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

TÚLIO ARAÚJO CUNHA

AVALIANDO O IMPACTO DO CONTROLE DE CONCORRÊNCIA DO REACT 18: UM ESTUDO DE CASO

CAMPINA GRANDE - PB

2023

TÚLIO ARAÚJO CUNHA

AVALIANDO O IMPACTO DO CONTROLE DE CONCORRÊNCIA DO REACT 18: UM ESTUDO DE CASO

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. Tiago Lima Massoni.

CAMPINA GRANDE - PB

2023

TÚLIO ARAÚJO CUNHA

AVALIANDO O IMPACTO DO CONTROLE DE CONCORRÊNCIA DO REACT 18: UM ESTUDO DE CASO

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. Tiago Lima Massoni

Orientador – UASC/CEEI/UFCG

Professor Dr. Eanes Torres Pereira

Examinador – UASC/CEEI/UFCG

Professora Dra. Eliane Cristina de Araujo

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 14 de fevereiro de 2023.

CAMPINA GRANDE - PB

RESUMO (ABSTRACT)

With version 18 of the React library, features were introduced that use the concept of concurrency to create web systems that provide a better experience for the user. Among the new tools offered is the *useDefferedValue* hook, which allows defining a value that will be updated with a delay. In this work, this hook was used to propose improvements in a system with problematic points. To perform a comparative analysis between a system with and without the use of this hook, a simulation of interactions was created that allowed collecting performance timelines. When analyzing these timelines, it was observed that the use of *useDefferedValue* led to advantages such as reducing the time required to perform actions and obtaining faster feedback on actions taken. In addition, browser resources were saved by avoiding the calculation of intermediate steps not desired by the user, optimizing the system's performance.

Avaliando o impacto do controle de concorrência do React 18: Um Estudo de Caso

Túlio Araújo Cunha
tulio.cunha@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Tiago Massoni
massoni@dsc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

Com a versão 18 da biblioteca React, foram apresentadas funcionalidades que usam o conceito de concorrência para criar sistemas web que proporcionem uma melhor experiência para o usuário. Dentre as novas ferramentas ofertadas, encontra-se o *hook* `useDefferedValue`[1] que permite definir um valor que será atualizado com atraso. Nesse trabalho, utilizou-se esse *hook* para propor melhorias em um sistema com pontos problemáticos. Para realizar uma análise comparativa entre um sistema com e sem a utilização desse *hook*, criou-se uma simulação de interações que permitiram coletar linhas temporais de performance. Ao analisar essas linhas, observou-se que o uso do `useDefferedValue` implicou em vantagens como a redução do tempo necessário para realizar ações e a obtenção de um *feedback* mais rápido às ações realizadas. Além disso, economizaram-se recursos do navegador ao evitar o cálculo de etapas intermediárias não desejadas pelo usuário, otimizando o desempenho do sistema.

Palavras-chave: React, Concorrência, Experiência do usuário.

1 INTRODUÇÃO

Com o passar dos anos, o React se tornou a biblioteca mais popular para criação de sites e sistemas *web*, sendo utilizada pela maioria dos projetos e desenvolvedores [9][10]. Com esse crescimento, alguns sistemas desenvolvidos podem apresentar situações que prejudicam a experiência do usuário. Etapas de processamento que demandam muito do navegador, podem deixá-lo indisponível para responder às ações do usuário, assim, causando frustrações.

Para facilitar a criação de sistemas mais performáticos, a biblioteca React na versão 18 apresentou diversas ferramentas que tiram benefício do conceito de concorrência por meio de componentes e *hooks* nativos. Essas novas funcionalidades auxiliam a pessoa desenvolvedora a solucionar problemas causados por longas tarefas de renderização que tendem atrapalhar a experiência do usuário.

Com o intuito de avaliar os impactos da utilização dessas ferramentas, escolheu-se um sistema que apresentasse um cenário problemático e aplicou-se nele uma conversão utilizando o *hook* `useDefferedValue`. Nesse artigo, investigou-se como essa funcionalidade pode impactar a performance da aplicação e por consequência, a experiência de um usuário. Para isso, criou-se um teste automatizado que simulasse um cenário de uso e a partir dele, obtiveram-se linhas temporais de performance que permitiram comprovar quantitativamente as vantagens do sistema convertido sobre o sistema original. As vantagens notadas cobriram pontos como as reduções do processamento necessário no navegador e do tempo necessário para realizar interações, além de um *feedback* mais responsivo às ações realizadas no sistema.

[1]Disponível em https://reactjs.org/docs/hooks-reference.html#usedeferredvalue

2 CONTEXTUALIZAÇÃO

Para entendimento das possíveis dificuldades encontradas pelas bibliotecas como o React, deve-se inicialmente compreender como os navegadores realizam seus processos para a exibição dos elementos HTML definidos em cada site. O processo de transformação de dados e códigos em elementos da página caracteriza-se como renderização [8].

2.1 Funcionamento dos Navegadores

O navegador é responsável por monitorar as ações do usuário e os processamentos desencadeados por elas, gerenciando possíveis mudanças no HTML e CSS da página, além de executar *scripts* JavaScript. Para organização dessas mudanças, o navegador utiliza de uma *pipeline* [7] que define a ordem de processamentos especializados para atingir o resultado desejado pelo site, que pode ser vista na figura 1.

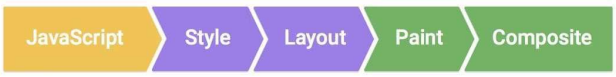


Figura 1: *Pipeline* de desenho do navegador

A *pipeline* acima pode ser resumida em três grupos principais:

- Grupo amarelo: responsável por executar o JavaScript que resultará em mudanças visuais na página;
- Grupo roxo: responsável por decidir as regras de estilização que serão aplicadas, além dos tamanhos e localizações dos elementos;
- Grupo verde: responsável por pintar os *pixels* na tela e ordenar corretamente as camadas desenhadas.

Como o navegador dispõe de apenas uma *thread* [2] para execução das cinco etapas da *pipeline* e cada uma delas pode apresentar gargalos, a experiência do usuário pode ser prejudicada. Isso ocorre porque o navegador estará ocupado processando a *pipeline*, logo, não estará disponível para processar as ações realizadas pelo usuário.

Cada unidade de processamento realizada pela *pipeline* define uma tarefa (*task*) do navegador que é necessária para cada *frame* exibido ao usuário. Contudo, nem toda tarefa precisa executar os cinco passos descritos na *pipeline*.

