CENTRO UNIVERSITÁRIO 7 DE SETEMBRO - UNI7

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

PEDRO HENRIQUE SILVEIRA RODRIGUES

ANÁLISE DE PERFORMANCE ENTRE *WEBASSEMBLY* E *JAVASCRIPT*

FORTALEZA – CEARÁ

2017

PEDRO HENRIQUE SILVEIRA RODRIGUES

ANÁLISE DE PERFORMANCE ENTRE *WEBASSEMBLY* E *JAVASCRIPT*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Centro Universitário 7 de Setembro - UNI7, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientador: André Jackson Gomes Bessa

FORTALEZA – CEARÁ

2017

PEDRO HENRIQUE SILVEIRA RODRIGUES

ANÁLISE DE PERFORMANCE ENTRE *WEBASSEMBLY* E *JAVASCRIPT*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Centro Universitário 7 de Setembro - UNI7, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

BANCA EXAMINADORA

André Jackson Gomes Bessa (Orientador)
Centro Universitário Sete de Setembro

Raimundo Sales Neto e Azevedo
Centro Universitário Sete de Setembro - UNI7

Robério Gomes Patricio
Centro Universitário Sete de Setembro - UNI7

Eduardo Mendes de Oliveira
Centro Universitário Sete de Setembro - UNI7

AGRADECIMENTOS

Agradeço a minha mãe, Virgínia Lúcia, que me deu apoio e incentivo em todos os momentos da minha vida, e que não me deixou desistir em momento algum.

Obrigado a minha namorada, Isabely Lima, por me fortalecer nos momentos de dificuldades, e estar ao meu lado quando mais precisei.

Meus agradecimentos aos meus fiéis amigos, Matheus Silva e Morgana Almeida, que fizeram parte desta jornada.

Agradeço à instituição pelo ambiente que proporcionou, pela oportunidade de fazer o curso, e a todo o corpo docente e administração.

Agradeço ao meu grande professor André Bessa, pela orientação e apoio, e pelo empenho dedicado à produção deste trabalho.

A todos que participaram direto ou indiretamente da minha formação acadêmica, o meu muito obrigado.

“Não importa o quanto tente, você sozinho não pode mudar o mundo, mas este é o lado bonito do mundo, simplesmente continuar tentando...”

(L - Death Note)

RESUMO

JavaScript é a única linguagem de programação suportada nativamente pelos navegadores, sendo bastante criticada, dentre outros motivos por ser extremamente dinâmica, o que a fez se tornar alvo de compilação para várias outras linguagens. Um projeto que tem se destacado principalmente pelo seu desempenho de execução, e que será abordado neste trabalho, é o *WebAssembly*, que também possui suporte nativo nos navegadores modernos, e que possui a participação dos principais fornecedores de navegadores até então. O propósito desta monografia é apresentar uma análise comparativa quantitativa de desempenho entre *WebAssembly* e *JavaScript*, embasando-se em seus tempos de execução para identificar em que casos, códigos escritos em *WebAssembly* possuem desempenho superior a códigos escritos em *JavaScript*.

Palavras-chave: *WebAssembly*. *JavaScript*. Performance. Tempo de execução.

ABSTRACT

JavaScript is the only programming language natively supported by browsers, being criticized, among other reasons for being extremely dynamic, which made it become the target of compilation for several other languages. A project that has been highlighted mainly by its execution performance, and that will be approached in this work, is the WebAssembly, which also has native support in modern browsers, and that has the participation of the main browsers providers until then. The purpose of this monograph is to present a quantitative comparative performance analysis between WebAssembly and JavaScript, based on its execution times to identify in which cases, codes written in WebAssembly perform better than codes written in JavaScript.

Palavras-chave: WebAssembly. JavaScript. Performance. Runtime.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 INTRODUÇÃO

A *web* começou como uma simples rede de troca de documentos, mas tem se tornado a plataforma de aplicativos mais onipresente de todos os tempos, acessível em uma vasta gama de sistemas operacionais e tipos de dispositivos. Por acidente histórico, *JavaScript* é a única linguagem de programação suportada nativamente na *web*, com exceção de algumas tecnologias disponíveis apenas através de *plugins*, como: *ActiveX*, *Java* ou *Flash*. Devido à "onipresença"do *JavaScript*, rápidas melhorias de desempenho em máquinas virtuais modernas têm sido necessárias, talvez por necessidade absoluta, acabou se tornando um alvo de compilação para outras linguagens. Através do *Emscripten*, mesmo programas escritos em *C* e *C++* podem ser compilados em um subconjunto estilizado de baixo nível de *JavaScript*, chamado *asm.js*. No entanto, *JavaScript* tem desempenho inconsistente e uma série de outras armadilhas, especialmente como um alvo de compilação. (HAAS *et al.*, 2017)

WebAssembly é uma alternativa proposta que tem como um de seus objetivos resolver problemas presentes na utilização de *JavaScript*, e que tem ganho bastante espaço entre desenvolvedores recentemente. *WebAssembly* é um novo tipo de código, que pode ser executado em navegadores modernos, trata-se de uma linguagem de baixo nível, como *assembly*, com um formato binário compacto que executa com performance quase nativa e que fornece um novo alvo de compilação para linguagens como *C*/*C++*, para que possam ser executadas na *web*. Também foi projetado para executar em conjunto com o *JavaScript*, permitindo que ambos trabalhem juntos. (ARAúJO, 2017)

WebAssembly trás consigo grandes implicações para a plataforma *web*, pois fornece uma maneira de executar códigos escritos em vários outros idiomas além de *JavaScript* na *web*, além de possibilitar uma velocidade de execução próxima da execução de um código nativo. Possui padrão aberto desenvolvido por um grupo comunitário da W3C que inclui representantes de todos os principais fornecedores de navegadores até então.

Neste trabalho é apresentada uma análise comparativa quantitativa, entre algoritmos escritos em *JavaScript* e *WebAssembly*, baseando-se em medições de seus tempos de execução, através de experimentos, sendo estes descritos na seção 3, após um embasamento teórico necessário para a compreensão deste trabalho descrito na seção 2, e finalizando com a seção 4 onde são feitas as considerações finais.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

A proposta deste trabalho é realizar uma análise comparativa entre algoritmos escritos em *WebAssembly* e *JavaScript*, utilizando como base seus tempos de execução para identificar em que casos, códigos escritos em *WebAssembly* possuem desempenho superior a códigos escritos em *JavaScript*.

1.1.2 Objetivos Específicos

a) Estudar o processamento realizado internamente em um motor *JavaScript*, e como ele processa algoritmos escritos em *WebAssembly* e *JavaScript*.
b) Elaborar um modelo de obtenção de métricas de desempenho de execução de códigos escritos em *JavaScript* e *WebAssembly*.
c) Desenvolver uma prova de conceito que sirva como base de comparação entre algoritmos escritos em *WebAssembly* e *JavaScript*, objetivando analisar a performance destes.
d) Realizar uma análise comparativa de desempenho entre algoritmos escritos em *WebAssembly* e *JavaScript*, utilizando os resultados obtidos na prova de conceito.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção se dedica a fornecer o embasamento teórico necessário para a compreensão das seções posteriores, focando principalmente no fluxo de processamento realizado por um motor *JavaScript*.

2.1 *JAVASCRIPT*

JavaScript é uma linguagem de programação, que surgiu em 1995, criada pelo desenvolvedor estadunidense, Brendan Eich, como parte do navegador *web Netscape Navigator* 2.0. O desenvolvimento da linguagem iniciou com o objetivo de fazer com que o navegador tivesse a habilidade de interpretar *scripts* e com isso a criação de páginas *web* dinâmicas. Ela apareceu em todos os navegadores subsequentes da *Netscape*, em todos os navegadores da Microsoft iniciando com o *Internet Explorer* 3.0, e desde 2002 tem sido suportada pelos navegadores mais populares como *Google Chrome*, *Mozilla Firefox*, *Apple Safari* e *Microsoft Internet Explorer*. (W3SCHOOLS, 2017)

O desenvolvimento da padronização da linguagem foi iniciado em novembro de 1996 pela *Nestcape*, que ao finalizar submeteu a especificação a Assembléia Geral da ECMA International, atualmente a maior autoridade da associação, que foi aceita e teve a definição do padrão chamada de ECMAScript com a especificação ECMA-262 e ISO/IEC 16262 (ECMA, 2017).

2.1.1 Motor *JavaScript*

Para que seja possível executar um *script* feito na linguagem *JavaScript*, é necessário que se tenha um motor *JavaScript*, em inglês *JavaScript Engine*, que é responsável por interpretar e executar códigos da linguagem. Embora existam vários usos para esse motor, ele é mais comumente utilizado em navegadores, mas existem outros projetos como *Node.js* que exercem esse trabalho no lado do servidor.

Segundo definição apresentada no site da *Mozilla Developer Network*, um motor *JavaScript* compila e executa *scripts* contendo instruções e funções escritos na linguagem *JavaScript*, lida com a alocação de memória para os objetos necessários para a execução dos *scripts* e desalocação de memória dos objetos que não são mais necessários. (MOZILLA, 2017)

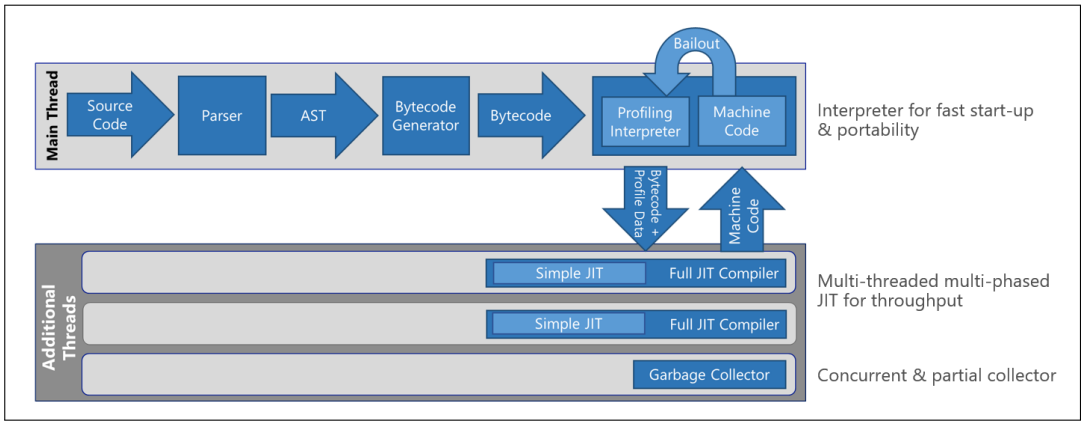
Não há uma especificação formal que padronize o que um motor *JavaScript* deve pos-

suir em sua implementação, entretanto alguns padrões podem ser percebidos nas implementações existentes até então:

- Analisador (Léxico + Sintático)
- Representação Intermediária
- Interpretador
- Compilador de linha de base
- Compilador de otimização
- Coletor de Lixo

O esquema lógico apresentado na Figura 1, mostra o fluxo de processamento feito pelo motor *JavaScript*, executando em um processador multi-core, desde o recebimento do código até sua execução.

