Investigando a aplicabilidade da tecnologia WebAssembly na implementação de Virtual DOMs mais eficientes para aplicações web

Vinícius G. C. Paulino[1], Vinicius C. Garcia[1]

Centro de Informática – Universidade Federal de Pernambuco (UFPE)[1]
Av. Jornalista Aníbal Fernandes, s/n - Cidade Universitária - Recife - PE

{vgcp,vcg}@cin.ufpe.br

Resumo. O React, uma ferramenta bastante utilizada no desenvolvimento de aplicações web, utiliza o mecanismo de Virtual DOM (*Virtual Document Object Model*) para otimizar atualizações à interface de usuário. Apesar de trazer benefícios, seu uso também exige um maior consumo de memória e processamento, o que pode afetar a qualidade da experiência do usuário. Com isso, novas iniciativas aplicam a tecnologia WebAssembly (Wasm) em virtual DOMs para atingir maior eficiência, porém exigem o uso de linguagens como C++ e Rust na construção de aplicações web. Portanto, este trabalho propõe o desenvolvimento do Warc, uma biblioteca baseada no React que permite que aplicações JavaScript adotem um virtual DOM implementado em WebAssembly. Foi realizada uma análise comparativa entre a eficiência do React e Warc com base em testes de benchmark para verificar a aplicabilidade de WebAssembly na implementação de Virtual DOMs mais eficientes. Com a análise dos resultados, concluiu-se que apesar do WebAssembly ter diminuído o tamanho e o tempo de carregamento da aplicação, limitações em sua interoperabilidade com o JavaScript causaram um desempenho inferior em comparação ao React. A perspectiva para futuros trabalhos é continuar avaliando o amadurecimento da tecnologia WebAssembly antes que seu uso em virtual DOMs reais seja indicado.

Palavras-chave: *Virtual DOM, WebAssembly, React.*

Abstract. React, a widely used web development tool, uses a Virtual DOM (Virtual Document Object Model) to optimize updates to the user interface. Although its use brings several benefits, it also requires a higher memory usage and processing, which can lower the user experience. To solve this problem, experimental tools aim to apply the WebAssembly (Wasm) technology in Virtual DOMs to reach higher performance, but they require languages such as C++ and Rust to build web applications. Therefore, this study presents Warc, a React based library that allows JavaScript applications to leverage a Wasm virtual DOM. A comparative analysis was carried out between the efficiency of React and Warc based on benchmark tests to verify the applicability of WebAssembly on building more efficient Virtual DOMs. On further analysis, it has been concluded that although WebAssembly was able to reduce the application size and load times, some limitations in its interoperability with JavaScript were the cause to a lower performance when compared to React. The perspective for future work is to continue evaluating the maturity of the WebAssembly technology before its use in real virtual DOMs is indicated.

Keywords: *Virtual DOM, WebAssembly, React.*

1. Introdução

Com o padrão *Single-Page Application* (SPA) [1] houve o surgimento de ferramentas que abstraem a sincronização entre a interface do usuário e os dados de uma aplicação web [2]. O React, uma das mais populares, permite que o desenvolvedor declare como a interface deve ser, sem se preocupar com como ela será construída [3][4].

Para isso, o React utiliza um Virtual DOM (*Virtual Document Object Model*) para determinar quais partes da interface devem ser atualizadas e efetuá-las de forma otimizada, processo conhecido como reconciliação. Apesar de entregar um desempenho adequado para a maioria dos casos [3], seu uso exige um maior processamento e consumo de memória, o que pode vir a afetar os tempos de carregamento e renderização da página [5][6].

As limitações do Virtual DOM sugerem que seja um campo interessante para a aplicação de WebAssembly (Wasm), um formato binário executável em navegadores e focado em portabilidade, segurança e alto desempenho [8] em aplicações web. Mesmo que a tecnologia esteja em seus primeiros passos, sua utilização no mecanismo de Virtual DOM do React tem sido cogitada para uma reconciliação mais eficiente [10].

Atualmente já existem virtual DOMs experimentais que utilizam WebAssembly, porém todas exigem o uso de linguagens como C++ e Rust [67, 68], impedindo que aplicações JavaScript se beneficiem com a adoção de Wasm. Como solução, este trabalho propõe e detalha o desenvolvimento do *Warc* (**W**eb**A**ssembly Powered **R**eact **C**lone), uma biblioteca compatível com a interface pública do React que utiliza um virtual DOM implementado em Wasm e pode ser usado por aplicações JavaScript.

Com o objetivo de analisar a aplicabilidade da tecnologia WebAssembly na implementação de Virtual DOMs mais eficientes, será realizada uma análise comparativa acerca do desempenho do React e Warc com base em dados coletados em testes de benchmark. A investigação será guiada pela metodologia GQM (*Goal, Question, Metric*) [45]. O resultado determinará se o uso de WebAssembly na ferramenta Warc é capaz de apresentar maior eficiência frente ao React em aplicações web JavaScript.

O trabalho está dividido em 5 seções, sendo estruturado da seguinte forma:

- **Seção 2 - Contexto**: Define o contexto teórico do trabalho, conceituando o React, o DOM, o Virtual DOM e a tecnologia WebAssembly;
- **Seção 3 - Desenvolvimento da Ferramenta Warc:** Detalha os objetivos, funcionalidades e a implementação da ferramenta proposta pelo trabalho;
- **Seção 4 - Metodologia**: Descreve a metodologia utilizada, o plano do GQM e os procedimentos de coleta de dados de desempenho das ferramentas React e Warc;
- **Seção 5 - Resultados**: Apresenta os resultados obtidos na coleta de dados de todas as métricas de desempenho e as respostas às questões do GQM;
- **Seção 6 - Discussão**: Discute os resultados apresentados e suas justificativas;
- **Seção 7 - Conclusão e direções futuras**: Apresenta a conclusão do trabalho e considerações para trabalhos futuros no ramo de WebAssembly e Virtual DOMs.

2. Contexto

2.1. De páginas estáticas a *Single-Page Applications*

Nos primeiros anos da Web, páginas eram simples e estáticas, formadas apenas por documentos HTML (*HyperText Markup Language*) [11]. Com a evolução dos navegadores e

a criação da linguagem JavaScript em 1995, páginas web se tornaram cada vez mais interativas com alterações ao conteúdo da página de forma dinâmica ao manipular o *Document Object Model* (DOM) [11, 12]. Tais práticas melhoraram a experiência do usuário, mas ainda eram limitadas em desempenho e interatividade em relação à aplicações nativas [13]. A demanda por aplicações mais ricas e interativas na web levou ao surgimento do padrão AJAX (*Asynchronous JavaScript and XML*) [13]. Aplicações AJAX eram capazes de criar ou recuperar dados realizando requisições ao servidor sem provocar recarregamentos da página.

Com a crescente maturação da plataforma web, o uso de AJAX foi sendo aperfeiçoado e *frameworks* JavaScript foram criados para diminuir a complexidade de aplicações [14, 15], como o Backbone.js, Knockout.js e Angular.js [15], que adotavam a estratégia de mover as lógicas de negócio e dados para o cliente para diminuir o processamento no servidor [14], além de abstrair a manipulação do DOM. Era o nascimento do padrão *single-page application* (SPA) [15, 16], aplicações executadas em sua totalidade no lado do cliente minimizando ainda mais o tempo de resposta para qualquer interação [17].

Segundo Mikito Takada [18], uma SPA possui uma experiência de usuário mais próxima de uma aplicação nativa devido a sua capacidade de atualizar qualquer parte da interface sem realizar recarregamentos ou comunicar-se com um servidor. A utilização desse padrão está altamente difundida e aplicações web se tornam cada vez mais complexas, fazendo uso extensivo de JavaScript para gerenciar a grande quantidade de dados e regras de negócio [17]. Aplicações como o Facebook, Twitter e Instagram são SPAs [19].

Em 2013 foi apresentado o React [15], uma ferramenta desenvolvida por engenheiros do Facebook para manter a interface de usuário sincronizada com a lógica de negócio e os dados de aplicações em larga escala [20]. Um dos seus diferenciais foi a abstração da manipulação do DOM por meio de uma estrutura de dados conhecida como Virtual DOM, com a qual seu algoritmo pode detectar mudanças na interface e aplicá-las em lote no DOM real, minimizando o impacto no desempenho [15, 20].

O React será o foco deste trabalho porque é a principal ferramenta do mercado a utilizar a abordagem de Virtual DOM [14]. Na próxima seção, seu funcionamento será abordado com mais profundidade.

2.2. React

O React é uma das principais bibliotecas de JavaScript para construção de interfaces de usuário na web, sendo o framework de front-end mais amado de acordo com o *Stack Overflow Developer Survey 2019* [21]. Atualmente, o React é utilizado por diversas companhias como Uber, Netflix, Reddit e Paypal [5]. Um de seus grandes diferenciais é a utilização de um estilo declarativo de programação [17], e de acordo com sua documentação oficial, *interfaces declarativas tornam o código mais previsível e mais fácil de depurar* [22].

Uma aplicação React é composta por vários componentes. Um componente é uma parte da interface de usuário que deve ser independente das demais e reutilizável, permitindo que o desenvolvedor possa pensar em cada parte isoladamente [23]. O código abaixo é um exemplo de componente React que renderiza um elemento *div* contendo o texto *Hello World*:

```
function HelloMessage(props) {
  return <div className="hello">Hello, {props.name}!</div>;
}
const root = document.getElementById('root');
ReactDOM.render(<HelloMessage name="World" />, root);
```

No trecho de código acima, utiliza-se o JSX [24], uma sintaxe semelhante ao HTML que facilita a construção de layouts. Como o JSX não é código JavaScript válido, é necessário

utilizar a ferramenta Babel [55] para convertê-lo em chamadas à função `React.createElement` que o navegador é capaz de interpretar.

```
function HelloMessage(props) {
  return React.createElement("div", { className: "hello"}, `Hello, ${props.name}!` );
}
```

No React, um componente pode criar e gerenciar seu estado. Quando o estado de um componente muda, ele é renderizado novamente e as mudanças são refletidas no DOM. Um componente pode atualizar seu estado utilizando o método *setState*, mas com a adição dos React *Hooks*, a partir da versão 16.8 [25], componentes funcionais podem criar e atualizar estado usando o hook `useState`.

```
function ClickCount() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>Clique-me</button>
    </div>
  );
}
```

Na próxima seção será aprofundado o funcionamento do Document Object Model (DOM) para que o mecanismo do Virtual DOM possa ser detalhado mais facilmente.

2.3. Document Object Model

O *Document Object Model* (DOM) é um padrão definido por uma especificação W3C [12] e implementado em todos os principais navegadores. O DOM define a estrutura lógica de documentos HTML e permite que um código JavaScript possa acessar um documento, navegar através de sua estrutura e modificar seu conteúdo e aparência [12, 5]. Na prática, o DOM é uma estrutura em árvore na qual cada nó representa um elemento HTML. É gerada internamente pelo navegador ao realizar a leitura de um documento HTML [5].