Os navegadores precisam dispor ao usuário uma nova imagem para cada *frame*. Como a maioria dos dispositivos atuais atualizam

suas telas 60 vezes a cada segundo[2], cada *frame* possui apenas 16*ms* (1 segundo / 60 = 16.66 milissegundos) para ser produzido. Contudo, o navegador também precisa realizar processamentos de manutenção, restando apenas 10*ms* para os processamentos do site [7].

Caso as tarefas demorem mais do que o tempo disponível, alguns *frames* não são processados e exibidos. Mesmo assim, se a tarefa desenceada por uma ação durar até 100*ms*, o usuário ainda sentirá que o resultado foi imediato. Caso contrário, o sentimento de ação e reação é quebrado, prejudicando a experiência do usuário [1].

2.2 Funcionalidades de Concorrência

Para evitar os problemas de travamento evidenciados anteriormente, a versão 18 da biblioteca React trouxe funcionalidades que aproveitam do conceito de concorrência. A implementação da API permite que as atualizações de renderização possam ser pausadas para continuarem posteriormente ou até serem canceladas completamente.

Com esse funcionamento, a biblioteca consegue processar em segundo plano mudanças como a criação de novos elementos e as lógicas dentro de componentes, sem bloquear a *thread* principal do navegador.

Dessa forma, as ações do usuário podem ser processadas pelo navegador de forma imediata e o React consegue atualizar a árvore de elementos da página apenas quando o processamento em segundo plano for finalizado.

3 METODOLOGIA

Para exemplificação das capacidades ofertadas pela versão 18 da biblioteca React, procuraram-se repositórios no GitHub[3] que utilizassem a biblioteca e investigaram-se cenários que pudessem se beneficiar das funcionalidades ofertadas.

Após a definição de um repositório que se adequasse aos critérios de busca, alterou-se o código do sistema para utilizar a nova API da biblioteca. Por fim, com as mudanças aplicadas, criou-se um teste automatizado para simular interações de um usuário no sistema. Com o teste automatizado criado, testou-se o sistema e gravaram-se linhas temporais de performance [5][6] em dois cenários: com e sem as funcionalidades de concorrência do React.

O uso das linhas temporais de performance permitem observar métricas como o tempo de processamento necessário para realização de tarefas e obter uma descrição detalhada dos momentos em que o navegador estava bloqueado sem responder às ações do usuário. Essas métricas servem para comparar os dois cenários e observar com detalhes os impactos das mudanças realizadas.

3.1 Repositório

Para o desenvolvimento do trabalho, buscou-se um repositório que se encaixa-se nos seguintes critérios:

- Utilizar a biblioteca React para a aplicação *frontend*, pelo menos a partir da versão 16;
- Ser um repositório popular, que apresentasse uma quantidade significativa de *forks* e/ou estrelas;
- Possuir um site demonstrativo com a aplicação em execução, facilitando a exploração do sistema e a identificação de pontos problemáticos;
- Apresentar algum cenário semelhante aos exemplos das documentações de desenvolvimento das funcionalidades da API de concorrência [4][3].

A partir dos critérios acima, escolheu-se o repositório *jira_clone*[4], do usuário *oldboyxx*, baseado na ferramenta de gerenciamento de tarefas Jira[5], onde uma equipe consegue monitorar o desenvolvimento de tarefas de um projeto.

O repositório disponibiliza a url https://jira.ivorreic.com/ para acessar o sistema, onde foi possível testá-lo e observar possíveis pontos de melhorias.

3.2 Sistema

A página inicial do sistema apresenta um quadro no estilo Kanban onde exibe-se tarefas que devem ser empenhadas por uma equipe em um projeto. Além disso, o quadro é dividido em 4 colunas que separam diferentes estágios para cada tarefa.

Além do quadro, a tela inicial também apresenta um menu lateral, textos informativos, botões de ações, um campo para o usuário pesquisar tarefas pelo nome e um conjunto de imagens circulares que lista todos os possíveis responsáveis para realizar tarefas.

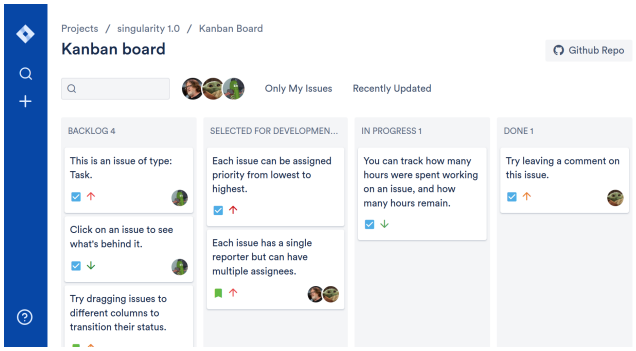


Figura 2: Tela inicial do jira_clone

As tarefas apresentadas no quadro podem ser filtradas de algumas formas. Além do campo de pesquisa anteriormente citado, o usuário pode interagir com um botão para exibir apenas as tarefas assinaladas a ele (Botão *Only My Issues*). Também é possível filtrar as tarefas recém-atualizadas (Botão *Recently Updated*). E por fim, é possível selecionar uma ou mais pessoas responsáveis pela tarefa por meio do conjunto de imagens circulares. O sistema também apresenta um botão de ação para limpar os filtros aplicados pelo usuário (Botão *Clear all*), como pode ser visto na figura 3.

Figura 3: Componentes para filtro de tarefas

[2]Disponível em https://insights.samsung.com/2022/03/07/how-does-refresh-rate-work-for-monitors/
[3]Plataforma de hospedagem de código-fonte. Disponível em https://github.com/
[4]Disponível em https://github.com/oldboyxx/jira_clone/
[5]Disponível em https://www.atlassian.com/br/software/jira

Nota-se na figura 3 que ao clicar em um responsável nas imagens circulares, a imagem correspondente é estilizada diferentemente com uma borda azul. Essa estilização representa a seleção realizada pelo usuário. Há também uma animação, provocada pela sobreposição do *mouse*, que suspende levemente a imagem circular da sua posição atual.

3.2.1 Dados. Quando um usuário acessa o sistema pela primeira vez, a aplicação cria um novo projeto. Todo projeto criado, apresenta inicialmente um conjunto de três responsáveis e os atribui oito tarefas predefinidas.

Toda vez que a página é carregada, o *backend* da aplicação fornece para o site todas as tarefas existentes do projeto e as pessoas que podem ser responsáveis por elas. A partir das tarefas recebidas, o site lista-as no quadro Kanban dependendo do seu estágio de completude. Além disso, as pessoas responsáveis também são listadas em imagens circulares que representam cada uma delas.

