dai fondatori del *tool Hyperfine* (con qualche piccola modifica da parte nostra per mostrare alcuni dettagli specifici).

4.2.1 Fibonacci

I grafici 4.3, 4.4, 4.5, sono chiamati "*scatter plot*". Essi mostrano la relazione tra il tempo di esecuzione (sull'asse delle ordinate) e il numero di iterazioni

(sull'asse delle ascisse), per un determinato benchmark effettuato con *Hyperfine*. Gli elementi singoli (*point*) rappresentano i singoli risultati ottenuti durante il benchmark. Questo tipo di grafico ci permette di visualizzare la distribuzione dei tempi di esecuzione e di identificare eventuali punti anomali.

Per favorire la lettura dei grafici, riportiamo la figura 4.2, che rappresenta la legenda dei vari grafici.

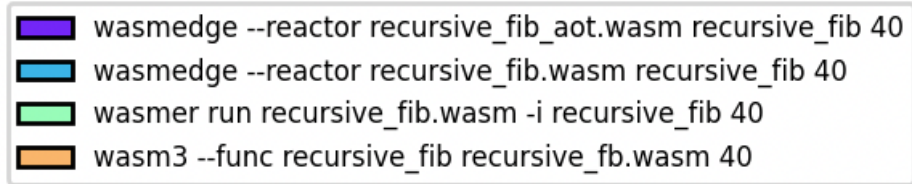


Figura 4.2: Legenda dei grafici.

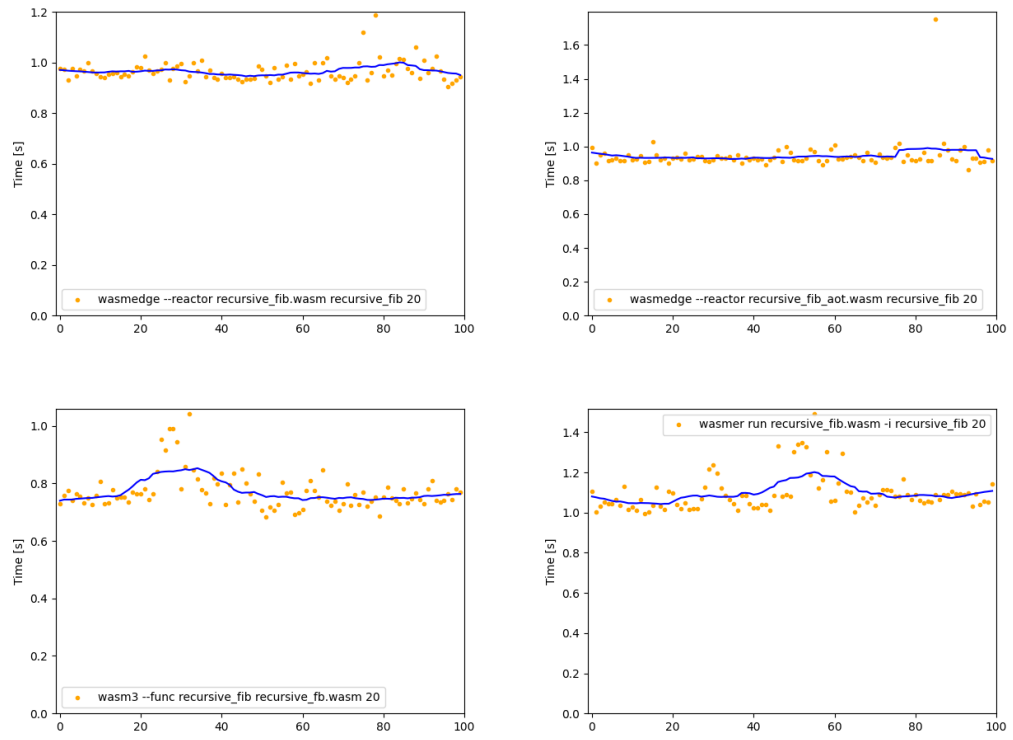


Figura 4.3: Scatter Plot Benchmarking $Fibonacci(20)$.