Figura 1 – Esquema lógico sequencial de processamento de um motor *JavaScript*



Fonte: Seth e Foresti

Iniciando o processamento do código, como mostrado no esquema acima, será utilizado um trecho de código que faz o cálculo do quadrado de um número e retorna o resultado desse cálculo, como prova de conceito para mostrar passo a passo os resultados de cada etapa do processo no motor *JavaScript*. O trecho de código pode ser visto no Código-fonte 1.

Código-fonte 1 – *Square* escrito em *JavaScript*

```
1 function square(x) {
2     return x * x;
3 }
```

O motor ao receber o trecho de código inicia o *parsing*, esse processo pode ser

dividido em dois subprocessos: Análise Léxica e Análise Sintática:

- **Análise Léxica**: processo de quebrar uma entrada em uma sequência de símbolos (*tokens*). *Tokens* são o vocabulário linguístico: uma coleção de blocos de construção válidos, de acordo com uma gramática.
- **Análise Sintática**: processo de aplicação de regras de sintaxe, de acordo com uma gramática formal.

Cada um desses subprocessos possui um componente responsável por realizar tarefas específicas de cada contexto, sendo eles:

- ***Lexer* ou *Tokenizer***: responsável por quebrar uma entrada em *tokens* válidos, exercendo assim o papel de analisador léxico. O *lexer* também sabe como tirar caminhos irrelevantes como espaços em branco e quebras de linha.
- ***Parser***: responsável pela construção da Árvore de Sintaxe Abstrata, em inglês *Abtract Syntax Tree* (AST), analisando a estrutura do código de acordo com as regras de sintaxe da linguagem, buscando por possíveis erros baseado na escrita padrão da linguagem, para evitar erros durante a compilação ou resultados divergentes na execução do código ao final do processo.

O processo de *parsing* é iterativo. O *parser* solicita ao *lexer* um novo *token*, e tenta combinar o *token* recebido com uma das regras de sintaxe. Se uma regra for correspondida, um novo nó correspondente ao *token* é adicionado a AST e o *parser* solicitará o próximo *token*, e assim consecutivamente.

Se nenhuma regra corresponder, o *parser* armazenará o *token* internamente e continuará pedindo *tokens* até encontrar uma regra que corresponda a todos os *tokens* armazenados internamente. Se nenhuma regra for encontrada, que corresponda aos *tokens* armazenados, o *parser* lançará uma exceção. Isso significa que o código fornecido não é válido e possui erros de sintaxe.

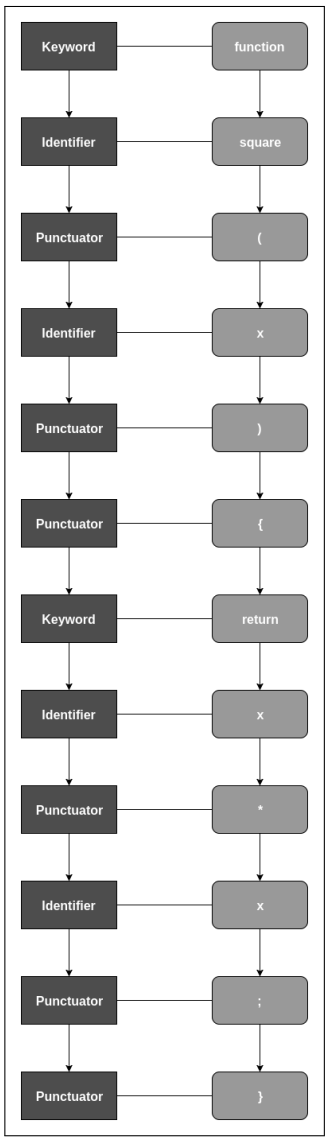
Com base na função *square* citada anteriormente, será obtido como resultado do *parsing* utilizando o primeiro subprocesso, *tokenizer* [1], os *tokens* que serão gerados e posteriormente analisados, cada um possuindo seu próprio tipo. Os *tokens* possem ser observados na Figura 2.

Após o resultado do *tokenizer* ser gerado, uma Árvore de Sintaxe Abstrata, é criada, para a função de potenciação conforme é mostrado na Figura 3.

Após a geração da AST, com base na implementação do motor *JavaScript*, o *bytecode*

[1] *Tokenizer* pode ser compreendido como o processo de analisar uma entrada de caracteres, como um código-fonte, e produzir uma sequência de símbolos que podem ser manipulados mais facilmente por um *parser*.

Figura 2 – Resultado do *Tokenizer* com base na função *square*



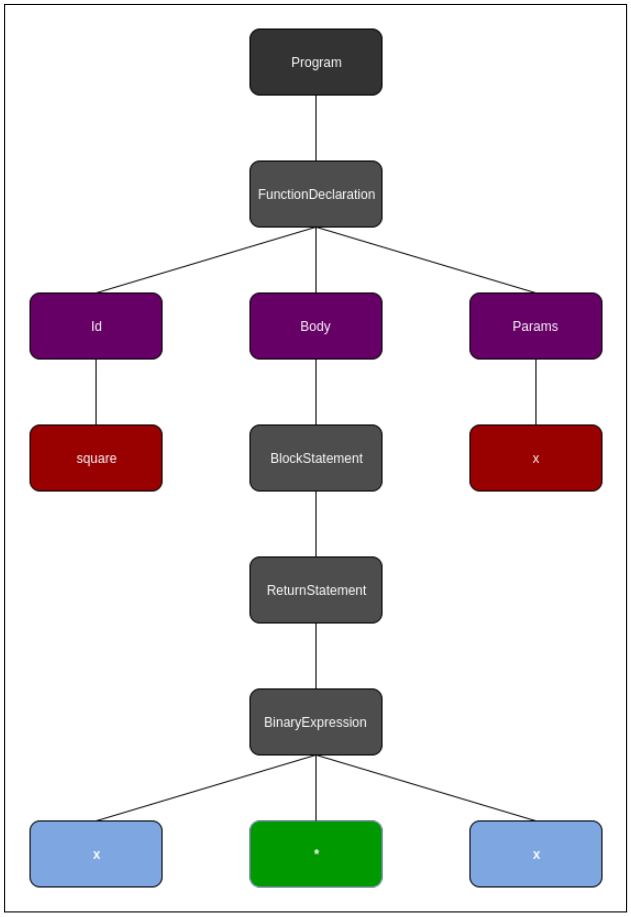
Fonte**:** Elaborada pelo autor.

generator a converte para uma linguagem intermediária ou para código nativo[2], e isso é feito para cada bloco de código representado na AST. Pode-se citar como exemplo, alguns motores implementados até então:

- ***V8***: implementado por um time da empresa Google e que é utilizado em seus navegadores, como *Google Chrome* e *Chromium*, que converte a AST diretamente para código nativo.
- ***SpiderMonkey***: o primeiro motor desenvolvido, criado pelo mesmo criador da linguagem *JavaScript*, Brendan Eich, na Netscape e atualmente mantido pela Mozilla e que é utilizado em seus navegadores como *Mozilla Firefox* e *Firefox Nightly*, que converte a AST para

[2] Conforme descrito acima, a etapa de geração do *bytecode* pode variar de acordo com a implementação do motor *JavaScript*. Deixando a decisão de utilizar ou não uma linguagem intermediária para o responsável pela implementação do motor.

Figura 3 – Árvore de Sintaxe Abstrata gerada com base na função *square*



Fonte**:** Elaborada pelo autor.

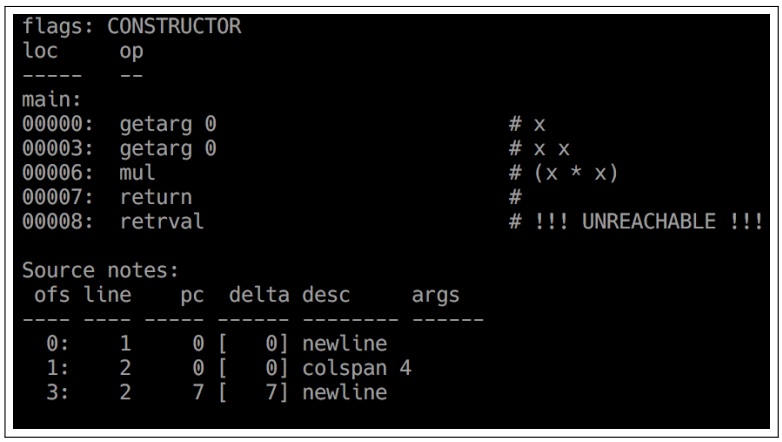
uma linguagem intermediária.

- ***Rhino***: implementado e mantido por um time da empresa Mozilla, que por ser escrito utilizando a linguagem *Java*, adiciona a etapa de tradução do código *JavaScript* para classes *Java*.
- ***Chakra***: implementado e mantido pela Microsoft, e que é utilizado no navegador *Microsoft Edge*, que converte a AST para uma linguagem intermediária.

Códigos escritos em forma de *bytecode* são formas canônicas de representação de códigos que são utilizados pelos motores *JavaScript*, são representações intermediárias, que são projetadas para obter uma execução eficiente através de um *software* interpretador. Na Figura 4, pode-se verificar o *bytecode* gerado, utilizando o motor *SpiderMonkey*, para a função *square* utilizada como referência.