3.3 Preparação do Sistema

Para execução local do sistema e aplicação das funcionalidades de concorrência ofertadas pela versão 18 da biblioteca React, algumas mudanças foram realizadas. Essas mudanças também serviram para definir uma base a partir da qual seria implementada a API de concorrência. Dessa forma, a única diferença entre as versões que serão testadas será o uso das novas funcionalidades.

A primeira mudança realizada foi mover o banco de dados da aplicação para um contêiner Docker[6] para facilitar sua execução e acesso. Além disso, no *backend*, foi necessário atualizar as versões das bibliotecas de dependência `pg`[7] e `typeorm`[8], uma vez que as versões utilizadas anteriormente não conseguiam realizar a conexão com o banco de dados.

Para simular um projeto em andamento de uma equipe com várias tarefas em diferentes estágios de desenvolvimento, utilizou-se a biblioteca `@faker-js/faker`[9]. Ela serviu para gerar dados falsos, mas realistas, das novas tarefas. Em seguida, foram criadas 75 tarefas para cada uma das três pessoas responsáveis no sistema.

Para o *frontend* da aplicação, atualizou-se a versão da biblioteca React da versão 16.12.0 para 18.2.0. Para isso, ambas as bibliotecas de dependência `react` e `react-dom` tiveram que ser atualizadas para a versão citada anteriormente. Além disso, no arquivo raiz da aplicação React, necessitou-se também utilizar a nova função de criação do nó raiz. As diferenças entre as versões podem ser vistas nas figuras 4 e 5.

Após as mudanças comentadas acima, para execução do projeto é necessária a execução do `docker-compose` que sobe o contêiner Docker com o banco de dados e a execução dos processos Node.js[10] para o *backend* e o *frontend* da aplicação. Após a realização dessas etapas, o sistema pode ser acessado na máquina local por meio da url http://localhost:8080/. Então, quando o usuário visita a url, ele

[6]Unidade de software que empacota código e dependências para execução de aplicações. Disponível em https://www.docker.com/resources/what-container/
[7]Biblioteca para comunicação com um banco de dados Postgres. Disponível em https://www.npmjs.com/package/pg
[8]Biblioteca ORM (*Object–relational mapping*) que facilita o uso do banco de dados em um sistema orientado a objetos. Disponível em https://www.npmjs.com/package/typeorm
[9]Disponível em https://www.npmjs.com/package/@faker-js/faker
[10]Ambiente de execução JavaScript. Disponível em https://nodejs.org/

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3
4  import App from 'App';
5
6  const container = document.getElementById('root');
7
8  ReactDOM.render(<App/>, container);
```

Figura 4: Código na versão 16

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3
4  import App from 'App';
5
6  const container = document.getElementById('root');
7
8  const root = ReactDOM.createRoot(container);
9
10 root.render(<App />);
```

Figura 5: Código na versão 18

consegue visualizar um projeto com 225 tarefas (75 tarefas para cada um dos 3 responsáveis).

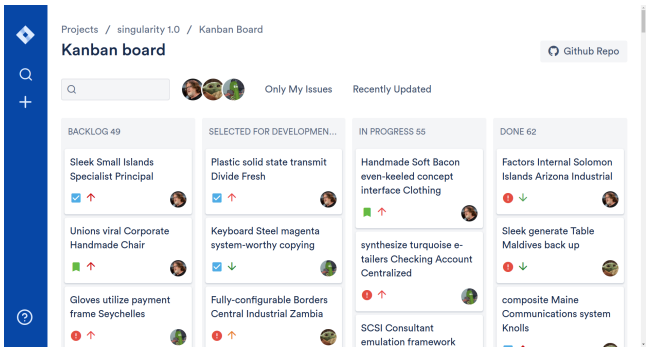


Figura 6: Sistema executando com as preparações realizadas

4 SISTEMA ORIGINAL

4.1 Problema

A partir desse momento, com o sistema apresentando centenas de tarefas, simulando um cenário provável de uso da aplicação, nota-se uma situação problemática em uma das interações do usuário com a página.

O problema se mostra evidente ao interagir com as imagens circulares dos responsáveis que filtram as tarefas do projeto. A cada clique realizado pelo usuário nas imagens, o sistema precisa filtrar e exibir apenas as tarefas correspondentes à atual seleção de pessoas responsáveis, além de aplicar mudanças na estilização dos elementos que contém as imagens.

Dependendo da quantidade de tarefas do sistema ou da configuração da máquina do usuário, essa ação pode levar mais do que os 10*ms* disponíveis para o processamento da *pipeline* de desenho do navegador. Além disso, caso a tarefa ultrapasse 100*ms*, o usuário não receberá um *feedback* instantâneo das suas ações [1], uma vez que o navegador estará ocupado.

Nesse momento, o sistema fica travado preparando o novo estado de tela para o usuário, sem processar as ações de sobreposição de *mouse* e cliques.

Esse comportamento gera inconsistências e possivelmente confusão, uma vez que o que é realizado pelo usuário não é refletido no sistema. Por exemplo, ao clicar em uma das imagens, o usuário não sabe se o clique foi realmente computado, uma vez que a estilização que sinaliza a seleção ainda não foi processada e exibida.

4.2 Causa do Problema

Para entendimento do problema, deve-se observar como funciona o componente responsável pela exibição das imagens circulares e listagem das tarefas.

A organização dos componentes foi feita da seguinte forma: o `ProjectBoard` gerencia o estado dos filtros utilizados para selecionar quais tarefas serão exibidas no quadro. Além disso, esse componente também é responsável pela exibição dos componentes `Filters` e `Lists`. A estrutura dos componentes pode ser observada na figura 7.

```
1  const ProjectBoard = ({ project, updateLocalProjectIssues }) => {
2    const [filters, mergeFilters] = useMergeState(defaultFilters);
3
4    return (
5      <Fragment>
6        <Filters
7          projectUsers={project.users}
8          filters={filters}
9          mergeFilters={mergeFilters}
10       />
11       <Lists
12         project={project}
13         filters={filters}
14         updateLocalProjectIssues={updateLocalProjectIssues}
15       />
16     </Fragment>
17   );
18 };
```

Figura 7: Código do componente ProjectBoard

Alguns trechos foram ocultados do código original, uma vez que não eram relevantes para o entendimento do cenário discutido

O componente `Filters` é responsável pela exibição e interações nas imagens circulares, aplicando o filtro conforme os cliques do usuário. Já o componente `Lists` é responsável pela listagem das tarefas que podem estar filtradas ou não.

Observando o código da figura 7, nota-se a dependência da variável `filters` para ambos os componentes `Filters` e `Lists`, uma vez que ambos precisam ter conhecimento do estado dessa variável para funcionarem corretamente.