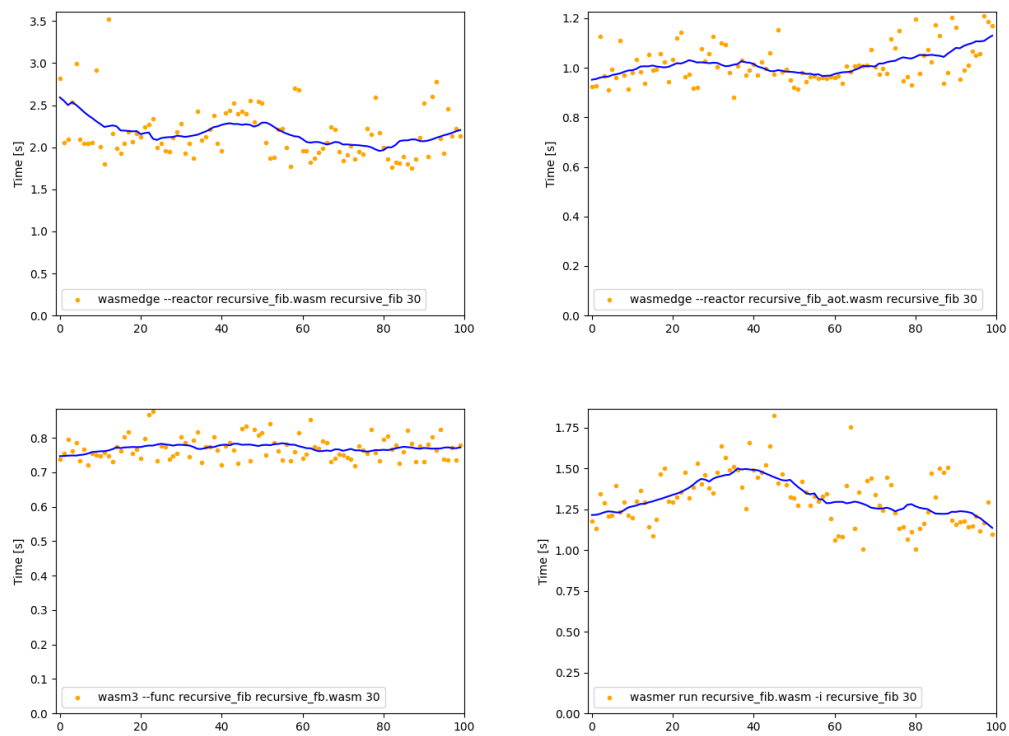


Figura 4.4: Scatter Plot Benchmarking $\text{Fibonacci}(30)$.