Após o *bytecode* ter sido gerado, passa-se para a etapa de interpretação e execução da representação intermediária, que é feita processando uma instrução por vez percorrendo o *bytecode* gerado. Quando o *bytecode* chega até a área de execução, observa-se a existência

Figura 4 – Informações do *bytecode* gerado com base na função *square*



Fonte**:** Elaborada pelo autor.

Nota**:** Informações obtidas utilizando a ferramenta *JavaScript Shell* fornecida pelo motor *SpiderMonkey*, usando internamente a função *dis* da ferramenta.

de dois componentes importantes para a performance de um motor *JavaScript*, que são o JIT (*Just-in-Time Compiler*) e o Coletor de Lixo (*Garbage Collector*).

JIT, ou *Just-in-Time compiler*, é o responsável por fazer a tradução de *bytecode* para código de máquina durante a execução do programa, ou seja, assim que um trecho de código é solicitado, o JIT faz a transformação do *bytecode* para código de máquina, referente ao bloco solicitado e o disponibiliza para execução, fazendo com que a disponibilização do resultado seja rápida e não consuma ciclos de processamento desnecessários, entregando somente os blocos que serão utilizados no momento em que forem solicitados.

O outro componente importante é o Coletor de Lixo, que é a área responsável pelo gerenciamento automático da memória que é ocupada por objetos, sendo denominados como "lixo"as posições de memória que estão sendo ocupadas com informações referentes ao programa em execução, mas que não são relevantes ou que não estão sendo utilizadas.

Ambos os componentes comentados anteriormente, JIT e Coletor de Lixo, são executados quando existe uma solicitação de um trecho de código *JavaScript*, e estão disponíveis durante toda a execução do motor *JavaScript*, por isso estão diretamente ligados ao passo de execução dentro do processo de interpretação de um código *JavaScript*.

Entretanto, no V8, que não utiliza uma representação intermediária, esse processo é um pouco diferente. Ele utiliza dois compiladores, que podem ser denominados como o compilador simples e o otimizador. O compilador simples é um compilador que não tem foco em otimização, apesar de ainda assim realizar algumas, tem como principal objetivo produzir código nativo tão rápido quanto possível, o que é importante para manter os tempos de carregamento

de uma página rápidos. *Crankshaft*, e o atual *TurboFan*, são exemplos de compiladores com foco em otimização, também conhecidos como otimizadores. O *V8* compila todo o código com o compilador simples e em seguida utiliza um *profiler*[1] interno para selecionar as funções a serem otimizadas, na maioria dos casos são as funções mais acessadas ou com maior custo de processamento, pelo otimizador.

2.2 *WEBASSEMBLY*

De acordo com a definição encontrada na especificação, ainda em desenvolvimento, do *WebAssembly*, é descrito que ele é um formato de código seguro, portátil e de baixo nível projetado para uma execução eficiente e que possui uma representação compacta. O objetivo principal é permitir que aplicações de alto desempenho sejam executados na *web*, mas não faz quaisquer pressupostos específicos da *web* ou requer quaisquer recursos específicos da *web*, sendo assim pode ser empregado em outros ambientes. *WebAssembly* é um padrão aberto desenvolvido por um grupo comunitário da W3C (*World Wide Web Consortium*) [2] que inclui representantes de todos os principais fornecedores de navegadores. A especificação descreve a versão 1.0 do padrão central do *WebAssembly*, e pretende-se que seja substituído por novos lançamentos incrementais com recursos adicionais no futuro. (GROUP, 2017)

WebAssembly é a primeira solução para a execução de código de baixo nível na *web* que oferece todos os objetivos abaixo. É o resultado de uma colaboração sem precedentes entre os principais fornecedores de navegadores e um grupo comunitário on-line para construir uma solução comum para aplicativos de alto desempenho. (HAAS *et al.*, 2017)

Segundo a atual especificação, o *WebAssembly* possui alguns objetivos principais em sua modelagem:

- Semântica rápida, segura e portátil:
  - **Rápido**: execução com o desempenho próximo do código nativo, aproveitando as capacidades comuns a todos os *hardwares* contemporâneos.
  - **Seguro**: código validado e executado em um ambiente autocontido e seguro no que se refere a memória, impedindo a corrupção de dados ou violações de segurança.
  - **Bem definido**: define de forma completa e precisa programas válidos e seu compor-

[1] *Profiling* é uma forma de análise de código dinâmica, que mede por exemplo, a complexidade espacial ou temporal de um algoritmo específico, ou a frequência e duração de chamadas de funções. Comumente, um *profiler* serve para auxiliar a otimização de aplicações, selecionando quais trechos de código darão um maior desempenho se otimizadas.

[2] <https://www.w3.org/community/webassembly/>

tamento, de forma que seja fácil argumentar formalmente e/ou informalmente.

  - **Independente de *Hardware***: código pode ser compilado em todas as arquiteturas modernas, dispositivos *desktop* ou móveis e em sistemas integrados.
  - **Independente de linguagem**: não privilegia nenhuma linguagem, modelo de programação ou modelo de objeto específico.
  - **Independente de plataforma**: pode ser incorporado em navegadores, pode ser executado como uma VM autônoma ou integrado em outros ambientes.
  - **Aberto**: programas podem interoperar com seu ambiente de forma simples e universal.
- Representação eficiente e portátil:
  - **Compacto**: possui um formato binário que é mais rápido de ser transmitido, sendo menor que os formatos de texto típicos ou os formatos de código nativo.
  - **Modularizado**: aplicações podem ser divididas em partes menores que podem ser transmitidas, armazenadas em *cache* e consumidas separadamente.
  - **Eficiente**: pode ser decodificado, validado e compilado em uma única passagem rápida, igualmente como é feito com compilação *just-in-time* (JIT) ou *ahead-of-time* (AOT).
  - **Fluido**: permite que os processos de decodificação, validação e compilação sejam iniciados o mais rápido possível, antes de todos os dados serem recebidos.
  - **Paralelizável**: permite que os processos de decodificação, validação e compilação sejam divididos em muitas tarefas paralelas independentes.
  - **Portável**: não faz nenhum pressuposto de arquitetura que não seja amplamente suportada por um *hardware* moderno.

Além disso, o *WebAssembly* também se destina a ser fácil de inspecionar e depurar, especialmente em ambientes como navegadores *web*, mas esses recursos não serão tratados na especificação atual. (GROUP, 2017)

Antes de prosseguir, é necessário que se tenha conhecimento de alguns conceitos-chave sobre *WebAssembly*, pois serão necessários para entender seu fluxo de processamento posteriormente:

- **Módulo**: aplicações *WebAssembly* são organizadas em módulos, que são a unidade básica de implantação, carregamento e compilação. Um módulo contém definições de tipos, funções, tabelas, memórias e variáveis globais. Além disso, pode declarar importações

e exportações e fornecer lógica de inicialização na forma de segmentos de dados[1] e elementos[2] ou uma função de início.

- **Funções**: o código interno de um módulo é organizado em funções individuais. Cada função possui uma sequência de valores como parâmetros e retorna uma sequência de valores como resultado, conforme definido pelo seu tipo de função.
- **Instruções**: o processamento de um código *WebAssembly* é baseado em uma máquina de pilha. O código de uma função consiste em uma sequência de instruções que manipulam valores em uma pilha de operandos implícita, desempilhando valores de argumentos e empilhando valores de resultados. No entanto, graças ao sistema de tipos, o leiaute da pilha de operandos pode ser determinado de forma estática em qualquer ponto do código, de modo que as implementações reais possam compilar o fluxo de dados entre as instruções diretamente sem nunca materializar a pilha de operandos. A organização da pilha é meramente uma maneira de conseguir uma representação compacta do programa, foi escolhido esse modelo pois se mostrou menor do que uma máquina registradora.
- **Memória**: o armazenamento principal de um módulo *WebAssembly* é um grande vetor de *bytes*, funcionando com uma memória linear. O tamanho do vetor sempre é um múltiplo do tamanho da página, páginas são subdivisões da memória para permitir-lhe uma utilização mais eficiente. Os *bytes* podem ser manipulados através de instruções de memória, a execução de um segmento de dados, ou por meios externos fornecidos por quem está acoplando o *WebAssembly*, um motor *JavaScript* ou um sistema operacional por exemplo, que pode ser chamado de *embedder*.
- **Exceções**: algumas instruções podem produzir um erro inesperado, que imediatamente aborta o processamento atual. As exceções atualmente não podem ser manipuladas por um código *WebAssembly*, mas o *embedder* normalmente fornecerá meios para lidar com essa condição. Se um código *WebAssembly* for utilizado por um motor *JavaScript* e ocorrer um erro interno, será lançada uma exceção *JavaScript* contendo um *stacktrace* com uma pilha de chamadas *JavaScript* e *WebAssembly*, que poderá ser manipulado por um código *JavaScript*.

---

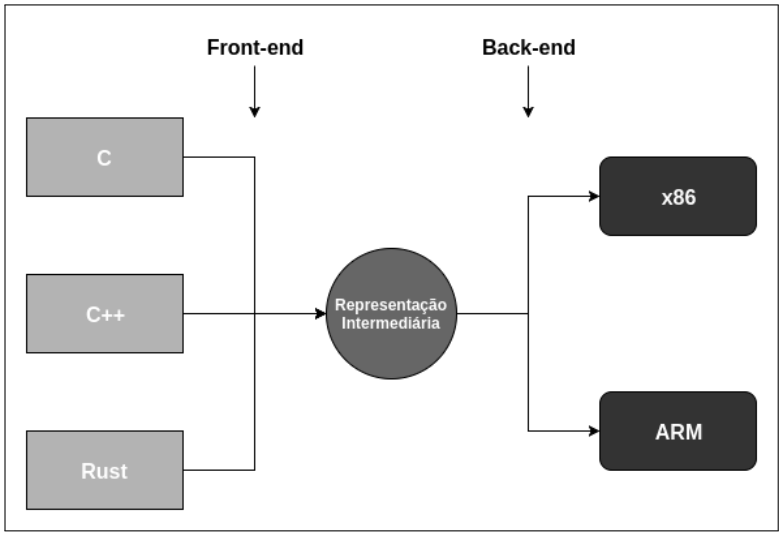
[1] O conteúdo inicial de uma memória são *bytes* "zero". O componente de dados de um módulo, *data*, define um vetor de dados que inicializa um intervalo de memória em um determinado deslocamento com um vetor estático de *bytes*.