Sendo assim, no momento em que o usuário ativa um filtro por meio de um clique em uma imagem circular, o componente `ProjectBoard` renderiza novamente. Por consequência, `Filters` e `Lists` também precisam ser renderizados novamente.

Além disso, nota-se na linha 12 da figura 7 que o componente `Lists` também recebe a variável `project`, que contém a lista de todas as tarefas do projeto. No momento que uma mudança ocorre no filtro, o componente `Lists` precisa percorrer todas as tarefas, a fim de filtrar apenas as correspondentes com o filtro ativo.

Em alguns casos, esse processamento pode levar mais tempo que o navegador tem disponível para processar a *pipeline* de desenho, causando o problema e até travando o sistema.

5 SISTEMA CONVERTIDO

5.1 Solução Proposta

Para solucionar o problema encontrado, utilizou-se como modelo uma solução proposta na documentação do time do React nomeada Padrões de Concorrência para Interfaces de Usuário (*Concurrent UI Patterns*) [3]. Na documentação, cita-se o cenário de separar estados de alta e baixa prioridade.

Essa solução se encaixa no problema descrito anteriormente, pois o componente `Filters`, responsável pela exibição e interação das imagens circulares, deve ter alta prioridade por ser um ponto de mudanças do estado do sistema. Como as mudanças são processadas a partir dele, o usuário deveria poder interagir com ele sempre que desejar. Dessa forma, o componente `Lists` deve aguardar as mudanças realizadas por `Filters` e o seu processamento deve ser secundário. Assim, configurando a baixa prioridade.

Para implementar a solução, deve-se utilizar o *hook*[11] nativo `useDeferredValue`. Esse *hook* permite definir um estado que terá seu processamento atrasado em detrimento de processamentos mais urgentes.

Para isso, o *hook* armazena o estado anterior de uma variável, passada como parâmetro, enquanto os processamentos das tarefas urgentes não são finalizados. Quando finalizados, o novo valor da variável é processado e o React atualiza a tela exibindo a alteração.

Sua utilização é bem simples, sendo necessário apenas passar como parâmetro a variável que deseja-se atrasar e ele retorna uma nova variável com o funcionamento descrito acima.

```
const deferredValue = useDeferredValue(value);
```

Figura 8: Código exemplificando uso do hook useDeffered-Value

Para o uso da solução no sistema *jira_clone*, definiu-se um novo valor suscetível a atrasos da variável `filters`, definida na linha 2 da figura 7. Assim, obtém-se do retorno do `useDefferedValue` um estado de menor prioridade nos processamentos que será utilizado como parâmetro para o componente `Lists`. As mudanças realizadas podem ser observadas na figura 9 nas linhas 4 e 15.

Após essas mudanças, o componente `Lists` só irá realizar a filtragem das tarefas do sistema após a finalização dos processamentos do componente `Filters`. Dessa forma, o usuário poderá visualizar as interações com o componente `Filters` sem aguardar o processamento, possivelmente longo, do componente `Lists`.

[11] Nomenclatura utilizada para descrever funções que manipulam estados de componentes funcionais dentro do ecossistema React

```jsx
const ProjectBoard = ({ project, updateLocalProjectIssues }) => {
  const [filters, mergeFilters] = useMergeState(defaultFilters);

  const deferredFilters = useDeferredValue(filters);

  return (
    <Fragment>
      <Filters
        projectUsers={project.users}
        filters={filters}
        mergeFilters={mergeFilters}
      />
      <Lists
        project={project}
        filters={deferredFilters}
        updateLocalProjectIssues={updateLocalProjectIssues}
      />
    </Fragment>
  );
};
```

Figura 9: Código mostrando as mudanças realizadas no componente ProjectBoard

Alguns trechos foram ocultados do código original, uma vez que não eram relevantes para o entendimento do cenário discutido

Outra mudança necessária para atingir o resultado desejado, foi a utilização da função `memo`[12], também nativa do React. Essa função permite que o React pule a etapa de desenho de um componente caso os parâmetros passados a ele não mudem.

No cenário analisado, o uso da função `memo` evita que o componente `ProjectBoard` desenhe desnecessariamente o componente `Lists` com o valor desatualizado da variável `filters`, presente na variável `deferredFilters`. Sem o uso dessa função, o React não conseguiria aproveitar o estado já desenhado do componente `Lists` e iria desperdiçar tempo e recursos com o valor antigo de `filters`.

Para tirar proveito da função, necessitou-se apenas envolver o retorno do componente `Lists` com a função `memo`. As mudanças podem ser comparadas nas figuras 10 e 11.

```jsx
export default ProjectBoardLists;
```

Figura 10: Código original da exportação do componente Lists

```jsx
export default React.memo(ProjectBoardLists);
```

Figura 11: Código da exportação do componente Lists com a função memo

6 AVALIAÇÃO

Para avaliação das possíveis melhorias realizadas com a solução implementada anteriormente, criou-se testes automatizados que simulassem a utilização de um usuário do sistema.

[12]Disponível em https://reactjs.org/docs/react-api.html#reactmemo

6.1 Ambiente

A biblioteca escolhida para implementação da simulação foi a Puppeteer[13], uma biblioteca Node.js que permite controlar um navegador Chrome[14] ou Chromium[15], onde pode-se acessar e interagir com o sistema. Além disso, a biblioteca permite a gravação das linhas temporais de performance que serão utilizadas para avaliação quantitativa dos resultados.

Para realização dos testes, utilizou-se uma máquina com as seguintes configurações:

- *Hardware*
  - Processador: Intel Core i7-8550U @ 1.80 GHz x 4
  - Memória RAM: 16 Gb
- *Software*
  - Sistema Operacional: Linux Mint 20.3 Cinnamon 5.2.7
  - Navegador: Google Chrome versão 108.0.5359.124 (Build oficial) (64-bit)

Uma vez que as configurações de *hardware* da máquina utilizada são robustas, optou-se por utilizar a funcionalidade de *throttling* presente nos navegadores Chrome, Chromium e na biblioteca Puppeteer. Essa funcionalidade permite simular um desempenho reduzido do navegador a partir de um fator numérico.

Escolheu-se utilizar um fator de 6 que reduz a velocidade do processador em 6 vezes. Essa redução permitiu simular dispositivos com menos desempenho como celulares e computadores mais antigos. Além disso, a redução da velocidade acentuou o problema encontrado e facilitou a avaliação das linhas temporais de performance.