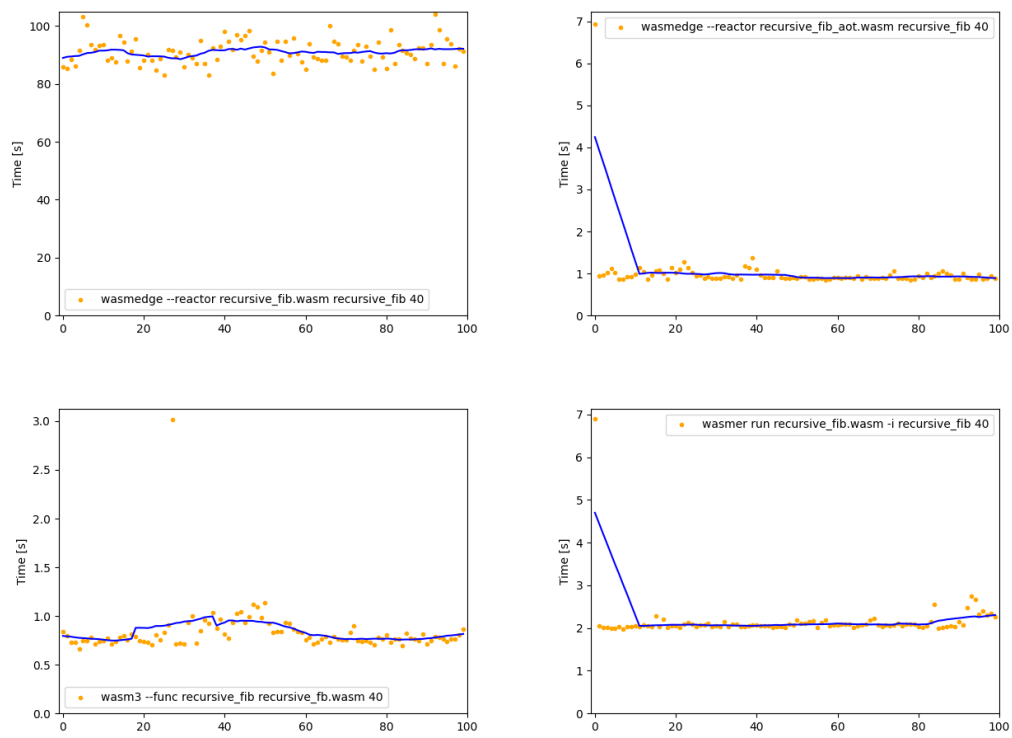


Figura 4.5: Scatter Plot Benchmarking Fibonacci(40).

I grafici 4.6, 4.7 e 4.10 mostrano l'**istogramma** del tempo di esecuzione dei runtime WebAssembly: *WasmEdge* (con modalità *AOT*), *WasmEdge* (con modalità *JIT*), *Wasm3* e *Wasmer*. Sono state eseguite *100* esecuzioni per ogni runtime e i risultati sono stati aggregati in un istogramma che mostra la distribuzione del tempo di esecuzione. Il tempo di esecuzione viene mostrato sull'asse delle ascisse e il numero di esecuzioni sull'asse delle ordinate. Ciascun *bar* rappresenta un intervallo di tempo di esecuzione e indica quante volte il tempo di esecuzione di un benchmark è caduto all'interno di quell'intervallo.

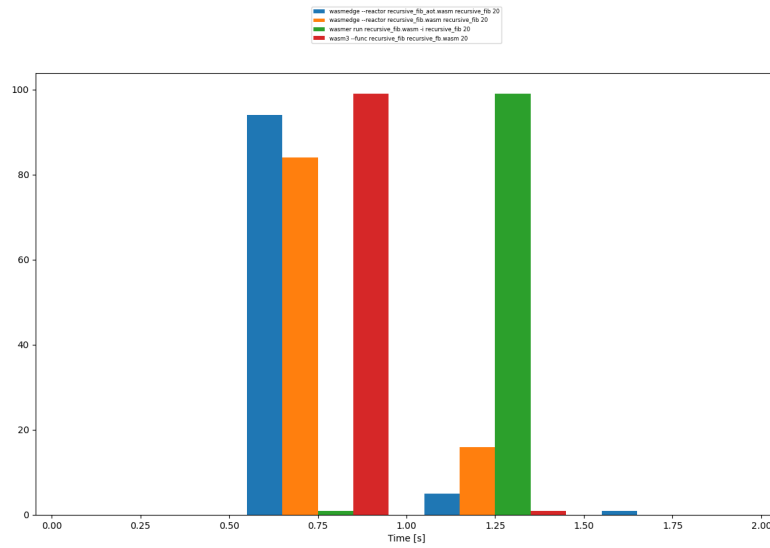


Figura 4.6: Histogram Plot Fibonacci(20).

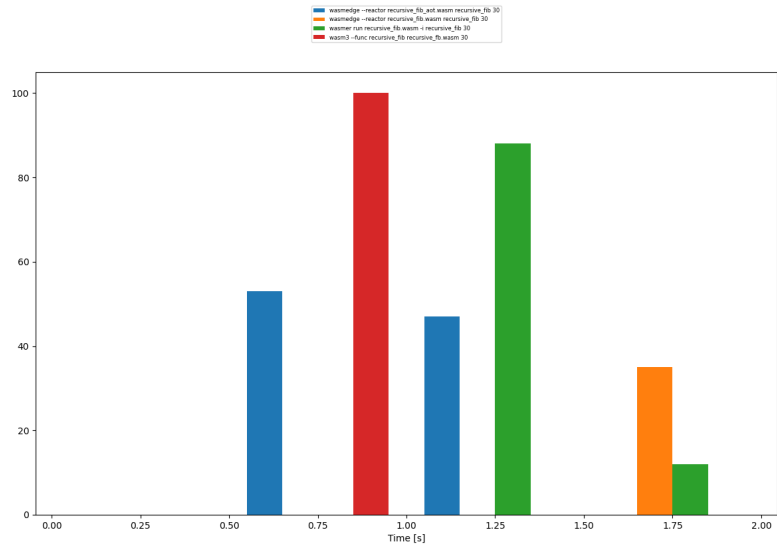


Figura 4.7: Histogram Plot Fibonacci(30).

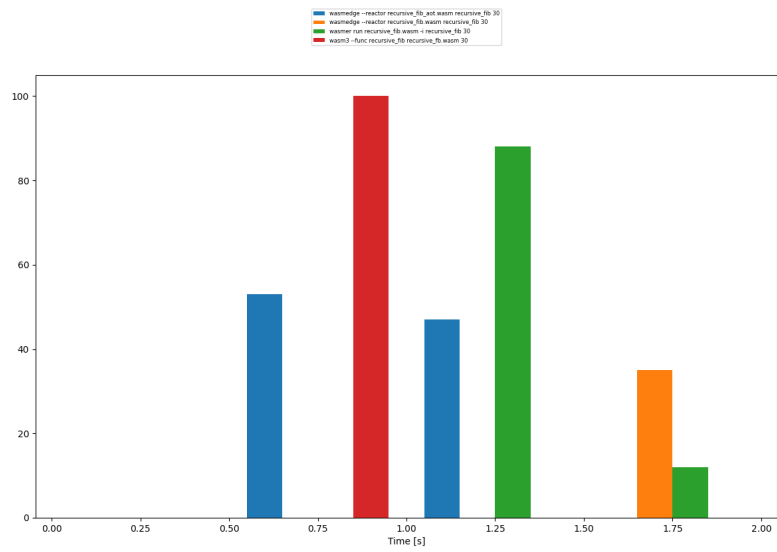


Figura 4.8: Histogram Plot Fibonacci(40).

L'istogramma è un buon modo per visualizzare tutti i benchmark in un unico grafico e capire in quale intervallo di tempo ricadono i risultati delle esecuzioni, ma non sempre si riesce a cogliere l'andamento preciso dei risultati. Proprio per questo motivo, ci è sembrato più adatto rappresentare i benchmark anche con dei *box whisker pilot*. Il *box whisker plot* mostra la distribuzione dei tempi di esecuzione attraverso la **mediana** (il valore centrale), la **prima** e la **terza quartile** (che rappresentano il 25% e il 75% dei dati) e i cosiddetti "*whisker*" che rappresentano i **valori estremi**.

Questa rappresentazione grafica aiuta a individuare variazioni significative delle performance e se sono presenti eventuali valori anomali che possono influire sui risultati finali.

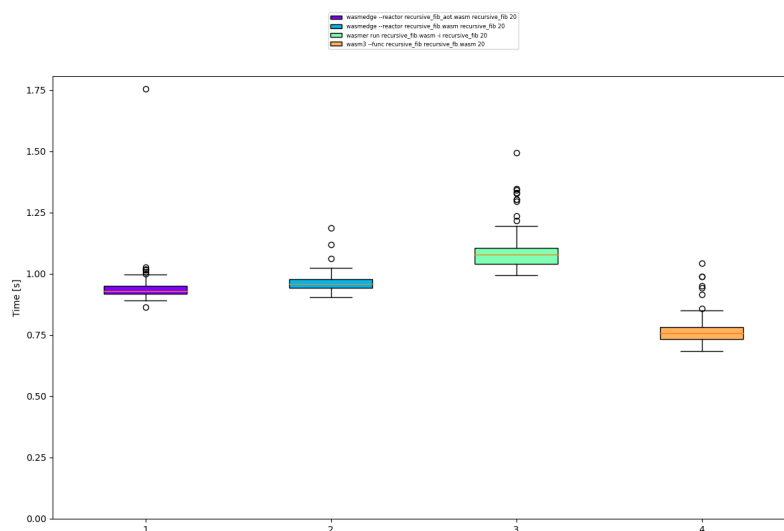


Figura 4.9: Box Whisker Plot Fibonacci(40).

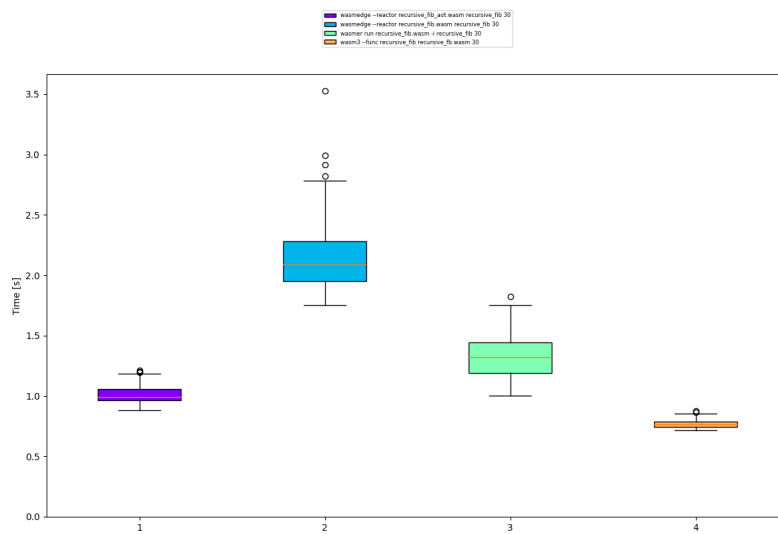


Figura 4.10: Box Whisker Plot Fibonacci(40).

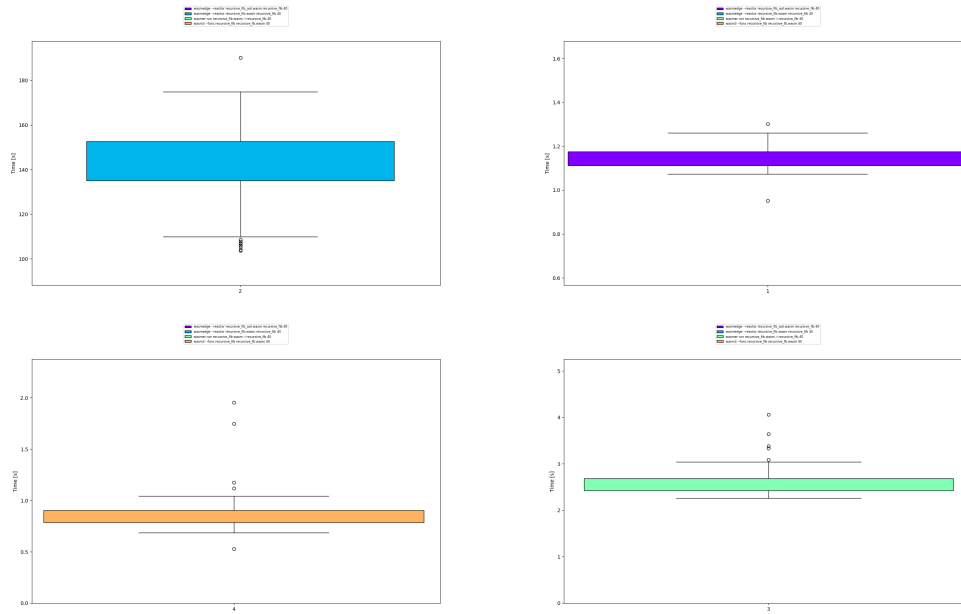


Figura 4.11: Box Whisker Plot Fibonacci(40).

Il grafico *box whisker* dell'esecuzione della funzione di Fibonacci con input 40, per una questione di visualizzazione, è stato diviso in quattro grafici (in figura 4.11) che rappresentano l'andamento dei runtime: *WasmEdge* con modalità *AOT*, *WasmEdge* con modalità *JIT*, *Wasm3* e *Wasmer*.

Rappresentiamo un riepilogo dei risultati attraverso le Tabelle 4.4, 4.5 e 4.6. Abbiamo effettuato il benchmark con la funzione di Fibonacci con input *20*, poi *30* e successivamente *40*. Per ogni runtime sono state effettuate *100* esecuzione.

Le Tabelle rappresentano i seguenti parametri per ogni runtime:

- **Media:** è il valore che rappresenta la somma di tutti i tempi di esecuzione divisi per il numero di esecuzioni. La media fornisce una misura della prestazione media del programma durante i benchmark.
- **Mediana:** è il valore centrale di un insieme di dati ordinati, ovvero il valore che divide l'insieme in due parti uguali. La mediana è meno influenzata da valori estremi rispetto alla media.
- **Minimo:** è il valore più piccolo tra tutti i tempi di esecuzione misurati durante i benchmark.
- **Massimo:** è il valore più grande tra tutti i tempi di esecuzione misurati durante i benchmark.
- **95 percentile:** è il valore che separa il 95% dei tempi di esecuzione più brevi dal 5% dei tempi di esecuzione più lunghi. Questa statistica fornisce una misura della massima prestazione che un programma potrebbe raggiungere in almeno il 95% delle esecuzioni.

Queste statistiche vengono calcolate sulla base dei tempi di esecuzione misurati durante i benchmark e sono utilizzate per descrivere le prestazioni dei runtime in modo più preciso e completo.

	WasmEdge (AOT)	WasmEdge (JIT)	Wasm3	Wasmer
Mean	0.945 s	0.966 s	0.772 s	1.095 s
Median	0.929 s	0.958 s	0.757 s	1.077 s
Min	0.864 s	0.905 s	0.683 s	0.994 s
Max	1.754 s	1.188 s	1.042 s	1.493 s
95° percentil	1.001 s	1.024 s	1.305 s	0.917 s

Tabella 4.4: Tabella riepilogo dei risultati Benchmarking Fibonacci(20).

	WasmEdge (AOT)	WasmEdge (JIT)	Wasm3	Wasmer
Mean	1.016 s	2.161 s	0.770 s	1.321 s
Median	0.994 s	2.088 s	0.765 s	1.323 s
Min	0.882 s	1.752 s	0.718 s	1.005 s
Max	1.211 s	3.526 s	0.877 s	1.827 s
95° percentil	1.170 s	2.701 s	0.828 s	1.572 s

Tabella 4.5: Tabella riepilogo dei risultati Benchmarking Fibonacci(30).

	WasmEdge (AOT)	WasmEdge (JIT)	Wasm3	Wasmer
Mean	1.145 s	141.915 s	0.869 s	2.619 s
Median	1.139 s	144.113 s	0.847 s	2.537 s
Min	0.952 s	103.748 s	0.531 s	2.257 s
Max	1.301 s	190.196 s	1.956 s	4.064 s
95° percentil	1.219 s	168.857 s	1.037 s	3.046 s

Tabella 4.6: Tabella riepilogo dei risultati Benchmarking Fibonacci(40).

Considerando il benchmarking della funzione di Fibonacci con input *20*, il runtime che ha il miglior tempo di esecuzione è *Wasmer* con *0.917* secondi mentre il runtime che ha il peggior tempo di esecuzione è *Wasm3* con *1.305* secondi. Nonostante *Wasm3* abbia la media più bassa di tutti *0.757* secondi, calcolando il *95° percentile* risulta il più lento. Questo potrebbe indicare una certa instabilità nella performance del runtime *Wasm3*.

Rispetto al benchmarking della funzione di Fibonacci con input *30*, il runtime con il minor tempo di esecuzione è *Wasm3* con *0.828* secondi mentre il peggiore è *WasmEdge* in modalità *JIT* con *2.701* secondi.

Infine, prendendo in considerazione il benchmarking della funzione di Fibonacci con input *40*, il runtime migliore è *WasmEdge* con modalità *AOT* con *1.219* secondi e il peggiore è *WasmEdge* con modalità *JIT* con ben *168.875* secondi.

Alcune considerazioni finali del benchmark:

- ***WasmEdge*** con la modalità ***JIT*** all'aumentare della complessità dell'esecuzione risponde sempre peggio, aumentando notevolmente il tempo di esecuzione. Mentre con la modalità ***AOT***, al crescere dell'input dato alla funzione di Fibonacci (ricordando che la funzione è ricorsiva) riesce a mantenere basso il tempo di esecuzione.
- ***Wasm3*** con input piccoli e quindi bassa complessità di esecuzione, tiene alti i tempi di esecuzione mentre all'aumentare della complessità, i tempi di esecuzione migliorano. Questo potrebbe essere dovuto dal fatto che *Wasm3* non utilizza una compilazione, ma ha una modalità ad "interprete".
- ***Wasmer*** all'aumentare della complessità, l'andamento del tempo di esecuzione cresce senza troppi picchi.

4.2.2 CoreMark 1.0

La teoria di funzionamento di *CoreMark* è basata su un insieme di task che vengono eseguiti per valutare le prestazioni della *CPU*. Questi task includono operazioni aritmetiche (ad esempio, l'incremento e la moltiplicazione), operazioni logiche (ad esempio, le operazioni *OR* ed *AND*), manipolazioni delle stringhe (ad esempio, la ricerca di una sottostringa in una stringa).

In sintesi, *CoreMark* è un benchmark che testa la **velocità** e l'**efficienza** della *CPU* di un micro-controllore (nel nostro caso dei runtime WebAssembly) eseguendo un insieme di operazioni specifiche. Il risultato finale del benchmark fornisce un valore di **performance** in termini di **iterazioni al secondo** che può essere utilizzato per confrontare le prestazioni di micro-controllori diversi.

Alcuni degli algoritmi principali utilizzati da *CoreMark* sono:

- **Elaborazione di stringhe:** include operazioni come la ricerca e la sostituzione in stringhe, che sono comuni in molte applicazioni *embedded*.
- **Manipolazione di matrici:** include operazioni di manipolazione di matrici, come la moltiplicazione di matrici, che sono utilizzate in molte applicazioni scientifiche e di calcolo.
- **Gestione dello stato:** include operazioni di gestione dello stato, come la gestione delle tabelle *hash*, che sono comuni in molte applicazioni *embedded*.

	WasmEdge (AOT)	WasmEdge (JIT)	Wasm3	Wasmer
90° percentil Total time	19.841 s	14.135 s	18.063 s	15.702 s
90° percentil Iterations/Sec	12966	94	11810	783

Tabella 4.7: Tabella riepilogo dei risultati Benchmarking *CoreMark 1.0*.

Il grafico 4.12 mostra i risultati riportati nella Tabella 4.7 con un grafico a barre, dove è mostrata la relazione tra il tempo totale e il numero di iterazioni al secondo.

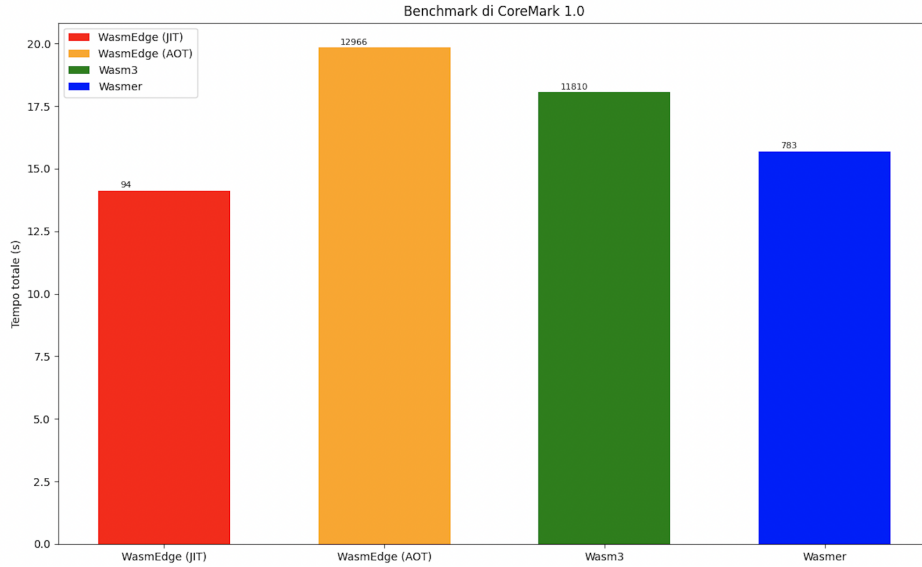


Figura 4.12: Grafico a barre dei risultati del Benchmarking Coremark 1.0.

Nella Tabella 4.7 è riportato il riepilogo dei risultati ottenuti. Ad ogni runtime è stato dato in input il module *WASM* chiamato "*coremark.wasm*", che rappresenta l'insieme delle operazioni effettuate dal benchmark *CoreMark 1.0*. Per ogni runtime sono state effettuate 10 esecuzioni del benchmark. Nello specifico la tabella riporta per ogni runtime, il 90° percentile del **tempo totale** sulle 10 esecuzioni e il 90° percentile delle **iterazione al secondo** sulle 10 esecuzioni. Il parametro significativo è proprio "*Iterations/Secs*", il quale rappresenta il numero di iterazioni completate di CoreMark al secondo. Più alto è il valore di "*Iteration/Sec*", più veloce è il sistema.

Per quanto riguarda il valore di "*Iterations/Sec*", il runtime *WasmEdge* in modalità *AOT* ha il valore più alto 12966, seguito da *Wasm3* con 11810, seguito da *WasmEdge* in modalità *JIT* con 94, e *Wasmer* con 783. Questo significa che *WasmEdge* in modalità *AOT* ha il miglior rendimento tra i quattro runtime testati, mentre *Wasmer* ha il peggior rendimento.

Per quanto riguarda il parametro "*Total Time*", il runtime *WasmEdge*

in modalità *JIT* ha il valore più basso *14.135*, seguito da *Wasmer* con *15.702*, *WasmEdge* in modalità *AOT* con *19.841* e *Wasm3* con *18.063*. Questo significa che *WasmEdge* in modalità *JIT* ha il miglior tempo totale di esecuzione tra i quattro runtime testati, mentre *WasmEdge* in modalità *AOT* ha il peggior tempo totale di esecuzione.

Però i valori "*Iterations/Secs*" e "*Total time*" non sono gli unici indicatori delle prestazioni. La memoria utilizzata, la velocità di caricamento, la compatibilità con le *API*, la scalabilità e altre metriche sono tutte importanti per valutare le prestazioni. Uno degli obiettivi futuri sarà quello di effettuare una valutazione completa e dettagliata dei vari runtime rispetto al benchmark CoreMark 1.0, per determinare quale sia il runtime migliore.

Capitolo 5

Conclusioni

In questo elaborato, abbiamo iniziato con una breve introduzione alla tecnologia WebAssembly, esplorando il suo funzionamento e le caratteristiche principali. Abbiamo poi presentato lo stato dell'arte delle tecnologie relative a WebAssembly e gli strumenti di benchmarking, descrivendo le soluzioni già esistenti e conosciute. Successivamente, abbiamo esaminato come sono stati implementati i vari benchmark e infine presentato i risultati ottenuti.

5.0.1 Obiettivi raggiunti

Siamo riusciti a fornire un punteggio delle *prestazioni* della tecnologia WebAssembly sui Browser *Chrome*, Safari e *Firefox*, reputando *Chrome* come l'attuale Browser che riesce a sfruttare al meglio questa tecnologia.

Abbiamo effettuato dei ***Performance Micro-benchmarking*** dei runtime *WasmEdge* (con modalità *JIT* e *AOT*), *Wasm3* e *Wasmer*, per cercare di comprendere a che punto della conoscenza si è con la tecnologia WebAssembly al di fuori del Web. Rispetto ai benchmarking della **funzione di Fibonacci** ricorsiva con input *20*, *30* e *40*, il runtime che sembra mantenere più bassi i tempi di esecuzione con la maggiore complessità (input *40*) è *WasmEdge* con la modalità *AOT*. Questo risultato ci può portare ad una visione futura del fatto che la modalità *AOT* possa essere un buon modo per sfruttare la tecnologia WebAssembly con il massimo del suo potenziale (nonostante ancora la tanta ricerca che c'è in questo ambito).

Infine, utilizzando un benchmark noto chiamato "***CoreMark 1.0***" adattato al WebAssembly, siamo riusciti a testare i runtime descritti in precedenza.

Però rispetto a quest'ultimo benchmark, bisogna effettuare un lavoro più dettagliato per decretare un runtime "vincitore".

5.0.2 Direzioni future

Un primo obiettivo futuro è analizzare dettagliatamente il benchmark *Core-Mark 1.0*, per vedere il comportamento dei runtime rispetto alla memoria utilizzata, la velocità di caricamento, la compatibilità con le *API*, la scalabilità e altre metriche.

Altro obiettivo futuro è ripetere gli stessi benchmark aggregando altri runtime esistenti ed estendere i benchmark con altri workload.