[2] O conteúdo inicial de uma tabela não está inicializada. O componente de elementos de um módulo, *elem*, define um vetor de elementos que inicializa um intervalo de uma tabela em um determinado deslocamento com um vetor estático de elementos.

2.2.1 Compilador

Quando se envia um código para ser executado na máquina de um usuário na *web*, não se sabe qual será a arquitetura de destino em que o código será executado. Então, o *WebAssembly* é um pouco diferente de outros tipos de *assembly*. É um idioma de máquina para uma máquina conceitual, e não para uma máquina física real, pode-se dizer então que instruções *WebAssembly* são instruções virtuais. Essas instruções têm um mapeamento muito mais direto para o código de máquina do que o código-fonte *JavaScript*. Eles representam uma espécie de interseção do que pode ser feito de forma eficiente em todo o *hardware* popular comum, mas eles não são mapeamentos diretos para o código de máquina específico de um *hardware* específico. Pode-se ver isso na Figura 5.

Figura 5 – Processo de compilação de um código *WebAssembly*



Fonte**:** Elaborada pelo autor.

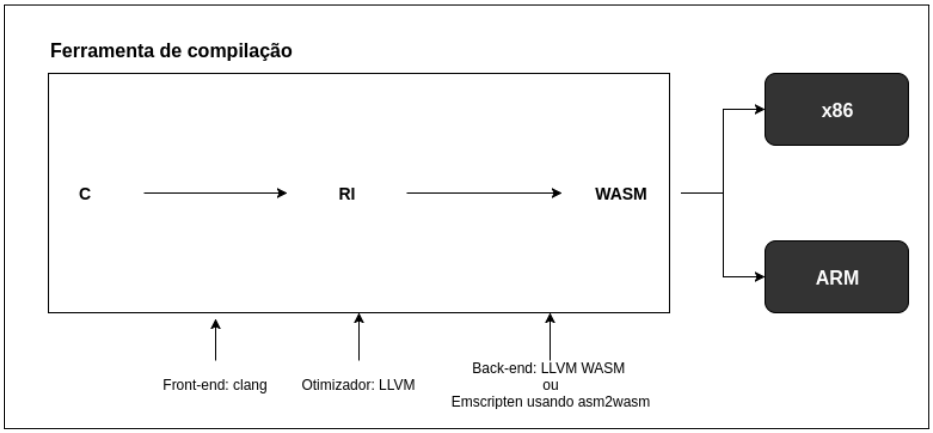
Até então, o projeto que possui maior suporte a *WebAssembly* é o LLVM, *Low Level Virtual Machine*, que é uma infraestrutura de compilação, em que, segundo Lattner, fornece componentes modulares e reutilizáveis para a construção de compiladores, que auxiliam na redução de custo e tempo na criação de um compilador particular. Essa infraestrutura possui uma representação intermediária própria, além de diversas bibliotecas e componentes com interfaces bem definidas e ferramentas construídas pelas próprias bibliotecas. LLVM fornece componentes totalmente independentes de linguagem e máquina alvo, permitindo assim, que algoritmos de diferentes linguagens possam ser conectados e otimizados em conjunto. (LATTNER, 2006)

LLVM é baseado na forma de atribuição única estática, em inglês *Static Single*

Assignment form (SSA), que conforme Lattner e Adve, fornece segurança de tipo, operações de baixo nível, flexibilidade e a capacidade de representar todas as linguagens de alto nível de forma simples. A forma SSA é uma propriedade de uma representação intermediária em que cada variável é atribuída exatamente uma vez, e cada variável é definida antes de ser utilizada. (LATTNER; ADVE, 2017)

Supondo que queira-se ir de *C* para *WebAssembly*, pode ser utilizado o *front-end* clang[1] para passar de *C* para a representação intermediária do LLVM. Uma vez que se está na RI do LLVM, ele o entende, e então pode executar algumas otimizações. Para ir da RI do LLVM para o *WebAssembly*, é preciso um *back-end*. Existe um que está atualmente em andamento no projeto LLVM, e está com a sua maior parte implementada e deve ser finalizado em breve. No entanto, ainda não é possível utilizá-lo. Existe outra ferramenta chamada *Emscripten*, que possui o seu próprio *back-end* que pode produzir *WebAssembly* compilando para outra representação, denominada *asm.js*, e em seguida, convertendo-o para *WebAssembly*. Pode-se ver isso na Figura 6.

Figura 6 – Ferramenta de compilação



Fonte**:** Elaborada pelo autor.

2.2.2 Representação Intermediária

Para permitir que o *WebAssembly* seja lido e editado mais facilmente, existe uma representação textual, além da representação binária (*WASM*) citada anteriormente, o *WAST*. É um formato intermediário projetado para ser utilizado em editores de texto, ferramentas de desenvolvimento, e até mesmo em navegadores.

[1] <https://clang.llvm.org>

Em ambos os formatos, binário e textual, a unidade fundamental de um código *WebAssembly* é um módulo. No formato de texto, um módulo é representado como uma grande expressão em S. As expressões *S* são um formato textual muito antigo e muito simples para representar árvores, e assim pode-se pensar em um módulo como uma árvore de nós que descrevem a estrutura do módulo e seu código. Ao contrário da árvore de sintaxe abstrata de uma linguagem de programação, a árvore de um código *WebAssembly* é bastante plana, principalmente composta por listas de instruções. Pode-se ver um exemplo no Código-fonte 2.

Código-fonte 2 – Exemplo de um módulo na representação intermediária textual

```
1 (module
2   (func (param $lhs i32) (param $rhs i32) (result i32)
3     get_local $lhs
4     get_local $rhs
5     i32.add))
```

Todo o código em um módulo *WebAssembly* é agrupado em funções, que possuem a seguinte estrutura de pseudo-código:

Código-fonte 3 – Estrutura de uma expressão *S*

```
1 ( func <signature> <locals> <body> )
```

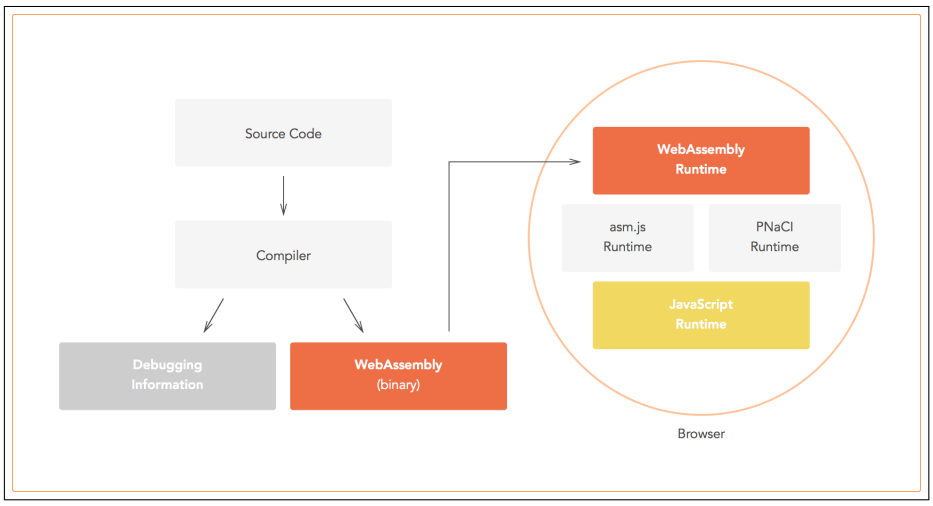
- ***Signature***: declara o que a função requer (parâmetros) e retorna (valores de retorno).
- ***Locals***: são definições de variáveis com tipos explícitos declarados.
- ***Body***: é apenas uma lista linear de instruções de baixo nível.

O formato textual também pode ser utilizado para simplificar a criação de um compilador de uma linguagem específica para *WebAssembly*, pois basta o novo compilador suportar a representação textual e posteriormente utilizar um outro compilador que utilize essa representação textual para a representação binária.

2.2.3 Fluxo de processamento de um código *WebAssembly*

O esquema lógico apresentado na Figura 7 mostra o fluxo de processamento de um código *WebAssembly*, desde a compilação até sua execução, destacando as etapas mais significativas para a explicação.

Figura 7 – Esquema lógico sequencial de processamento de um código *WebAssembly*



Fonte: Peyrott

Iniciando o processamento do código, conforme mostrado no esquema acima, será utilizada a mesma função utilizada na seção anterior, alterando apenas a linguagem, *C*, que executa o cálculo do quadrado de um número e retorna o resultado desse cálculo, como prova de conceito para mostrar passo a passo os resultados de cada etapa do processo realizado pelo motor *JavaScript*. O trecho de código pode ser visto no Código-fonte 4.

Código-fonte 4 – *Square* escrito em *C*

```
1 int square(int n) {
2   return n * n;
3 }
```

Se for adicionado nesse processo a etapa de compilar a função *square*, escrita em *C*, para a representação intermediária textual de *WebAssembly*, serão obtidos como saída as expressões *S* exibidas na Figura 8.

A próxima etapa é compilar da representação intermediária textual para a representação intermediária binária, que é o formato que será utilizado durante a execução pelo motor JavaScript. Pode-se observar a saída dessa etapa, onde é exibida a representação binária da função *square* na Figura 9.

O último passo do processamento é carregar o módulo e executar a função *square*. Para isso será utilizado o código descrito no Código-fonte 5 que é responsável por carregar, compilar e instânciar o módulo referente a essa função.