6.2 Interações automatizadas

A simulação testada consiste nas seguintes interações realizadas pelo Puppeteer:

- Dois cliques na última imagem circular, simulando um clique não intencional que necessitou de outro clique para remover a seleção;
- Um clique na imagem do segundo responsável;
- Um clique na imagem do primeiro responsável.

De forma resumida, antes das interações, o sistema não possuía filtro ativo. Ao fim das interações, objetiva-se que o sistema apresente as tarefas filtradas de dois dos responsáveis.

Além dos passos de interações realizados, adicionou-se uma espera de 2 segundos para gravação completa da linha temporal de performance.

Vale ressaltar que cada passo realizado pelo Puppeteer aguarda o navegador ficar disponível para processar a interação. Sendo assim, nos momentos que o navegador está processando o novo estado da tela, o Puppeteer não realiza nenhuma interação.

A partir dos passos definidos acima, realizou-se a simulação gravando a linha temporal de performance para os dois cenários: com e sem as mudanças realizadas na seção 5.1.

[13]Disponível em https://pptr.dev/

[14]Navegador criado pelo Google. Disponível em https://www.google.com/intl/pt-BR/chrome/

[15]Navegador base a partir do qual o Chrome foi implementado. Disponível em https://www.chromium.org/Home/

7 RESULTADOS E DISCUSSÕES

A partir dos testes realizados na seção 6, foram obtidas as linhas temporais de performance em formato *json*. Para exibição das linhas temporais de forma visual e interativa, utilizou-se a ferramenta *DevTools* disponível nos navegadores Chrome e Chromium. Na aba `perfomance` é possível carregar o arquivo *json* e interagir com a linha temporal de performance. Os arquivos das simulações criados podem ser encontrados no repositório https://github.com/tulioac/tcc.

A partir da observação das linhas temporais de execução, avaliou-se três cenários:

- Impacto no processamento do navegador
- Impacto nas imagens circulares
- Impacto na listagem das tarefas

Para o primeiro cenário, utilizou-se o gráfico de performance e o sumário da execução para avaliar o comportamento do navegador na simulação realizada pelo Puppeteer. Para o segundo e terceiro cenários, utilizou-se também das seções `Frames`, `Interactions` e `Animation` da linha temporal de performance.

A seção `Frames` permite visualizar o estado da tela em cada momento durante a linha temporal. Já a seção `Interactions` lista as interações realizadas pelo usuário, ou no caso analisado, pelo Puppeteer. Por fim, a seção `Animation` lista as animações de estilo realizadas nos elementos da página.

7.1 Impacto no processamento do navegador

Em ambos os cenários, o sistema levou pouco mais de 1 segundo para ser inicializado. Como pode ser visto na seção *System* (representada pela cor cinza) nas figuras 12a e 13a para o código original, e nas figuras 12b e 13b para o código convertido.

Além disso, o navegador passou um tempo ocioso. Esse tempo foi ocasionado pelo aguardo de 2 segundos inseridos no fim das interações realizadas pelo Puppeteer. Os tempos ociosos estão representados pela seção *Idle* (representada pela cor branca) nas figuras 12a e 14a para o código original, e nas figuras 12b e 14b para o código convertido.

A partir das figuras, nota-se que o código original ficou 1.9 segundos ocioso e o convertido apenas 1.0 segundo. Essa diferença de tempo já era esperada, uma vez que no código original o fim das interações só é computado quando as renderizações são finalizadas. Sendo assim, o navegador não possui nenhuma tarefa a realizar no fim das interações.

Já no código convertido, as interações são finalizadas após o navegador processar o último clique nas imagens circulares. Após isso, o sistema já está disponível para uso e a interação é finalizada. Porém, nesse momento a renderização da listagem das tarefas filtradas ainda não finalizou, pois estava executando em segundo plano.

Para realizar a comparação, subtraiu-se dos dois cenários os tempos em que o sistema estava inicializando e ocioso. Sendo assim, serão comparados apenas os trechos que o navegador estava realizando algum processamento.

Para o cenário original, subtraindo os tempos de inicialização e ociosidade do total ($6605ms - 1197ms - 1935ms$), resulta em 3473 milissegundos de processamento.

Para o cenário convertido, realizou-se os mesmos cálculos ($3746ms - 1264ms - 1011ms$), resultando em 1471 milissegundos de processamento.

Subtraindo os dois resultados, encontra-se uma diferença de 2002 milissegundos ou 2 segundos. Essa diferença representa um tempo significativo que o navegador ficou disponível sem consumir recursos.

Percebe-se que a diferença de tempo encontrada é resultado de um menor tempo nas seções amarela (*Scripting*) e roxa (*Rendering*) no código convertido. Isso ocorreu, porque o navegador não precisou processar essas etapas nos momentos que as interações com as imagens circulares ainda estavam ocorrendo. Sendo assim, etapas intermediárias de processamento não foram computadas, economizando no tempo total da simulação.

Vale ressaltar que se o usuário realizasse mais ações, a diferença de tempo de processamento entre os cenários também iria crescer. Assim, a experiência do usuário seria ainda mais prejudicada.

7.2 Impacto nas imagens circulares

Para observar o impacto nas imagens circulares que exibem a ativação do filtro, focou-se no primeiro clique das interações definidas na seção 6.2. A partir do clique, observou-se quanto tempo foi necessário para que se visualizasse a estilização que representa a ativação do filtro.

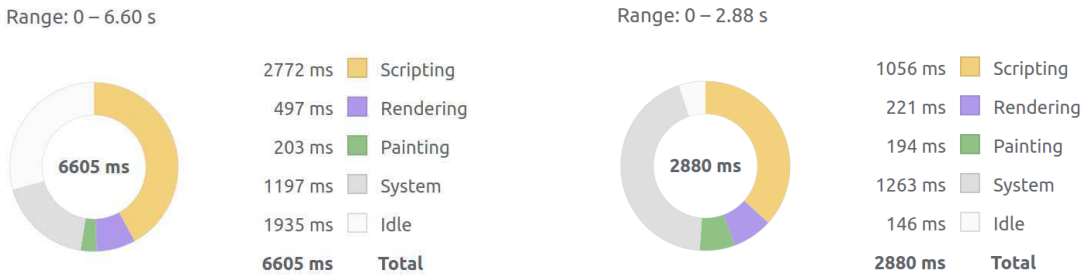
7.2.1 Código original. Para o cenário do código original, observa-se que no momento $1078ms$ a interação do primeiro clique é registrada, como pode ser visto na figura 15. Logo após, no momento $1082ms$ inicia-se a animação de sobreposição do *mouse* que se estende até o momento $1229ms$, como pode ser visto nas figuras 16 e 17. O sistema fica indisponível processando as mudanças ocasionadas pelo clique até o momento $1662ms$ e somente a partir desse instante exibe a estilização de borda azul na imagem circular, como pode ser visto na figura 18. Sendo assim, aguardou-se $433ms$ para visualizar a aplicação do filtro nas imagens circulares, prejudicando a percepção do *feedback* da interação [1].

