UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS EDI CORDEIRO DE BRITO

ANÁLISE COMPARATIVA ENTRE *WEBASSEMBLY* E *JAVASCRIPT* COMO ALVOS DE COMPILAÇÃO

CAMPINA GRANDE - PB

2019

LUCAS EDI CORDEIRO DE BRITO

ANÁLISE COMPARATIVA ENTRE *WEBASSEMBLY* E *JAVASCRIPT* COMO ALVOS DE COMPILAÇÃO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Matheus Gaudencio do Rêgo.

CAMPINA GRANDE - PB

2019

B862a Brito, Lucas Edi Cordeiro de.
Análise comparativa entre WebAssembly e JavaScript como alvos de compilação. / Lucas Edi Cordeiro de Brito. – 2019.

11 f.

Orientador: Prof. Dr. Matheus Gaudencio do Rêgo
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. WebAssembly. 2. JavaScript. 3. Emscripten. 4. Asm.js - Mozila. 5. Navegadores web. 6. Linguagem de programação - web. 7. Aplicativos web – linguagem de programação. I. Rêgo, Matheus Gaudencio do. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

LUCAS EDI CORDEIRO DE BRITO

ANÁLISE COMPARATIVA ENTRE *WEBASSEMBLY* E *JAVASCRIPT* COMO ALVOS DE COMPILAÇÃO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Matheus Gaudencio do Rêgo
Orientador – UASC/CEEI/UFCG

Professor Dr. Fábio Jorge Almeida Morais
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Examinador – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

Análise comparativa entre WebAssembly e JavaScript como alvos de compilação

Trabalho de Conclusão de Curso

Lucas Edi Cordeiro de Brito
lucas.brito@ccc.ufcg.edu.br
Universidade Federal de Campina Grande

Matheus Gaudencio do Rêgo
matheusgr@computacao.ufcg.edu.br
Universidade Federal de Campina Grande

RESUMO

Atualmente, a Web faz parte da vida de várias pessoas, seja sendo utilizada como uma forma de acesso a serviços, quanto como uma plataforma de desenvolvimento acessível e universal. Desde o começo, JavaScript era a única linguagem de programação disponibilizada para desenvolver aplicativos que executam na Web. Com isso, essa linguagem se tornou facilmente um alvo de compilação de outras linguagens de alto nível. Em 2013, a Mozilla criou um subconjunto de JavaScript chamado de asm.js, que é executado de forma mais performática pelo navegador. Porém, essa linguagem não foi padronizada, e apenas alguns navegadores implementaram o suporte para tal. Para resolver isso, em 2017 foi criado o WebAssembly, um formato de instrução binário, feito para ser executado na Web. Dado isso, nos perguntamos se seria mais vantajoso utilizar WebAssembly ou asm.js nos navegadores que suportem ambos. Analisamos o tempo de execução em 8 navegadores, em 3 Sistemas Operacionais distintos, e observamos que WebAssembly é cerca de 2,8 vezes mais rápido que asm.js. Também comparamos o tamanho total do código das duas, e vimos um ganho de espaço de aproximadamente 47% quando utilizamos a versão em WebAssembly, dado sua natureza de ser um formato binário.

PALAVRAS-CHAVE

WebAssembly, JavaScript, Performance, Emscripten, asm.js

1 INTRODUÇÃO

A Web começou apenas como uma rede de troca de documentos e hoje se tornou uma das plataformas mais acessíveis e universais que existe. Estando disponível em diversos tipos de sistemas operacionais e para diferentes tipos de arquiteturas de computadores, é uma das principais formas de desenvolvimento e distribuição de software moderno. Historicamente, JavaScript [8] era a única linguagem de programação disponível para se desenvolver na Web, e com isso ela se tornou facilmente um alvo de compilação de outras linguagens [4]. Projetos como o Emscripten [19] e o Portable Native Client (PNaCl) [17] são alguns exemplos de soluções voltadas à compilação de C e C++ para JavaScript.

O Emscripten é um projeto feito para compilar código diretamente do bitcode do LLVM [9] para JavaScript, mais especificamente para asm.js [1], um subconjunto de JavaScript feito para ser executado mais eficientemente pelo navegador. Ele também disponibiliza várias ferramentas comumente utilizadas nos projetos feitos em C e C++ para facilitar o trabalho com esse ecossistema. Algumas dessas ferramentas são o `emconfigure` e o `emmake`, que servem como *wrappers* para o script `configure` [12] e o programa `make` [13], respectivamente. Elas são úteis para configurar automaticamente as variáveis de ambiente para utilização do compilador do Emscripten. Fora isso, o projeto possui várias adaptações de APIs nativas de C/C++ para se trabalhar na Web. Algumas delas são as APIs do *OpenGL*, que são convertidas para *WebGL*, e as funções de sistemas de arquivos da biblioteca padrão, que são adaptadas para utilizar o *IndexedDB* [2], um banco de dados transacional do navegador capaz de guardar qualquer objeto JavaScript.

Neste contexto surge o WebAssembly (WASM), um novo formato de instruções binárias feito para rodar em uma máquina virtual baseada em pilha, disponível nos principais navegadores. Ele foi feito com o intuito de ser um novo alvo de compilação de linguagens de alto nível, focando inicialmente em linguagens com memória não-gerenciada, como C e C++. Também poderíamos utilizar linguagens que necessitam de um *Garbage Collector* [5], porém isso requer compilar junto todo o *runtime* da linguagem, o que pode aumentar significativamente o tamanho do binário final. O WASM roda na mesma *engine* [6] que o navegador usa para executar o JavaScript, aumentando assim a interoperabilidade entre as duas linguagens e facilitando a troca de contexto. Sendo uma evolução do asm.js, a adaptação de projetos como o Emscripten é feita de forma natural, sendo adicionada apenas mais um alvo de compilação para toda a pipeline já existente. Também existe o esforço para adicionar o WebAssembly como um alvo de compilação nativo no backend do LLVM e atualizar o Emscripten para usar esse backend por padrão [20].

Dado que o Emscripten trabalha diretamente na tradução do bitcode do LLVM, e que ele suporta JavaScript e WebAssembly como alvos de compilação, podemos utilizar uma mesma biblioteca e compilar para ambas linguagens, permitindo a coleta de diferentes métricas sobre os dois programas simultaneamente. Com isso, surge a dúvida se é mais vantajoso compilar o código da biblioteca de C ou C++ para WebAssembly ou para JavaScript. Qual é o mais performático em tempo de execução? Qual possui o menor executável? Ainda, considerando que em um ambiente Web os usuários são bastante sensíveis com a velocidade de carregamento de uma página [3][7], cada byte extra nos arquivos que são baixados pode contar negativamente para a experiência. Esses são os tipos de perguntas que devemos nos preocupar quando estamos desenvolvendo para essa plataforma, e é o que este artigo procura responder.

A avaliação foi realizada com a construção de um aplicativo prova de conceito que utiliza as tecnologias de WebAssembly e

Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original.

JavaScript. Este aplicativo foi feito utilizando o FFMPEG, um software para conversão de vídeos, compilado usando o Emscripten. Coletamos as métricas de tempo de execução da conversão de um vídeo e o tamanho total do executável das duas implementações. Para a nossa coleta, utilizamos 4 navegadores em 3 Sistemas Operacionais distintos. Fizemos as análises sobre as observações coletadas e chegamos a conclusão de que WebAssembly é cerca de 2,8 vezes mais rápido que JavaScript nos ambientes que executamos nossas coletas. O WebAssembly também possui um tamanho de arquivo aproximadamente 47% menor que JavaScript, devido sua natureza de ser um formato binário.

Na próxima Seção iremos apresentar a fundamentação teórica, que contém os principais conceitos necessários para o entendimento das Seções posteriores. Em seguida, apresentamos nossa metodologia, detalhando como foi feita a implementação da prova de conceito utilizada para a coleta de métricas e quais navegadores e Sistemas Operacionais foram utilizados. Posteriormente, apresentamos nossos resultados em conjunto com a discussão, mostrando nossas análises sobre as observações coletadas. Por último, compartilhamos as limitações sobre o trabalho realizado, e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta Seção iremos apresentar os conceitos fundamentais para o melhor entendimento dos tópicos apresentados neste artigo. Começaremos apresentando o navegador e alguns conceitos fundamentais, como seu sistema de *threads*, que serão necessários nas próximas seções. Depois, vamos apresentar o `asm.js` e a motivação para sua criação, e como ele difere do JavaScript tradicional. Em seguida, mostraremos o WebAssembly e como é o seu funcionamento em mais baixo nível, e como o `asm.js` influenciou suas características. Por último, apresentamos pequenos conceitos que ajudarão com o entendimento da nossa metodologia.

2.1 Navegador

O navegador é um software capaz de exibir páginas Web. Uma página Web contém, entre outras coisas, códigos feitos em JavaScript que são baixados e executados em forma de *scripts*. Esses scripts são geralmente executados em um ambiente que chamamos de *Main Thread*. A Main Thread é a forma principal de um script realizar alterações na interface de usuário (UI), porém, essa thread também é utilizada pelo navegador para realizar as operações de cálculo de estilo e *layout*, pintura e composição, que são necessários para a exibição de uma página. Por conta disso, o navegador fica impossibilitado de realizar quaisquer um desses passos enquanto o código do script estiver rodando na Main Thread. Isso pode ser um problema para scripts que executam por muito tempo, visto que, para o usuário, a página fica em um estado não-responsivo e dá a impressão de que o navegador está "congelado".

Para resolver esse problema, os navegadores criaram o que chamamos de *Web Workers*. Cada página Web pode ter diversos Workers executando ao mesmo tempo. Cada Worker desse executa em uma thread separada, que chamamos de Worker Thread. Essas threads são isoladas, e rodam independentemente de cada uma, inclusive da Main Thread. A diferença em utilizar os Workers e a Worker Thread é que eles executam em plano de fundo, ou seja, não impede do navegador de rodar os passos necessários para a exibição da página. Para a execução de uma computação longa, é preferível utilizar Web Workers ao invés de executar o código na Main Thread, a fim de evitar o congelamento da interface.

2.2 asm.js

O asm.js é uma tecnologia que foi projetada pela Mozilla em 2013, e é um subconjunto de JavaScript. Essa tecnologia foi feita primariamente não para ser escrita diretamente pelos desenvolvedores, mas sim como um alvo de compilação de linguagens de alto nível, como C e C++. O asm.js impõe restrições sobre o código a fim de que o navegador consiga utilizar um compilador *ahead-of-time* para uma melhor geração de código, e com uma performance melhor do que JavaScript tradicional. O asm.js também possui um modelo de memória que não necessita de um *Garbage Collector*, fazendo com que o navegador não precise fazer pausas na execução do código para realizar a liberação de recursos, o que pode ter um grande impacto na performance.

A seguir, temos a implementação de um módulo em asm.js, que contém uma função que incrementa uma unidade a um número passado como parâmetro.

```
function module(stdlib, ffi, heap) {
  'use asm';

  function increment(i) {
    i = i|0;
    return (i + 1)|0;
  }

  return {
    increment: increment,
  };
}
```

Código 1: Módulo asm.js

Como podemos ver no Código 1, o módulo asm.js recebe três parâmetros. O primeiro deles é o `stdlib`, que representa a biblioteca padrão do asm.js, a qual é um subconjunto da biblioteca padrão de JavaScript. O segundo parâmetro, `ffi`, representa uma *foreign function interface*, e é um objeto que contêm um conjunto de funções em JavaScript nativo que são disponibilizadas para o módulo asm.js. Por último, temos a `heap`, e como o nome sugere, representa a *heap* do módulo. Essa *heap* é o que possibilita o asm.js de não precisar de um *Garbage Collector*, porém requer que o módulo gerencie toda sua memória, como toda linguagem de alto nível sem memória gerenciada.

2.3 WebAssembly

O WebAssembly é um formato de instrução binária, feito para ser executado em uma máquina virtual baseada em pilha. Ele foi projetado para ser possível a sua execução em um ambiente Web, e para ser um novo alvo de compilação de linguagens de alto nível. Temos apenas 4 tipos de dados no WebAssembly, a fim de simplificar as possíveis instruções que podem ser realizadas, que são inteiros de 32 e 64 bits, e números de ponto flutuante, também de 32 e 64 bits. Ele é o sucessor do asm.js, e suas metas e características

são bastante parecidas, porém o WebAssembly tem a vantagem de utilizar instruções de CPU mais baixo nível do que é possível com JavaScript, como por exemplo popcount e copysign [18].

O WebAssembly possui a vantagem de ser um padrão que foi projetado pelas empresas por trás dos principais navegadores, e que pode ser melhor otimizado de acordo com sua especificação, enquanto asm.js tem apenas uma especificação informal e foi implementado apenas por alguns navegadores. Também será possível a paralelização de instruções com WebAssembly com o suporte a threads, que utilizará as Worker Threads, o que não é possível fazer apenas com JavaScript. A motivação do WebAssembly foi de permitir que os desenvolvedores preencham as falhas da Web como uma plataforma, utilizando módulos que foram feitos em outras linguagens para satisfazer essas demandas, sem sacrificar a performance.

O WebAssembly funciona utilizando uma pilha para guardar os valores durante a execução de uma função. Por exemplo, em uma instrução de adição, a máquina virtual que está executando o código em WebAssembly irá retirar os dois valores que estão no topo da pilha (duas operações de *pop*), irá fazer a soma dos valores, e irá colocar o resultado de volta no topo da pilha (uma operação de *push*). Dessa forma, podemos abstrair todas as operações para apenas fazer manipulações na pilha, e decidir qual instrução executar baseando-se no resultado que está no topo da pilha.

Visto que o WebAssembly foi projetado com o pensamento na Web, ele também possui uma variante em forma de texto, para possibilitar a fácil depuração do código, chamado de WebAssembly Text (WAT). Essa forma em texto utiliza S-Expressions [14] para definir o nome da instrução e seus argumentos, de forma similar às linguagens Closure e Lisp.

Abaixo temos a implementação de um módulo em WebAssembly, que exporta uma função que incrementa em uma unidade um número recebido por parâmetro, da mesma forma que foi visto anteriormente no exemplo em asm.js. Esse módulo está na forma de texto do WebAssembly.

```
(module
  (func $increment (param $i i32) (result i32)
    i32.const 1
    local.get $i
    i32.add)
  (export "increment" (func $increment))
  (type $t0 (func (param i32) (result i32))))
```

Código 2: Módulo WebAssembly

Como podemos ver no Código 2, o módulo em WebAssembly possui uma sintaxe bem diferente de JavaScript. A forma que um módulo é estruturado é bem definida, porém o compilador de WebAssembly Text para WebAssembly possui alguns açúcares sintáticos [16] para facilitar a escrita. Como foi descrito anteriormente, o WebAssembly funciona utilizando uma pilha para guardar as informações pertinentes a execução do código. Por exemplo, as instruções de `i32.const` e `local.get` empurram, respectivamente, um número inteiro de 32 bits e o valor de uma variável local para o topo da pilha. A instrução `i32.add` faz a adição dos dois primeiros valores no topo da pilha, retirando-os e empurrando de volta o resultado da adição.

2.4 LLVM

O projeto do LLVM consiste de compiladores modulares e um conjunto de ferramentas utilitárias. O compilador do LLVM, também referenciado como um *backend* de compilador, é o componente responsável por receber a Representação Intermediária (RI) do código fonte e transformá-lo em código objeto, que é executável pelo computador. Esse compilador também tem a capacidade de realizar a compilação dessa RI para WebAssembly, porém, essa implementação ainda está em fase experimental na época de escrita deste artigo. Essa Representação Intermediária do LLVM também é chamada de bitcode.

2.5 Processamento de Vídeo

Outros conceitos que são necessários para o entendimento desse artigo são *codecs* e *transcoding*. Um codec é um software que é capaz de realizar a codificação e decodificação de um formato "cru", utilizado para a exibição do conteúdo, para um formato mais compacto, utilizado para armazenamento. Exemplos de tipos de codecs de vídeos são MP4 e H264. Softwares de exibição de vídeos usam os codecs para decodificar e transmitir o conteúdo de um arquivo de vídeo, enquanto softwares de edição de vídeo podem utilizar codecs para armazenar o conteúdo vídeo em disco. Transcoding é um termo utilizado nos softwares de conversão de vídeos que define o processo da mudança de um formato de *container* e codec para outro tipo de container e codec. Exemplos de formato de container são Matroska e MPEG-4.

3 METODOLOGIA

Nesta Seção apresentamos a implementação de um aplicativo usado como base para comparar as tecnologias de WebAssembly e JavaScript, assim como sua arquitetura e funcionamento. Também iremos explicar em detalhes como foi feita a coleta de métricas, e quais sistemas operacionais e navegadores foram utilizados.

3.1 Prova de Conceito

Nesta pesquisa construímos um aplicativo[1], chamado Convideo, que utiliza as tecnologias de WebAssembly e JavaScript. Utilizamos este aplicativo como base para a comparação entre as duas tecnologias propostas. O Convideo tem como propósito ser um aplicativo web para a conversão de vídeos. Para realizar tal conversão, o aplicativo usa a biblioteca FFMPEG. Esta biblioteca é muito utilizada pelo mercado, o que torna o uso da aplicação próximo à realidade dos usuários.

Utilizamos o Emscripten, citado anteriormente, para a compilação do código fonte do FFMPEG para WebAssembly e JavaScript. O Emscripten utiliza o bitcode gerado para o LLVM e faz a compilação para asm.js e para WebAssembly. Como ambos são gerados a partir da mesma fonte, temos a garantia que os mesmos algoritmos são aplicados em ambas as versões da ferramenta. É importante ressaltar precisamos desabilitar algumas funcionalidades do Emscripten, que na época da escrita deste artigo ainda estão em fase experimental, como suporte a *threads* e SIMD [15].

[1] Disponível em https://github.com/lucasecdb/convideo

3.2 Fluxo de conversão

Nesta Seção, iremos detalhar os componentes principais da arquitetura da aplicação e o fluxo da conversão de um vídeo, a fim de descrever como foi desenvolvido o aplicativo e como seus componentes comunicam entre si.

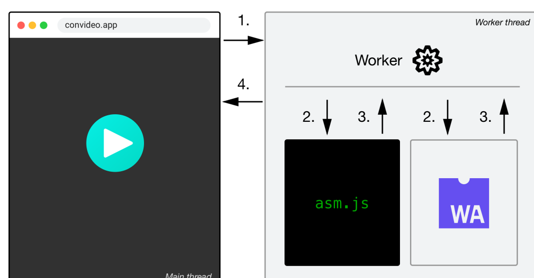

Figura 1: Fluxo de conversão de um vídeo

O sistema é composto de 3 componentes principais. O primeiro deles é o navegador que roda o front-end da aplicação, onde temos nossa Main Thread. O segundo é o *Web Worker*, que contém um script no qual executa uma pequena lógica para saber qual versão do algoritmo iremos utilizar. Por último, temos o algoritmo propriamente dito. Esse algoritmo possui duas implementações, uma em WebAssembly e outra em asm.js, como foi discutido anteriormente. O fluxo da conversão de um vídeo, como podemos ver na Figura 1, segue os seguintes passos:

(1) A Main Thread se comunica com o Web Worker, que é onde iremos rodar o código do FFMPEG para a conversão do vídeo, passando o arquivo do vídeo em formato binário.
(2) Em seguida, o Worker verifica se estamos querendo realizar a conversão utilizando o asm.js ou WebAssembly, e faz a conversão com a implementação correspondente, utilizando o arquivo enviado pela Main Thread.
(3) O FFMPEG nesse momento faz a conversão propriamente dita e, quando finalizado, passa o resultado para o Web Worker, também em formato binário.
(4) O Worker, por sua vez, repassa para a Main Thread, que então realiza o download do arquivo para salvar localmente no computador do usuário.

3.3 Coleta de métricas

Para a coleta de métricas, foram adicionadas algumas opções no aplicativo para a realização da coleta do tempo de execução da conversão do vídeo, como por exemplo um botão para fazer o download de todas as métricas em um arquivo no formato CSV, e uma caixa de entrada para poder especificar quantas vezes a conversão do vídeo seria executada. No arquivo CSV tem-se as informações do nome do arquivo convertido, tamanho original em bytes, tamanho final em bytes, tempo total para a conversão, nome do formato utilizado, nome do codec de vídeo utilizado, nome do codec de áudio utilizado, índice de execução, e uma flag indicando se foi convertido utilizando o WebAssembly ou não. Um exemplo do arquivo de métricas pode ser visto na Figura 2. Também adicionamos manualmente um campo para indicar qual foi o navegador e o Sistema Operacional utilizado em cada arquivo CSV. Para cada vídeo utilizamos um total de 50 conversões por implementação. Ou seja, 50 vezes na versão em WebAssembly e 50 vezes com o asm.js, totalizando uma execução de 100 conversões por vídeo. Escolhemos coletar essa quantidade pois se mostra como um tamanho suficiente para gerar relevância estatística.

filename	elapsed_time	input_size	output_size	format	video_codec	audio_codec	wasm	index	browser
earth.mp4	7.158709999988787	1570024	1322592	matroska	mpeg4	aac	1	1	Chrome 77
earth.mp4	6.380874999973457	1570024	1322592	matroska	mpeg4	aac	1	2	Chrome 77
earth.mp4	6.411274999962188	1570024	1322592	matroska	mpeg4	aac	1	3	Chrome 77
earth.mp4	6.520250000001397	1570024	1322592	matroska	mpeg4	aac	1	4	Chrome 77
earth.mp4	6.430869999981951	1570024	1322592	matroska	mpeg4	aac	1	5	Chrome 77
earth.mp4	6.399135000014212	1570024	1322592	matroska	mpeg4	aac	1	6	Chrome 77
earth.mp4	6.449459999974351	1570024	1322592	matroska	mpeg4	aac	1	7	Chrome 77
earth.mp4	6.1296800000127405	1570024	1322592	matroska	mpeg4	aac	1	8	Chrome 77
earth.mp4	6.697159999981523	1570024	1322592	matroska	mpeg4	aac	1	9	Chrome 77
earth.mp4	6.9253549999557436	1570024	1322592	matroska	mpeg4	aac	1	10	Chrome 77
earth.mp4	6.395094999985304	1570024	1322592	matroska	mpeg4	aac	1	11	Chrome 77
earth.mp4	6.5508649999974295	1570024	1322592	matroska	mpeg4	aac	1	12	Chrome 77
earth.mp4	6.428959999990184	1570024	1322592	matroska	mpeg4	aac	1	13	Chrome 77
earth.mp4	6.4271849999786355	1570024	1322592	matroska	mpeg4	aac	1	14	Chrome 77
earth.mp4	6.428630000038538	1570024	1322592	matroska	mpeg4	aac	1	15	Chrome 77

Figura 2: Exemplo do arquivo de métricas

Utilizamos o formato Matroska [10], que é um formato de vídeo genérico bastante utilizado, que pode comportar vários *streams* de vídeo e áudio em um mesmo arquivo. A escolha desse formato se dá por sua popularidade e flexibilidade como um *container* de vídeos. Para o codec de vídeo, escolhemos usar o MPEG-4 Part 2, dado sua popularidade e velocidade na conversão nos vídeos, visto que ele é um codec lossy (i.e. que não garante a preservação da qualidade do vídeo). Também testamos utilizar o codec H.264, mas optamos por não prosseguir dado sua lentidão na conversão dos vídeos no navegador, o que pode ser causado pela falta suporte às instruções específicas utilizadas pela implementação do codec. Não alteramos os parâmetros dos codecs e formato, assim utilizando os valores padrões provenientes do FFMPEG. Para a avaliação, utilizamos o arquivo `earth.mp4`, disponível no repositório do aplicativo, que é um arquivo em MP4 de duração de 30 segundos da rotação da terra. A escolha desse arquivo foi de ser um vídeo relativamente pequeno, porém que possui movimentação, assim exercitando o algoritmo de compressão do codec utilizado.

Para nossa coleta, foram utilizados 4 navegadores, em 3 Sistemas Operacionais distintos, como podemos observar em detalhes nas Tabelas 1 e 2. Antes de inicializarmos a coleta, fechamos todos os outros aplicativos no computador para evitar interferência com a pesquisa. Também utilizamos uma janela anônima no navegador, para evitar que as extensões instaladas atrapalhem o resultado. Cada coleta foi realizada individualmente, mais uma vez, para evitar interferências.

Tabela 1: Ambientes de execução

Sistema Operacional	CPU	Memória
Windows 10	Intel Core i7 1,8 GHz	16 GB
macOS 10.15	Intel Core i5 2,7 GHz	8 GB
Ubuntu 18.04	Intel Core i7 1,8 GHz	16 GB

Tabela 2: Navegadores utilizados

Navegador	Versão
Google Chrome	78
Microsoft Edge[2]	18
Mozilla Firefox	70
Safari[3]	13

4 RESULTADOS E DISCUSSÕES

Nesta Seção iremos apresentar os resultados de nossas análises. Também iremos apresentar nossas discussões sobre as análises feitas.

Primeiro iremos responder nossas duas perguntas de pesquisa que foram introduzidas no começo do artigo. Vamos analisar tanto a performance do tempo de execução das duas implementações, como também o tamanho de código gerado para as duas. Em seguida, tentaremos dar uma explicação para os resultados encontrados, utilizando outros estudos feitos na área como base para nossas justificativas.

4.1 Performance do tempo de execução

Para responder nossa primeira pergunta, utilizamos a metodologia descrita na Seção anterior, onde tivemos um total de 800 observações coletadas. Com esses dados, calculamos a média de todas as execuções, agrupadas por linguagem, a fim de observar se existia alguma indicação de que uma implementação teria um tempo de execução muito diferente da outra.

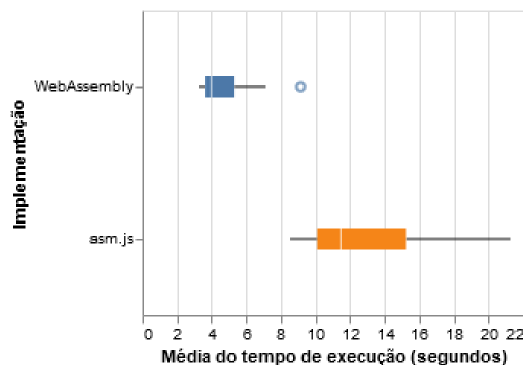

Figura 3: Boxplot da média do tempo de execução dos navegadores

A Figura 3 mostra o boxplot da média do tempo de execução das duas abordagens para todos os navegadores e Sistemas Operacionais que utilizamos para coletar as métricas. Nela podemos observar que praticamente todas as execuções em WebAssembly apresentam um melhor tempo quando comparadas a implementação em asm.js, independente do Sistema Operacional ou navegador. Notamos que a média da execução do algoritmo em JavaScript é de **2,78** a **2,99** (confiança de 95%) vezes maior do que a do algoritmo em WebAssembly.

Em seguida, agrupamos os dados por Sistema Operacional, destacando os navegadores e implementações, e analisando o tempo de execução para identificar se a relação observada no gráfico anterior também se aplica a cada Sistema Operacional individualmente. Com isso, montamos 3 gráficos, um para cada Sistema Operacional, em relação ao navegador utilizado.

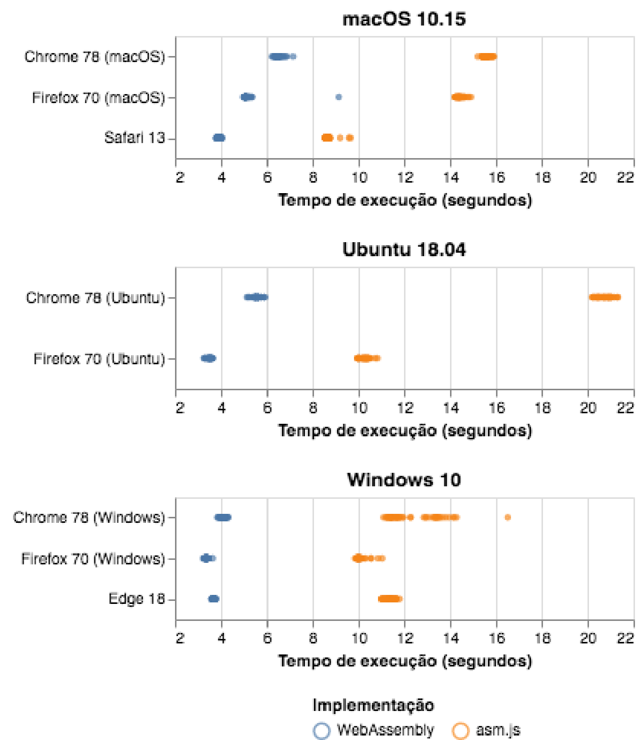

Figura 4: Tempo de execução por Sistema Operacional

Na Figura 4 temos o tempo de execução de cada observação realizada para cada navegador especificamente. Podemos observar que a premissa de que JavaScript seria, em geral, mais lento que WebAssembly continua verdadeira quando analisamos os navegadores individualmente.

Analisando os Sistemas Operacionais individualmente também chegamos na conclusão que WebAssembly é mais eficiente do que JavaScript. No caso do macOS, podemos ver que o navegador Safari possui a melhor performance com a implementação utilizando WebAssembly, quando comparado com os demais. Já no Ubuntu, que aqui representa o Linux, temos que o Firefox possui uma métrica melhor quando comparado com o Google Chrome. Por último, temos no Sistema Operacional Windows uma distinção não muito clara do navegador com a melhor métrica de tempo de execução, visto que todos os três navegadores possuem resultados muito similares na implementação em WebAssembly.

É interessante observar também que a métrica de tempo de execução da implementação em WebAssembly é, em geral, menos variável do que a implementação em JavaScript. Podemos ver isso

[2] Utilizado apenas no Sistema Operacional Windows 10
[3] Utilizado apenas no Sistema Operacional macOS

mais claramente quando observamos na Figura 4 a dispersão das observações feitas. No caso do Google Chrome no Windows, temos um valor mínimo de 11 segundos e um valor máximo de 16 segundos. Em contraste, na implementação em WebAssembly temos um valor mínimo de 3,8 segundos, e um valor máximo de 4,3 segundos. Esta última possui uma variação média entre **0,00** e **2,06** segundos (confiança de 95%), enquanto JavaScript possui uma variação média entre **0,15** e **2,83** segundos (confiança de 95%).

4.2 Tamanho de código

Para responder nossa segunda pergunta de pesquisa, montamos um gráfico que compara o tamanho total do código das duas implementações utilizadas.

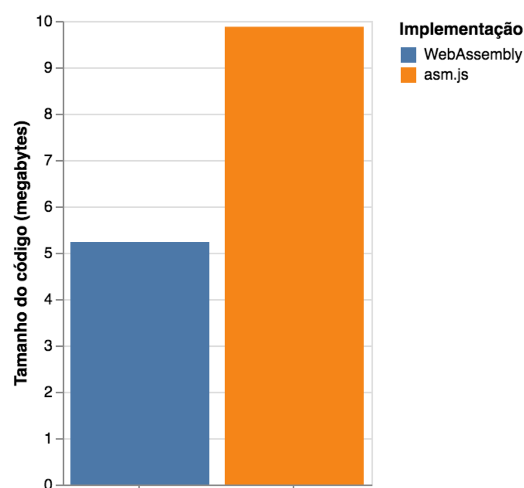

Figura 5: Tamanho do código por implementação

A Figura 5 descreve o tamanho total, em bytes, dos executáveis das duas implementações. O tamanho da implementação em WebAssembly também inclui um pequeno *runtime* que é gerado pelo Emscripten na geração do código, que é feito em JavaScript. Esse mesmo runtime também é incluso na versão em asm.js, porém ele existe em conjunto com o arquivo que possui a implementação do algoritmo do FFMPEG. Podemos observar que o tamanho total da implementação em WebAssembly é aproximadamente 47% menor que o asm.js.

4.3 Discussão

O asm.js utilizado para a comparação não é uma linguagem com uma especificação padrão, ou seja, os navegadores podem decidir implementar ou não um compilador apropriado para executar esse subconjunto da linguagem, possivelmente otimizando o código de maneiras diferentes. Essa falta de padronização pode ter sido um dos fatores pelo qual podemos observar as diferenças nas observações entre os navegadores utilizando a implementação em asm.js, visto que cada implementação do navegador pode diferir na forma de otimização. Um exemplo disso seria a disparidade do tempo de execução na implementação em asm.js no Sistema Operacional Ubuntu, onde o navegador Google Chrome foi em média duas vezes mais lento do que o Firefox.

WebAssembly, por outro lado, é uma linguagem com uma especificação padrão, de baixo nível, e estaticamente e fortemente tipada. Com isso, é possível utilizar apenas um compilador *ahead-of-time* e otimizadores para realizar a compilação do código fonte, visto que possuímos todas as informações de tipos de forma estática. Isso justifica o resultado que vimos na Seção anterior, onde as diferentes observações do tempo de execução ficaram bem próximas de um valor, quando comparado com sua versão em asm.js. Navegadores como o Google Chrome [11] implementaram um compilador base para WebAssembly utilizando essa técnica. Com isso, podemos apenas compilar esse código uma única vez, otimizando o máximo possível, sem a necessidade de vários níveis de otimizações diferentes. Visto que esse passo de otimização pode demorar um pouco para ser executado, o Google Chrome faz esse passo em plano de fundo, e utiliza uma compilação sem otimização enquanto a otimização não é completada. Depois que ela está pronta, o navegador pode então apenas trocar a implementação que está executando, sem nenhuma perda de performance.

Não existe uma justificativa trivial para as diferenças observadas nos resultados de tempo de execução das duas implementações. Ambas as técnicas podem ser bastante performáticas na medida que é possível fazer uma compilação com *ahead-of-time*. Entretanto, as primitivas de operações e de manipulação de memória do WebAssembly parecem permitir uma melhor otimização de código, ao contrário do asm.js que utiliza de primitivas mais complexas, como foi mostrado no exemplo do Código 1.

Como vimos na Seção anterior, o tamanho do executável do WebAssembly é consideravelmente menor que o de JavaScript. Isso, se dá pela natureza do WebAssembly ser um formato binário, que naturalmente ocupa menos espaço em disco. Por consequência, o custo de transferência do WebAssembly é bem menor do que do JavaScript, o que faz com que a página que está utilizando a implementação em WebAssembly seja carregada mais rapidamente pelo navegador. Essa diferença de carregamento pode ter um impacto bastante positivo na experiência de usuário, como foi visto nos estudos feitos e compartilhados pela Google e Amazon.

5 LIMITAÇÕES E TRABALHOS FUTUROS

Nesta Seção iremos apresentar as limitações encontradas no desenvolvimento desse trabalho e análise das observações coletadas, e o que poderia ser realizado em trabalhos futuros para mitigar os problemas encontrados.

5.1 Limitações

Uma grande limitação encontrada com o desenvolvimento desse trabalho foi que encontramos apenas um compilador que realiza a compilação de C ou C++ para JavaScript e WebAssembly. Dessa forma, nossas análises podem ter sido enviesadas pelo compilador utilizado, visto que os resultados encontrados podem ter sido apenas causados por uma má otimização para a implementação de JavaScript, e não uma limitação da tecnologia em si. Sendo assim, estamos efetivamente comparando apenas o quão bem o Emscripten consegue otimizar o código para as duas implementações.

5.2 Trabalhos futuros

Em trabalhos futuros, seria interessante utilizar outros compiladores que seguem o mesmo princípio do Emscripten e realizar as mesmas coletas sobre o código gerado dessas outras ferramentas, para ver se chegaríamos na mesma conclusão.

Também seria interessante realizar em trabalhos futuros uma carga de trabalho mais específica (e.g. estresse apenas de memória ou determinadas operações básicas) a fim de entender melhor quais são as mais impactantes e onde cada ferramenta performa melhor.

Outro trabalho que poderia ser realizado seria encontrar uma forma de analisar a métrica de memória em conjunto com o tempo de execução, visto que não encontramos uma API pública e padronizada do navegador para a coleta do consumo de memória de uma *worker thread*. Assim, poderíamos observar melhor se os ganhos observados na implementação em WebAssembly possuem ou não um impacto na memória, quando comparado com o asm.js.

REFERÊNCIAS

[1] [n. d.]. asm.js. http://asmjs.org/. Acessado em 2019-09-21.

[2] [n. d.]. Indexed Database API. https://www.w3.org/TR/IndexedDB/. Acessado em 2019-09-21.

[3] Daniel An. [n. d.]. New Industry Benchmarks for Mobile Page Speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/

[4] Jeremy Ashkenas. [n. d.]. List of languages that compile to JS. https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS. Acessado em 2019-09-24.

[5] Wikipedia contributors. 2019. Garbage collection (computer science) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Garbage_collection_(computer_science)&oldid=915289420. Acessado em 2019-09-24.

[6] Wikipedia contributors. 2019. JavaScript engine — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=JavaScript_engine&oldid=913993771. Acessado em 2019-09-24.

[7] Kit Eaton. [n. d.]. How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[8] David Flanagan. [n. d.]. JavaScript: The Definitive Guide, 6th Edition. O'Reilly Media, 1096.

[9] LLVM Foundation. [n. d.]. LLVM. https://llvm.org. Acessado em 2019-09-21.

[10] Alexander Noé. 2007. Matroska file format (under construction!). *Jun* 24 (2007), 1–51.

[11] V8 project authors. [n. d.]. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff

[12] Wikipedia contributors. 2019. Configure script — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Configure_script&oldid=915660397. Acessado em 2019-11-20.

[13] Wikipedia contributors. 2019. Make (software) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Make_(software)&oldid=923349997. Acessado em 2019-11-20.

[14] Wikipedia contributors. 2019. S-expression — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=S-expression&oldid=920113692. Acessado em 2019-11-10.

[15] Wikipedia contributors. 2019. SIMD — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SIMD&oldid=921111808 Acessado em 2019-10-30.

[16] Wikipedia contributors. 2019. Syntactic sugar — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Syntactic_sugar&oldid=920452329. Acessado em 2019-11-10.

[17] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25

[18] Alon Zakai. [n. d.]. Why WebAssembly is Faster Than asm.js. https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/

[19] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224

[20] Alon Zakai. 2019. Emscripten and the LLVM WebAssembly backend. https://v8.dev/blog/emscripten-llvm-wasm. Acessado em 2019-09-24.