Figura 8 – Representação intermediária textual da função *square*

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "square" (func $square))
  (func $square (param $0 i32) (result i32)
    (i32.mul
      (get_local $0)
      (get_local $0)
    )
  )
)
```

Fonte**:** Elaborada pelo autor.

Nota**:** Expressões *S* que representam a função *square*.

Figura 9 – Representação intermediária binária da função *square*

```
> $ hexdump -C square.wasm
00000000  00 61 73 6d 01 00 00 00  01 86 80 80 80 00 01 60  |.asm...........`|
00000010  01 7f 01 7f 03 82 80 80  80 00 01 00 04 84 80 80  |................|
00000020  80 00 01 70 00 00 05 83  80 80 80 00 01 00 01 06  |...p............|
00000030  81 80 80 80 00 00 07 97  80 80 80 00 02 06 6d 65  |..............me|
00000040  6d 6f 72 79 02 00 0a 5f  5a 36 73 71 75 61 72 65  |mory..._Z6square|
00000050  69 00 00 0a 8d 80 80 80  00 01 87 80 80 80 00 00  |i...............|
00000060  20 00 20 00 6c 0b                                 | . .l.|
00000066
```

Fonte**:** Elaborada pelo autor.

Nota**:** Imagem gerada utilizando a ferramenta *hexdump* que exibe o conteúdo do arquivo utilizando notação hexadecimal.

Código-fonte 5 – Carregamento e execução da função *square*

```
1 fetch(url)
2     .then(response => response.arrayBuffer())
3     .then(bytes => WebAssembly.instantiate(bytes, {}))
4     .then(response => {
5         const functions = response.instance.exports;
6         console.log(functions.square(10));
7     });
```

Na linha 1 é utilizada a função "fetch"fornecida pelo motor *JavaScript* que carrega o arquivo descrito, neste exemplo seria "square.wasm", e retorna um *promise* com o resultado. Na linha 2, é feita a leitura do conteúdo do arquivo e é retornado um novo *promise* com o

conteúdo na forma de um vetor de *bytes*. Na linha 3, é utilizada a função *instantiate* do objeto *WebAssembly* fornecida pelo motor *JavaScript*, que compila o código fornecido para código de máquina e retorna um objeto com dois campos, sendo o primeiro campo o módulo representando o módulo *WebAssembly* compilado e o segundo campo sendo uma instância desse módulo que contém todas as funções exportadas por esse mesmo módulo *WebAssembly*. Vale ressaltar que nessa etapa podem ser lançadas exceções caso ocorra algo inesperado durante a compilação, um exemplo disso pode ser erros de tipo. Nas linhas 5 e 6, é utilizado o campo *exports* da instância do módulo que contém todas as funções exportadas e é executada a função *square*. Após o final da execução o valor 100 é exibido no *console* do navegador, através da função *log* do objeto *console* fornecida pelo motor *JavaScript*.

3 DESENVOLVIMENTO

Esta seção se dedica a descrever como realizar uma implementação utilizando *WebAssembly*. Para tanto, foi definida uma prova de conceito que consiste em uma aplicação simples com a finalidade de mensurar a performance entre *WebAssembly* e *JavaScript*, para que posteriormente seja possível realizar uma análise comparativa entre os tempos de execução de algoritmos escritos em *WebAssembly* e *JavaScript* utilizando o método *StopWatch*.

Essa aplicação foi construída utilizando as três principais linguagens suportadas pelos navegadores até então: *HTML*, *CSS* e *JavaScript*. Entretanto para a análise de desempenho foram construídos alguns algoritmos utilizando as linguagens *JavaScript* e *C*, para que pudesse ser feita a comparação entre ambas as execuções. Todos os testes foram realizados utilizando o navegador *Google Chrome*, que utiliza o motor *JavaScript V8*, e os algoritmos escritos em *C* foram compilados para a representação binária de *WebAssembly* utilizando o compilador *Emscripten*.

Para a análise de desempenho foi utilizado o método *StopWatch* que consiste em medir o tempo decorrido entre o início e fim de uma tarefa. Para facilitar a compreensão deste método, um exemplo de utilização desta técnica pode ser visto no Código-fonte 6.

Código-fonte 6 – Exemplo de utilização da técnica *StopWatch*

```
1 const startTime = performance.now();
2 task();
3 const endTime = performance.now();
4 const elapsedTime = endTime - startTime;
```

No código acima, pode-se ver que primeiro é obtido o tempo de início da tarefa antes de executá-la, utilizando o método *now* do objeto *performance*, fornecido pelo motor JavaScript, que retorna o tempo decorrido desde a criação do contexto de uma página, em milissegundos. Logo em seguida, é executada a tarefa, através da função *task*, no exemplo citado. Posteriormente, é obtido o tempo final da tarefa, utilizando novamente o método *now* do objeto *performance*, e logo depois, é feita a subtração desses valores, obtendo então o tempo decorrido da execução da tarefa em questão.

3.1 EXECUÇÃO DOS ALGORITMOS

Para a execução dos testes, foram selecionados 3 algoritmos, eles foram utilizados para realizar uma análise simples sobre a execução de códigos escritos em *JavaScript* e *WebAssembly*. São eles:

- N-ésimo termo da sequência *Fibonacci*
- *ShellSort*
- *QuickSort*

Para a execução dos algoritmos foram criados alguns objetos, com funções que simplificam a manipulação dos algoritmos durante suas execuções. Entretanto, serão explicados mais profundamente apenas os três principais objetos, selecionados devido sua importância para a aplicação. Os objetos são:

- *Benchmark*
- *Arrays*
- *Runner*

O primeiro objeto, *Benchmark*, foi o responsável por implementar a técnica *StopWatch* na aplicação, foi através dele que foi obtido o tempo decorrido de cada algoritmo executado. Entretanto, ele não se limita apenas ao tempo de execução, sendo responsável também por informar o resultado de cada algoritmo. A implementação desse objeto pode ser vista no Código-fonte 7.

Código-fonte 7 – Objeto *Benchmark* utilizado na execução dos testes

```
1  const Benchmark = {
2      run: (callback) => {
3          const startTime = performance.now();
4          const result = callback();
5          const endTime = performance.now();
6
7          return {
8              result,
9              time: (endTime - startTime).toFixed(5)
10         }
11     }
12 };
```

O objeto *Benchmark*, possui apenas uma única função, que recebe como parâmetro o algoritmo a ser executado e retorna dois valores: o tempo decorrido e o resultado dessa tarefa.

O segundo objeto, *Arrays*, foi utilizado para realizar uma validação simples no resultado de cada algoritmo: verificar se o resultado obtido através da implementação em *JavaScript* e *WebAssembly* foi o mesmo. Esse objeto fazendo apenas essa checagem, foi responsável por verificar quaisquer falha de manipulação de memória que pudesse nos retornar um resultado falso positivo. A implementação desse objeto pode ser vista no Código-fonte 8.

Código-fonte 8 – Objeto *Arrays* utilizado na execução dos testes

```
1 const Arrays = {
2     isDifferent: (a, b) => (a && !b)
3         || (!a && b)
4         || (!a && !b)
5         || (a.length !== b.length)
6         || a.some((element, index) => element !== b[index])
7 };
```

O objeto *Arrays* possui apenas uma única função, assim como o objeto *Benchmark*, que aceita dois vetores como parâmetro e retorna um valor do tipo *Boolean*, informando se os dois vetores possuem os mesmo valores ou não.

Da linha 2 a linha 4 são realizadas as validações mais simples, onde é verificado se um dos dois vetores não possui valor, sendo retornado o valor verdadeiro em caso positivo e falso em caso negativo. Como *JavaScript* possui avaliação de curto-circuito, em que os argumentos posteriores são avaliados apenas se os argumentos anteriores não forem suficientes para determinar o valor da expressão, caso seja informado algum vetor vazio, por exemplo, o motor já identificará nas primeiras condições que os vetores são diferentes. Na linha 5, é verificado se o tamanho desses vetores é diferente, pois caso sejam, os vetores em si já serão identificados como diferentes pois não possuem o mesmo número de elementos. Na linha 6, é utilizado o método *some* para checar elemento por elemento de ambos os vetores, caso seja identificado em algum índice que os elementos dos vetores são diferentes a iteração irá parar e será retornado o valor verdadeiro, caso contrário será retornado o valor falso.

O terceiro objeto, *Runner*, foi o responsável por implementar todo o fluxo de execução dos algoritmos, manipulando memória quando necessário, gerando os vetores iniciais que

foram utilizados pelos algoritmos de ordenação, verificando o resultado de cada algoritmo e calculando a relação entre o tempo decorrido entre os algoritmos implementados em *WebAssembly* e em *JavaScript*. A implementação desse objeto pode ser vista no Código-fonte 9.

Código-fonte 9 – Objeto *Runner* utilizado na execução dos testes

```
1  const Runner = {
2      runFibonacci: (functions, value) => {
3          ...
4      },
5      runQuickSort: (functions, value) => {
6          ...
7      },
8      runShellSort: (functions, value) => {
9          ...
10     }
11 };
```

O objeto *Runner* diferente dos dois objetos anteriores, possui três métodos, cada um responsável pela execução de um algoritmo específico. Dada a importância da compreensão desses métodos, serão explicados cada um deles individualmente a seguir.

Em cada uma das funções de execução, os parâmetros representam os mesmos objetos, alterando apenas seus significados de acordo com o algoritmo analisado.

O primeiro parâmetro representa um objeto simples que possui três atributos:

- ***module***: um objeto que representa a instância do módulo *WebAssembly* relacionado ao algoritmo em questão.
- ***wasm***: uma função escrita em *WebAssembly* relacionada ao algoritmo a ser analisado.
- ***js***: uma função escrita em *JavaScript* relacionada ao algoritmo a ser analisado.

O segundo parâmetro representa um valor numérico, tendo significados distintos de acordo com a função de execução.

Esta seção se dedica apenas a explicar a execução de cada um dos algoritmos propostos, e não os próprios algoritmos em si, com a finalidade de fornecer todos os detalhes sobre toda a manipulação necessária para a execução dos mesmos, mais detalhes de implementação de cada uma das duas abordagens, *JavaScript* e *WebAssembly*, serão explicados e analisados na próxima seção.

Código-fonte 10 – Função de execução do algoritmo *Fibonacci*

```
1  function runFibonacci(functions, value) {
2      const wa = Benchmark.run(() => functions.wasm(value));
3      const js = Benchmark.run(() => functions.js(value));
4
5      if (wa.result !== js.result) {
6          return;
7      }
8
9      const jsTime = js.time, waTime = wa.time;
10     const ratio = (jsTime / waTime).toFixed(5);
11     return {term: value, jsTime, waTime, ratio};
12 }
```

Iniciando a execução do primeiro algoritmo, seu objetivo é encontrar o n-ésimo termo da sequência *fibonacci*, sendo o termo a ser encontrado representado pelo segundo parâmetro da função.

Analisando o código descrito no Código-fonte 10, pode-se ver que nas linhas 2 e 3, são executados ambos os algoritmos, utilizando o objeto *Benchmark* já mencionado anteriormente, em que são obtidos os seus resultados e tempos de execução. Na linha 5 é verificado se os resultados obtidos são diferentes, retornando um valor nulo caso essa regra seja obedecida, caso contrário a execução do método continua. Na linha 10, é calculada a relação entre os tempos de execução entre os algoritmos escritos em *JavaScript* e *WebAssembly*. Na linha 11, são retornados os valores obtidos de ambas as execuções.

Código-fonte 11 – Função de execução do algoritmo *ShellSort*

```
1  function runShellSort(functions, value) {
2      const a = new Float64Array(value);
3      const b = new Float64Array(value);
4
5      for (let i = 0; i < value; i++) {
6          a[i] = b[i] = Math.random() * 20000 - 10000;
7      }
```

```
8
9       const bytes = 8
10          , end = a.length - 1
11          , pointer = functions.module._malloc(a.length * bytes)
12          , offset = pointer / bytes;
13
14      functions.module.HEAPF64.set(a, offset);
15      const wa = Benchmark.run(() => functions.wasm(pointer, value))
            ;
16      a.set(functions.module.HEAPF64.subarray(offset, offset + end +
            1));
17      functions.module._free(pointer);
18
19      const js = Benchmark.run(() => functions.js(b, value));
20
21      if (Arrays.isDifferent(a, b)) {
22          return;
23      }
24
25      const jsTime = js.time, waTime = wa.time;
26      const ratio = (jsTime / waTime).toFixed(5);
27      return {value, jsTime, waTime, ratio};
28  }
```

ShellSort é um algoritmo de ordenação, de complexidade quadrática, que na aplicação citada anteriormente, tem como objetivo colocar os elementos dos vetores em ordem ascendente, ou seja, do elemento de menor valor para o de maior valor, sendo o tamanho do vetor representado pelo segundo parâmetro da função.

Analisando o código descrito no Código-fonte 11, pode-se ver que da linha 2 a linha 7 são gerados dois vetores de mesmo tamanho e mesmo valores, que serão utilizados posteriormente. Na linha 9, foi criada uma variável representando a quantidade de *bytes* ocupados por um tipo flutuante, em *WebAssembly* tipos flutuantes podem ser de 32 ou 64 *bits*, e como no algoritmo escrito em *C* foi utilizado o tipo *double*, que em *C* representa 64 *bits*, quando for gerada a representação binária do algoritmo, utilizando o *Emscripten*, essas mesmas variáveis

de tipo *double* serão representadas com 64 *bits* em *WebAssembly*, por isso foi atribuído o valor 8 a essa variável representando os 8 *bytes*. Na linha 10, foi criada uma variável para conter o último índice desses vetores, que será utilizado posteriormente. Na linha 11, é utilizado o método *_malloc* para alocar a memória necessária para os elementos do vetor "a"que será passado para o código *WebAssembly* posteriormente, ocupando 8 *bytes* por elemento do vetor. Na linha 12, é criada uma variável responsável por conter a quantidade de deslocamento de *bytes*, de acordo com a quantidade de *bytes* por elemento do vetor. Na linha 14, é utilizado o campo *HEAPF64* que representa a *heap* responsável por armazenar campos flutuantes de 64 *bits*. Abaixo segue uma tabela de mapeamento, de que *heap* utilizar com base em que tipo de dado:

Tabela 1 – Tabela de mapeamento de *Heap* para o tipo específico

Heap	C++	JavaScript
HEAP8	int8_t	Int8Array
HEAPU8	uint8_t	Uint8Array
HEAP16	int16_t	Int16Array
HEAPU16	uint16_t	Uint16Array
HEAP32	int32_t	Int32Array
HEAPU32	uint32_t	Uint32Array
HEAPF32	float	Float32Array
HEAPF64	double	Float64Array

Fonte**:** Elaborado pelo autor.

Conforme explicado anteriormente, *WebAssembly* utiliza uma memória linear, que pode ser entendida como um simples vetor de elementos. Na tabela acima, pode-se ver que tipo de vetor é utilizado em *JavaScript* na implementação do *WebAssembly* para armazenar que tipo de dados. Até o momento, cada tipo de dado possui sua própria *heap*.

Continuando na linha 14, os dados são copiados do vetor para a *heap*, de forma que o código *WebAssembly* possa manipulá-lo. Na linha 15, é executada a função em *WebAssembly* referente ao algoritmo *ShellSort*, passando como parâmetro para a função, a variável representando um ponteiro para o vetor, e o tamanho do vetor. Na linha 16, são obtidos os valores que estão na *heap* e são atribuídos de volta ao vetor "a". Na linha 17, é removido da memória o espaço alocado anteriormente, dado que não será utilizado posteriormente.

Código-fonte 12 – Função de execução do algoritmo *QuickSort*

```
1 function runQuickSort(functions, value) {
2     const a = new Float64Array(value);
3     const b = new Float64Array(value);
```

```
4
5       for (let i = 0; i < value; i++) {
6           a[i] = b[i] = Math.random() * 20000 - 10000;
7       }
8
9       const start = 0
10          , end = a.length - 1
11          , bytes = 8
12          , pointer = functions.module._malloc(a.length * bytes)
13          , offset = pointer / bytes;
14
15      functions.module.HEAPF64.set(a, offset);
16      const wa = Benchmark.run(() => functions.wasm(pointer, start,
            end));
17      a.set(functions.module.HEAPF64.subarray(offset, offset + end +
            1));
18      functions.module._free(pointer);
19
20      const js = Benchmark.run(() => functions.js(b, start, end));
21
22      if (Arrays.isDifferent(a, b)) {
23          return;
24      }
25
26      const jsTime = js.time, waTime = wa.time;
27      const ratio = (jsTime / waTime).toFixed(5);
28      return {value, jsTime, waTime, ratio};
29  }
```

A implementação da função de execução do algoritmo *QuickSort* é semelhante a função do *ShellSort*, tendo como única diferença os parâmetros que são passados para as implementações dos algoritmos, enquanto no *ShellSort* são passados apenas dois parâmetros, que são o ponteiro para o vetor e o tamanho do vetor, no *QuickSort* são passados três parâmetros: o ponteiro representando o vetor, o índice inicial do vetor e o índice final do vetor.

3.2 COMPARAÇÃO

Como descrito na seção anterior, foram selecionados 3 algoritmos para a realização da análise do tempo de execução, sendo estes escritos nas linguagens *JavaScript* e *C*:

- N-ésimo termo da sequência *Fibonacci*
- *ShellSort*
- *QuickSort*

A análise consiste da implementação desses algoritmos em cada linguagem, para a obtenção do tempo de execução de cada um individualmente, utilizando a técnica *StopWatch*, alterando apenas os dados de entrada. Todos os testes foram realizados utilizando o navegador *Google Chrome*, que utiliza o motor *JavaScript V8*, e os algoritmos escritos em *C* foram compilados para a representação binária de *WebAssembly* utilizando a ferramenta *Emscripten*.

O primeiro algoritmo a ser analisado foi o responsável por calcular o n-ésimo termo da sequência *Fibonacci*, para tanto foi utilizada a implementação em *JavaScript* que pode ser vista no Código-fonte 13.

Código-fonte 13 – Algoritmo responsável por calcular o n-ésimo termo da sequência *Fibonacci* em *JavaScript*

```
1 function fibonacci(n) {
2     if (n === 1 || n === 2) {
3         return 1;
4     }
5     return fibonacci(n - 1) + fibonacci(n - 2);
6 }
```

O mesmo algoritmo foi implementado em *C* utilizando recursão e pode ser visto no Código-fonte 14.

Código-fonte 14 – Algoritmo responsável por calcular o n-ésimo termo da sequência *Fibonacci* em *C*

```
1 int fibonacci(int n) {
2   if (n == 1 || n == 2) {
3     return 1;
4   }
```

```
5     return fibonacci(n - 1) + fibonacci(n - 2);
6 }
```

Como foi possível observar nas implementações exibidas nos códigos-fonte 13 e 14, o mesmo algoritmo foi implementado em *JavaScript* e em *C*, fazendo com que fosse possível obter o tempo decorrido de cada execução isoladamente. Os tempos obtidos para cada implementação com base no termo a ser encontrado, pode ser visto na Tabela 2.

Tabela 2 – Tempos de execução do algoritmo que calcula o n-ésimo termo da sequência *Fibonacci*

Termo	JavaScript (ms)	WebAssembly (ms)	Razão (JS / WA)
5	0.11500	0.05500	2.09091
10	0.01500	0.02000	0.75000
15	0.16000	0.01000	16.00000
20	1.20000	0.05000	24.00000
25	0.82000	0.36500	2.24658
30	8.63500	3.13500	2.75439
35	94.79500	32.33500	2.93165
40	1098.46000	360.71000	3.04527
45	11907.52500	4212.73500	2.82655

Fonte**:** Elaborado pelo autor.

Ambiente**:** Teste realizado no navegador *Google Chrome* que utiliza o motor *JavaScript V8*.

Como foi possível perceber, com base na Tabela 2, *WebAssembly* se mostrou mais eficiente que *JavaScript* em quase todos os casos, com exceção da execução do algoritmo que busca pelo décimo termo da sequência *Fibonacci*, em que a execução de *JavaScript* foi de aproximadamente 30% mais eficiente que a execução em *WebAssembly*, entretanto, no melhor caso, *WebAssembly* utilizou apenas aproximadamente 4% do tempo decorrido pelo mesmo algoritmo escrito em *JavaScript*, onde buscou-se pelo vigésimo termo da sequência.

O segundo algoritmo trata-se de um algoritmo de ordenação denominado *ShellSort*, que na aplicação proposta tem como objetivo colocar os elementos de um vetor em ordem ascendente. A implementação feita em *JavaScript* pode ser vista no Código-fonte 15.

Código-fonte 15 – *ShellSort* em *JavaScript*

```
1 function shellSort(array, n) {
2     let h = 1;
3     while (h <= n / 3) {
```

```
4           h = h * 3 + 1;
5       }
6
7       while (h > 0) {
8           for (let i = h; i < n; i++) {
9               let aux = array[i];
10              let j = i;
11
12              while (j > h - 1 && array[j - h] >= aux) {
13                  array[j] = array[j - h];
14                  j -= h;
15              }
16              array[j] = aux;
17          }
18
19          h = (h - 1) / 3;
20      }
21  }
```

A mesma implementação do algoritmo *ShellSort* foi realizada em *C*, e pode ser visto no Código-fonte 16.

Código-fonte 16 – *ShellSort* em *C*

```
1  void shellSort(double *array, int n) {
2    int h = 1;
3    while (h <= n / 3) {
4      h = h * 3 + 1;
5    }
6
7    while (h > 0) {
8      for (int i = h; i < n; i++) {
9        double aux = array[i];
10       int j = i;
11
12       while (j > h - 1 && array[j - h] >= aux) {
```

```
13            array[j] = array[j - h];
14            j -= h;
15          }
16          array[j] = aux;
17        }
18
19        h = (h - 1) / 3;
20      }
21  }
```

O mesmo algoritmo foi implementado em *JavaScript* e em *C*, e foram obtidos os seguintes resultados referentes ao tempo de execução e dados de entrada:

Tabela 3 – Tempos de execução do algoritmo *ShellSort*

Tamanho do vetor	JavaScript (ms)	WebAssembly (ms)	Razão (JS / WA)
10	0.10500	0.10500	1.00000
100	0.06500	0.01000	6.50000
1000	3.64500	0.09500	38.36842
2000	1.88500	0.20500	9.19512
3000	0.38000	0.31500	1.20635
4000	0.52000	0.47000	1.10638
5000	0.82000	0.67000	1.22388
25000	5.00500	4.09500	1.22222
50000	10.26000	8.42000	1.21853

Fonte**:** Elaborado pelo autor.

Vetor**:** Itens do vetor gerados aleatoriamente.

Ambiente**:** Teste realizado no navegador *Google Chrome* que utiliza o motor *JavaScript V8*.

Com base na Tabela 3, pode-se observar que a execução do algoritmo escrito em *WebAssembly* teve performance superior ao escrito em *JavaScript* em quase todos os casos. No pior caso, o algoritmo em *WebAssembly* utilizou aproximadamente o mesmo tempo decorrido pelo algoritmo em *JavaScript*, entretanto, no melhor caso, o algoritmo escrito em *WebAssembly* chegou a utilizar apenas aproximadamente 3% do tempo utilizado pelo algoritmo em *JavaScript*, em que foi utilizado um vetor com 1000 elementos.

O terceiro algoritmo também trata-se de um algoritmo de ordenação, denominado *QuickSort*, que na aplicação proposta tem como objetivo colocar os elementos de um vetor em ordem ascendente. A implementação feita em *JavaScript* pode ser vista no Código-fonte 17.

Código-fonte 17 – *QuickSort* em *JavaScript*

```
function quickSort(array, start, end) {
    if (start >= end) {
        return;
    }

    let pivot = array[end], left = 0, right = 0;
    while (left + right < end - start) {
        let num = array[start + left];
        if (num < pivot) {
            left++;
        } else {
            array[start + left] = array[end - right - 1];
            array[end - right - 1] = pivot;
            array[end - right] = num;
            right++;
        }
    }

    quickSort(array, start, start + left - 1);
    quickSort(array, start + left + 1, end);
}
```

O mesmo algoritmo foi implementado em *C* e pode ser visto no Código-fonte 18.

Código-fonte 18 – *QuickSort* em *C*

```
void quickSort(double *array, int start, int end) {
  if (start >= end) {
    return;
  }

  double pivot = array[end];
  int left = 0, right = 0;
  while (left + right < end - start) {
```

```
9       double num = array[start+left];
10      if (num < pivot) {
11        left++;
12      } else {
13        array[start+left] = array[end-right-1];
14        array[end-right-1] = pivot;
15        array[end-right] = num;
16        right++;
17      }
18    }
19
20    quickSort(array, start, start + left - 1);
21    quickSort(array, start + left + 1, end);
22  }
```

O mesmo algoritmo foi implementado em *JavaScript* e em *C*, e foram obtidos os seguintes resultados referentes ao tempo de execução e dados de entrada:

Tabela 4 – Tempos de execução do algoritmo *QuickSort*

Tamanho do vetor	JavaScript (ms)	WebAssembly (ms)	Razão (JS / WA)
10	0.27000	0.06500	4.15385
100	0.07500	0.01000	7.50000
1000	2.37500	0.09000	26.38889
10000	1.23500	1.06500	1.15962
100000	15.78000	13.14000	1.20091
1000000	179.01000	160.65500	1.11425
10000000	2171.02000	1861.00500	1.16658
20000000	4224.75500	3724.26000	1.13439
30000000	6662.29500	5761.54500	1.15634
40000000	9594.70000	7917.12000	1.21189
50000000	11478.08000	9998.39500	1.14799

Fonte**:** Elaborado pelo autor.

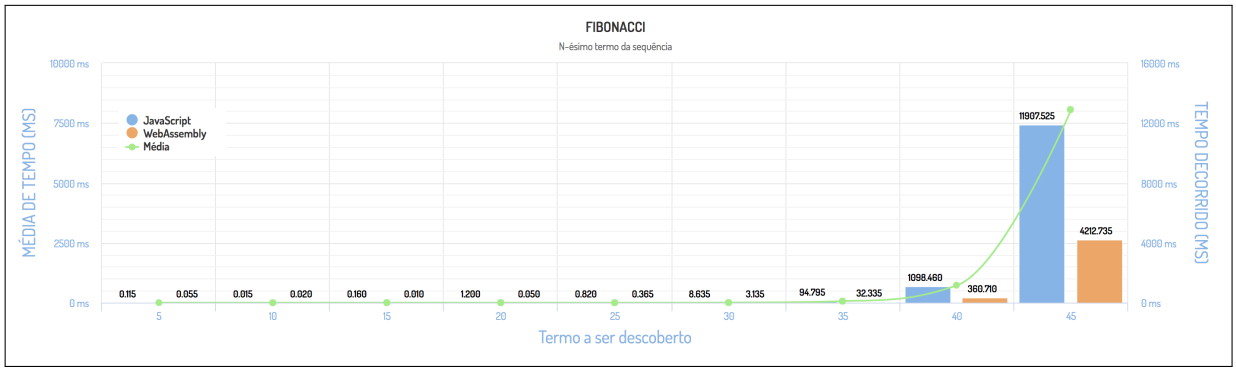
Vetor**:** Itens do vetor gerados aleatoriamente.

Ambiente**:** Teste realizado no navegador *Google Chrome* que utiliza o motor *JavaScript V8*.

Analisando a Tabela 4, pode-se observar que o algoritmo escrito em *WebAssembly* teve tempo de execução superior ao escrito em *JavaScript* em todos os casos, em que na melhor execução utilizou apenas aproximadamente 4% do tempo utilizado pelo mesmo algoritmo escrito em *JavaScript*, onde foi utilizado um vetor com 1000 elementos.

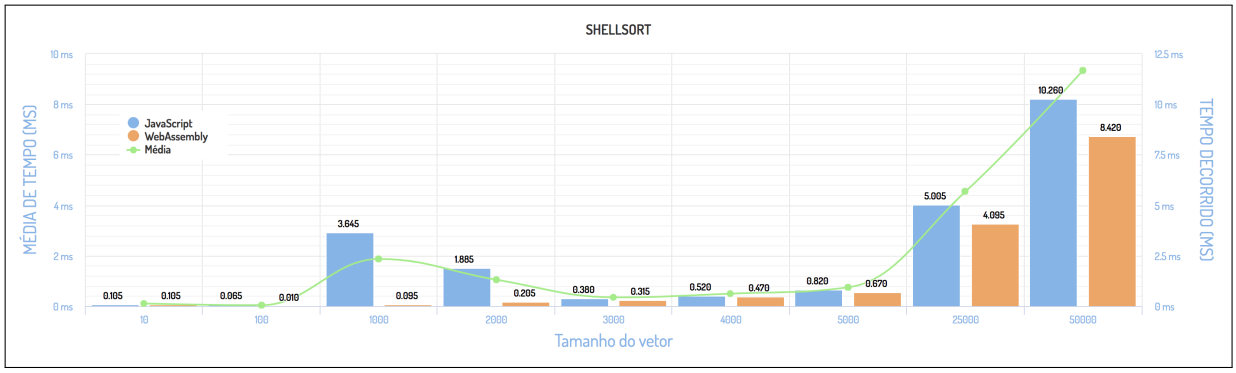
Com base nos resultados obtidos exibidos nas tabelas 2, 3 e 4, foram gerados os gráficos exibidos nas figuras 10, 11 e 12. Esses gráficos possuem 4 dados-chave: os dados de entrada, o tempo utilizado por ambas as implementações para conclusão da tarefa proposta e a média de tempo gasto com base nos tempos de execução obtidos.

Figura 10 – Resultados do algoritmo que descobre o n-ésimo termo da sequência *Fibonacci*



Fonte**:** Elaborada pelo autor.

Figura 11 – Resultados do algoritmo *ShellSort*

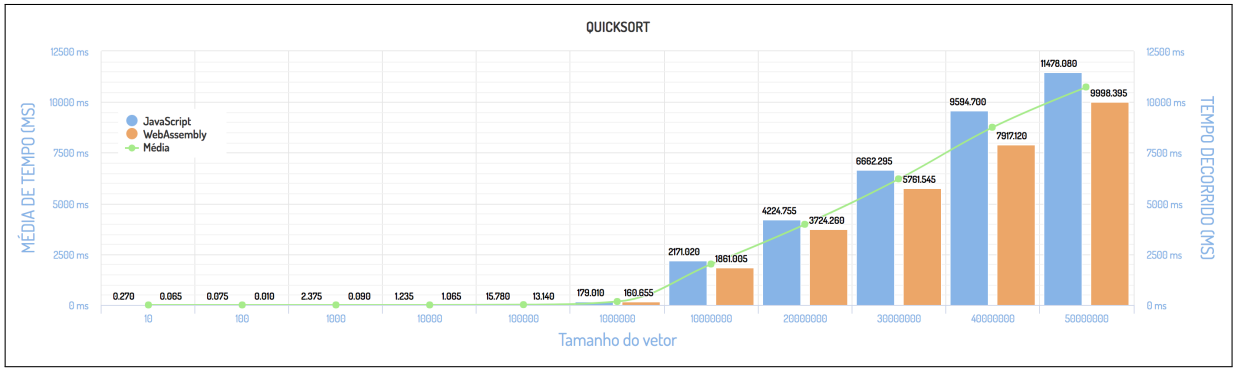


Fonte**:** Elaborada pelo autor.

Com base nos resultados exibidos nas figuras 10, 11 e 12, pode-se observar que em quase todos os casos, os algoritmos escritos em *WebAssembly* tiveram performance superior aos escritos em *JavaScript*, com apenas duas exceções: a primeira, na execução do algoritmo que busca o n-ésimo termo da sequência *Fibonacci*, em que a implementação escrita em *WebAssembly* obteve resultado inferior a implementação escrita em *JavaScript*; e a segunda exceção, obtida na implementação do algoritmo *ShellSort*, em que o algoritmo escrito em *WebAssembly* obteve aproximadamente o mesmo tempo de execução que o algoritmo escrito em *JavaScript*.

Em cada algoritmo testado, houveram execuções que apresentaram disparidade

Figura 12 – Resultados do algoritmo *QuickSort*



Fonte**:** Elaborada pelo autor.

bastante significativa:

- **N-ésimo termo da sequência *Fibonacci***: na busca pelo vigésimo termo da sequência, o tempo utilizado pelo algoritmo escrito em *JavaScript* foi de aproximadamente 2300% superior ao utilizado pelo algoritmo escrito em *WebAssembly*.
- ***ShellSort***: na execução que utilizou um vetor com 1000 elementos, o tempo utilizado pelo algoritmo escrito em *JavaScript* foi de aproximadamente 3700% superior ao utilizado pelo algoritmo escrito em *WebAssembly*.
- ***QuickSort***: na execução que utilizou um vetor com 1000 elementos, o tempo utilizado pelo algoritmo escrito em *JavaScript* foi de aproximadamente 2500% superior ao utilizado pelo algoritmo escrito em *WebAssembly*.

3.3 DISCUSSÃO

De acordo com o que foi apresentado na seção anterior, é possível perceber que o desempenho de um algoritmo escrito em *WebAssembly* é na maioria dos casos, superior ao mesmo algoritmo escrito em *JavaScript*. Isso é devido a sua execução, enquanto um código escrito em *JavaScript* precisa ser transformado em uma Árvore de Sintaxe Abstrata e posteriormente convertido em uma representação intermediária, um código escrito em *WebAssembly* não precisa realizar esses passos, pois ele já é uma representação intermediária, precisando apenas ser decodificado e verificado para garantir que não há erros em sua estrutura interna.

Um código escrito em *JavaScript* é compilado durante sua execução, utilizando o compilador *Just-in-Time*, e dependendo dos tipos que são utilizados em tempo de execução, múltiplas versões do mesmo código precisam ser compilados utilizando o compilador de otimização, para garantir que um bloco de código será otimizado independente dos tipos de dados utilizados

internamente, e isso possui um custo elevado devido ter que monitorar os tipos de dados em execução. Outra observação importante é que os compiladores JIT precisam gerenciar a relação custo-benefício entre tempos de carregamento mais rápidos e tempos de execução mais rápidos. Se for gasto mais tempo compilando e otimizando com antecedência, isso irá acelerar a execução do código, mas irá deixar a inicialização desse mesmo bloco de código mais lento.

Um fator importante a ser notado sobre *WebAssembly*, é que ele já possui uma representação intermediária mais próxima do código de máquina, o que acelera sua execução devido ter que fazer traduções menos complexas de instruções. Outro ponto é que os tipos de dados fazem parte do algoritmo, o que permite novas abordagens, como por exemplo paralelizar o trabalho de compilação e execução.

O tempo de carregamento de um arquivo *WebAssembly* também é inferior ao de um arquivo *JavaScript*, pois possui uma representação binária desenvolvida para ser mais compacta. Além disso, atualizações mais recentes nos motores *JavaScript* permitem iniciar o processo de compilação enquanto os *bytes* do arquivo ainda estão sendo recebidos pelo navegador.

4 CONCLUSÃO

4.1 CONSIDERAÇÕES FINAIS

Este trabalho se dispôs a fazer uma análise de uma tecnologia emergente no mercado, chamada de *WebAssembly*, que tem como principal objetivo permitir que aplicações de alto desempenho sejam executados na *web*, porém não faz quaisquer pressupostos específicos da *web* ou requer quaisquer recursos específicos da *web*, sendo assim pode ser empregado em outros ambientes.

Com base no que foi apresentado neste trabalho, pode-se observar que *WebAssembly* se destaca em relação a performance devido alguns aspectos diretamente ligados a seus objetivos de projeto. Um ponto a ser destacado em relação a *WebAssembly*, é que o mesmo possui uma representação intermediária binária extremamente compacta, o que faz com que sua transmissão seja mais eficiente que um arquivo textual comum, como por exemplo um arquivo *JavaScript*. Além disso, cada arquivo binário representa um único módulo e é dividido em seções, e cada seção é dividida em funções. Isso significa que a latência de carregamento de um arquivo pode ser minimizada, ao iniciar o processo de compilação a medida em que as funções desse arquivo estão sendo recebidas. Junto com essa abordagem, é possível paralelizar o processo de compilação, distribuindo o processamento dessas funções, assim como é feito pelos motores *V8* e *SpiderMonkey*.

Foi analisado neste trabalho, por meio de experimentos, que a execução de um código escrito em *WebAssembly* possui performance superior a um código escrito em *JavaScript* na maioria dos casos, através da execução dos algoritmos: *ShellSort*, *QuickSort* e o algoritmo que busca o n-ésimo termo da sequência *Fibonacci*. O motivo dessa eficiência pode ser compreendida devido diferenças cruciais entre sua execução e a execução de um algoritmo escrito em *JavaScript*. Uma diferença importante entre um algoritmo escrito em *WebAssembly* e *JavaScript*, é que enquanto um algoritmo em *JavaScript* precisa ser analisado, para posteriormente ser gerada a Árvore de Sintaxe Abstrata e em seguida a representação intermediária, para só então possa ser convertido em código de máquina, um algoritmo em *WebAssembly* já é uma representação intermediária, e ainda mais eficiente que a representação gerada para um código *JavaScript* em alguns motores, precisando apenas ser convertido em código de máquina.

4.2 TRABALHOS FUTUROS

Como continuação deste trabalho, deseja-se obter mais métricas de desempenho na execução de cada algoritmo, de forma que se consiga realizar uma análise mais precisa sobre diferenças de performance entre *WebAssembly* e *JavaScript*. Além disso, estabelecer uma metodologia de análise de desempenho voltada a *WebAssembly*, através de uma biblioteca criada com esse propósito, em que seja possível informar as funções a serem testadas e as regras dos dados de entrada e essa biblioteca execute as funções passando os dados corretos obedecendo as regras estabelecidas e gere todas as métricas obtidas com base na execução de cada função isoladamente.

REFERÊNCIAS

ARAúJO, A. C. **WebAssembly**. 2017. <https://developer.mozilla.org/pt-BR/docs/WebAssembly>. Acessado: 03-12-2017.

ECMA. **ECMAScript® 2017 Language Specification**. Rue du Rhone 114, CH-1204 Geneva: Ecma International - European association for standardizing information and communication systems, 2017.

GROUP, W. C. **WebAssembly Specification**. [S.l.]: WebAssembly Community Group, 2017. <https://webassembly.github.io/spec/_download/WebAssembly.pdf>. Acessado: 22-10-2017.

HAAS, A.; ROSSBERG, A.; SCHUFF, D. L.; TITZER, B. L.; GOHMAN, D.; WAGNER, L.; ZAKAI, A.; HOLMAN, M.; BASTIEN, J. Bringing the web up to speed with webassembly. 6 2017.

LATTNER, C. Introduction to the llvm compiler infrastructure. Gelato Itanium Conference and Expo (ICE). 2006. Disponível em: <https://llvm.org/pubs/2006-04-25-GelatoLLVMIntro.pdf>.

LATTNER, C.; ADVE, V. **LLVM Language Reference Manual**. 2017. <https://llvm.org/docs/LangRef.html>. Acessado: 22-11-2017.

MOZILLA. **JSAPI User Guide**. [S.l.]: Mozilla Developer Network, 2017. <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_User_Guide>. Acessado: 21-10-2017.

PEYROTT, S. **7 Things You Should Know About WebAssembly**. [S.l.]: Auth0, 2017. <https://auth0.com/blog/7-things-you-should-know-about-web-assembly/>. Acessado: 22-10-2017.

SETH, G.; FORESTI, A. **Microsoft Edge's JavaScript engine to go open-source**. [S.l.]: Microsoft, 2017. <https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/#UPQmPCbSOAfW8gkK.97>. Acessado: 21-10-2017.

W3SCHOOLS. **Famous Month-by-Month Browser Statistics Since 2002**. [S.l.]: W3Schools, 2017. <https://www.w3schools.com/browsers/>. Acessado: 21-10-2017.

GLOSSÁRIO

A

AST: Abstract Syntax Tree (Árvore de Sintaxe Abstrata).

C

Canônico: Adjetivo que caracteriza aquilo que está de acordo com os cânones, com as normas estabelecidas ou convencionadas.

R

RI: Representação Intermediária.

S

Script: Conjunto de instruções e tarefas a serem seguidas.

SSA: Static Single Assignment form (Forma de Atribuição Única Estática).