Além disso, observa-se na figura 23a, um agrupamento de trechos de processamento de *Rendering* e de *Paiting* (representados respectivamente pelas cores roxo e verde), separados por um trecho de *Scripting* (representado pela cor amarelo). Esses trechos representam, respectivamente:

- 1º Conjunto verde e roxo: desenho (*Rendering* e *Painting*) da animação de sobreposição do *mouse* na imagem circular;
- Trecho amarelo intermediário: processamento das mudanças no sistema após o clique - mudança no estado do filtro e execução de códigos dependentes desse estado;
- 2º Conjunto verde e roxo: desenhos da borda que representa o filtro ativo na imagem circular e da nova listagem de tarefas com o filtro ativo.

Tal comportamento se repete durante as outras interações realizadas pela simulação, como pode ser visto na figura 24a. Porém, o tempo de processamento de cada etapa pode variar, dependendo da quantidade de elementos e mudanças que precisam ser processadas e/ou renderizadas.

7.2.2 Código convertido. Observa-se que o primeiro clique é registrado no momento $1104ms$, como pode ser visto na figura 19. No



(a) Código original **(b) Código convertido**

Figura 12: Sumários de execução das linhas temporais de performances

(a) Código original **(b) Código convertido**

Figura 13: Trechos iniciais das linhas temporais

(a) Código original **(b) Código convertido**

Figura 14: Trechos finais das linhas temporais

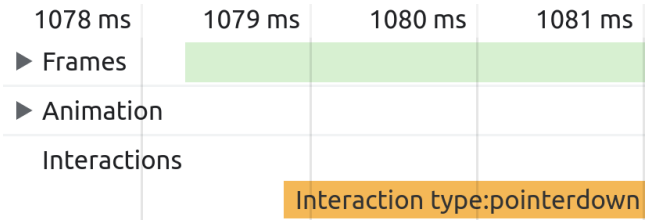


Figura 15: Código original: Registro do primeiro clique

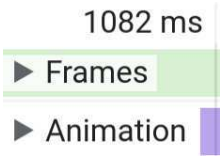


Figura 16: Código original: Início da animação

momento 1106*ms* inicia-se a animação de sobreposição do *mouse* que é finalizada no momento 1242*ms*, como pode ser visto nas figuras 20 e 21. Já no momento 1307*ms* o sistema exibe a estilização da borda azul, indicando a ativação do filtro (visto na figura 22). Assim, aguardou-se 65*ms* para aplicação do filtro, provendo ao usuário do sistema um *feedback* válido da sua ação [1].

Na figura 23b, nota-se o agrupamento de mesmo comportamento ao apontado no tópico 7.2.1: um conjunto verde e roxo, trecho amarelo intermediário e outro conjunto verde e roxo. Porém, nessa simulação, esse agrupamento apresenta o trecho amarelo (*Scripting*) bem reduzido, pois o navegador não fica bloqueado, logo, o Puppeteer consegue realizar outro clique em seguida que desencadeia as etapas de desenho (*Rendering* e *Painting*). Além disso, no segundo conjunto verde e roxo, não se realiza o desenho da listagem das tarefas.

7.2.3 Comparação. Comparando os dois cenários, nota-se que o tempo necessário entre o registro do clique até a visualização do filtro ativo nas imagens circulares, foi consideravelmente menor para o código convertido. Obteve-se uma diferença de 368*ms*, comparando 433*ms* do código original e 65*ms* do código convertido.

7.3 Impacto na listagem das tarefas

Para avaliar o impacto na listagem das tarefas, considerou-se o contexto geral da simulação: a partir do primeiro clique até a exibição da listagem das tarefas filtradas do último filtro realizado pelas interações. Ou seja, observou-se para ambos os cenários todo o trecho que o navegador estava realizando algum processamento.

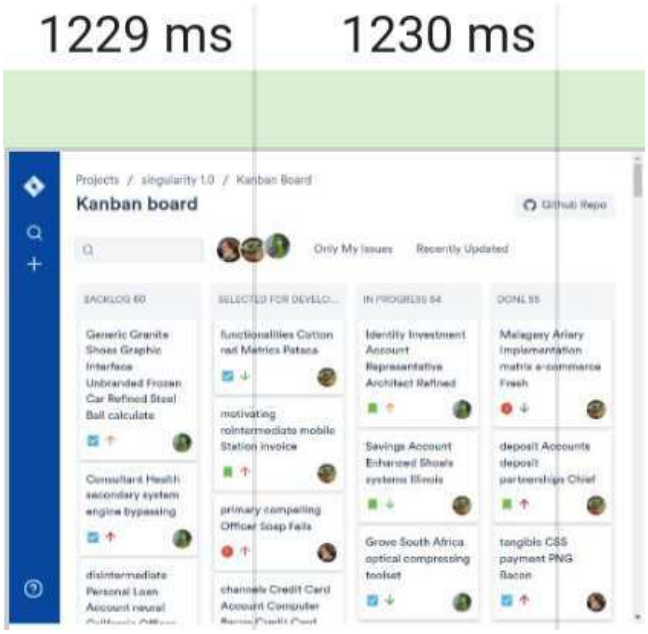


Figura 17: Código original: Frame com a animação finalizada

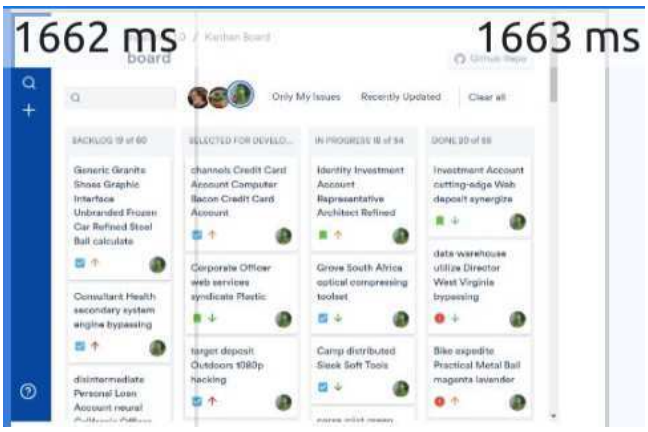


Figura 18: Código original: Fim da renderização do primeiro clique - imagens circulares com um filtro ativo