Antes que um navegador renderize uma página, é necessário processar o documento HTML obtido através da web. É criado um objeto **`Node`** para cada elemento [5, 26] após a transformação dos caracteres do arquivo HTML em tokens [26]. Por fim, as relações de parentesco entre os elementos são criadas, e o resultado é uma estrutura de dados em árvore a qual o navegador utiliza como modelo para a renderização [26]. A árvore fica acessível via JavaScript a partir da variável **`window.document`**. Com isso, é possível utilizar uma série de métodos da DOM API para atualizar a árvore, como **`Document.createElement`** para instanciar novos elementos e **`Node.appendChild`** para inserir elementos na página [27].

Após a primeira renderização, atualizações ao conteúdo da página dispararão rotinas conhecidas como *reflow* e *repaint* [5]. O reflow faz o recálculo das posições e tamanhos dos elementos da página, e o repaint "pinta" a tela conforme as demarcações de cada elemento [28]. Ambas as rotinas são custosas e podem afetar o desempenho da aplicação se executadas frequentemente. Logo, é importante minimizar a ocorrência de reflows e repaints [15], o que pode ser alcançado diminuindo a frequência de manipulações do DOM [17].

2.4. Virtual DOM

O Virtual DOM é uma representação virtual do DOM utilizada internamente pelo React. É composto por uma árvore de objetos JavaScript, na qual as operações de adição, atualização e remoção de objetos não são custosas [5], sendo uma maneira de minimizar o custo de

manipulação do DOM real [17]. Nessa árvore, cada elemento do DOM real é representado por um objeto JavaScript. A cada mudança de estado uma nova árvore do virtual DOM é gerada e comparada à árvore atual na reconciliação [4]. O resultado é o conjunto mínimo de mudanças a serem realizadas no DOM real para sincronizá-lo ao virtual DOM mais recente [17]. Assim, o DOM sofre mudanças apenas onde e quando necessário [5].

Nas próximas subseções o algoritmo de reconciliação será explicado com mais detalhes, assim como as principais otimizações realizadas pelo React. Após isso, serão detalhadas algumas desvantagens dessa abordagem.

2.4.1. Algoritmo de reconciliação

Em um componente React, o método `render` gera uma árvore de elementos que farão parte do virtual DOM. Uma mudança de estado do componente provoca a geração de uma nova árvore [4], e com isso, é preciso encontrar a maneira mais eficiente de atualizar o DOM para sincronizá-lo com a árvore mais recente. O algoritmo de *diffing*, ou reconciliação, é usado para encontrar as diferenças entre as árvores.

De acordo com a documentação oficial do React, a reconciliação pode ser resumida da seguinte forma [4]: o elemento raiz da árvore atual é comparado com o elemento raiz da nova árvore. Caso os elementos tenham tipos diferentes (por exemplo, o primeiro tem o tipo *main* enquanto o outro tem o tipo *div*), React irá destruir toda a árvore antiga e construir uma nova a partir do elemento raiz da nova árvore. Todo o estado dos componentes antigos também é destruído. Caso ambos tenham tipos iguais, o nó do DOM pode ser reaproveitado, aplicando-se as devidas modificações em seus atributos.

Após isso, as listas de elementos filhos de ambos os elementos são iteradas ao mesmo tempo. Para cada par de elementos filhos a mesma comparação é realizada. Caso não seja possível fazer o mapeamento de um elemento da árvore atual para a nova árvore, seu nó do DOM é destruído. Da mesma forma, se não for possível fazer o mapeamento de um elemento da nova árvore para um existente na árvore atual, significa que é um novo elemento e deve ser inserido no DOM. A comparação continua até que não haja elementos a serem comparados e todas as diferenças entre as árvores sejam detectadas.

2.4.2. Otimizações

Sabe-se que encontrar o conjunto mínimo de diferenças em duas árvores de virtual DOM pode ser realizado em tempo $O(n^3)$ [4]. Nessa complexidade, atualizar 1000 elementos exigiria 1 bilhão de comparações, mas o algoritmo de reconciliação do React utiliza algumas heurísticas e otimizações que o aproximam de tempo $O(n)$ [17]:

2.4.2.1. Comparação nível a nível

Duas árvores de virtual DOM são comparadas nível a nível. Na Figura 1, a árvore *before* representa o estado atual do DOM e a árvore *after* é uma árvore gerada com uma mudança de estado. Primeiro compara-se os elementos da raíz (azul escuro). Após isso, comparam-se seus filhos de mesmo nível (em rosa), e assim sucessivamente. No fim da comparação nenhuma mudança foi detectada (árvores iguais), logo, nenhuma atualização ao DOM é programada.

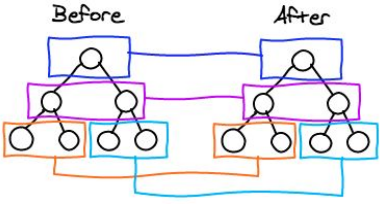


Figura 1: Comparação de árvore nível a nível [29]

2.4.2.2. Uso do atributo key

Outra otimização no algoritmo de reconciliação é o uso do atributo `key`. No React, para renderizar uma lista de componentes é necessário prover, para cada um, uma chave de valor único que permite identificar o mesmo componente em duas árvores [24].

Usar *keys* para o mapeamento de componentes durante a reconciliação é mais eficiente do que mapear pela ordem, o que poderia acarretar mais alterações no DOM do que o necessário caso um novo componente seja inserido no meio de outros pré-existentes [24, 34]. Na Figura 2, o uso de *keys* em todos os elementos da comparação torna fácil a identificação do novo elemento (*z*). Logo, apenas uma mudança ao DOM pode ser programada (inserção).

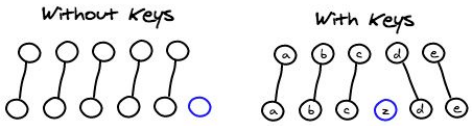


Figura 2: Uso de chaves [29]

2.4.2.3. Renderização de subárvores

Assim que o estado de um componente é atualizado com a invocação do método `setState`, o React o marca como *dirty* (sujo), o que causará a geração de uma nova subárvore de virtual DOM para apenas aquele componente e seus filhos [4]. Na Figura 3, apenas as subárvores de componentes marcados como sujos (em vermelho), terão as subárvores geradas novamente (marcadas em azul). Com isso, os elementos em branco não terão sofrido mudanças.

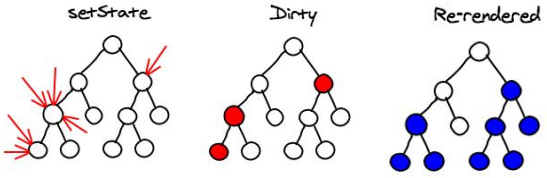


Figura 3: Renderização de subárvores [29]

2.4.2.4. Atualizações em lote

Além disso, ao finalizar o processo de reconciliação, todas as atualizações necessárias no DOM, que manualmente seriam feitas com várias chamadas a DOM API, são realizadas em lote em poucas chamadas. Com isso, o DOM é atualizado apenas uma vez durante um loop de eventos [29], característica essencial para minimizar repaints e reflows.

2.4.3. Desvantagens

A abordagem do virtual DOM exige que uma representação virtual do DOM seja gerada a cada mudança de estado. Logo, dependendo do tamanho da aplicação e da quantidade de elementos na página, o consumo de memória acaba aumentando [17][5].

Além disso, apesar do desempenho do React em manipulação do DOM ser relativamente superior a outros frameworks e bibliotecas, testes de benchmark revelam que a necessidade de realizar a reconciliação de árvores do virtual DOM afeta o tempo de renderização quando comparada a uma aplicação em JavaScript puro [5].

2.5. WebAssembly

Durante a última década, a linguagem JavaScript foi forçada a evoluir em desempenho e usabilidade para suportar a demanda de aplicações complexas na web [30]. Mesmo com o maior desempenho trazido pela *engine* V8 [32], utilizada no Chromium, e o uso de técnicas de JIT (*just in time compilation*) e outras otimizações, JavaScript ainda é considerado

inadequado para aplicações de computação intensa devido a limitações inerentes à linguagem, como sua tipagem dinâmica e desempenho inconsistente e imprevisível.

Tais limitações motivaram o surgimento do **asm.js**[1], um subconjunto altamente otimizável da linguagem JavaScript no qual todas as operações são estaticamente tipadas [33]. Asm.js possibilitou que aplicações desenvolvidas em C/C++ fossem executadas em navegadores com o compilador Emscripten[2], porém ainda sujeitas a várias desvantagens inerentes ao JavaScript [30].

Entendendo que JavaScript não é adequado para a execução de código de alta performance, engenheiros integrantes dos quatro maiores navegadores do mercado [31] se uniram para projetar um formato de código binário de baixo nível executável em navegadores e focado em desempenho, portabilidade e segurança [9]. O projeto se tornou o WebAssembly (Wasm), anunciado em 2019 pela World Wide Web Consortium (W3C) como a quarta linguagem padrão da web [7].

Sua adoção abre portas para outras linguagens de programação em navegadores de forma nativa além de JavaScript, já que seu binário é alvo de compilação de C, C++, Rust, Go e outras linguagens [34]. Atualmente, a versão 1.0 da tecnologia é suportada em todos os principais navegadores [35] e já é utilizada em aplicações reais, como o Google Earth [36] e Figma [37].

Nas subseções seguintes, serão apresentadas as principais decisões de projeto da tecnologia WebAssembly, suas características e limitações, o que está planejado para as futuras versões e o que motiva a sua aplicabilidade em virtual DOMs.

2.5.1. Objetivos arquiteturais

A tecnologia WebAssembly foi projetada a partir dos seguintes objetivos [31, 38]:

- **Alto desempenho**: deve oferecer desempenho consistente e próximo ao nativo;
- **Segurança**: deve ser executado de forma controlada, em *sandbox*, sem permissões ou acesso a recursos da máquina;
- **Portabilidade**: deve ser independente de hardware, linguagem e plataforma;
- **Representação compacta**: deve ter um formato binário compacto para minimizar o tempo de transferência na rede e também garantir eficiência em decodificação e compilação.

2.5.2. Conceitos

Segundo sua especificação oficial, WebAssembly é um formato de código binário, seguro e portátil, arquitetado para execução eficiente e representação compacta [31]. Essa tecnologia tem como objetivo permitir aplicações de alto desempenho na Web. Basicamente se trata de uma linguagem de programação de baixo nível, semelhante ao Assembly, que é estruturada a partir dos conceitos detalhados nas subseções seguintes.

- **Tipos de dados:** Há apenas quatro tipos de dados suportados no Wasm: números inteiros (*i32* e *i64*, possuindo 32 e 64 bits respectivamente) e números de ponto flutuante (*f32* e *f64*, com 32 e 64 bits respectivamente). Inteiros de 32 bits também são utilizados para representar valores *booleanos* e endereços de memória [38];
- **Conjunto de instruções:** O modelo de execução do Wasm funciona como uma máquina de pilha (*stack machine*) [31]. Seu código é formado por instruções binárias

[1] asm-js. Disponível em: <http://asmjs.org/>
[2] Emscripten. Disponível em: <https://emscripten.org/>

que realizam operações básicas em dados e alteram o controle do fluxo de execução (blocos, *loops* e condicionais) [38];

- **Funções:** O código Wasm é dividido em funções. Uma função pode receber uma sequência de valores como parâmetros e pode retornar um ou mais valores. Funções podem ser chamadas recursivamente e também podem declarar variáveis locais [38];
- **Memória linear:** A memória linear é um array de bytes mutáveis que pode crescer dinamicamente. Um programa Wasm pode guardar valores na memória linear e depois recuperá-los a partir de endereços de memória [38];
- **Módulos:** Um binário WebAssembly é chamado de *módulo*, o qual contém declarações de funções, variáveis globais e memória [38]. Para ser executado, um módulo Wasm precisa ser instanciado utilizando a WebAssembly API acessível via JavaScript, que possibilita a criação de um contexto de execução. [31]

2.5.3. Funcionamento

Para criar um módulo WebAssembly é necessário escolher uma linguagem que suporte a compilação para a arquitetura Wasm [39]. Utilizando o Emscripten, é possível compilar um código C++ para WebAssembly e executá-lo em um navegador. Na Tabela 1 é possível ver o código fonte em C++, resultado equivalente em Wat (*WebAssembly Text Format*) e no formato binário. O formato de texto representa um módulo WebAssembly e permite a leitura e inspeção do binário na web [38].

C++	Binário (hexadecimal)	WAT (formato textual)

`int factorial(int n) {`
`  if (n == 0)`
`  return 1;`
`  else`
`  return n * factorial(n-1);`
`}`
	`20 00 42 00 51 04 7e`
	`42 01 05 20 00 20 00`
	`42 01 7d 10 00 7e 0b`
		`(func (param i64) (result i64)`
		`  local.get 0`
		`  i64.eqz`
		`  if (result i64)`
		`  i64.const 1`
		`  else`
		`  local.get 0`
		`  local.get 0`
		`  i64.const 1`
		`  i64.sub`
		`  call 0`
		`  i64.mul`
		`  end)`

Tabela 1: Comparação do código C++, binário WebAssembly e formato textual (Wat)

Após compilado e disponibilizado na web, um módulo pode ser carregado numa página. Com uma chamada a *WebAssembly.instantiate*, o módulo será decodificado, compilado e inicializado para que então suas funções possam ser utilizadas [39].

```
fetch('factorial.wasm')
  .then((response) => response.arrayBuffer())
  .then((bytes) => WebAssembly.instantiate(bytes))
  .then((result) => console.log(result.instance.factorial(5)));
```

2.5.4. Desempenho

O artigo *Bringing the Web Up to Speed* (2018) [31], que introduziu a tecnologia WebAssembly, reportou benchmarks iniciais que indicam que a execução de código Wasm é cerca de 30% mais rápida que execução de código asm.js (JavaScript otimizado). Outros benchmarks indicam que comparado à execução de código JavaScript comum (ES6), WebAssembly pode chegar a ser entre 30% e 60% mais rápido, a depender do navegador e dispositivo em questão [66]. Além disso, WebAssembly pode ser duas vezes mais lento em relação a um código nativo equivalente [31], um resultado ainda competitivo e satisfatório.

2.5.5. Linguagens que compilam para WebAssembly

Além do projeto Emscripten, que oferece um ferramental para a compilação de C/C++ para Wasm, diversas linguagens de programação também estão adotando WebAssembly como alvo de compilação, tais como Rust [40], AssemblyScript [41], C# [42] e Go [43]. Uma lista de linguagens que fazem parte do ecossistema Wasm pode ser encontrada em *Awesome Wasm*[3].

2.5.6. Limitações e direções futuras

A primeira versão do WebAssembly foi um mínimo produto viável (MVP) [35], logo possui certas limitações como o suporte apenas a tipos numéricos e falta de acesso direto a APIs do navegador [44]. Atualmente há várias propostas de modificação à sua especificação sendo trabalhadas para a adição de novas funcionalidades [35], como principalmente:

- *Garbage collection*;
- *Threads*;
- Acesso direto ao DOM e outras APIs do navegador;
- Suporte à tipos de dados dinâmicos do JavaScript (*reference types*);
- Tratamento de exceções.

3. Desenvolvimento da Ferramenta Warc

O uso de WebAssembly na implementação de virtual DOMs não é uma ideia nova. A engenheira de software Lin Clark, numa apresentação intitulada *What WebAssembly Means for React* no evento React Europe 2017 [10] afirmou que Wasm possui potencial para fazer parte da reconciliação do React eventualmente, mas ainda não há planos concretos para isso.

Atualmente existem projetos de código aberto que buscam utilizar a tecnologia Wasm em virtual DOMs, como o **asm-dom** [67] e o **dodrio** [68]. Porém, ambos exigem o desenvolvimento de aplicações web nas linguagens C++ e Rust respectivamente, impossibilitando que aplicações JavaScript possam adotar WebAssembly em Virtual DOMs.

Para preencher essa lacuna, este trabalho propõe o desenvolvimento de um protótipo de biblioteca para construção de interfaces web, compatível com a interface pública do React, que utiliza um virtual DOM implementado em WebAssembly. Assim, será possível que aplicações JavaScript adotem a tecnologia em seus virtual DOMs sem a necessidade de configurações adicionais ou outras linguagens.

A ferramenta a ser desenvolvida será referida como *Warc*, uma abreviação de ***W**eb**A**ssembly Powered **R**eact **C**lone*, que pode ser entendida como um protótipo de uma possível versão do React, que em vez de implementar os mecanismos de reconciliação em JavaScript, o faz com o uso de WebAssembly internamente.

Esta seção descreve todos os aspectos técnicos a respeito do desenvolvimento do Warc. É importante ressaltar que o React não possui uma especificação técnica de implementação, logo, a construção do protótipo foi realizada com base na análise de seu código-fonte (a partir de seu repositório no Github[4]) e sua documentação oficial [4].

Além disso, outros materiais de referência técnica foram de suma importância para a conclusão do protótipo, como os artigos *Inside Fiber: in depth overview of the new reconciliation algorithm in React*, de Max Koretskyi [48] e *Build Your Own React*, de Rodrigo Pombo [49], os quais exploram detalhadamente o funcionamento interno do React.

---

[3] Awesome Wasm*: Curated list of awesome things regarding the Wasm ecosystem.* Disponível em: <https://github.com/mbasso/awesome-wasm>

[4] Repositório do React no Github. Disponível em: <https://github.com/facebook/react/>

Devido ao tempo limitado para o desenvolvimento do presente trabalho, o protótipo implementa apenas um subconjunto das funcionalidades que o React oferece atualmente, mas ainda será possível a existência de componentes cujos códigos possam ser compartilhados entre as duas bibliotecas com mudanças mínimas. Uma lista de funcionalidades do React que não fazem parte do escopo pode ser lida na subseção 3.2.

3.1. Objetivos em alto nível

Usuários hipotéticos do Warc que utilizem-no para desenvolver aplicações devem esperar que ele seja capaz de realizar três tarefas básicas: manipulação do DOM, componentização e criação de estado local, as quais serão descritas com mais detalhes nas subseções seguintes.

3.1.1. Manipulação do DOM

O Warc deve ser capaz de criar, atualizar e remover elementos no DOM de acordo com o que é definido pelo usuário. A principal maneira de declarar elementos é utilizar o JSX [24]. Toda aplicação desenvolvida com o React, e também o Warc, deve gerar uma árvore de elementos que serão processados e renderizados automaticamente.

3.1.2. Componentização

O Warc deve permitir a composição de elementos em componentes. Compor diversos componentes é a principal estratégia no React para construir interfaces robustas, e esse mesmo padrão deve ser utilizado também no Warc.

3.1.3. Gerenciamento de estado

O Warc deve oferecer o hook `useState` para criar estado local em componentes, concluindo a tríade das tarefas básicas de responsabilidade da biblioteca. Dessa forma, uma aplicação desenvolvida com o Warc será capaz de construir seu layout e manipulá-lo para refletir seu estado.

3.2. Funcionalidades fora do escopo

O React oferece diversas outras funcionalidades além das discutidas anteriormente, mas nem todas se encontrarão dentro do escopo do que será implementado no Warc devido ao tempo restrito para o desenvolvimento do protótipo. São elas:

- **Componentes via classes**: componentes no React também podem ser declarados como classes ES6 [23]. No Warc apenas componentes funcionais serão suportados;
- **Ciclo de vida**: classes no React permitem o uso de métodos como `componentDidMount` e `shouldComponentUpdate`, que fazem parte do ciclo de vida de um componente [50]. Porém, isso não será suportado pelo Warc.
- **Eventos sintéticos**: o React oferece a classe `SyntheticEvent`, usado para igualar o comportamento de eventos entre diferentes navegadores e otimizar o desempenho de *event listeners* [51]. No Warc, somente eventos nativos serão utilizados.
- **Atributo `key`**: No React utiliza-se o atributo *key* ao renderizar uma lista de elementos [52], importante para otimizações na reconciliação. No Warc, essa otimização não será implementada, logo o uso do atributo `key` não é necessário.
- **Outros hooks**: Alguns hooks importantes para a manutenção do estado e do ciclo de vida, como o `useEffect` e o `useContext`, e outros menos comuns como `useCallback`, `useReducer` e `useMemo` [53] não serão implementados.

3.3. Arquitetura interna

O Warc será utilizado por usuários como uma biblioteca JavaScript comum, mas seu objetivo final é delegar as operações complexas do virtual DOM a um módulo WebAssembly. O Warc é formado por dois módulos, JS e Wasm, e sua arquitetura interna reflete as interações entre esses módulos.

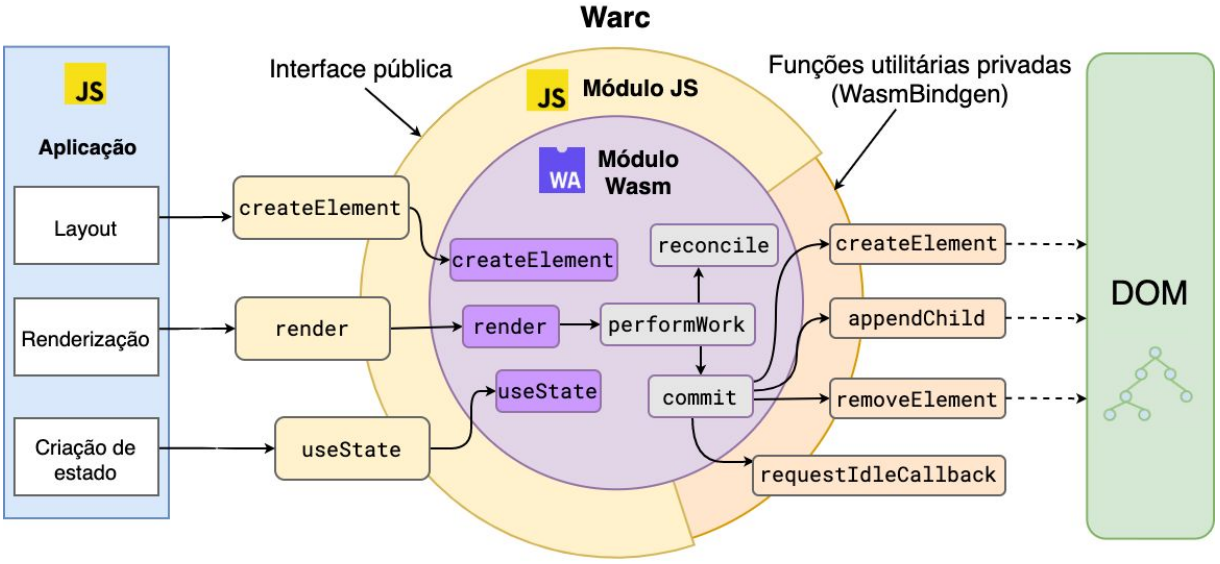


Figura 4: Diagrama da arquitetura interna do Warc

No diagrama da Figura 4 é possível visualizar o funcionamento interno do Warc. A aplicação utiliza funções do módulo JS para criar elementos, renderizá-los e criar estado, que por sua vez utilizam funções do módulo Wasm para criar o virtual DOM e executar a reconciliação. Então, o módulo Wasm utiliza funções geradas pela ferramenta WasmBindgen [54] para ter acesso ao DOM e aplicar as mudanças na tela. Esse processo será detalhado nas subseções seguintes.

3.3.1. Módulo JS

O Warc contará com uma camada de código JavaScript, que será chamada de Módulo JS, que funciona como intermediária ao módulo Wasm. Com isso, usuários não terão a necessidade de lidar diretamente com o WebAssembly - que passa a ser um mero detalhe de implementação - e simplesmente fazem o uso de funções JavaScript.

O módulo JS também conterá uma série de funções de uso privado do módulo Wasm (geradas automaticamente pela ferramenta WasmBindgen [54], a ser detalhada na próxima subseção), permitindo-o manipular o DOM e acessar outras APIs do navegador. A interface pública do módulo JS deverá conter as seguintes funções:

`createElement`. Será responsável pela declaração de elementos que serão gerenciados pelo Warc para a construção do layout da aplicação. Essa função não deve ser usada diretamente pelo usuário da biblioteca, pois um transpilador como o Babel [55] pode transformar a utilização da sintaxe JSX [24] para chamadas a `createElement`:

```
let element = <div className="greeting">Hello world!</div>;
// resultado após o uso do Babel:
let element = createElement("div", { className: "greeting" }, createElement("Hello world!"));
```

`render`. Como discutido anteriormente, renderiza os elementos na tela. Internamente, sua chamada informará ao virtual DOM a árvore de elementos que deverá ser renderizada e qual elemento já existente no DOM deverá conter essa árvore.

`useState`. É um dos Hooks [53] introduzidos na versão 16.8 do React [25], que permite que um componente funcional possa conter e controlar dados dinâmicos que são persistidos a cada ciclo de renderização.

`load`. É responsável por carregar o módulo WebAssembly em memória para compilação das instruções binárias [39]. Com isso, é apenas necessário envolver a chamada a `render` em um callback:

```
load().then(() => {
  render(<App />, document.getElementById("root"));
});
```

3.3.2. Módulo Wasm

O núcleo do Warc será um módulo WebAssembly, responsável por guardar o contexto interno e executar todas as operações do virtual DOM. O funcionamento do virtual DOM, juntamente com as principais funções desse módulo, podem ser descritos a partir dos seguintes passos:

Criação de elementos. Cada elemento criado por JSX resultará numa chamada a `createElement`, que irá alocar em sua memória linear um novo objeto `VolatileElement`, um elemento volátil que servirá de base para a criação do virtual DOM. Seu retorno pode ser passado como argumento para a função `render` para que o elemento correspondente seja renderizado.

Processamento de elementos. A função `render` do módulo JS realiza uma chamada à função homônima do módulo Wasm, a qual inicia a criação ou manutenção do virtual DOM, uma árvore de objetos `Fiber` onde cada um representa um elemento do DOM. Na primeira chamada a `render`, para cada `VolatileElement` um novo `Fiber` é criado e é anexado à árvore do virtual DOM.

Reconciliação. Quando há mudanças de estado, uma nova árvore de objetos `VolatileElement` é gerada. Logo, os objetos `Fiber` da árvore do virtual DOM existente são utilizados na função `reconcile` para verificar se os novos `VolatileElement` representam alguma mudança no layout da aplicação. Assim, todas as propriedades e elementos filhos são comparados. Quando uma mudança é detectada em um `Fiber`, é adicionado a uma lista de mudanças. Após todos os fibers serem comparados, a reconciliação termina.

Aplicação de mudanças no DOM. Neste último passo, a função `performWork` do módulo WebAssembly itera sobre os fibers marcados com mudanças e invoca a função `commit` para cada um. `commit` irá utilizar as funções utilitárias do módulo JS para realizar cada mudança no DOM. Após esse processo, o DOM real e o virtual DOM estarão sincronizados.

3.4. Ferramentas de desenvolvimento

O módulo Wasm do Warc foi desenvolvido na linguagem **Rust** [40] devido ao seu foco em performance e confiabilidade, além de seu amplo suporte ao desenvolvimento de WebAssembly [56]. Com isso, também foram utilizadas várias ferramentas de seu ecossistema, como as bibliotecas **WasmBindgen** [54] (geração automática de código intermediário entre JS e Wasm), **web_sys** e **js_sys** (APIs Web e JavaScript via WebAssembly), e o utilitário de linha de comando **wasm-pack**.

O **Babel** é um compilador de JavaScript de código-aberto [55] que permite utilizar versões mais recentes da linguagem sem a necessidade de lidar diretamente com a

compatibilidade com navegadores que não suportam funcionalidades novas. Existem diversos plugins e *presets* para o Babel, e um deles, o *preset-react* foi utilizado no Warc para fazer a transformação da sintaxe JSX em chamadas à função *createElement*.

O **Webpack** [57] é outra ferramenta amplamente usada no ecossistema JavaScript como empacotador de módulos, sendo capaz de unir arquivos de diferentes extensões em um único *bundle* (pacote). A sua estratégia de importação dinâmica [58] de módulos foi essencial para o carregamento do módulo Wasm junto ao código JavaScript.

3.5. Resultado do desenvolvimento

O desenvolvimento da ferramenta Warc durou aproximadamente três meses, sendo a maior parte desse tempo dedicada à construção do módulo WebAssembly. Seu código-fonte é aberto na licença MIT e pode ser encontrado no repositório: https://github.com/gilesv/warc.

4. Metodologia

Este trabalho foi conduzido com base na metodologia GQM (*Goal/Question/Metric*), proposta por Victor Basili [45] como um processo de medição de dados orientado a objetivos a partir dos quais são derivadas questões, que devem ser respondidas para determinar se os objetivos foram alcançados. Essa metodologia servirá como guia para a investigação acerca do uso de WebAssembly em virtual DOMs.

Para isso, serão realizados testes de benchmark em aplicações desenvolvidas com as ferramentas React e Warc para possibilitar uma análise acerca do desempenho das ferramentas com base nos resultados obtidos. Assim, ficará claro se o uso de WebAssembly é capaz de prover maior eficiência em um virtual DOM para aplicações JavaScript e quais características ou limitações dessa tecnologia são responsáveis pela conclusão.

4.1 Plano GQM

O plano do GQM definido para o trabalho será detalhado nas próximas subseções.

4.1.1. Objetivo

O objetivo primário do trabalho é *Analisar a aplicabilidade da tecnologia WebAssembly na implementação de um Virtual DOM a fim de melhorar a eficiência de aplicações web do ponto de vista do usuário.*

4.1.2. Questões

Com base no objetivo do trabalho, foram elaboradas as seguintes questões:

Questão 1: ***O quão eficiente é o Virtual DOM implementado pelo React?***

Nesta primeira questão queremos obter uma visão objetiva da eficiência da biblioteca React no que diz respeito ao funcionamento de seu mecanismo de virtual DOM, desenvolvido na linguagem JavaScript, e estabelecer uma base para comparação para os resultados dos testes de benchmark.

Questão 2: ***O quão eficiente é o Virtual DOM implementado pelo Warc?***

Nesta segunda questão, o objetivo é obter uma visão objetiva da eficiência da ferramenta experimental Warc, que utiliza um virtual DOM desenvolvido com a tecnologia WebAssembly.

Questão 3: ***É possível obter maior eficiência com o Virtual DOM do Warc frente ao React em aplicações web?***

Nesta última questão, buscamos comparar as duas abordagens de virtual DOMs (React e Warc) com base em benchmarks em aplicações de teste para entender os ganhos trazidos ao caso de uso do virtual DOM pela tecnologia WebAssembly.

4.1.3. Métricas

A partir das questões acima levantadas, um total de 12 métricas foram identificadas:

4.1.3.1. Tamanho da aplicação (S)

O tamanho de uma aplicação (***S**ize*) refere-se à soma dos tamanhos, em *kilobytes*, de todos os artefatos necessários para executar uma aplicação. Tal informação é importante para mensurar o impacto do tamanho da implementação do virtual DOM no código da aplicação final.

- $\mathbf{S^R}$ será definida pelo tamanho da aplicação de teste desenvolvida com o React e está associada com a questão **Q1**.
- $\mathbf{S^W}$ será definida pelo tamanho da aplicação de teste desenvolvida com o Warc e está associada com a questão **Q2**.

4.1.3.2. Tempo de carregamento (L)

O tempo de carregamento de uma página (***L**oading*) é um fator importante para o desempenho de uma aplicação do ponto de vista do usuário. É possível mensurá-lo com o *First Contentful Paint* (FCP) [47], fornecido pelo Google Chrome, que mede o intervalo de tempo entre o momento que o usuário acessa a página da aplicação até que o primeiro conteúdo no DOM seja renderizado. No contexto do trabalho, o FCP permite entender quanto tempo é preciso para que o React e o Warc criem seus virtual DOMs e apliquem mudanças no DOM real.

- A métrica $\mathbf{L^R}$ será definida pelo FCP médio da aplicação de teste desenvolvida com o React e está associada com a questão **Q1**.
- A métrica $\mathbf{L^W}$ será definida pelo FCP médio da aplicação de teste desenvolvida com o Warc e está associada com a questão **Q2**.

4.1.3.4. Tempo de execução e renderização (E)

Tempo de execução e renderização (***E**xecution*) refere-se ao tempo, em milissegundos, utilizado pelo navegador em operações e eventos específicos durante o funcionamento da aplicação. Esses eventos podem ser classificados nas seguintes categorias [46, 14]:

- **Scripting**: execução de código JavaScript e WebAssembly; alocação e liberação de memória (*garbage collection*); captura de eventos do DOM;
- **Rendering**: processamento do layout da página (*reflows*);
- **Painting**: coloração da página com base em cálculos de layout (*repaints*).

Para o escopo desta investigação as categorias de *scripting*, *rendering* e *painting* serão agregadas na métrica **SRP**, que é definida por:

$$\mathbf{SRP = T_S + T_R + T_P}$$

Dado que

- $\mathbf{T_S}$ = Tempo médio de *scripting*
- $\mathbf{T_R}$ = Tempo médio de *rendering*
- $\mathbf{T_P}$ = Tempo médio de *painting*

Logo,

- A métrica $\mathbf{E^R}$ será definida pela média aritmética das medianas de SRP em cada cenário de teste executado na aplicação desenvolvida com o React e está associada com a questão **Q1**.
- A métrica $\mathbf{E^W}$ será definida pela média aritmética das medianas de SRP em cada cenário de teste executado na aplicação desenvolvida com o Warc e está associada com a questão **Q2**.

4.1.3.5. Consumo de memória (M)

As métricas de consumo de memória (***M****emory*) são definidas pela média do valor máximo, em megabytes, ocupado pela aplicação durante o seu funcionamento. Esse valor refere-se a métrica *JS Heap* utilizada pelo Google Chrome, que mede o tamanho de todos os objetos JavaScript alocados pela aplicação [46].

- A métrica $\mathbf{M^R}$ será definida pela média do consumo máximo de memória da aplicação de teste desenvolvida com o React e está associada com a questão **Q1**.
- A métrica $\mathbf{M^W}$ será definida pela média do consumo máximo de memória da aplicação de teste desenvolvida com o Warc e está associada com a questão **Q2**.

4.1.3.6. Variação percentual de eficiência (V)

Para responder à questão **Q3**, os impactos do uso de WebAssembly na ferramenta Warc serão avaliados a partir do cálculo da variação percentual entre cada par de métricas de mesmo tipo. Caso haja uma redução percentual em dada métrica, o Warc foi mais eficiente; caso a haja um um aumento percentual, o Warc foi menos eficiente.

- A métrica $\mathbf{V^S}$ determinará a variação percentual entre o tamanho das duas aplicações de teste ($\mathbf{S^R}$ e $\mathbf{S^W}$). Logo, $\mathbf{V^S = (S^W / S^R - 1) \times 100}$.
- A métrica $\mathbf{V^L}$ determinará a variação percentual entre os tempos de carregamento das duas aplicações de teste (L^R e L^W). Logo, $\mathbf{V^L = (L^W / L^R - 1) \times 100}$.
- A métrica $\mathbf{V^E}$ determinará a variação percentual entre os tempos de SRP das duas aplicações de teste (E^R e E^W). Logo, $\mathbf{V^E = (E^W / E^R - 1) \times 100}$.
- A métrica $\mathbf{V^M}$ determinará a variação percentual entre o consumo de memória das duas aplicações de teste (M^R e M^W). Logo, $\mathbf{V^M = (M^W / M^R - 1) \times 100}$.

4.2. Coleta de dados

Após a conclusão do desenvolvimento do protótipo Warc, foi necessário implementar aplicações e elaborar cenários para os testes de benchmark. Esta seção descreve todos os passos do processo de coleta de dados deste trabalho.

4.2.1. Desenvolvimento das aplicações de teste: *TodoMVC*

Para realizar a coleta de dados de desempenho, foi desenvolvida a aplicação *TodoMVC* [59] para exemplificar cenários de manipulação do DOM (criação, atualização e remoção de elementos). TodoMVC é um projeto bastante conhecido na comunidade que é implementado em cada framework existente para comparar as diferentes tecnologias. É uma lista de afazeres, sendo composta por um campo de texto, uma lista de itens e botões de filtragem. É possível criar, editar, marcar itens como concluídos ou removê-los.

Para o propósito de comparação, duas versões da aplicação TodoMVC foram desenvolvidas. Uma delas utiliza o React (*bundle* de produção, versão 16.13.1) como

dependência, e a outra utiliza o Warc, a ferramenta desenvolvida neste trabalho. Ambas as versões compartilham as mesmas funcionalidades e até mesmo grande parte do código fonte.

4.2.2. Procedimento de coleta

Para realizar a coleta de dados dos testes de benchmark nas aplicações foi utilizada a ferramenta e o Chrome DevTools [60] para monitorar métricas de desempenho da aplicação durante as interações e gerar relatórios, e a biblioteca Puppeteer [61] para executar interações na aplicação de forma automatizada.

O Chrome DevTools é um conjunto de ferramentas de desenvolvimento providas pelo navegador Chrome para auxiliar na otimização de páginas web, capaz de mensurar informações da aplicação e exibi-las em gráficos interativos [14] (ver Figura 5). O Puppeteer é uma biblioteca Node.js capaz de simular ações de um usuário na aplicação (clicar em botões, preencher formulários) de forma automatizada.

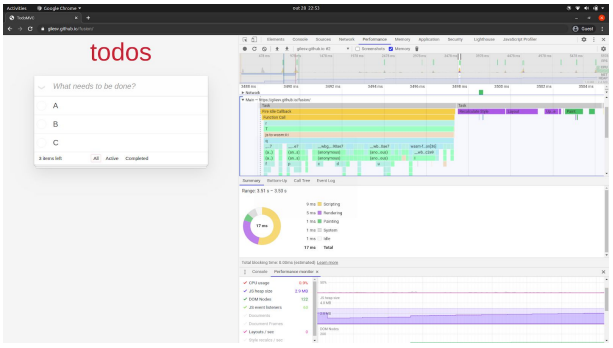


Figura 5: Painel de performance do Chrome DevTools na aplicação de teste TodoMVC

Dessa forma, utilizando o Puppeteer para automatizar interações e Chrome DevTools para monitorar e gerar relatórios de desempenho, é possível coletar todos os dados necessários para a análise comparativa entre Warc e React.

4.2.3. Cenários de teste

Os testes de benchmark executados pelo script do Puppeteer tanto na aplicação TodoMVC desenvolvida com Warc e a TodoMVC desenvolvida com React estão detalhados na Tabela 2.

Cenário	Objetivos	Parâmetros
(A) Carregamento	Obter o FCP médio de FCP após a primeira construção do VDOM/DOM	50, 150 e 300 itens
(B) Criação de itens	Obter métricas (SRP, memória) durante a construção do DOM/VDOM	50, 150 e 300 itens
(C) Filtragem de itens	Obter métricas (SRP, memória) durante reutilização de elementos DOM/VDOM	50, 150 e 300 itens
(D) Remoção de itens	Obter métricas (SRP, memória) durante remoção de elementos DOM/VDOM	50, 150 e 300 itens

Tabela 2: Cenários realizados nos testes de benchmark

Para cada cenário foram considerados 50, 150 e 300 itens na aplicação, nos quais cada item é um componente `TodoItem` renderizado. Cada cenário foi realizado 10 vezes seguidas para calcular a média dos valores das métricas de desempenho. O código do *script* que executa os cenários pode ser encontrado no repositório do trabalho no Github[5]. A execução foi realizada com as seguintes configurações:

- Processador: Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
- Placa gráfica: Intel® UHD Graphics 620 (Kabylake GT2)
- Memória RAM: 8GB
- Disco: 240 GB SSD PCIe
- Sistema operacional: Ubuntu 19.10 64-bit
- Navegador: Google Chrome - Versão 85.0.4183.121 (Official Build) (64-bit)

5. Resultados

Nesta seção serão apresentados os resultados coletados a partir da execução dos testes de benchmark nas aplicações. Cada resultado será condensado nas métricas definidas no plano GQM. A discussão acerca desses resultados será apresentada na seção posterior.

5.1. Q1 - *O quão eficiente é o Virtual DOM implementado pelo React?*

Nas subseções seguintes, serão apresentados os resultados dos testes de benchmark compilados nas métricas relacionadas à Questão 1: tamanho da aplicação ($\mathbf{S^R}$), tempo de carregamento ($\mathbf{L^R}$), tempo de execução e renderização ($\mathbf{E^R}$) e consumo de memória ($\mathbf{M^R}$).

5.1.1. Tamanho da aplicação ($\mathbf{S^R}$)

Os valores dos tamanhos dos artefatos integrantes da aplicação TodoMVC desenvolvida com o React são apresentados abaixo na Tabela 3. Logo, concluímos que $\mathbf{S^R}$ = **146 kB**.

Artefato	Tamanho
HTML	6 kB
CSS	3 kB
JavaScript	137 kB
Total (S^R)	**146 kB**

Tabela 3: Tamanho da aplicação React

5.1.2. Tempo de carregamento ($\mathbf{L^R}$)

Os valores médios do *First Contentful Paint* (FCP) na aplicação TodoMVC desenvolvida com o React foi coletada a partir do cenário A, que consiste em acessar a página com 50, 150 e 300 *TodoItems* preexistentes para mensurar o tempo necessário para que o React crie o virtual DOM e aplique as mudanças no DOM, sendo desconsiderado o tempo de download dos artefatos (arquivos HTML, CSS, JavaScript). Na Figura 6 é possível observar que o valor de FCP cresce de acordo com a quantidade de itens. Calculando a média aritmética entre os valores de FCP dos três cenários, concluímos que $\mathbf{L^R}$ = **230,3 ms**.

[5] Repositório da coleta de dados no Github: https://github.com/gilesv/tg/tree/master/measuring

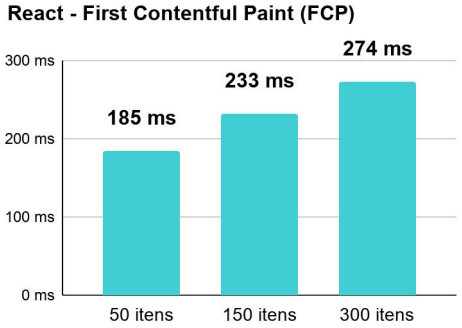


Figura 6: FCP médio para a aplicação React em cada cenário

5.1.3. Tempo de execução e renderização (E^R)

Os valores médios de *scripting*, *rendering* e *painting* (SRP) na aplicação TodoMVC desenvolvida com o React foi coletada a partir dos cenários B, C e D, e cada um com 50, 150 e 300 *TodoItems* preexistentes na página. Na Figura 7 é possível observar que o cenário B teve os valores mais altos, seguidos dos cenários D e C. Ao calcular a média aritmética dos valores de mediana de cada cenário apresentados na Tabela 4, conclui-se que **E^R = 1088,1 ms**.

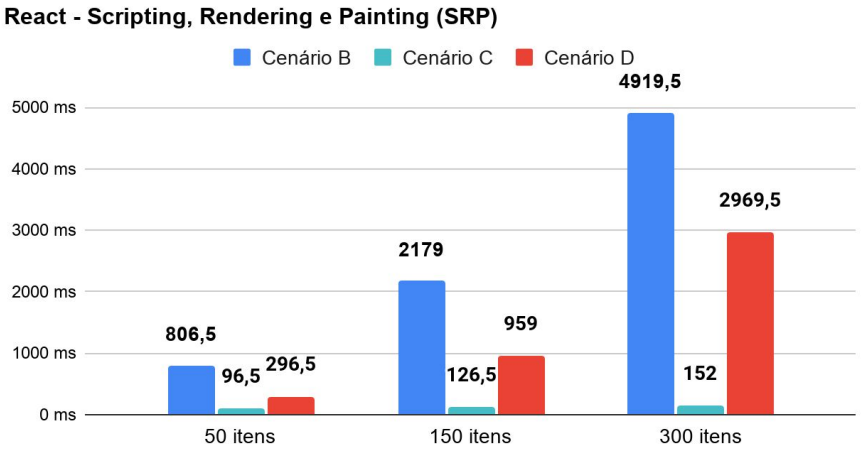


Figura 7: SRP médio para a aplicação React em cada cenário

Cenário	Mediana do SRP
B	2179 ms
C	126,5 ms
D	959 ms
Média (E^R)	**1088,1 ms**

Tabela 4: SRP médio total da aplicação React

5.1.4. Consumo de memória (M^R)

Os valores médios do consumo de memória na aplicação TodoMVC desenvolvida com o React foi coletada também a partir dos cenários B, C e D, cada um com 50, 150 e 300 *TodoItems* preexistentes na página. Na Figura 8 é possível observar a progressão do consumo

de memória. Ao calcular a média aritmética dos valores de cada cenário apresentados na Tabela 5, conclui-se que **M^R = 6,5 MB**.

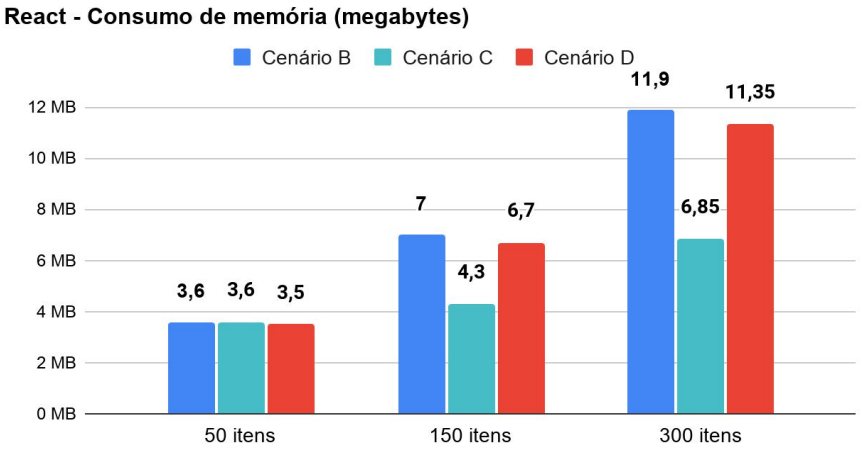


Figura 8: Consumo médio de memória para a aplicação React em cada cenário

Cenário	Consumo médio de memória
B	7,5 MB
C	4,9 MB
D	7,1 MB
Média (M^R)	**6,5 MB**

Tabela 5: Consumo médio de memória da aplicação React

5.1.5. Resumo da Questão 1

Na Tabela 6 são resumidos os valores das métricas associadas à Questão 1:

Tamanho (S^R)	Carregamento (L^R)	Execução (E^R)	Memória (M^R)
146 kB	230,3 ms	1088,1 ms	6,5 MB

Tabela 6: Resumo de métricas da aplicação React (Questão 1)

5.2. Q2 - *O quão eficiente é o Virtual DOM implementado pelo Warc?*

Nas subseções seguintes, serão apresentados os resultados dos testes de benchmark compilados nas métricas relacionadas à Questão 2: tamanho da aplicação (**S^W**), tempo de carregamento (**L^W**), tempo de execução e renderização (**E^W**) e consumo de memória (**M^W**).

5.2.1. Tamanho da aplicação (S^W)

Os valores dos tamanhos dos artefatos integrantes da aplicação TodoMVC desenvolvida com o Warc são apresentados abaixo na Tabela 7.

Artefato	Tamanho
HTML	6 kB
CSS	3 kB
JavaScript	7,8 kB
WebAssembly	35, kB
Total (S^W)	**51,8 kB**

Tabela 7: Tamanho da aplicação Warc

Com isso, concluímos que $\mathbf{S^W = 51,8\ kB}$.

5.2.2. Tempo de carregamento (L^W)

Os valores médios do *First Contentful Paint* (FCP) na aplicação TodoMVC desenvolvida com o Warc foi coletada a partir do cenário A com 50, 150 e 300 *TodoItems* preexistentes na página, a fim de mensurar o tempo necessário para que o Warc crie o virtual DOM e aplique as mudanças no DOM, sendo desconsiderado o tempo de download dos artefatos (arquivos HTML, CSS, JavaScript e WebAssembly). Calculando a média aritmética entre os valores de FCP dos três cenários vistos na Figura 9, concluímos que $\mathbf{L^W = 213,3\ ms}$.

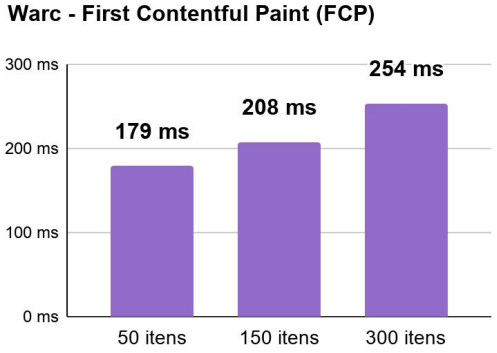


Figura 9: FCP médio para a aplicação Warc em cada cenário

5.2.3. Tempo de execução e renderização (E^R)

Os valores médios de *scripting*, *rendering* e *painting* (SRP) na aplicação TodoMVC desenvolvida com o React foi coletada a partir dos cenários B, C e D, cada um com 50, 150 e 300 *TodoItems* preexistentes na página. Na Figura 10 é possível observar que o cenário B teve os valores mais altos, seguidos dos cenários D e C. Ao calcular a média aritmética dos valores de mediana de cada cenário apresentados na Tabela 8, conclui-se que $\mathbf{E^W = 2043,8\ ms}$.

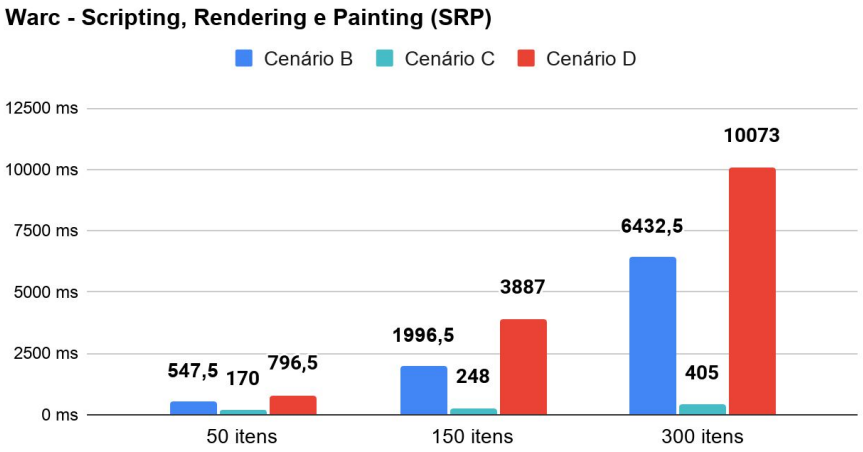


Figura 10: SRP médio para a aplicação Warc em cada cenário

Cenário	Mediana do SRP
B	1996,5 ms
C	248 ms
D	3887 ms
Média (E^R)	**2043,8 ms**

Tabela 8: SRP médio total da aplicação Warc

5.2.4. Consumo de memória (M^W)

Os valores médios do consumo de memória na aplicação TodoMVC desenvolvida com o React foi coletada também a partir dos cenários B, C e D, cada um com 50, 150 e 300 *TodoItems* preexistentes na página. Na Figura 11 é possível observar a progressão do consumo de memória. Ao calcular a média aritmética dos valores de cada cenário apresentados na Tabela 9, conclui-se que **M^R = 12,7 MB**.

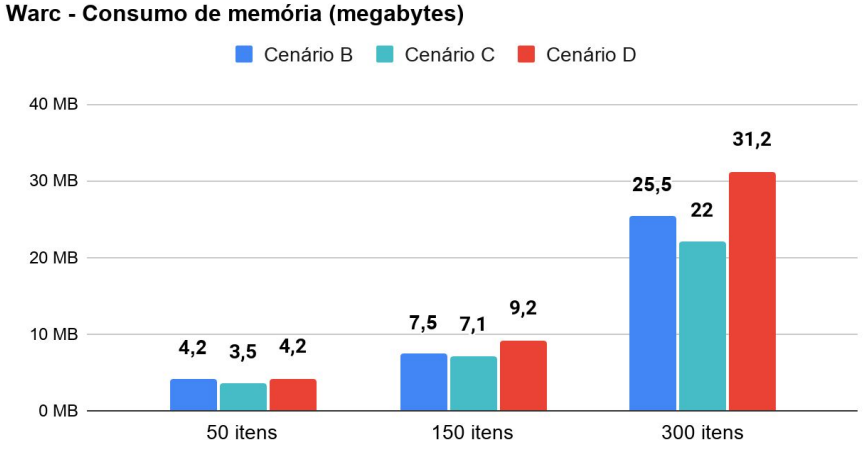


Figura 11: Consumo médio de memória para a aplicação Warc em cada cenário

Cenário	Consumo médio de memória
B	12,4 MB
C	10,8 MB
D	14,8 MB
Média (M^W)	**12,7 MB**

Tabela 9: Consumo médio de memória da aplicação Warc

5.2.5. Resumo da Questão 2

Na Tabela 10 são apresentadas de forma resumida os valores das métricas associadas à Questão 2:

Tamanho (S^W)	Carregamento (L^W)	Execução (E^W)	Memória (M^W)
51,8 kB	213,3 ms	2043,8 ms	12,7 MB

Tabela 10: Resumo de métricas da aplicação Warc (Questão 2)

5.3. Q3 - *É possível obter maior eficiência com o Virtual DOM do Warc frente ao React em aplicações web?*

A partir de todos os valores das métricas de tamanho (S^R e S^W), carregamento (L^R e L^W), execução e renderização (E^R e E^W) e consumo de memória (M^R e M^W), foi possível calcular a variação percentual (V) de cada uma de acordo com a seção 4.1.3. Os resultados das variações percentuais as quais podem ser visualizadas na Tabela 11.

V^S	V^L	V^E	V^M
-0,645	-0,073	0,878	0,953

Tabela 11: Resumo de métricas de variação percentual entre React e Warc

Com base nos dados coletados em testes de benchmark em aplicações de testes desenvolvidas com o React e o Warc, pode-se observar na Tabela 12 que o Warc, a ferramenta proposta por este trabalho, foi capaz de aumentar a eficiência nas métricas de tamanho da aplicação (S) e tempo de carregamento (L), nas quais possibilitou reduções percentuais de **-64,5%** e **-7,3%** respectivamente. Por outro lado, o protótipo piorou a eficiência nas métricas de execução e renderização (E) e consumo de memória (M), nas quais houve aumentos percentuais de **87,8%** e **95,3%** respectivamente.

Logo, apesar de haver ganhos notáveis em alguns pontos, a análise do presente trabalho indica que a utilização da tecnologia WebAssembly, em seu estado atual, no Virtual DOM da ferramenta Warc não apresentou melhora substancial em eficiência frente ao React de forma geral. Na próxima seção, serão apresentados e discutidos alguns pontos técnicos que podem justificar o resultado dessa análise no que diz respeito aos impactos positivos e negativos do uso de WebAssembly no virtual DOM do Warc.

Métrica	React (JS)	Warc (WebAssembly)	Variação (V)
Tamanho (S)	146 kB	51,8 kB	**-64,5%**
Carregamento (L)	230,3 ms	213,3 ms	**-7,3%**
Execução e renderização (E)	1088,1 ms	2043,8 ms	**+87,8%**
Consumo de memória (M)	6,5 MB	12,7 MB	**+95,3%**

Tabela 12: Visão geral da comparação de eficiência entre React e Warc

6. Discussão

O Warc, a ferramenta experimental desenvolvida neste trabalho, possibilitou a criação de uma aplicação web desenvolvida em JavaScript utilizando um virtual DOM implementado na tecnologia WebAssembly. Essa aplicação foi submetida a vários testes de benchmark com o objetivo de analisar a aplicabilidade de Wasm em busca de um virtual DOM mais eficiente.

Os resultados obtidos indicaram que o uso de WebAssembly provocou uma diminuição no tamanho e no tempo de carregamento da aplicação. Porém, ocorreram elevações nos tempos de execução e renderização e no consumo de memória. Em posterior análise, descobriu-se que limitações existentes na interoperabilidade entre JavaScript e Wasm são responsáveis pelos resultados negativos. A expectativa atual é que essas limitações são contornáveis, e serão resolvidas com a adição de novas funcionalidades à especificação do WebAssembly [62, 63, 65].

A seguir estão detalhadas justificativas e observações realizadas durante a análise dos resultados de cada métrica de eficiência.

6.1. Carregamento e tamanho da aplicação

O uso do Warc resultou em uma diminuição de 64% no tamanho total da sua aplicação, visto que ela possui 35 kB de módulo Wasm e 7,8 kB de código Javascript, enquanto a aplicação desenvolvida com o React possui um total de 137 kB de código JavaScript. Com isso, entende-se que o Warc pôde diminuir o tamanho total devido a representação binária do módulo WebAssembly torná-lo menor e mais compacto.

Os resultados obtidos também indicam que o Warc ofereceu um tempo de carregamento 7,3% menor em relação ao React. O tempo de carregamento é composto pela análise e compilação do código (JavaScript ou Wasm) e do processamento do virtual DOM. Visto que a análise e compilação de um módulo WebAssembly é mais rápida [31], o Warc é capaz de inicializar o virtual DOM mais rapidamente, contribuindo para um menor tempo de carregamento.

6.2. Execução do Virtual DOM

A execução das operações do Virtual DOM realizada pelo módulo WebAssembly, no Warc, mostrou-se mais lenta em comparação ao React, com um aumento de 87% nos tempos de SRP (*scripting*, *rendering* e *painting*). Para descobrir a causa desse aumento foi utilizada a ferramenta *JavaScript Profiler*, do Chrome DevTools, para visualizar os gargalos na execução do código durante as interações dos cenários B, C e D.

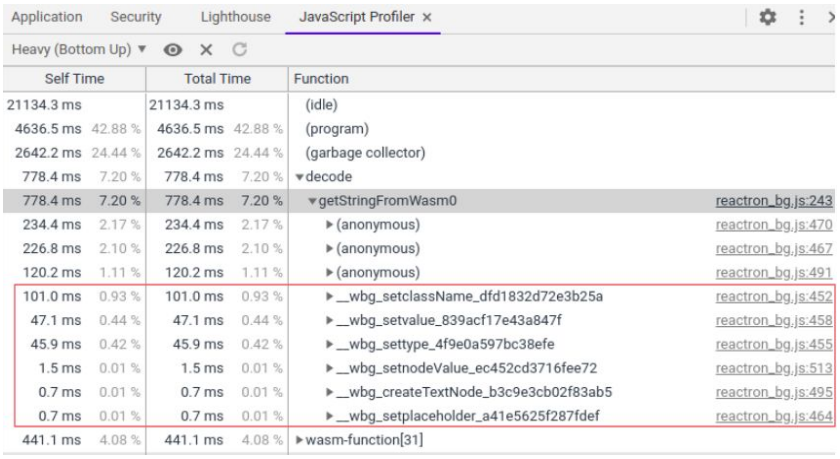


Figura 12: Relatório que indica o tempo de execução de `getStringFromWasm`. Em vermelho estão destacadas as operações da DOM API que necessitam da decodificação de strings

Na Figura 12 é possível visualizar uma lista ordenada de funções mais executadas pelo Warc. Indicado em vermelho está um conjunto de funções geradas pela ferramenta WasmBindgen, responsáveis por executar operações da DOM API, como alterar a classe, tipo, placeholder ou valor de um elemento. Todas elas requerem manipulação de *strings*, logo utilizam a função `getStringFromWasm`, destacada em cinza, responsável por copiar bytes da memória linear do módulo Wasm e decodificá-los em *strings* do JavaScript.

Essa decodificação é necessária porque o WebAssembly não é capaz de lidar com strings do JavaScript, logo, sempre que o virtual DOM precisa de um texto, como o tipo de um elemento, nome de uma classe ou placeholder, uma transformação do objeto *String* para um array de bytes é realizada. Essa operação torna-se custosa quando executada frequentemente, o que está acontecendo no Warc a cada mudança de estado e geração de uma nova árvore de elementos no Virtual DOM.

Na versão atual da tecnologia WebAssembly, sua interoperabilidade com o JavaScript ainda é limitada [31] devido a necessidade da codificação e decodificação binária dos dados. Isso mudará no futuro com a proposta de *Reference Types* [62], que permitirá que objetos JavaScript sejam representados diretamente na memória do WebAssembly por meio de referências.

Já a proposta de *Interface Types* [63] tornará possível que o módulo Wasm tenha acesso direto à DOM API. Dessa forma, o WasmBindgen não precisará gerar uma camada auxiliar de código JavaScript já que haverá uma maneira mais eficiente de acessar o DOM evitando a troca de contexto entre JS e Wasm [62].

Portanto, a perspectiva futura é que os problemas na interoperabilidade entre JavaScript e Wasm sejam resolvidos com o suporte às duas propostas apresentadas, que atualmente encontram-se em fase experimental. Com uma melhor interoperabilidade, espera-se que os tempos de SRP com a utilização de WebAssembly no Virtual DOM diminuam consideravelmente.

6.3. Consumo de memória do Virtual DOM

A partir dos dados coletados nos cenários B, C e D é possível perceber um alto consumo de memória na aplicação desenvolvida com o Warc, com um aumento de 95% em comparação ao consumo de memória do React. Para descobrir a causa desse aumento foi utilizada a ferramenta *Heap Snapshot*, do Chrome DevTools, capaz de monitorar o uso de memória durante a execução do Warc.

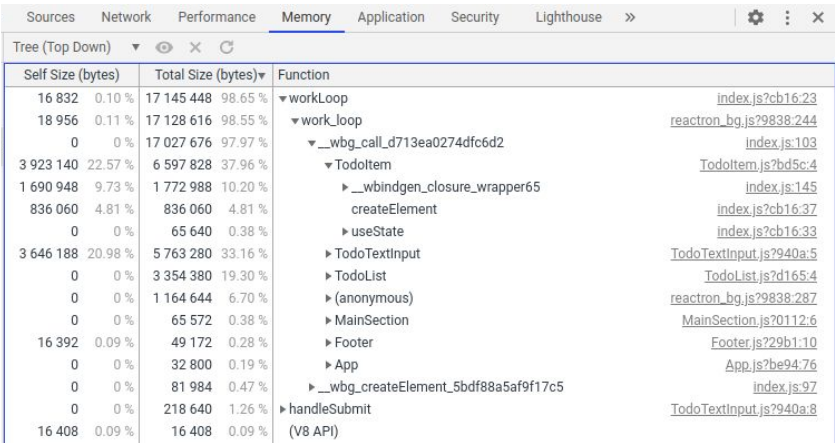


Figura 13: Relatório que indica o tempo de execução de `wbindgen_closure_wrapper65`.

Na Figura 13 é possível visualizar que o componente `TodoItem` é responsável por alocar a maioria dos objetos. A maior parte da alocação desses objetos é realizada a pela chamada da função `wbindgen_closure_wrapper65`, gerada pelo WasmBindgen para permitir que closures (funções anônimas do Rust compiladas no módulo Wasm) possam ser invocadas pelo módulo JS.

A utilização de closures foi necessária na implementação do hook `useState`: cada vez que `setState` é chamada, executa uma closure no módulo Wasm para iniciar a reconstrução do virtual DOM. Porém, closures criadas com a função `Closure::into_js_value` não podem ser descartadas pois não é possível determinar quando elas são executadas. Esse comportamento é explicado na documentação do WasmBindgen, motivo pelo qual o uso de closures deve ser feito com cautela: "*Por padrão essa função vazará memória. Isso pode ser perigoso caso seja chamada várias vezes numa aplicação porque poderá sobrecarregar o módulo WebAssembly"* [64].

Portanto, o aumento de cerca de 95% no consumo de memória do Warc em relação ao React acontece devido a um vazamento de memória causado pela função `Closure::into_js_value`, conforme novas árvores do virtual DOM são geradas a cada mudança de estado. Para resolver esse problema, a proposta *Function References* [65] permitirá a troca nativa e eficiente de funções anônimas entre o contexto JavaScript e um módulo Wasm, o que impedirá que o WasmBindgen vaze memória ao compartilhar closures.

7. Conclusão e direções futuras

O objetivo deste trabalho foi analisar a aplicabilidade da tecnologia WebAssembly (Wasm) na implementação de Virtual DOMs a fim de melhorar a eficiência de aplicações web do ponto de vista do usuário. O meio de investigação foi realizar a implementação do Warc, uma biblioteca experimental baseada no React capaz de delegar as operações do virtual DOM a um módulo Wasm.

Resultados obtidos a partir de testes de benchmark em aplicações desenvolvidas com as ferramentas React e Warc mostraram que a utilização do Warc promoveu uma diminuição no tamanho total da aplicação e seu tempo de carregamento, mas algumas limitações da tecnologia WebAssembly provocaram aumento do processamento e consumo de memória. Dessa forma, conclui-se que a tecnologia ainda precisa de melhorias em sua interoperabilidade com o JavaScript para que seu uso em virtual DOMs seja indicado. Com isso, pode-se considerar que o objetivo do trabalho definido no plano GQM foi cumprido.

Esse trabalho apenas toca a superfície da utilização de WebAssembly em virtual DOMs, então sua conclusão não deve ser encarada como definitiva. Logo, é necessário continuar avaliando o amadurecimento da tecnologia na sua utilização em virtual DOMs. Além disso, como apresentado na discussão, a implementação do Warc apresenta diversos pontos de melhorias. Abaixo estão listadas possíveis oportunidades para trabalhos futuros:

Utilização de propostas do Wasm. Atualmente as propostas de *reference types* [62], *interface types* [63] e *function references* [65], que melhoram a interoperabilidade entre JS e Wasm, estão em fase experimental, mas assim que estiverem disponíveis o Warc poderá ser atualizado para usar as novas funcionalidades. Com isso, sua eficiência poderá ser mensurada em uma nova análise.

Otimizações no Warc. Portar toda a base de código do React em WebAssembly não foi o objetivo deste trabalho. Muitas otimizações importantes e funcionalidades implementadas pelo React, como o uso do atributo *key* e eventos sintéticos, não foram incluídas no escopo de desenvolvimento do Warc, mas podem ser adicionadas futuramente. Além disso, abordagens mais eficientes para o algoritmo de reconciliação podem melhorar os casos em que o Warc se mostrou pior que o React.

Integrar soluções existentes. Além do Warc, outras possibilidades podem ser exploradas a partir da integração de soluções mais robustas, como **asm-dom** [67] ou **dodrio** [68], em uma biblioteca que utilize a API declarativa do React para a criação de aplicações ainda em JavaScript, porém beneficiadas por WebAssembly.

8. Referências bibliográficas

[1] MOLIN, Eric. *Comparison of Single-Page Application Frameworks: A method of how to compare Single-Page Application frameworks written in JavaScript*. 2016.

[2] GIMENO, A. *The deepest reason why modern JavaScript frameworks exist*. Medium, 2018. Disponível em: <https://medium.com/the-reason-why-js-frameworks-exist>. Acesso em 17 Set. 2020.

[3] GROV, Marianne. *Building User Interfaces Using Virtual DOM*. 2015. Dissertação de Mestrado.

[4] Facebook Inc. *Reconciliation*. Disponível em: <https://reactjs.org/docs/reconciliation.html>. Acesso em: 12 Set. 2020.

[5] PERSSON, Morgan. *JavaScript DOM Manipulation Performance: Comparing Vanilla JavaScript and Leading JavaScript Front-end Frameworks*. 2020.

[6] NIELSEN, J. *Website response times*. 2010. Disponível em: <https://www.nngroup.com/articles/website-response-times/>. Acesso em 15 Set. 2020.

[7] World Wide Web Consortium. *W3C brings a new language to the Web*. 2019. Disponível em <https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en>. Acesso em: 12 Set. 2020.

[8] WebAssembly. *High-Level Goals*. Disponível em: <https://webassembly.org/high-level-goals/>. Acesso em: 15 Set. 2020.

[9] WebAssembly. *Use Cases*. Disponível em: <https://webassembly.org/docs/use-cases/>. Acesso em: 12 Set. 2020.

[10] CLARK, Lin. *What WebAssembly means for React*. ReactEurope 2017. Vídeo (33 min). Disponível em: <https://youtube.com/watch?v=3GHJ4cbxsVQ>. Acesso em: 15 Nov. 2020.

[11] MILLER, Derek. *The power of JavaScript*. Cavendish Square Publishing, LLC, 2017.

[12] World Wide Web Consortium: *Document Object Model*. Disponível em: <https://www.w3.org/TR/DOM-Level-2-Core/introduction.html>. Acesso em 4 Ago. 2020.

[13] PAULSON, Linda Dailey. *Building rich web applications with Ajax*. Computer, v. 38, 2005.

[14]. CHĘĆ, Dariusz; NOWAK, Ziemowit. *The performance analysis of web applications based on virtual DOM and reactive user interfaces*. In: KKIO Software Engineering Conference. Springer, Cham, 2018. p. 119-134.

[15] MUYLDERMANS, Daisy. *How does the virtual DOM compare to other DOM update mechanisms in JavaScript frameworks?*. 2019.

[16] MOLIN, Eric. *Comparison of Single-Page Application Frameworks: A method of how to compare Single-Page Application frameworks written in JavaScript*. 2016.

[17] GROV, Marianne. *Building User Interfaces Using Virtual DOM*. 2015. Dissertação de Mestrado.

[18]. TAKADA, Mikito. *Single page apps in depth*. Disponível em: http://singlepageappbook. com/index. html, 2012. Acesso em 15 de novembro de 2020.

[19]. HAKIM, Malek. *Speed index and critical path rendering performance for isomorphic single page applications*. 16th Winona Computer Science Undergraduate Research Symposium. 2016.

[20] STAFF, C. A. C. M. *React: Facebook's functional turn on writing Javascript*. Communications of the ACM, v. 59, n. 12, p. 56-62, 2016.

[21] Stack Overflow Survey 2019. *Most Loved and Dreaded Web Frameworks*. Disponível em: <https://insights.stackoverflow.com/survey/2019>. Acesso em 12 Nov. 2020.

[22] Facebook Inc. *React*. Disponível em: <https://reactjs.org/>. Acesso em 4 Nov. 2020.

[23] Facebook Inc. *Components and Props*. Disponível em: <https://reactjs.org/docs/components.html>. Acesso em 4 Nov. 2020.

[24] Facebook Inc. *Introducing JSX*. Disponível em: <https://pt-br.reactjs.org/docs/introducing-jsx.html>. Acesso em 4 Nov. 2020.

[25] Facebook Inc. *React v16.8.0*. Disponível em: <https://pt-br.reactjs.org/blog/react-v16.8.0.html>. Acesso em 4 Nov. 2020.

[26] Google Inc. Constructing the Object Model. Disponível em: <https://developers.google.com/performance/constructing-the-object-model>. Acesso em 4 Ago. 2020.

[27] MDN. *Web APIs*. Disponível em: <https://developer.mozilla.org/docs/Web/API/Document>. Acesso em 20 Out. 2020.

[28] Google Inc. *Browser reflow*. Disponível em: <https://developers.google.com/speed/docs/insights/browser-reflow>. Acesso em 4 Ago. 2020.

[29] CHEDEAU, Christopher. *React's diff algorithm*. 2017. Disponível em: <https://calendar.perfplanet.com/2013/diff/>. Acesso em 9 de setembro de 2020.

[30] REISER, Micha; BLÄSER, Luc. *Accelerate JavaScript applications by cross-compiling to WebAssembly*. In: Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. 2017. p. 10-17.

[31] HAAS, Andreas et al. *Bringing the web up to speed with WebAssembly*. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. p. 185-200.

[32] Google Inc. V8. Disponível em: <https://v8.dev/>. Acesso em 12 de novembro de 2020.

[33] Asm.js. *FAQ*. Disponível em: <http://asmjs.org/faq.html>. Acesso em: 14 de novembro de 2020.

[34] WebAssembly. *Developers Guide*. Disponível em: <https://webassembly.org/getting-started/developers-guide/>. Acesso em 12 de novembro de 2020.

[35] WebAssembly. *Roadmap*. Disponível em: <https://webassembly.org/roadmap/>. Acesso em 12 de novembro de 2020.

[36] Chromium Blog. *WebAssembly brings Google Earth to more browsers*, 2019. Disponível em: <https://blog.chromium.org/webassembly-brings-google-earth.html>. Acesso em 12 Nov. 2020.

[37] Figma. *WebAssembly cut Figma's load time by 3x*, 2017. Disponível em: <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>. Acesso em 12 Nov. 2020.

[38] WebAssembly Spec. Disponível em: <https://webassembly.github.io/spec/introduction.html> Acesso em 12 de novembro de 2020.

[39] MDN. *WebAssembly: Using the JavaScript API.* Disponível em: <https://developer.mozilla.org/docs/WebAssembly/Using_the_JS_API>. Acesso em 20 Out. 2020.

[40] The Rust Programming Language. Disponível em: <https://www.rust-lang.org/>. Acesso em 12 de novembro de 2020.

[41] AssemblyScript: *A Language Made for WebAssembly*. Disponível em: <https://www.assemblyscript.org/>. Acesso em 12 de novembro de 2020.

[42] Microsoft Corporation. *Blazor: Build client web apps with C#*. Disponível em: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>. Acesso em 12 de novembro de 2020.

[43] Golang. *Go support for WebAssembly*. Disponível em: <https://github.com/golang/go/wiki/WebAssembly>. Acesso em 12 de novembro de 2020.

[44] EBERHARDT, Colin. *The future of WebAssembly - A look at upcoming features and proposals*. Scott Logic. Disponível em: <https://blog.scottlogic.com/2018/07/20/wasm-future.html>. Acesso em: 16 Nov. 2020.

[45] BASILI, V. R., CALDIERA, G., and ROMBACH, D. (1994). *The Goal Question Metrics Approach*, Volume 1, páginas 528–532. John Wiley & Sons.

[46] Google Developers. *Referência aos eventos de linha do tempo*. Chrome DevTools. Disponível em: <https://developers.google.com/chrome-devtools/performance-reference>. Acesso em: 16 Nov. 2020.

[47] WebDev. *First Contentful Paint*. Disponível em: <https://web.dev/first-contentful-paint/>. Acesso em: 16 Nov. 2020.

[48] KORETSKYI, Max. *Inside Fiber: in-depth overview of the new reconciliation algorithm in React*. Disponível em: <https://indepth.dev/inside-fiber-in-react>. Acesso em: 16 Nov. 2020.

[49] POMBO, Rodrigo. Build your own React. Pomb.us. Disponível em: <https://pomb.us/build-your-own-react/>. Acesso em: 16 Nov. 2020.

[50] Facebook Inc. *State and Lifecycle*. Disponível em: <https://reactjs.org/docs/state-and-lifecycle.html>. Acesso em: 16 Nov. 2020.

[51] Facebook Inc. *Events*. Disponível em <https://reactjs.org/docs/events.html>. Acesso em: 16 Nov. 2020.

[52] Facebook Inc. *Lists and Keys*. Disponível em <https://reactjs.org/docs/lists-and-keys.html>. Acesso em: 16 Nov. 2020.

[53] Facebook Inc. *React Hooks*. Disponível em <https://pt-br.reactjs.org/docs/hooks-intro.html>. Acesso em: 16 Nov. 2020.

[54] Wasm Bindgen: *Facilitating high-level interactions between Wasm modules and JavaScript*. Disponível em: <https://github.com/rustwasm/wasm-bindgen >. Acesso em: 16 Nov. 2020.

[55] Babel: *Use Next Generation JavaScript Today*. Disponível em: <https://babeljs.io/>. Acesso em 11 Nov. 2020.

[56] The Rust Programming Language: WebAssembly. Disponível em: <https://www.rust-lang.org/pt-BR/what/wasm>. Acesso em 12 Nov. 2020.

[57] Webpack. Disponível em <https://webpack.js.org/>. Acesso em 22 Out. 2020.

[58] Webpack: *Dynamic Imports*. Disponível em: <https://webpack.js.org/guides/code-splitting/#dynamic-imports>. Acesso em 22 Out. 2020.

[59] TodoMVC. Disponível em: <http://todomvc.com/>. Acesso em 23 Out. 2020.

[60] Chrome DevTools. Disponível em: <https://developers.google.com/web/tools/chrome-devtools>. Acesso em: 16 Nov. 2020.

[61] Puppeteer. Disponível em: <https://pptr.dev/>. Acesso em: 16 Nov. 2020.

[62] WebAssembly: *Reference Types Proposal*. Disponível em: <https://github.com/WebAssembly/reference-types/>. Acesso em 9 Nov. 2020.

[63] WebAssembly: *Interface Types Proposal.* Disponível em: <https://github.com/WebAssembly/interface-types/>. Acesso em 9 Nov. 2020.

[64] WasmBindgen Documentation. *Closure - into_js_value*. Disponível em: <https://docs.rs/wasm-bindgen/0.2.68/wasm_bindgen/closure/struct.Closure.html#method.into_js_value>. Acesso em 5 Out. 2020.

[65] WebAssembly: *Function References Proposal*. Disponível em: <https://github.com/WebAssembly/function-references>. Acesso em 12 Nov. 2020.

[66] TURNER, Aaron. *WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6*. Medium. Disponível em: <https://medium.com/@torch2424/webassembly-is-fast-a-real-world>. Acesso em: 8 Dec. 2020.

[67] Asm-dom. *A minimal WebAssembly virtual DOM to build C++ SPAs*. Disponível em <https://github.com/mbasso/asm-dom>. Acesso em: 8 Dec. 2020.

[68] Dodrio. *A fast, bump-allocated virtual DOM library for Rust and WebAssembly*. Disponível em <https://github.com/fitzgen/dodrio>. Acesso em: 8 Dec. 2020.