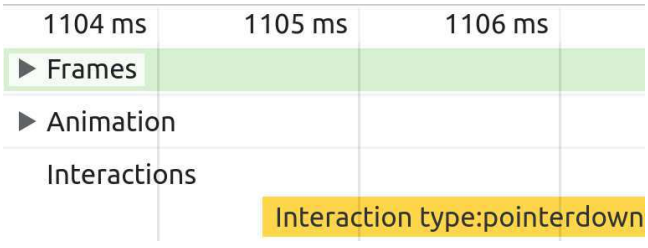


Figura 19: Código convertido: Registro do primeiro clique

7.3.1 Código original. Na figura 24a, observa-se que durante a execução da simulação todas as etapas de desenho (*Rendering* e

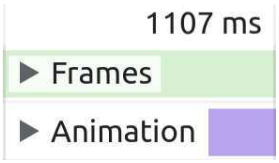


Figura 20: Código convertido: Início da animação

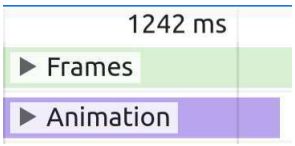


Figura 21: Código convertido: Fim da animação

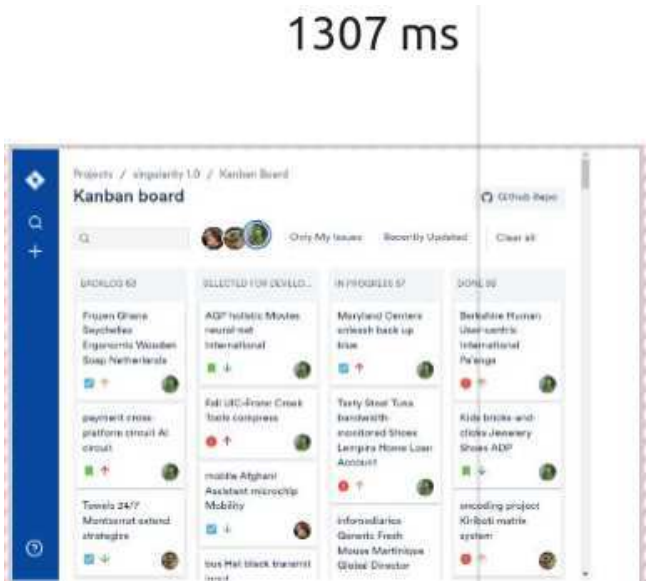


Figura 22: Código convertido: Fim da renderização do primeiro clique - imagens circulares com um filtro ativo

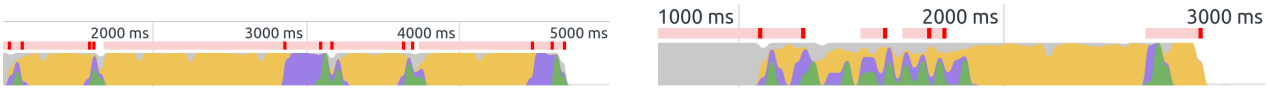
Painting) são espaçadas por grandes trechos de *Scripting*. Durante esses trechos, o sistema fica irresponsivo enquanto processa as mudanças desencadeadas pelas interações. Além disso, boa parte dos trechos de processamento são desperdiçados, preparando telas intermediárias que não foram utilizadas, pois não representavam o resultado final desejado.

Considerando do início do primeiro clique, realizado no momento 1078*ms*, até a exibição das tarefas filtradas, no momento 4677*ms* (visto na figura 25), levou-se 3599*ms* para que se atingisse a listagem das tarefas filtradas desejadas.

7.3.2 Código convertido. Na figura 24b, nota-se que entre o intervalo aproximado de 1100*ms* e 1900*ms*, o navegador realiza diversos processamentos de desenho (*Rendering* e *Painting*) espaçados por pequenas etapas de *Scripting*. Essas etapas de desenho são resultados

(a) Código original **(b) Código convertido**

Figura 23: Intervalos dos gráficos de processamento do primeiro clique até o filtro ativo nas imagens circulares



(a) Código original **(b) Código convertido**

Figura 24: Trechos das simulações nos gráficos de processamento

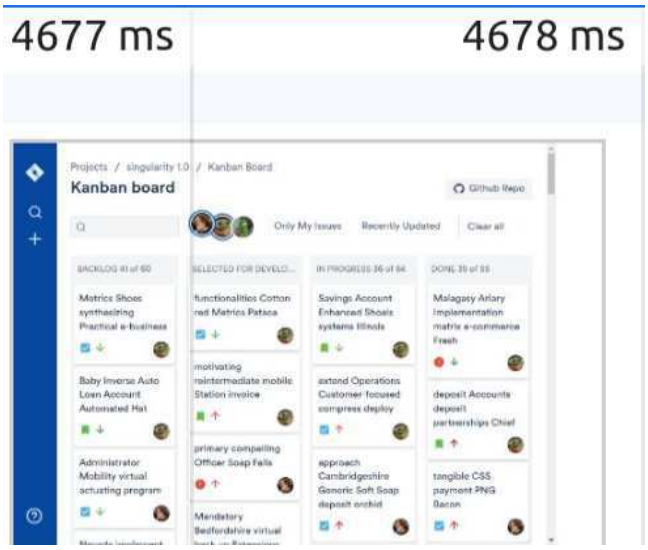


Figura 25: Código original: Filtros ativos nas imagens circulares e listagem de tarefas filtradas

Figura 26: Código convertido: Gráfico de processamento das etapas de desenho da listagem das tarefas filtradas

Nessa simulação, a primeira interação ocorreu no momento 1104*ms* e no momento 2622*ms* o sistema exibiu as tarefas filtradas a partir das seleções realizadas, como pode ser visto na figura 27. Sendo assim, levou-se 1518*ms* para que a simulação atingisse o resultado desejado.

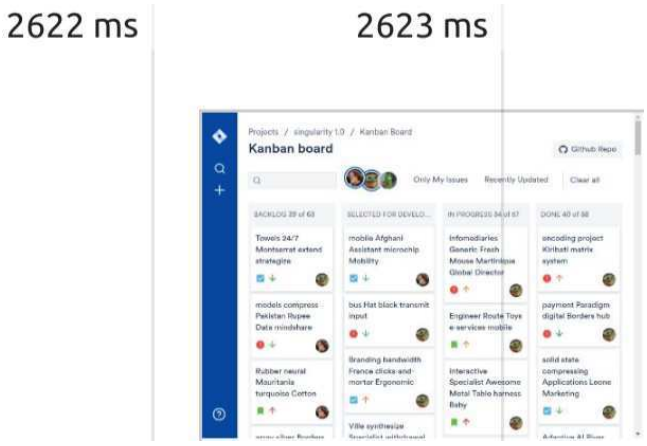


Figura 27: Código convertido: Filtros ativos nas imagens circulares e listagem de tarefas filtradas

das animações e estilizações desencadeadas pelas ações realizadas pelo Puppeteer ao interagir com as imagens circulares.

Além disso, a partir do momento aproximado de 1900*ms*, o sistema passa por um grande trecho de *Scripting*. Nesse momento, as mudanças dos estados no sistema, desencadeadas pelas interações, são calculadas. Para finalmente no momento aproximado de 2500*ms* o navegador voltar a realizar etapas de desenho. Após essa etapa, no momento aproximado de 2600*ms*, o navegador atualiza a tela com a listagem das tarefas filtradas, como pode ser visto nos trechos roxo e verde da figura 26.

7.3.3 Comparação. Observando a figura 24b, nota-se que todas as interações realizadas pelo Puppeteer puderam ser exibidas quanto antes na tela para o cenário convertido. Por outro lado, no cenário original, toda interação resultava em um grande intervalo de tempo em que o navegador permanecia ocupado, como pode ser visto na figura 24a. Dessa forma, no cenário convertido o sistema exibe

um *feedback* quase instantâneo das ações sobre ele, uma vez que o navegador economiza o processamento das etapas intermediárias.

De forma geral, as etapas de processamento economizadas diminuem o tempo total necessário para realizar a simulação, enquanto proporcionam uma melhor experiência para o usuário do sistema que visualiza o resultado das suas ações sem precisar aguardar longos carregamentos. Isso também é evidenciado pela diferença de 2081*ms* entre os cenários para que se visualizasse a tela com as tarefas filtradas.

8 CONCLUSÃO

Esse trabalho teve como objetivo avaliar os impactos do uso do *hook* `useDefferedValue` em um sistema *web* que utilizasse a biblioteca React. A partir da escolha do sistema *jira_clone*, criou-se um teste automatizado que realizasse interações a fim de simular um cenário de uso. Executou-se a simulação para dois cenários: o sistema original e o sistema convertido que utiliza o *hook* citado anteriormente. A partir das execuções, obtiveram-se linhas temporais de performance que permitiram a análise quantitativa dos cenários.

Na análise feita na seção 7, foram apresentadas as vantagens da solução implementa. Solução esta que consistiu em dividir componentes entre alta e baixa prioridade, fazendo uso do *hook* nativo `useDefferedValue` combinado com a função `memo`. A partir da análise, pode-se pontuar vantagens como: economia de processamento por parte do navegador, redução do tempo necessário para realizar interações com o sistema e obtenção de um *feedback* responsivo às ações realizadas. As vantagens encontradas foram ocasionadas por evitar as renderizações das telas intermediárias do sistema, que não eram necessárias para o objetivo final do usuário. Essas renderizações eram responsáveis por tarefas de longo processamento que ocupavam o navegador, logo, impossibilitando-o de responder rapidamente às ações realizadas sobre o sistema.

8.1 Trabalhos futuros

Além do *hook* utilizado nesse trabalho, a versão 18 da biblioteca React apresentou outros *hooks* e componentes que também podem agregar no desenvolvimento de sistemas *web*. Sendo assim, recomenda-se para trabalhos futuros a investigação do uso das novas ferramentas disponibilizadas em outros sistemas. Esses trabalhos poderiam validar as vantagens ofertadas pelas ferramentas na área de experiência do usuário em outros cenários de uso, como também, em outras áreas como a expansão de novas possibilidades do usuário interagir com o sistema.

9 AGRADECIMENTOS

A realização desse trabalho só foi possível com o apoio de diversas pessoas que participaram diretamente na minha vida. Dentre eles, estão os professores do curso de Ciência da Computação da UFCG que fortaleceram meu conhecimento, permitindo compreender diversos conceitos novos para mim ao longo desses anos. Dentre os professores, tenho um reconhecimento especial para meu orientador Tiago Massoni, por me guiar nessa jornada e me esclarecer diversas dúvidas que permitiram a conclusão desse trabalho.

Além dos professores, conheci diversos colegas desde o curso de Engenharia Elétrica até Ciência da Computação que me motivaram para continuarmos nossa caminhada em equipe, dividindo as frustrações e conquistas. Dentre os colegas, um obrigado especial para Sara Andrade, que durante vários anos me apoia imensamente e me ensina a ser uma pessoa melhor diariamente. Agradeço também a ela por presentear-nos com nossa filha, Mariana, que desde o início se mostrou uma pessoa forte em constante evolução que nos enche de orgulho. Toda essa jornada de aprendizado seria muito mais difícil se não fosse por Joseilda, que está presente desde o início da vida de Mariana, cuidando dela e nos ajudando muito a guiá-la no seu crescimento.

Por fim, agradeço aos meus pais e irmão, que durante minha vida sempre apoiaram minhas decisões e proveram uma vida de oportunidades com foco em educação de qualidade.

REFERÊNCIAS

[1] Chrome DevRel. 2020. Meça o desempenho com o modelo RAIL. Acessada em Dez, 2022 de https://web.dev/rail/
[2] MDN Web Docs. 2023. Main Thread. Acessada em Jan, 2023 de https://developer.mozilla.org/en-US/docs/Glossary/Main_thread
[3] React Docs. [Sem data]. Concurrent UI Patterns (Experimental). Acessada em Dez, 2022 de https://17.reactjs.org/docs/concurrent-mode-patterns.html
[4] React Docs. [Sem data]. Suspense for Data Fetching (Experimental). Acessada em Dez, 2022 de https://17.reactjs.org/docs/concurrent-mode-suspense.html
[5] Meggin Kearney e Flavio Copes. 2015. Timeline event reference. Acessada em Dez, 2022 de https://developer.chrome.com/docs/devtools/performance/timeline-reference/
[6] Kayce Basques e Sofia Emelianova. 2023. Performance features reference. Acessada em Jan, 2023 de https://developer.chrome.com/docs/devtools/performance/reference/
[7] Paul Lewis. 2018. Rendering Performance. Acessada em Dez, 2022 de https://web.dev/rendering-performance/
[8] Seobility. [Sem data]. Rendering. Acessada em Dez, 2022 de https://www.seobility.net/en/wiki/Rendering
[9] Stackoverflow. 2022. 2022 Developer Survey. Acessada em Dez, 2022 de https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe
[10] Stackoverflow. 2023. Stackoverflow Trends. Acessada em Jan, 2023 de https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs