Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Examining the rationale behind virtualised DOM applications in modern web frameworks

Bachelorarbeit im Studiengang Elektro- und Informationstechnik

vorgelegt von

Daniel Gran

Matrikelnummer 3559822

Erstgutachter: Prof. Dr. Matthias Hopf

Zweitgutachter: Prof. Dr. Joerg Arndt

© 2024

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

ohm Technische Hochschule Nürnberg

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Gran Vorname: Daniel Matrikel-Nr.: 3559822

Fakultät: efi Studiengang: Elektro- und Informationstechnik (B. Eng.)

Semester: 8

Titel der Abschlussarbeit:

Examining the rationale behind virtualised DOM applications in modern web frameworks

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit
- [x] genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
- [ ] genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format an die Betreuer übermittelt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: https://www.th-nuernberg.de/datenschutz/

Contents

1. Introduction . . . **1**
1.1. JavaScript . . . 1
1.2. The Evolution of Web-Based Programming . . . 3
1.2.1. Direct JavaScript Programming Methods . . . 3
1.2.2. jQuery . . . 3
1.2.3. Component-Based Web Programming . . . 4
1.3. Software Architecture Metrics . . . 4
1.4. Node.js, Node Package Manager (NPM) . . . 6
1.5. Memory Management (The Stack) . . . 7
1.6. Tree Data Structures . . . 9
1.7. Computing Theory . . . 10

2. State of the Art . . . **13**
2.1. Component-Based Programming . . . 13
2.1.1. React.js . . . 13
2.1.2. Vue.js . . . 14
2.1.3. Angular . . . 15
2.2. Advantages of JavaScript Libraries/Frameworks . . . 15
2.3. Libraries vs. Frameworks . . . 16
2.3.1. Libraries (e.g., React) . . . 16
2.3.2. Frameworks (e.g., Angular) . . . 16

3. React.js in-depth . . . **17**
3.1. Architectural structure of React 18.3.1 . . . 18
3.1.1. Project Anatomy . . . 18
3.1.2. Component Diagram . . . 19
3.2. React for the user (react) . . . 20
3.3. The virtual DOM (react-dom) . . . 21
3.4. Absolute maximum execution timings . . . 23
3.5. React Stack (v. 15.6.2, < v. 16.0.0) . . . 25
3.6. React Fiber (react-reconciler) . . . 28
3.6.1. Internal Types . . . 29
3.6.2. The React Fiber Reconciler . . . 32
3.6.3. React Update Handling . . . 36

4. Experiments . . . 41
4.1. Code Setup and Implementation of the Testing Lab Setup . . . 41
4.1.1. Testing Lab Setup . . . 41
4.1.2. Testing Project Anatomy . . . 41
4.2. Methodology . . . 44
4.3. Testing Mechanics . . . 45
4.3.1. Component Rendering/Reconciliation . . . 45
4.3.2. Development Metrics . . . 46
4.4. Findings . . . 47
4.4.1. Component Rendering/Reconciliation . . . 48
4.4.2. Development Metrics . . . 50

5. Results . . . 51
5.1. Evaluation of Results . . . 51
5.1.1. Rendering/Reconciliation Timings . . . 51
5.1.2. Development Metrics . . . 52

6. Summary . . . 55

7. Outlook . . . 57
7.1. Enhancement of the Analysis . . . 57
7.2. Improvement Investigation of React . . . 58
7.3. Serverside React . . . 58
7.4. Own opinion . . . 59

A. Detailed File Information . . . 61
A.1. Python scripts . . . 61
A.1.1. Static dependency analyser . . . 61
A.2. Configuration Files . . . 62
A.2.1. Standard package.json . . . 62
A.2.2. Reacts 18.3.1 package.json . . . 62
A.3. React Sourcecode Snippets . . . 63
A.3.1. Type definition of a Fiber . . . 63
A.3.2. v18.6.2 Lane Definition . . . 68

List of Figures . . . 71

List of Tables . . . 73

List of Listings . . . 75

Bibliography . . . 77

Kurzdarstellung

Moderne Browser-Anwendungen haben in den letzten Jahrzehnten aufgrund ihrer vielseitigen Nutzung erheblich an Bedeutung gewonnen. Dies spiegelt sich auch in den gestiegenen technischen Anforderungen seitens der Industrie und der Verbraucher wider. Die zu verarbeitende Datenmenge im Browser nimmt kontinuierlich zu, ebenso wie die Rechenkomplexität der anzuzeigenden Daten. Web-Frameworks müssen sich daher diesen erhöhten Anforderungen anpassen.

React, ein virtualisiertes DOM-Web-Framework, wurde ursprünglich für den News Feed von Facebook entwickelt. Durch die Übernahme von Instagram und die rasche Verbreitung von React nahm dessen Bedeutung weiter zu. Dank der Open-Source-Politik haben Entwickler weltweit Zugriff auf den Quellcode und können zur Weiterentwicklung beitragen.

Im Laufe der Jahre erfuhr der Kern-Rendering-Algorithmus von React ein bedeutendes Update, das moderne Leistungsstandards, eine verbesserte Entwicklererfahrung und neue Funktionen ermöglichte. Diese Arbeit untersucht die neuen Funktionsprinzipien des React Fiber Reconciliation Algorithmus und vergleicht die Leistungsfähigkeit der alten und neuen Version des Algorithmus.

Abstract

Modern Browser Applications have increasingly gained popularity for their versatile use over the last decades. So are the technical requirements from the industry and the consumer. The data size which has to be handled by the Browser is increasing. Also the computing complexity of data being displayed within an application is on it's rise. Thus, web frameworks have to adapt to higher requirements.

React is a virtualised DOM web framework which was initially developed for Facebook's News Feed. Due to Facebook's aquistition of Instagram and the fast adoption of React, it grew with a even higher pace. With it's open-source policy, developers from the whole world have access to the codebase and can contribute to it.

Over the years, the core rendering algorithm faced a significant update which enabled modern performance standards, a better developer experience and new features. This work researches the new working principles of the React Fiber Reconciliation Algorithm. Also, it compares the old and the new version of the algorithm performance-wise.

Chapter 1.

Introduction

Modern web applications have gained significant popularity. This rise in popularity is reflected in the technical requirements from both the industry and end users. The amount of data that browsers need to handle is continuously growing, as is the complexity of the data being displayed within applications. As a result, web frameworks must adapt to meet these higher demands. React, a web framework developed by Facebook, is widely used for web frontend development and faces these evolving challenges. A key technical principle for ensuring the necessary performance is managing rendering within an abstract UI state representation in memory.

This chapter aims to contextualise the evolution of modern web applications within the broad technological landscape and clarify the foundational concepts and terminology relevant to this work. By summarising key points, this chapter will lay the groundwork for understanding Reacts internal structures.

1.1. JavaScript

The JavaScript Language and its implementations are responsible for the basis of the concept of interactive browser content. JavaScript was first released for browsers in 1995. [Acke 17, p. 43] It was invented to provide a lightweight, interpreted language that could enhance the dynamic of web pages. Initially, JavaScript's goal was to allow web developers to create interactive content directly within the browser.

JavaScript's use in the browser quickly grew due to its ability to manipulate the DOM[1], validate forms, and handle user events without requiring a page reload. This client-side scripting capability transformed static HTML pages into interactive web applications, setting the stage for the rich internet applications we see today.

[1] DOM: Document Object Model

JavaScript in Hindsight

The first JavaScript version was initially released in December 4, 1995 by NetScape and Sun. [WIRF 20, p. 136] It was first used to serve as a programming language in a browser called *NetScape Navigator*. Because of the corporation between NetScape and Sun[2] it was later called *JavaScript*. Because shortly after Microsoft also developed a use-case similar language called *JScript* for Internet Explorer 3.0 in parallel, two different, use-case identical, languages were available. [Acke 17, p. 43] This ultimately triggered a race. “With Microsoft taking browser competition seriously, Netscape and Sun feared that Microsoft might try to dominate development of Web scripting standards and attempt to refocus it on a Visual Basic-based language. ”[WIRF 20, p. 31] With the goal to avoid this, Netscape and Sun researched a way on how to put an “umbrella [over] a JavaScript standard [so it] could be quickly drafted with Microsoft participation but not [with] Microsoft domination”[WIRF 20, p. 31]. Netscape is having personal connections to *ECMA International*, which ”is an industry association dedicated to the standardisation of information and communication systems“[Inte 24]. With it also approved by the International Standard Organization (ISO), they used their chance and applied for the standardisation process. In November 1996, the start-up meeting for the procedure was initiated with the project name *TC39*. In the course of this, Microsoft also joined ECMA as a member. [WIRF 20, p. 31]

With Netscape, Sun, Microsoft and other firms unifying on their opinions, on December 16, 1999 the first language standard *ECMAScript 3rd Edition* got released to the public. Shortly after, the current browsers at that time adopted *ES3* as their JavaScript language standard. [WIRF 20, p. 47]

With the first version in 2016 the ECMA TC36 group began releasing new standards on a yearly schedule. Each new edition brings enhancements, optimisations, and new features that contribute to the language’s robustness and versatility. [TC36 24] Table 1.1 outlines the various versions of ECMAScript, along with their respective release years and short forms.

Long Form	Short Form	Year
ECMAScript 3rd Edition	ES3	1999
ECMAScript 5th Edition	ES5	2009
ECMAScript 2015	ES6	2015
ECMAScript 2016	ES7	2016
ECMAScript 2017	ES8	2017
ECMAScript 2018	ES9	2018
...	...	...

Table 1.1.: Evolution of ECMAScript versions and their release years [TC36 24].

[2] Sun firstly introduced the programming language *Java* to the market

As the version of supplied JavaScript packages and the global amount of possible end users indirectly correlate, development of JavaScript can not just focus on the latest ECMA standard "ESNext". To enable the use of new language features in older browsers, JavaScript compilers have been developed. These compilers translate JavaScript code written in a newer ECMAScript version into a version that is compatible with the target browser. This is done by rewriting the code to use older language features by using a technique called *transpilation*. A well established transpiler is *Babel*, which is widely used in the JavaScript community. [Team 24a]

1.2. The Evolution of Web-Based Programming

Programming methods of browser-based applications have faced several ground breaking changes since the rise of the internet. In this section, we provide a brief introduction to the history of the technological development of web-based programming. We'll look at the early practices of realising end-user applications, the rise of libraries and frameworks, and how these changes have influenced modern approaches. In section 1.3 we elaborate on why these approaches greatly changed.

1.2.1. Direct JavaScript Programming Methods

Before the advent of frameworks, web developers relied on direct JavaScript programming methods. This period was characterised by manually writing JavaScript code to manipulate the DOM, handle events, and create interactive features. While this approach provided flexibility, it also led to code that was often difficult to maintain and debug.

1.2.2. jQuery

The development of jQuery marked a significant milestone in web programming. jQuery simplified many common JavaScript tasks, such as DOM manipulation and traversing, event handling, and AJAX[3] calls. It provided a more intuitive and concise syntax, making it easier for developers to write robust and cross-browser compatible code. The reasons behind its development were to address the complexities and hurdles of raw JavaScript, making web development more efficient. [York 15, p. 4]

[3] AJAX: Asynchronous JavaScript And XML; used i.E. for asynchronous Client-side Server calls

1.2.3. Component-Based Web Programming

The introduction of component-based web development revolutionised how developers build user interfaces. The main idea of that, which is obeyed by most modern web-frameworks, is to break the user interface down into parts, which later form the application. This technique comes with many advantages over the traditional approach. Components seek to logically separate code while encapsulating related concerns as styling and logic, therefore pushing the developer in sophisticated programming approaches. Historically implied, Angular.js is considered as the first web frontend framework following this approach, which is developed by a bigger firm. [Spri 23, p. 27]

1.3. Software Architecture Metrics

The main aspect of software architecture is to provide a design plan that describes the structure of a software system. [Hofm 00, p. 31] Abstractly speaking, the goal of a proper architecture is to combine interests by different stakeholders of a software system. This can be the end user, the web app developer, the React Library developer, Meta, Facebook or Instagram itself.

This study employs the terminology of *software architecture* to understand the offerings of the React Library to developers and, indirectly, to end users. The focus is on the quality attributes (QAs) of the React Library, which are the characteristics that define the overall metrics of the system. In the following section, quality attributes that are relevant to the React Library are discussed, along with how they influence the development of web applications.

System Quality Attributes

Quality attributes are the characteristics of a system that determine its overall metrics, such as performance, security, or maintainability. A software system can be measured in those. Laying focus on such attributes heavily influences the development of a software system. That ranges from thinking about the current state of the system to the future state and the possible changes that might be needed. So overall, the impact of setting focus on quality attributes ranges from the low level implementation to the high level component design.

The following will discuss the quality attributes that are relevant to the React Library and how they influence the development of web applications.

QA: Developer ease

The official React documentation states that React is a declarative, efficient, and flexible JavaScript library for building user interfaces [Team 24b]. That means, that low level implementation details are abstracted away from the developer. This is a key aspect of the React Library, as it allows developers to focus on building the user interface without worrying about the underlying implementation details. These mostly include rendering and UI updating mechanics, which are handled by the React Library.

QA: Performance

With the increasing capabilities of web applications, performance became a critical factor. More data needs to be processed and displayed efficiently in the frontend. This work focuses on parts of the core algorithms of React. This topic is key to later research in this thesis, referenced in chapter 4.

QA: Testability

As web applications have grown in complexity, automated testing has become more important. Frameworks now provide tools and interfaces for adding tests to ensure the reliability and correctness of components.

This is more of a general aspect of modern web development, but it's worth mentioning, because the modular nature of React Components makes it easier to test them in isolation.

This work later uses Performance and User Experience attributes to provide reasoning for the research questions in chapter 4 Experiments. With the development of React Fiber, these QAs were key driving factors for the React Team to improve the React Library.

1.4. Node.js, Node Package Manager (NPM)

Several tools exist to provide developers with a more integrated development experience. Integration on a file system level is crucial, because it defines rules on how to structure a project and how to manage dependencies.

Given that all source code examined in this study is written in JavaScript, an understanding of the tools utilised for managing JavaScript projects is essential. Consequently, the focus in this section is to understand the role of Node.js and the Node Package Manager (NPM) in the development of JavaScript-based software applications.

The concept of both *node* and *npm* is provide a better overall development experience. Node.js' project and module management solves another *architectural concern* which comes up in most development scenarios. That is the ability to reuse already written code and to rapidly adapt to changing requirements. It does that by offering an extensive toolset, which handles project metadata and dependency management.

NPM acts as the package manager required for developing JavaScript based software applications. Inter alia, it adds the capability to connect to remote repositories, from where built javascript projects can extend the local project.

Node, in contrast, empowers developers to execute JavaScript beyond the confines of the browser. Its API enables seamless interaction with the local system, offering methods to manipulate the filesystem, manage network communications, and execute low-level binaries directly from the local disk. Within the scope of our project, Node.js serves as the backbone for running code independently of the browser environment, eliminating the need for constant page refreshes. Furthermore, libraries like webpack enhance the development workflow through features such as Hot Module Replacement, which dramatically expedites the process by instantly reflecting code modifications in the browser.

Project Structure

With the introduction of node into a local development project, there comes a fixed file and folder structure. The following file diagram shows a minimal file structure, which is manifested in any JavaScript Project, which uses node as the project managing system:

```
./minimal-project
├── /node_modules
│   └── ...
├── /src
│   └── ...
├── package.json
└── package-lock.json
```

Figure 1.1.: File structure of a typical Node project called 'minimal-project'.

- **/node_modules**: This folder is auto-generated and contains module code that extends the capabilities of the local program. When installing modules, Node searches for the required dependency imports within this folder.
- **/src**: The standard source directory designated for production code. This directory is not mandatory; source code does not have to be placed here.
- **package.json**: The primary information file for any Node.js project. This file contains standard metadata. Detailed information about the most critical configuration content can be found in A.2.1 Standard package.json.
- **package-lock.json**: This file contains information about the exact versions of packages that were installed. Since package versions can conflict with each other, more detailed settings for the packages can be specified here.

1.5. Memory Management (The Stack)

The Computer's Stack is a critical part of the memory. Any program that runs on a computer uses the stack to store data and manage function calls. To later understand the key concepts introduced in React Fiber, we need to understand the stack and how it is used in computing.

The Processor's Stack is a region of memory that operates in a last-in, first-out (LIFO) manner. The Stack pointer is a register that points to the top of the stack. When a function is called, a new frame is pushed onto the stack.

This low level concept is extended in JavaScript. "Frame objects, also simply called frames, contain information about a single function call, including which line of code called the function, so the execution can move back there when the function returns.“ [Swei 22, 'What is the Call Stack?']

Sequent Functions

The following code snippet 1.1 shows a simple example of sequent function calls. The functions *a()*, *b()*, and *c()* are called in a sequent manner.

```
1 function a() { ... }
2 function b() { ... }
3 function c() { ... }
4 a(); b(); c();
```

Listing 1.1: Example of sequentially executed functions.

The following figure 1.2 shows the stack after each function call:

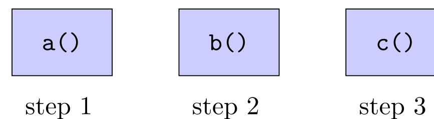

Figure 1.2.: Strongly simplified stack representation after each function call.

Recursive Functions

Running functions recursively is a common pattern in programming. Especially in dealing with tree-like data structures, recursive functions are often used to traverse the tree.

Every time a function calls itself, a new frame is pushed onto the stack. This is repeated until the abort condition is met.

The following listing 1.2 shows a simple example of a recursive function.

```
1 function d() {
2   if(meetsAbortCondition(payload)) {
3     return;
4   }
5   d(payload);
6 }
```

Listing 1.2: Recursive function example in JavaScript.

Which results in the following stack:

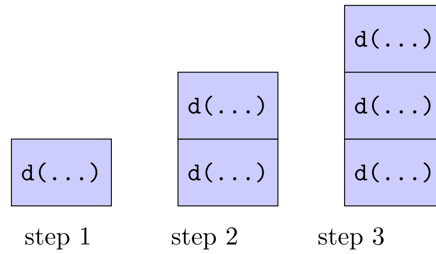

Figure 1.3.: Strongly simplified growing stack after each function call.

In a logical manner, this implies that each step of the algorithm is fixed and can not be manipulated. In the context of React Fiber, this behaviour is not desired and leads to several problems when rendering a user interface. This is later fully elaborated in 3.6 React Fiber (react-reconciler).

1.6. Tree Data Structures

When following the component based approached, tree structures come into play. By the nature of HTML, every HTML node can possibly have parents, siblings or children. A parent P of a node is considered the node, which is more close to the root element of the DOM. Every node can have only one parent. The Root Element of a webpage does not have a Parent. Child Elements describe the set of elements which share the same parent. A Child can have 0 to n possible siblings, which is the set of elements which also share the same parent.

Parent Element:
The parent of an HTML element E is the element that directly contains E within its tags in the HTML markup. Mathematically,

$$P = \text{parent}(E)$$

Child Elements:
The children of an element P (denoted as children(P)) are the elements directly nested within P. Mathematically,

$$\text{children}(P) = \{C_1, C_2, \ldots, C_n\}$$

where $C_1, C_2, \ldots, C_n$ are the child elements of P.

Sibling Elements:
Siblings are elements that have the same parent. If E and F are sibling elements, they share the same parent P. Mathematically,

$$\text{parent}(E) = \text{parent}(F) = P$$

Singly Linked List

Because of the nature of a Tree Data Structure, a Linked List is a efficient way to store the data in a computational context. A Singly Linked List is a data structure that consists of a sequence of elements, where a element field points to the next element in the sequence. [Good 14, p. 122ff.]

Figure 1.4.: Singly Linked List Structure. A singly linked list consists of a sequence of elements, where each element points to the next element in the sequence.

To apply the Linked list to a tree structure, the general approach is to make each node store a list of references to it's children. [Good 14, p. 333ff.] In the case of React Fiber, the tree structure is not followed by the general approach. Instead, the tree structure is realised by a dual singly linked list (not a doubly linked list[4]). That means, that each node stores a reference to its first child and its next sibling. This is later elaborated in 3.6 React Fiber (react-reconciler).

1.7. Computing Theory

The algorithms which are later discussed in this thesis will get analysed by different metrics in 4 Experiments. For that, it's important to understand the concept of time complexity in computing theory.

In algorithm analysis, our main concern is understanding how the running time scales with respect to the input size n. This involves taking a holistic view to grasp the overall growth rate of the execution time rather than precise timings. These algorithms have the main goal to differentiate UI state, so it can be updated effectively. This concept introduces some nomenclature, including *Time Complexity* and *Runtime Complexity* as a metric to assess the performance of an algorithm.

[4]Doubly Linked List: A Doubly linked list stores in each node both a reference to its predecessor and its successor.

In computational theory, an algorithm processes a number of inputs. The amount of steps f of an algorithm to complete, where

$$f : \mathcal{N} \rightarrow \mathcal{N}$$

depends on the finite input size x, where $x \in \mathcal{N}$ and $\mathcal{N} = \mathbb{N}$. The maximum number n of steps required to solve the problem is represented using "big-O" notation, where $O(n) = f(n)$ for $n \in \mathcal{N}$.

Here, $f(n)$ denotes the function describing the maximum steps involved of the algorithm to complete, and $O(n)$ indicates that the running time grows linearly with the size of the input n. This notation provides a precise and mathematical way to compare the efficiency of different algorithms based on their worst-case performance. [Sips 06, p. 247ff.]

In the following diagram, you can see the comparison of common resulting time complexities in algorithms, illustrating the problem with the growth rate of each complexity as the input size n increases.

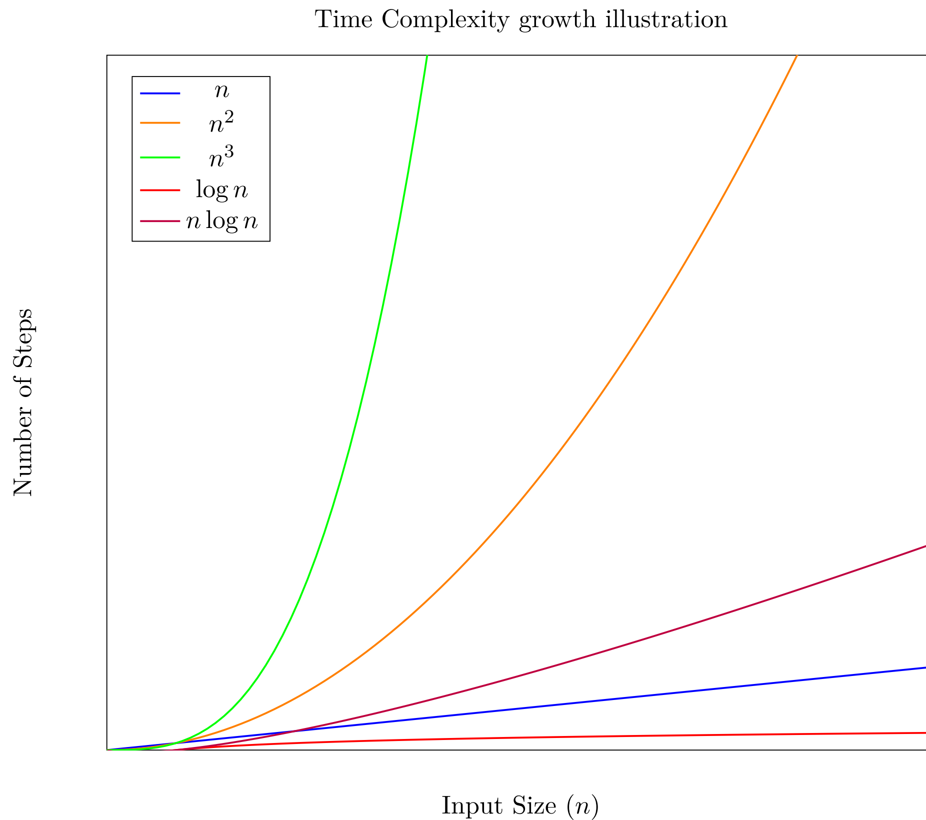

Figure 1.5.: Comparison of Various Time Complexities in Algorithms. The plot illustrates the growth rates of linear (n), quadratic (n^2), cubic (n^3), logarithmic ($\log n$), and linearithmic ($n \log n$) time complexities with respect to input size n. As n increases, each complexity demonstrates its characteristic rate of growth in the number of computational steps.

For the example of Binary Trees, where each parent P has two children, the following time complexities can be derived:

Access Method	**Running Time**
size, isEmpty	O(1)
numChildren	O(1)
height	O(n)

Table 1.2.: Time Complexity of Binary Trees. The table shows the time complexity of various methods for binary trees, including size, root, and height [Good 14, p. 333ff.].

React Fiber is able to provide a time complexity of $O(n)$ for the reconciliation process, which is a part of the process of updating the UI state. This is achieved by using several assumptions and optimisations, which are discussed in 3.6 React Fiber (react-reconciler).

Chapter 2.

State of the Art

This chapter provides an overview of the current state of web development technologies, focusing on JavaScript libraries and frameworks. The goal is to offer a comprehensive overview of why web frameworks exist in their current forms and why multiple frameworks have emerged instead of a single dominant one.

The start of the chapter will introduce the concept of component-based programming and its significance in modern web development. Additionally, it will explain what web frameworks aim to abstract away and why programming front-end sites solely with plain JavaScript, CSS, and HTML is less common today.

Furthermore, the chapter will delineate the key differences between a library, such as React, and a framework, focusing on the concept of *bounded programming* that frameworks enforce.

2.1. Component-Based Programming

Component-based programming has become the industry standard of modern web development. This paradigm shifts the focus from manipulating the Document Object Model (DOM) directly to building reusable components that encapsulate both logic and presentation.

2.1.1. React.js

React.js, developed by Facebook, is one of the most popular libraries for building user interfaces. It emphasises the creation of reusable UI components, which can manage their own state. React introduced the concept of a virtual DOM, which optimizes updates by calculating the minimal set of changes needed to keep the actual DOM in sync with the virtual representation. This approach significantly improves performance, especially in complex applications with frequent updates.

React's component-based architecture allows developers to break down complex UIs into smaller, manageable pieces. Each component can be developed, tested, and debugged independently, promoting better code organisation and maintainability. React also supports a unidirectional data flow, making the state management more predictable and easier to debug.

Key Authors

Because this study cites several sources, which come from non-official React documentation but are still considered reliable, it is essential to acknowledge the primary authors of the React project. Over the years, several pivotal contributors have been recognised in the core files of the React project. These individuals, primarily full-time employees at Facebook and integral members of the React Team, include Jordan Walke, Andrew Clark, Dan Abramov, and Sebastian Markbage.

React embodies the open-source spirit, fostering a collaborative environment where developers from around the globe can contribute to its evolution. This collective effort has resulted in a robust and versatile library that benefits from diverse insights and expertise. While Walke, Clark, Abramov, and Markbage have made the most significant contributions, as measured by the number of commits to the React repository, they are part of a larger community of developers who continually enhance and refine React. This extensive involvement from a broad array of contributors not only enriches the library but also ensures its continued relevance and adaptability in the ever-changing landscape of web development. Their extensive involvement and expertise lend substantial credibility to subsequent research in this domain [Meta 24].

2.1.2. Vue.js

Vue.js is a popular framework for building user interfaces, created by Evan You. Vue is designed to be incrementally adoptable, which means developers can use as much or as little of the framework as needed. This flexibility makes Vue an attractive option for both small and large projects. Vue's core library focuses on the view layer, and it can be easily integrated with other libraries or existing projects. It also provides a powerful ecosystem with officially maintained libraries for state management (Vuex) and routing (Vue Router). Vue's reactivity system and component-based structure offer a straightforward way to manage application state and build reusable components [vuej 24].

The key difference with React is, that it offers a more restrictive way of programming. With Vue.js, a framework, the project file anatomy is already set. As a front end developer, you

have to adapt to it according its documentation. Component files, testing files, configuration files already have their predefined filepath are not subject to the developer.

2.1.3. Angular

Angular, developed and maintained by Google, is a full-fledged framework for building web applications. Unlike React, which is primarily a library for building UIs, Angular provides a complete solution that includes tools for routing, state management, and form handling. Angular uses TypeScript, a statically typed superset of JavaScript, which adds type safety and enhances the development experience. The framework's architecture is based on components and services, promoting the separation of concerns and reusability. Angular also employs a hierarchical dependency injection system, which makes it easy to manage dependencies and create testable code. Angular's comprehensive nature and strong typing make it suitable for large-scale enterprise applications where maintainability and scalability are crucial. Its opinionated structure enforces best practices and provides a robust foundation for building complex applications [Angu 24].

2.2. Advantages of JavaScript Libraries/Frameworks

JavaScript libraries and frameworks offer several key advantages that have contributed to their widespread adoption:

- **Abstraction of Complexity**: Frameworks abstract away the complexities of DOM manipulation, event handling, and state management, allowing developers to focus on building features rather than dealing with low-level implementation details.
- **Reusability and Modularity**: By promoting component-based architecture, frameworks encourage the development of reusable and modular code. This modularity leads to better maintainability and scalability of applications.
- **Enhanced Performance**: Features like virtual DOM (React) and efficient reactivity systems (Vue) optimise the rendering process, resulting in faster and more responsive applications.
- **Community and Ecosystem**: Popular frameworks have large communities and ecosystems that provide a wealth of resources, libraries, and tools, accelerating development and problem-solving.
- **Consistency and Best Practices**: Frameworks often enforce best practices and consistent patterns, reducing the risk of common errors and improving code quality.

2.3. Libraries vs. Frameworks

In the world of JavaScript and node packages, the terms Library and Framework often get crumbled together. This section provides a clear distinction between those two.

2.3.1. Libraries (e.g., React)

A library provides specific functionality that can be called upon when needed. It offers more flexibility, as the developer is in charge of the flow of the application. React is an example of a library, as it focuses solely on building user interfaces and leaves other aspects, such as routing and state management, to external libraries.

2.3.2. Frameworks (e.g., Angular)

A framework provides a complete solution, enforcing a specific architecture and workflow. It dictates the structure of the application and integrates various aspects like routing, state management, and form handling. Angular is an example of a framework, as it offers a comprehensive suite of tools and enforces a specific way of building applications, known as *bounded programming*.

In conclusion, both libraries and frameworks play crucial roles in modern web development. Libraries offer flexibility and simplicity, while frameworks provide a structured and comprehensive solution for building complex applications. The choice between using a library or a framework depends on the chosen system attributes for the project.

Chapter 3.

React.js in-depth

React.js was initially developed in 2011 under the name FaxJS by Jordan Walke, who has been with Facebook since September 2010 [Spri 23, p. 27] [Walk 24]. The framework was first implemented in Facebook's Newsfeed application. Following Facebook's acquisition of Instagram in 2012, Instagram adopted the framework for their Newsfeed as well [Spri 23, p. 27]. As the framework's use cases grew rapidly (even to native mobile appliances), it became necessary to abstract it further, leading to the expansion of the codebase.

React was officially open-sourced in May 2013. Meta (formerly Facebook) announced this release at JSConf, a popular conference for JavaScript developers [JSCo 13]. Since then, the project has benefited not only from its full-time contributors but also from a large and active community that contributes to its development. Currently, the React project is publicly accessible on GitHub (`https://www.github.com/facebook/react`).

The core idea of React is providing an extensive performance by the *virtualisation of the Browser DOM*. That means, that the whole UI state lies both in the browser, and in a different form as a memory object. Because of the abstraction of this UI State, in 2015, react-native got released to the public. With React Native, not only Browser Applications could be supported. Native app programming opened the door for Cross-Plattform apps in the mobile sector. [Spri 23, p. 27]

With the technological evolution of React, a significant breaking change occurred within the Reconciliation module, which is responsible for updating and diffing changes between two virtual DOM (vDOM) versions. The major version 16.0.0, released on September 26, 2017, introduced a "[n]ew core architecture"[Clar 17], marking the beginning of the Fiber Reconciliation Logic, which remains the state-of-the-art today.

Over the years, the module has seen various enhancements, but its fundamental concept, detailed in 3.6, has stayed consistent. This release aimed to achieve better overall performance, improved error management, and the introduction of fragments[5].

[5] Fragment: A React-internal element that is not considered in the later tree depth, being left out in the browser DOM. (`<>...</>`)

The core architecture change revolutionised how React handled the rendering process. Prior to this, React relied on synchronous rendering algorithms. The implementation of Fiber made asynchronous rendering possible, significantly improving the efficiency and flexibility of the rendering process [Clar 17].

The current release by the time of writing this thesis is version *18.3.1* with the git branch name "18-3-1". The focus of this work is on the production code of the react project. This version already includes future features. These are existent in the codebase, but not yet released to the public, thus out of the scope of this work.

3.1. Architectural structure of React 18.3.1

Running the following bash command in the */packages* directory of the React project sums up the *total lines of code of .js files*.

```
$> find . -name *.js -type f -exec wc -l  + | tail -n 1
```

The output of this command is `1065996 total`. With over one million lines of code, it is crucial to define focus points within the codebase. An object-oriented programming approach is followed by React, allowing for a clear separation of concerns. This is particularly useful because various sub-projects, each with its own responsibility related to the UI Rendering Engine, are located in the `/packages` directory.

As the codebase grows, the importance of effective code structuring is highlighted by employing the Lines of Code (LOC) metric. The codebase is primarily written using an object-oriented approach. This metric as a root cause to ensure that large amounts of code remain manageable and changeable. This organisation can be further validated through subsequent evaluations.

In the following sections, the project structure of React will be delved into, providing an overview of the current codebase. Later, the focus will be on the code that handles the UI rendering process, which will be explained in detail.

3.1.1. Project Anatomy

The project anatomy describes the way how the source code files are logically separated. As mentioned before, the extensive functionality of the react projects needs logical separation.

Figure 3.1 shows the simplified folder and file structure of the react project, according to git release tag `v18.3.1`.

```
/react-sourcecode/v18.3.1
├── /packages
│   ├── /react
│   ├── /react-dom
│   ├── /react-reconciler
│   └── ...
├── /scripts
├── package.json
├── README.md
├── LICENSE
└── ...
```

Figure 3.1.: Overview of the react folder structure.

- ***/packages***: Holds the different Components (Modules) of the React project. For simplicity, we focus on the core components which are responsible for rendering HTML-based web applications. These are described in 3.1.2 Component Diagram.
- ***/scripts***: Automated workflows which are important for local build processes, Unit Testing Wrappers and DevOps Tasks including CI/CI (pipeline) scripts.
- ***package.json***: The main npm package file, listed at A.2.2. Its main purpose is not to include any production code. This project is a wrapper around the actual production code, which can be found in the */packages* directory. This is marked by the *workspaces* key in the package.json file. [Docs 24]

3.1.2. Component Diagram

Given the modular structure within the `/packages` folder, it is feasible to create a dependency diagram to further evaluate the code flow of the project. The react repository houses various packages that interact in multiple ways. To provide an overview of the package structure, we present a component diagram highlighting the integration of different submodules. This diagram exclusively analyses static, so-called "code-time" dependencies. These dependencies were extracted using a Python function, which can be found in Appendix A.1.

Code-time dependencies refer to the dependencies present in the code during its development phase, as opposed to runtime dependencies, which are only resolved during execution. These are crucial for understanding the architectural structure of a software system, as they reflect the direct relationships between different components within the codebase.

The Python function `analyze_imports(package_path: str)` takes one argument to the specified `/packages` path. It iterates over each component (package) in the specified directory. For each package, it constructs the full path and scans the directory for files. It skips entries that are testing files, do not have a ".js" extension, or are located within the

`/node_modules` directory. For each JavaScript file within a component, it reads the file content line by line. It checks each line for the presence of `require` or `import` statements. Further, it checks if the line contains "react-" to filter non-production packages. If both conditions are met, the line is added to the list of dependencies for the corresponding package in the dictionary. The function returns a dictionary with the package name as the key and a list of dependencies as the value.

After additional manual code analysis, it could be determined that three components are relevant for this work. This was done by back-tracing the high level React functions, which are used for building web applications. Relevant dependencies were extracted and compiled into the following diagram 3.2.

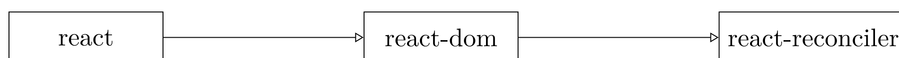

Figure 3.2.: Simplified component of React 18.3.1, applicable for the web application.

The following section aims to provide a deeper insight into those three components of React. These are separated in three sections, each focusing on one of the components. This offers to understand the core mechanics of the Rendering engine, the Reconciliation process and the intended usage of React.

Following structure is embodied in the following sections:

- `react`: Section 3.2
- `react-dom`: Section 3.3
- `react-reconciler`: Section 3.6

3.2. React for the user (react)

The highest level of abstraction for the developer is the *React* Component. This component serves as a central part in React applications, offering a well-documented interface that encapsulates both functionality and presentation logic. Developers leverage code from this part of the source code to construct intricate user interfaces efficiently, utilising a range of methods and lifecycle functions.

Within React's Component paradigm, developers find core functionalities such as the Hook system, Data Providers, and Component Lifecycle Hooks. These tools are essential for managing state, handling data flow, and controlling application component behaviour.

The Hook system, including `useState()` and `useEffect()`, enables functional components to manage local state and side effects effectively.

Data Providers, whether through Redux[6] or React's Context API, facilitate centralised state management across the application, ensuring seamless data propagation.

Meanwhile, Component Lifecycle Hooks like `componentDidMount` and `componentDidUpdate` provide precise control over component initialisation, updates, and cleanup, enhancing responsiveness and performance.

In essence, React Components embody the foundation of building blocks in React applications, offering developers powerful tools to create dynamic and scalable user interfaces.

3.3. The virtual DOM (react-dom)

The vDOM is the virtual representation of an (web) application in their UI structure. When changes are detected to the virtual DOM, the React library handles updating to the actual Browser by copy. Its main purpose is to be able to handle dynamic interface behaviour in the most efficient way possible.

Realising this concept has historically been changed significantly. The two React versions, where the focus of this work is on, follow the same concept idea, but differ indefinitely in their actual implementation. This section describes the virtual DOM mechanic by abstraction. That means, all information both applies to the old and the new version.

Data structure

The vDOM is a tree data complex. Both versions structure their code in that manner, so that the children (and possible) siblings can be located by directing through the component tree. Figure 3.3 shows the structural concept of a HTML appliance, where some items contain others.

[6] React Redux: Common external state management system for React

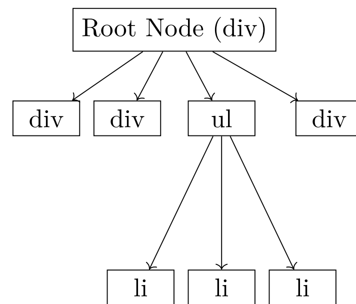

Figure 3.3.: This is a simple tree-like structure of a vDOM.

Listing 3.1 shows the JavaScript implementation of a sample React app, on which the `createRoot` methods returns a object structure of the virtual DOM.

```
1  <body>
2    <div id="root">
3      <div>Div 1</div>
4      <div>Div 2 </div>
5      <ul>
6        <li>Item 1</li>
7        <li>Item 2</li>
8        <li>Item 3</li>
9      </ul>
10     <div>Div 3</div>
11 </body>
12 <script>
13   const RootContainer = document.getElementById("root");
14   const root = ReactDOM.createRoot(RootContainer)
15   console.log(root)
16 </script>
```

Listing 3.1: An example application which prints a React Fiber Root object to the console.

The resulting console out can be found in figure 3.4.

Figure 3.4.: Console printout from listing 3.1, Safari Browser.

The key property of the attached JavaScript object for the “virtual DOM” from above is the `current` object, as this marks the root element of the tree structure. The concept of this is elaborated in 3.6.

3.4. Absolute maximum execution timings

A benchmark metric established by the React team aims to optimise the rendering engine’s performance, ensuring timely execution for fluid UI representation. [C Do 16, 5:55 min, ff.] The key metric in this context is the frame rate during extensive UI updates. The objective is to maintain consistent levels of frames per second (FPS), thereby providing a smooth and responsive user experience.

To be able to display smooth UI updates, their goal was to be able to render changes in at least $f_{min} = 60\ fps$. [Lin 17, 4:45 min, ff.] This means, that the rendering time of a single

frame can't take longer than $T_{max} = \frac{1}{f_{min}} \approx 16.67\ ms$ and hence is seen as an upper timely limit.

Historically, React utilised a recursive synchronised diffing algorithm, referred to as "Stack" (analysed in detail in section 3.5). This algorithm encountered limitations in handling rendering times, leading to insufficient UI update timings. Consequently, the React foundation team decided to develop a more controllable approach, now known as *React Fiber*.

In figure 3.5 the range of considered "good" algorithm runtime complexities is displayed concerning the resulting frames per second (FPS). This figure highlights the correlation between algorithm efficiency and UI performance, illustrating how various runtime complexities impact the achievable FPS. The diagram underscores the importance of optimising algorithm performance to ensure smooth and responsive user interactions, emphasising that maintaining high FPS is crucial for providing a seamless user experience in complex UI environments.

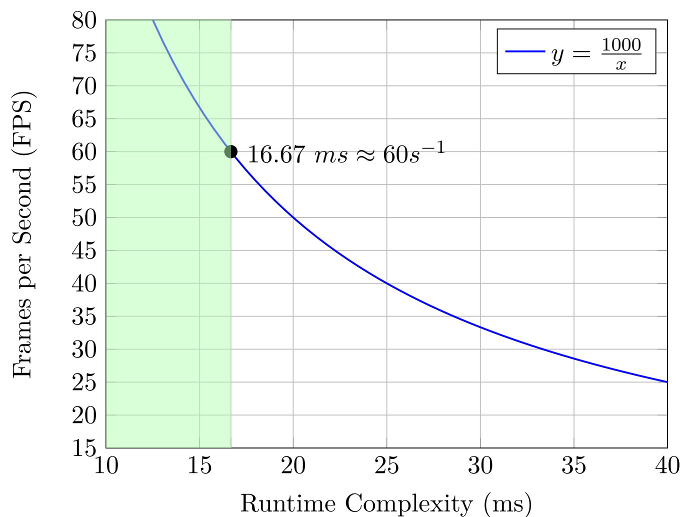

Figure 3.5.: Theoretical maximum frames per second according to rendering time.

The introduction of React Fiber brought various technical advancements, enabling better control over execution times. This refined control over the execution stack facilitated the implementation of a key concept called *React Hooks*, allowing for more granular management of component state and lifecycle.

3.5. React Stack (v. 15.6.2, < v. 16.0.0)

The last official react version which intendedly used the react stacking reconciler was version 15.6.2. The official release date was September 25, 2017 [Team 17]. This version with the commit SHA `ffbc2db0e7860ee1a96511578235dec7eaccc8d3` is focused here.

This section aims to provide proof for the function stacking algorithm which was introduced in 1.2.

In version 15, a React application is being rendered in the DOM, where a unique DIV node with $id =' root'$, in the following manner:

```
1 const rootElement = document.getElementById("root");
2 ReactDOM.render(
3   <div>
4     <SampleComponent/>
5   </div>,
6   rootElement
7 )
```

Listing 3.2: Minimal approach on how a react v15 application is created.

The `render()` function is conceptually designed to return a tree of React Elements representing the browser DOM. [Team 20]. Because of the Linked List structure of the *React-Component* type, no actual Array or JavaScript internal List type is being returned.

```
567   render: function(nextElement, container, callback) {
568     return ReactMount._renderSubtreeIntoContainer(
569       null,
570       nextElement,
571       container,
572       callback,
573     );
574   },
```

Listing 3.3: react-sourcecode/v15.6.2/src/renderers/dom/client/ReactMount.js - Implementation of the `render()` method.

When analyzing the JavaScript call stack of an example application, we can pinpoint key functions which get executed sequentially. They are displayed in figure 3.6.

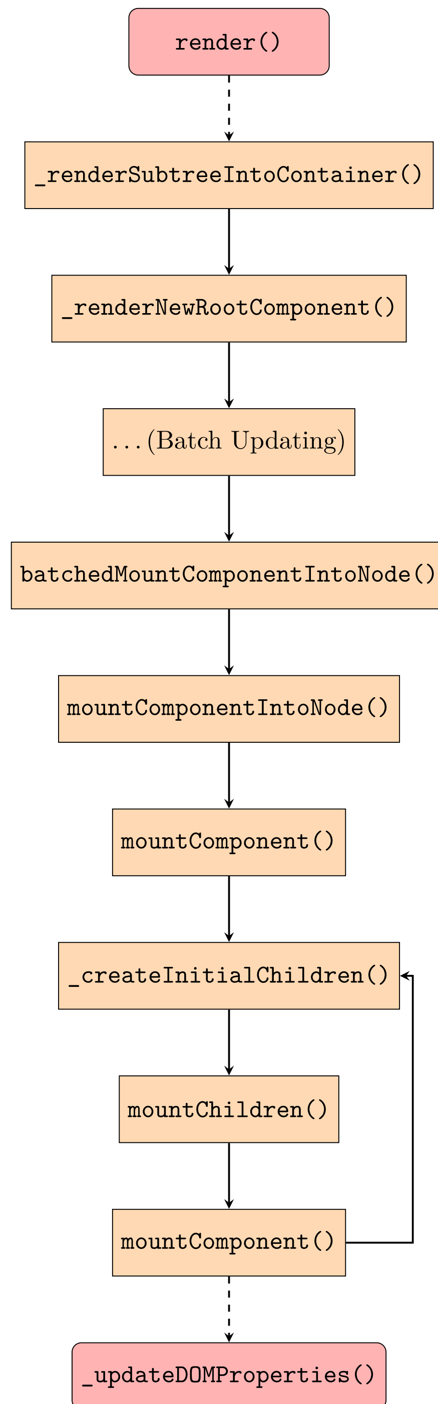

Figure 3.6.: Initial rendering process of a React v15 application.

React differs from the initial rendering process and the later UI updates. Initially, reaching from the `render()` function (which was called in the user code) to the first call of the `mountComponent()` function (internal react function), application buildup is being handled.

This call structure outlines the thesis of the initial “stacking” approach of react versions prior to 16.

3.6. React Fiber (react-reconciler)

Historically, Fiber marked a significant update to the React ecosystem. The React Team introduced numerous breaking changes aimed at enhancing features and ultimately improving rendering performance [Spri 23, p. 28].

React Fiber completely overhauled the previous virtual DOM methodology by representing each part of the UI tree with a Fiber. Previously handled React elements are now encapsulated within the React Fiber tree. The core mechanic is a double *Singly Linked List Tree* structure to depict the UI state
[Meta 24, v18.3.1, *packages/react-reconciler/src/ReactInternalTypes.js*].

The most important feature updates include:

- **Scheduling and prioritising Work**: Since not all re-renders have the same priority of being processed and JS was designed to run in a single thread, react introduces the ability to prioritise work. This gets being done by introducing several different concepts including Processing Lanes, custom call stacking and a WorkLoop
- **Optimized Callstack**: React Fiber abandoned the idea of recursive UI buildups. Instead, they now rely on a enterprise programming pattern. Leveraging the concept of *Unit of Work* with a comprehensive Workloop, this becomes possible. This is further described in section 3.6.3.
- **Caching work**: Recursive algorithms can be made significantly faster by Memoization[7] [Swei 22, "Memoization and dynamic programming"]. While not utilising recursion, because of the input/output approach of UI updates, memoization is a performance concept which is applicable in this area, too. This is made possible with a strict Unit of work separation by encapsulating UI updates in local objects.

These advancements collectively enable React to handle a larger number of rendering updates in the same time, concurrently compared to its predecessor, thereby enhancing the dynamism and responsiveness of user interfaces. This statement is later tested in chapter 4.

[7] Memoization: A technique of storing (caching) the output of a function with its input. This can avoid executing the same logic multiple times.

Analysis

The goal of the subsequent analysis is to provide a deeper insight into the working principles of React Fiber. The structure of the next sections is summarised as follows:

First, the internal types of React Fiber will be examined, as outlined in section 3.6.1. Next, the Fiber Reconciler, which is the core mechanism responsible for managing updates and maintaining the virtual DOM, will be discussed in section 3.6.2. Finally, section 3.6.3 will delve into the update queue and the lane system, which are crucial for understanding how React Fiber schedules and prioritises rendering updates.

The logic, which is explained in this chapter applies both to web frameworks and to the react-native (mobile apps) use case. This means, the react-reconciler is independent from any type of UI Element structure. Since the focus of this work is the examination behind modern web frameworks, the native use case gets obeyed.

3.6.1. Internal Types

This section focuses on familiarising with the common types used by react-reconciler for data processing, providing a broader understanding of its workings.

Type: Fiber

A Fiber in React is defined as a unit of work that specifies UI tasks to be executed or completed. It leverages aspects of the enterprise pattern "Unit of Work", as described in Fowler's work, where a unit of work tracks operations within a business transaction that impact the database [Fowl 03, p. 184]. In the context of React, the database metaphorically represents the current UI state of the DOM, which is further discussed in section 3.6.3.

Note: The relevant Fiber definition can be found in the appendix subsection A.3.1.

These properties add up to the major functionality of the React Fiber algorithm:

- `child: Fiber` (l. 197),
  `sibling: Fiber` (l. 108): The double *Singly Linked List* structure. This enables the tree structure.
- `current: Fiber` (l. 205) ,
  `alternate: Fiber | null` (l. 159): References both the currently active root Fiber and a possible alternative version of the Fiber tree for UI updating.
- `memoizedProps: any` (l. 120),
  `memoizedState: any` (l. 126): Memoization enablement

- `updateQueue: mixed` (l. 123): The Update Queue which is local to the current Fiber. Later specified in 3.6.3.

Type: FiberRoot

In the introduced Fiber Tree, the root element is assigned a specialised type that contains additional information about the entire working tree. To enhance code readability and maintainability, the code authors chose to employ a composite class notation using dot notation (...). An example of such an object can be found in the earlier figure 3.4. Listing 3.4 shows the source code.

```
331 // Exported FiberRoot type includes all properties,
332 // To avoid requiring potentially error-prone :any casts
        throughout the project.
333 // The types are defined separately within this file to ensure
        they stay in sync.
334 export type FiberRoot = {
335   ...BaseFiberRootProperties,
336   ...SuspenseCallbackOnlyFiberRootProperties,
337   ...UpdaterTrackingOnlyFiberRootProperties,
338   ...TransitionTracingOnlyFiberRootProperties,
339   ...
340 };
```

Listing 3.4: packages/react-reconciler/src/ReactInternalTypes.js - Definition of a *FiberRoot*.

The FiberRoot includes some special properties, which mostly are out-of-scope of this work. The most important one for understanding the concept is the `current` property, which has the same job as described earlier. This is the mutable root of the tree [*react-sourcecode/v18.3.1/packages/react-reconciler/src/ReactInternalTypes.js; l. 204*]. It does not alternate. Only one FiberRoot exists per virtual DOM.

Types: Lane, LaneMap and Lane

With each Update having an own Update Lane assigned, the Lane type has to be defined. This is interesting, because the internal prioritisation logic solely relies on the JavaScript `number` type. These numbers later get implemented by JavaScript's binary number presentation (`0bxxx`). This can be seen in listing 3.5.

```
16 export type Lanes = number;
17 export type Lane = number;
18 export type LaneMap<T> = Array<T>;
```

Listing 3.5: v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Type Definition of Lane, Lanes and LaneMap.

Type: Update

Since the introduction of the updating system in `react-reconciler` with version 16, prioritisation, delayed execution, and measurement of UI updates have become critical mechanics. The core mechanism enabling this functionality is encapsulated within the *Update* type. Updates are managed within another type called `UpdateQueue`, represented as a Singly Linked List structure for efficient storage and retrieval. Listing 3.6 shows the code implementation.

```
120 export type Update<State> = {|
121   // TODO: Temporary field. Will remove this by storing a map of
122   // transition -> event time on the root.
123   eventTime: number,
124   lane: Lane,
125
126   tag: 0 | 1 | 2 | 3,
127   payload: any,
128   callback: (() => mixed) | null,
129
130   next: Update<State> | null,
131 |};
```

Listing 3.6: packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Definition of React UI Updates and the subsequent Queues.

Type: UpdateQueue

The previous core types `Update` and `Lanes` get bundled in the `UpdateQueue` type in listing 3.7.

```
133 export type SharedQueue<State> = {|
134   pending: Update<State> | null,
135   interleaved: Update<State> | null,
136   lanes: Lanes,
```

```
137 |};
138
139 export type UpdateQueue<State> = {|
140   baseState: State,
141   firstBaseUpdate: Update<State> | null,
142   lastBaseUpdate: Update<State> | null,
143   shared: SharedQueue<State>,
144   effects: Array<Update<State>> | null,
145 |};
```

Listing 3.7: packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Defintion of the React UpdateQueue.

After analysing the exact work behaviour of the rendering process in 3.6.2, the updating logic will be focused on in 3.6.3.

3.6.2. The React Fiber Reconciler

Virtual DOMs, which are generated by the `react-dom` component, represent the current state of the application's user interface. This encompasses all content and styling properties necessary to convert the virtual DOM into plain HTML. When UI updates trigger re-renders in the React application, the virtual DOM is read and updated. The intermediate mechanism responsible for managing these updates and providing the described features is handled by Fibers.

Each React root instance is associated with a `ReactFiberRoot` instance, but the `ReactFiberRoot` itself is not the Fiber. Each `FiberRoot` is associated with at least one Fiber, which contains information about the root element of the virtual DOM. This specific Fiber is stored in `FiberRoot.current`, as explained in 3.6.1 Type: FiberRoot.

The call stack

The reconciliation process is started by the application creation. Before UI rendering occurs, a Fiber tree is built, initiated by the `beginWork()` function. This function is responsible for initially traversing the DOM for all kinds of elements. The goal is that every element is assigned a unique Fiber within the tree, and this is achieved without any recursive methods. Technically, not all recursive structures are removed, but these are only called within edge cases. In the Fiber buildup analysis, we focus on the synchronous WorkLoop implementation for better understanding.

The reconciliation process is initiated during the application creation. Before UI rendering occurs, a Fiber tree is constructed, beginning with the `beginWork()` function. This function is responsible for traversing the DOM to process all types of elements. The objective is to assign a unique Fiber to every element within the tree, and this is accomplished without relying on recursive methods. While not all recursive structures are eliminated, their use is limited to edge cases. In the analysis of Fiber construction, the focus is on the synchronous WorkLoop implementation to provide a clearer understanding.

There are three source files which are most important. These are `ReactFiberWorkLoop.old.js`, `ReactFiberBeginWork.old.js`, and `ReactChildFiber.old.js`. The `beginWork()` function is triggered from the work loop, which also is triggered upon application mount. `ReactFiberWorkLoop` holds information about several different work tasks, which happen during the UI updating process. `beginWork` handles Fiber creation and covers several different use cases. This includes Fragment Elements, normal React elements, and others, like LazyLoadingComponents, etc. When the `beginWork()` function from `ReactFiberBeginWork.old.js` is triggered, it can be stated that the reconciliation and therefore the diffing algorithm has started. An overview of the call stack can be found in figure 3.7.

The `beginWork()` function is called with the two major fiber tree props, which are `current` and `workInProgress`, both of type Fiber. Since `current` is mutable, the `beginWork` function handles updating the immutable `workInProgress` Fiber Tree. `beginWork()` also returns a Fiber instance. It does so because it is called by the `performUnitOfWork` function, which is part of the `ReactFiberWorkLoop` and is responsible for switching the ReactFiber tree between the `alternate` and the `current` working tree.

Having the basis cleared up, it is now interesting to take a closer look at how React exactly manages to create the initial Fiber tree, detects changes within the tree, and subsequently build up a new one, which handles the `beginWork` function. As already mentioned, the mechanic of the `beginWork` function includes that it returns the object type of Fiber. But not only that. It returns the child of the Fiber if there is one. If not, `null` is returned. Possibly, that structure would make it very easy to execute recursively. But the `beginWork` function not only covers the logic for the execution of an isolated Fiber element. In addition, it also covers some execution logic for its children.

Firstly, `beginWork()` differs based on the Fiber tag. The Fiber tag specifies the type of the component. This can be a React Functional Component, a Class component (these two types are the two possible ways of creating a custom React component in code; in this study, we focus on class components), a lazy-load component, or several other ones, which are out-of-scope for this work. For a closer examination, it is assumed that we are in the midst of the reconciliation process, with the current Fiber having the class component tag.

In that case, the actual update method
`updateClassComponent()` is being triggered. This logic does alters the current `workInProgress` Fiber Node for further processing. After that, it returns the finished `workInProgress` Fiber object. Just before returning, after the complete updating logic happened, React executes the function `reconcileChildFibers()`. As the function name implies, that function touches the child elements of the ongoing Fiber element.

As stated, this is consistent across all different Fiber tag type cases. At first, this may seem paradoxical, given that React's logic inherently avoids recursive structures. However, the "next step" information is still addressed for both optimisation and preparation for the subsequent step, where the child of the current element is processed through this call stack. Thus, the function does not provide complete Fiber creation but aims to improve next-step performance. By describing the behaviour through the call stack, performance checks are maintained. These checks include treating subsequent React Fragment children (empty HTML tags) as mere arrays or performing overall empty-children cleanup, ensuring that no unnecessary work is generated for these children.

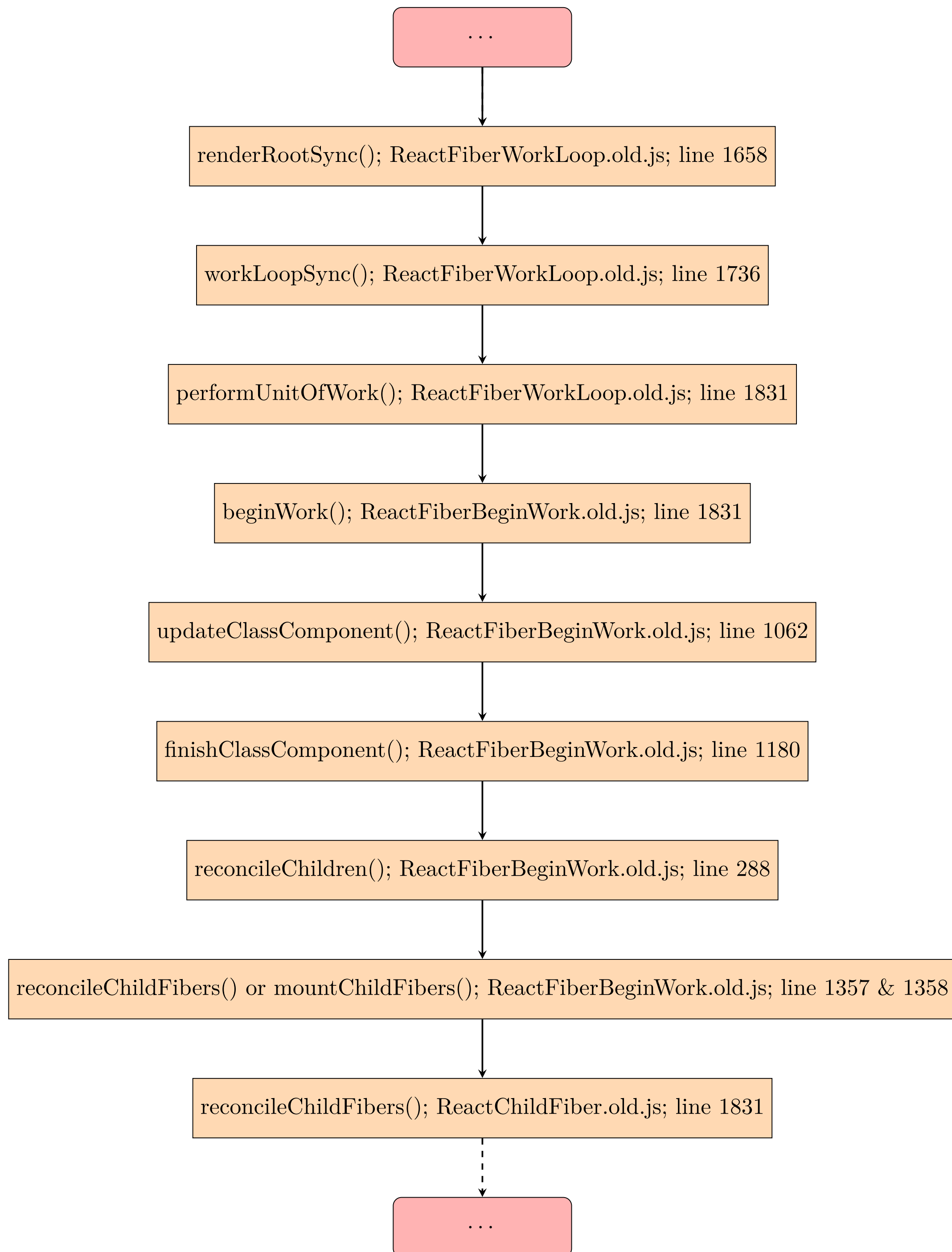

Figure 3.7.: Call Stack sequence of creating a Fiber, with the node being a React Class Component.

3.6.3. React Update Handling

React offers a highly optimised Update Scheduling System for handling virtual DOM updates within the Fiber tree. The goal of this system is to prioritise UI updates (Units of Work) and execute them sequentially. For example, when rendering UI state, handling dynamic movement is prioritised over updating text to maintain consistent frame rates.

At first glance, updating UI state in a non-sequential order might seem to cause UI state to become non-deterministic. The challenge with this approach is that some Units of Work might depend on prior ones.

React realises prioritisation by assigning a `Lane` to an `Update`. A `Lane` represents the priority of an UI state change. When the work loop runs, it initially prefers items in the lane with the highest priority. The deterministic outcome of the UI is secured by cross dependency detection of each update. Updates, which intersect with each other, will be executed multiple times, until a deterministic outcome can be ensured.

An `UpdateQueue` is constructed during the initial application render for the `current` Fiber of the FiberRoot. This is triggered by the initial `createRoot()` call in the user code (listing 3.1). This initial function call progresses through several intermediate functions. The important point is that the `FiberRoot` element, receives its `UpdateQueue` via the `initializeUpdateQueue` function (listing 3.8). This function is initially called in `createFiberRoot()` [*ReactFiberRoot.old.js*; l. 204], which acts as part of a factory method, initially assembling the root element.

```
168 export function initializeUpdateQueue<State>(fiber: Fiber): void {
169   const queue: UpdateQueue<State> = {
170     baseState: fiber.memoizedState,
171     firstBaseUpdate: null,
172     lastBaseUpdate: null,
173     shared: {
174       pending: null,
175       interleaved: null,
176       lanes: NoLanes,
177     },
178     effects: null,
179   };
180   fiber.updateQueue = queue;
181 }
```

Listing 3.8: v18.3.1/packages/react-reconciler/src/ReactFiberRoot.old.js - Function implementation of `initializeUpdateQueue()`.

Lane Logic

Overall, the `UpdateQueue` maintains a list of UI updates, which are enqueued by the `ReactFiberReconciler.old.js` and the `ReactFiberWorkLoop.old.js`. In general, a Fiber represents a unit of an UI. For example, this can be any UI element, ranging from an empty `<div/>` tag to a modal user interface, which may also have complex structure attached.

Each Fiber contains information, a `current` key (which in theory should be the same but can differ in practice) [*ReactFiberClassUpdateQueue.old.js*; l. 493f.] and an `alternate` key, each of which can have different states and their own `UpdateQueue` attached. With three `UpdateQueues`, each potentially containing a different number of enqueued updates, the `processUpdateQueue()` function in `ReactFiberClassUpdateQueue.old.js` is responsible for uniting them.

Because React Fiber offers UI update prioritisation, this function must also account for this behaviour. Both queues utilise a persistent, singly-linked list structure. When scheduling an update, it is appended to the end of both queues (`current.updateQueue` and `alternate.updateQueue`). Each queue maintains a pointer to the first unprocessed update in this persistent list (`updateQueue.firstBaseUpdate`). The work-in-progress pointer always points to a position equal to or beyond that of the current queue, as updates are always processed from there. The pointer of the current queue is only updated during the commit phase, when the work-in-progress is swapped in [*ReactFiberClassUpdateQueue.old.js*; l. 18ff.].

Update prioritisation is realised using the lane system from React. In listing 3.9, several different static lane definitions can be found. Appendix A.3.2 shows a complete list of all Lanes.

```
30 export const SyncLane: Lane = /*                         */ 0
       b0000000000000000000000000000001;
31
32 export const InputContinuousHydrationLane: Lane = /*    */ 0
       b0000000000000000000000000000010;
33 export const InputContinuousLane: Lane = /*              */ 0
       b0000000000000000000000000000100;
```

Listing 3.9: v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Binary number definitions of Lane priorities.

In total, React offers 31 different Lanes, where the exact behaviour and use case of each Lane are beyond the scope of this work. However, it is interesting to understand how the overall Lane prioritisation functions.

As already stated earlier, the Lane is just a alias type of the JavaScript number type. The lanes are implemented in a binary pattern, ranging from `0b0...1` (2^0, SynchronousLane) to `0b1....0` (2^{30}, OffscreenLane).

As the names imply, these two lanes serve different UI concerns. The SynchronousLane is crucial for ad-hoc updates that the user can see, while the OffscreenLane handles updates that are important for establishing the deterministic UI state of the library but are not as critical to the user's immediate view.

The details of how prioritisation works can be found in the official `ReactFiberClassUpdateQueue.old.js` file. Nevertheless, to maintain the completeness of this work, the example from this file is further illustrated in the following figure 3.8.

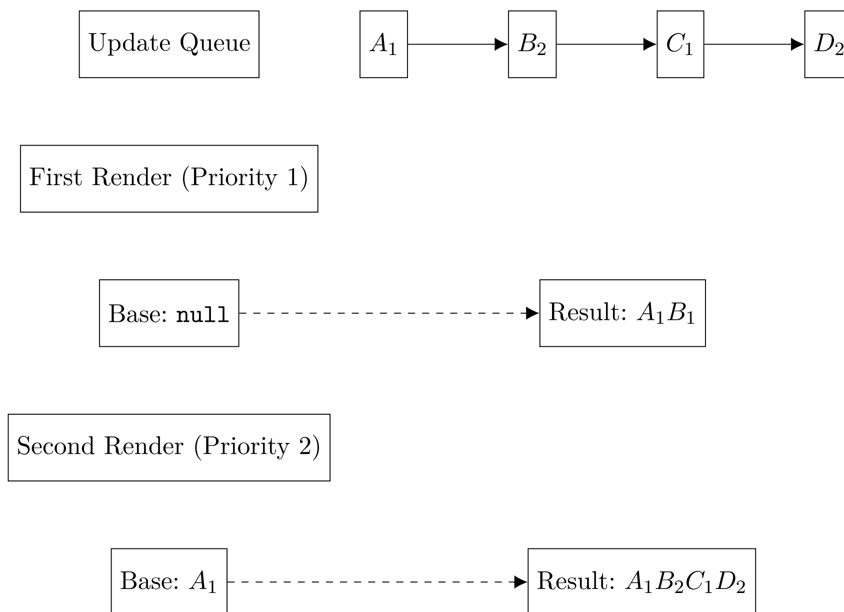

Figure 3.8.: Illustration of the official prioritization code example; Indices stand for priority; the lower the higher priority.

In this example, the two main UpdateQueues are represented by the first line (`Update Queue`) and the changing `BaseState` (initially `null`), where no updates have yet been processed. The rendering process can be executed with different priorities (Lanes).

During the initial run, all updates with the highest priority are sorted out and appended to the list, resulting in the state A_1B_1. Because B_2 was skipped, the renderer must run again to maintain the integrity of the UI. Before the next run starts, the lane is cut off where an update was skipped, changing the base state to A_1. Subsequently, in the run with the

highest priority, all subsequent UI updates are stored, as the second priority also includes updates from the first priority.

Listing 3.10 shows the head of the function, which implements the behaviour.

```
457 export function processUpdateQueue<State>(
458   workInProgress: Fiber,
459   props: any,
460   instance: any,
461   renderLanes: Lanes,
462 ): void {
```

Listing 3.10: v18.3.1/packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Function parameters of `processUpdateQueue()`.

The `processUpdateLane()` method, with the `renderLanes` parameter, takes into account which Lanes (and thus which priorities) need to be handled. Multiple rendering lanes that should be handled in one update can be achieved through binary operations of multiple render lanes. The new state of the active Fiber is set in the `getStateFromUpdate` method [*ReactFiberClassUpdateQueue.old.js*; l. 573ff.].

Chapter 4.

Experiments

In this chapter, we delve into the analysis of two fundamental React algorithms: Stack and Fiber. By comparing these algorithms, we aim to provide a detailed understanding of their respective strengths and weaknesses, focusing on their impact on application performance and user experience.

This analysis will systematically explore various facets of both algorithms, shedding light on their core functionalities, efficiency in rendering components, and overall impact on development workflows. By rigorously evaluating these aspects, we seek to offer valuable insights into how React's evolution from Stack to Fiber has influenced modern web development practices.

4.1. Code Setup and Implementation of the Testing Lab Setup

In this section, an overview is provided of the circumstances under which the tests are run.

4.1.1. Testing Lab Setup

Table 4.1 shows the used components within the testing setup. This setup was chosen to ensure consistency and reliability throughout the testing process. The MacBook Air M2 2023 provides sufficient computational power and memory capacity to handle the experimental workloads effectively. The use of Chrome guarantees a stable browsing environment, essential for accurate and reproducible test results across different scenarios.

4.1.2. Testing Project Anatomy

Using several *node.js* Projects, the testing code for react is being used. They all are implemented in a wrapper project, called `domruntime`. As the other code examples, this project can be found in the data repository of the project.

Specification	Computer 1
Computer	MacBook Air M2 2023
Memory	8 *GB*
CPU	Apple M2 Processor
Browser	Chrome Version 126.0.6478.63 (Official Build) (arm64)

Table 4.1.: Technical Specifications of the Testing Lab.

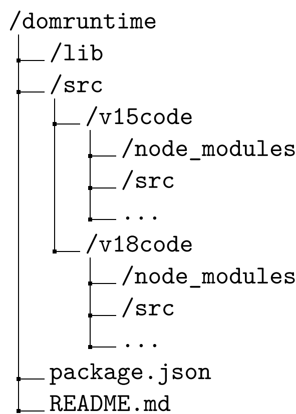

Figure 4.1.: Project Anatomy of the Testing Project “domruntime”.

- ***/v15code** and **/v18code***: Implementation of the code separated by react major version. These two folders are standalone Node projects. They are listed as submodules in the main `package.json`. The React versions used are *15.6.2* and *18.3.1*.
- *package.json*: Main Project meta information. In here, the sub-modules are listed in the `workspaces` key.

Both testing projects reference the dependencies in their own `package.json` file. The repository used for this work is *npmjs.com*. The code, which the local project then uses, is stored in the `node_modules` directory.

The React module versions is specified. The *webpack* module is used in both version projects. It’s minimal implementation enables both HMR[8] and a final build output of the application. The build is being stored in `/dist` and is used for the final elaboration. Also, *babel* is used in both versions and minimally configured to enable a professional developer experience. It enables using `.jsx` files, which enables the combinement of `HTML` and `JavaScript` in the

[8] HMR: Hot Module Replacement; Live updates in the browser during development

same directory. Also, as described in 1.1, it serves the use of translating the *ECMAScript* version. In this case, the version used is *ESNext*. Thus, no version conversion happens.

The following section describes core mechanics which are being used in order to test the high level metrics introduced at the beginning of this chapter.

4.2. Methodology

The methodology for testing both algorithms in this study involves treating the React framework as a facade, limiting the use of debugging techniques to maintain focus on the algorithms themselves. According to the facade pattern, this approach allows for controlling complex functionality from another system with a simplified interface [Gamm 95, p. 185]. This decision was made to ensure the study remains manageable in scope and to direct attention squarely on the algorithms.

In the subsequent analysis, findings will be justified based on the evaluations conducted in 3 React.js in-depth.

The evaluation will focus on metrics crucial for assessing both versions of React:

- **Component Rendering/Reconciliation**: As described in 1.7 Computing Theory, the time complexity of both algorithms will be analysed, with their performance during the component rendering and reconciliation phases being examined.
- **Development Metrics**: Compile times, developer ease, ease of codebase maintenance, and the quality of documentation for both algorithms will be evaluated.

The testing will be conducted using a simple React application that triggers key mechanics discussed earlier.

The results from these tests will provide insights into the performance and efficiency of both React versions in handling common application mechanics. This methodology ensures a thorough and focused analysis of the algorithms within the React framework, without jumping into debugging processes of the framework itself.

4.3. Testing Mechanics

The metrics which got described in section 4.2 will now be described in detail.

4.3.1. Component Rendering/Reconciliation

To trigger x amount of component updates, which have to happen all subsequently, the following mechanic is being used. In the testing codebase, a custom Component "Number-PositionComponent" is being introduced. This has the sole task to be able to respond to an user Click Event. To both test text and positional re-rendering, upon a user click, component state variables i and $offset$ get incremented. To able to control x, a recursive build-up algorithm is being used, which puts n amount of the same component into each other:

```
1  function RecNumberPositionComponent(i, j) {
2    if (i <= 0)
3      return <div/>
4    return (
5      <NumberPositionComponent side={j} number={i}>
6        {
7          RecNumberPositionComponent(i - 1, j)
8        }
9      </NumberPositionComponent>
10   )
11 }
```

Listing 4.1: Recursive implementation of building the test interface.

Figure 4.2 shows the resulting Component structure.

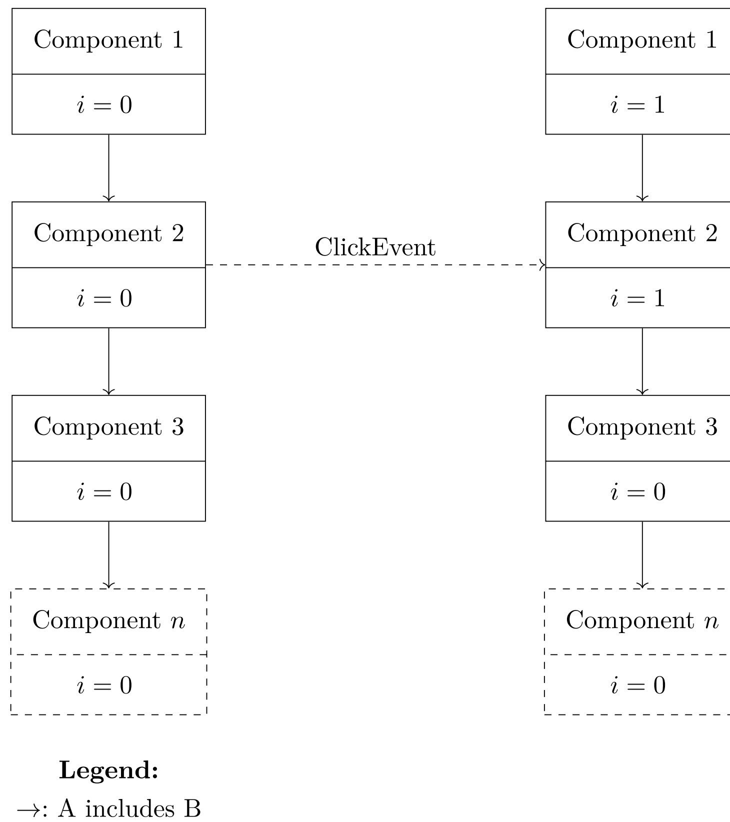

Figure 4.2.: Illustration of component relationships in a hierarchical structure. Component 1 includes Component 2, which in turn triggers Component 3. Component 3 further triggers component n, represented by a dashed arrow.

4.3.2. Development Metrics

Building, loading, or testing the React internal code requires execution time. In this section, an overview is provided on how these metrics can be obtained.

Build Time

Build time refers to the duration it takes for a software application to compile and construct its executable or deployable output. As already mentioned, the official React project transpiles the written source code into the final language version. The actions which are to be triggered in there, can be looked up in the *package.json* file. Running the correct *npm*

script for production (the end user) also delivers the required time for that task. The JSON key scripts>build within the appended React 18 package.json in appendix A.2.2 shows the concept of this.

Total package require time

The initial loading time for React and React-Dom can be measured using the built-in JavaScript function performance.now(). This method captures the precise timestamps before and after the loading of these libraries. The difference between these timestamps provides an accurate measurement of the time taken to import the React library into the application. Listing 4.2 shows the concept of this.

```
1 let t0 = performance.now();
2 const React = require("react");
3 const ReactDOM = require("react-dom");
4 let t1 = performance.now();
5 console.log("Call to require('react') and require('react-dom/
    client') took " + (t1 - t0) + " milliseconds.");
```

Listing 4.2: Measuring the time taken to require React and ReactDOM.

Test suite execution time

The React production code is backed by an extensive Test Suite. Running the complete suite is part of a CI/CD[9] process and takes a significant amount of time. The test run can be triggered using the build in package.json scripts, using the test script.

Package size

The compiled react packages differ in size. The folder size in both React, and React-DOM get measured in the corresponding node_modules folder. The folder sizes are measured by the bash command du -h -d 0 ./<folder>.

4.4. Findings

In the following, the results of the tests get shown.

[9] CI/CD: Continuous Integration / Continuous Development is a term for describing practices used in software engineering to automate the process of integrating code changes and deploying applications.

4.4.1. Component Rendering/Reconciliation

A Component Tree with a depth of 900 is created (React limits the component depth to 1000). Every node is also given an *id* tag, to programatically identify the node. A javascript program triggers and measures the time it takes to complete a single Click Event. The resulting nested div structure triggers n amount of updates in one batch, enabling this concept to work synchronously.

Diagram 4.3 and 4.4 are showing the complete execution time when a render is triggered in relation to the amount of components which have to be re-rendered in the DOM. The dotted red line at $T_{max} = 16.67\ ms \approx 60\ s^{-1}$ references the introduced fps limit in subsection 3.4.

Also, both diagrams 4.3 and 4.4 show a spike at the end of the x-axis. This is because the tests are triggered in reverse. This spike represents the time before render. It is required because of internal React buildup in addition to the initial React require time.

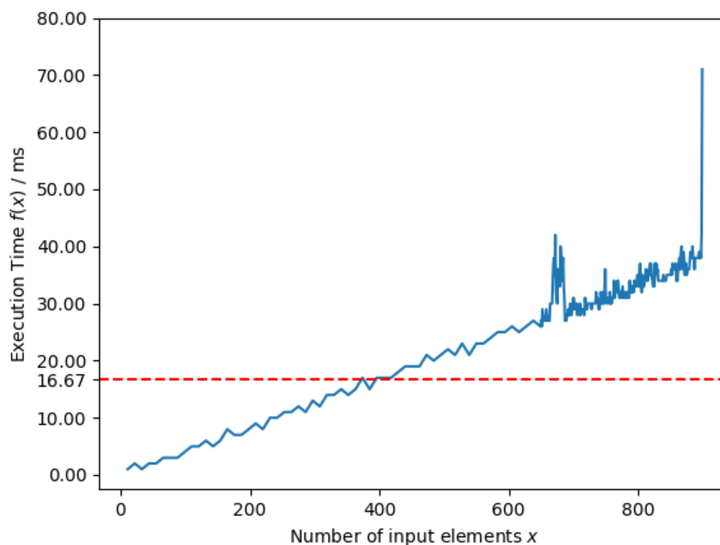

Figure 4.3.: Resulting Rendering timings in version 15.6.2.

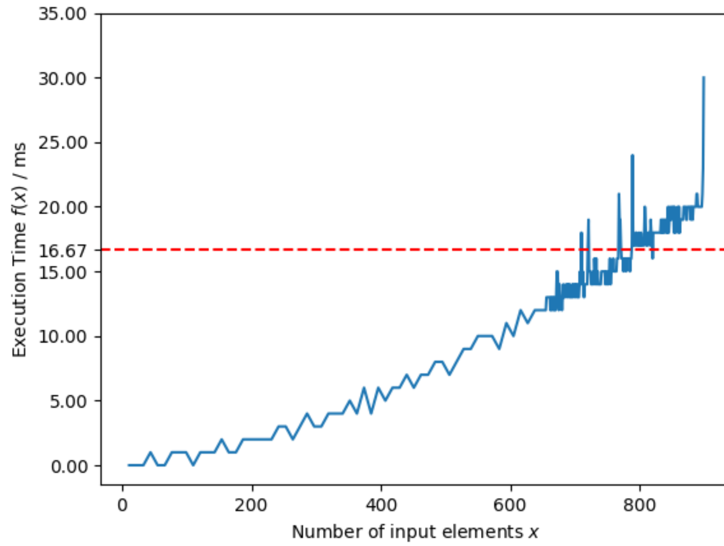

Figure 4.4.: Resulting Rendering timings in version 18.3.1.

The following information can be extracted from these diagrams:

Metric	**15.6.2**	**18.3.1**
Max. possible renders $f(T_{max})$	≈ 390	≈ 780
Time before render	$\approx 34\ ms$	$\approx 10\ ms$
Time Complexity of the rendering algorithm	$O(n)$	$O(n^v), v > 1$

Table 4.2.: Performance comparison between React versions 15.6.2 and 18.3.1 across various metrics.

4.4.2. Development Metrics

In the following, the results of development metrics are described.

Metric	**15.6.2**	**18.3.1**
Build time	47.82 s *1	197.49 s *2
Test suite execution time	299.08 s	102.20 s
Total package `require(...)` time	$\approx$ 34 ms	$\approx$ 17 ms
Package size "react"	688 kB	452 kB
Package size "react-dom"	2.5 MB	5.6 MB

Table 4.3.: Comparison of Algorithm 1 and Algorithm 2 based on various metrics.

**1*: This React versions requires an older version of *node* to run. The version used is *lts/boron, node version* 6.17.1, *npm version* 3.10.10. The bash command to measure the execution time is `time npm run build`.

**2*: The version used is *lts/iron, node version* 20.15.1, *npm version* 10.7.1. The bash command used it `yarn run build`. It automatically displays the build time.

Chapter 5.

Results

In this chapter, the findings from 4 are further analysed and justified.

5.1. Evaluation of Results

5.1.1. Rendering/Reconciliation Timings

Rendering Timings

With the excessive use of the rendering engine, executing 900 re-renders pushes React to its operational limits. Both versions comfortably handle a larger number of shallow re-renders within a tree depth of up to hundreds, adhering to the execution limit set at 16.67 ms. However, in real-world applications where component states depend on extensive data, API calls, or other dependencies requiring more time, these numbers can significantly decrease.

The contrast between the Stack and Fiber reconciler versions remains impressive. The Stack version efficiently manages component depths up to 390, whereas the Fiber Reconciler extends this capability to 780, nearly doubling the performance. This significant improvement underscores the cumulative effect of React's ongoing performance enhancements over the years.

Several factors contribute to this advancement. A pivotal factor is the introduction of the sequential call stack in Fiber, which optimizes the rendering process by prioritising and scheduling tasks more efficiently. Additionally, Fiber's architectural design allows for better management and prioritisation of UI updates, enhancing overall responsiveness and user experience.

Furthermore, Fiber's implementation introduces data structures and algorithms that streamline the reconciliation process. These innovations, such as the use of work scheduling and prioritization through lanes, further contribute to its ability to handle deeper component trees with increased efficiency.

Time Complexity

Another very interesting point. Beside the measurement inaccuracies, the graph in figure 4.3 clearly follows a linear pattern, proving that the rendering algorithm follows an $O(1)$ time complexity.

On the other hand, taking a closer look onto figure 4.4, the expected linear pattern in React Fiber clearly differs. Much more, the graph follows a slight quadratic character. This is made very clear, when taking a closer look onto the very little rising timings in the x area from 0 to 150, beside the significantly higher rising timings starting at $x = 200$. While the overall rendering timings are clearly better, the advertised linear $O(n)$ time complexity is hereby proven wrong.

Time Before Render

Another improvement of React Fiber is the initial time of react, building up the Rendering Engine. Beside the package size being $\frac{5.6+0.452-(2.5+0.688)}{0.688+2.5} \cdot 100\% \approx 90\%$ bigger, the initial loading time could be made approximately made 3 times lower.

5.1.2. Development Metrics

Build Time

The build time of React drastically increased from $\approx 48\ s$ to about $200\ s$, with a time difference of about $152\ s$.

Test suite execution time

Running the complete test suite of the react project also decreased drastically. With version 15 taking $\approx 299\ s$, version 18 can glance with a nearly a third of it, taking $\approx 102.20\ s$.

Total package `require(...) time`

The import time of React also got drastically smaller. Sparing around 18 ms, this directly influences the *First Contentful Paint (FCP)* time.

Package sizes

As already stated, the package size increased by approx. 90%. Paradoxically, React managed to extensively dampen the above measures, which speaks for itself.

Chapter 6.

Summary

This work analyses the React Fiber Rendering Engine and compares their functionality to the previously used React Stacking algorithm.

In Chapter 1, the required topics to understand the full breadth of React and its involved mechanics are introduced. Chapter 2 provides a broad overview of React's origins and offers a quick overview of the current technical state of the art. This includes a brief introduction to other web frameworks, a general explanation of how they work, and a focus on why React has achieved its current status. Chapter 3 details the React framework, thoroughly researching the React Fiber version. It delivers the most important code examples related to Stack and Fiber to enable an understanding of how the rendering mechanisms operate. Chapter 4 presents metrics on key comparison points, explains the testing project setup, and shows the results. Lastly, Chapter 5 evaluates the results from chapter 4 and provides justifications. This final chapter 6 summarises the key findings in general.

React is a component-based web framework initially used for the social network Facebook. After historically proving to be a reliable rendering engine for this use case, it was further expanded by Facebook (now Meta). Following Facebook's acquisition of Instagram, React's use case expanded through its implementation into the photo-sharing platform.

The next significant milestone for the React library was its open-sourcing. This move by the Facebook engineering team not only made a strong statement but also enabled React to be used for virtually any web project worldwide. Consequently, React has become the most widely used web rendering framework globally. This development has led to an ever-expanding variety of applications for React, increasing its usage indefinitely.

With the complexity of modern JavaScript development and the variety of different browsers React aims to support, a groundbreaking change to the core rendering engine became necessary. This led to an overhaul of the core rendering algorithm with the introduction of React 16.

The former React Stack algorithm derived its name from its approach to handling rendering updates. Executing the main rendering function of a component generated the rendering

tree in a single call. This behaviour not only caused performance issues but also resulted in longer response times for the entire user interface.

This behaviour was ultimately changed by the introduction of React Fiber. Fiber solved these issues in an elegant way by decoupling rendering updates from each other and placing them into a sequential work loop. This concept allowed for a more performant way of handling rendering updates. Additionally, the new sequential rendering system enabled the prioritisation of updates, rendering UI changes based on their importance to the user, thereby aiming to eliminate clunky behaviour.

The performance analysis proved the performance gain with the new rendering algorithm, nearly doubling the rendering execution timings in the minimal example used in this work.

Chapter 7.

Outlook

React is constantly evolving. Besides the analysis presented in this work, it is also possible to either **enhance the ongoing analysis** or **investigate potential improvements**.

7.1. Enhancement of the Analysis

In general, this work provides an overview of the two algorithms implemented in the web framework. The first approach to enhance the technical analysis in this work is to further investigate the runtime behavior of React. Detailed profiling and performance benchmarking under various conditions can reveal deeper insights into the core mechanics of the algorithm. This includes concepts such as the scheduling system, the performance of different types of rendering updates, and potential memory optimization.

Additionally, the standard React framework includes many more features, such as the Hooking system, React Native, and the Context API. These features also have underlying code that can be analyzed for functionality, performance, and potential improvements.

Another approach is to compare React to other frameworks mentioned in Chapter 2. Such comparative analysis could involve evaluating React's performance, scalability, and ease of use against those of other popular frameworks. This would help in understanding React's strengths and weaknesses relative to its competitors, providing a broader context for its development and usage.

A direct example is to use static code analysis tools, which commonly come into play when developing larger software projects. Example tools include SonarQube or Embold. By using these, an analysis can systematically identify and address issues in the codebase regarding code quality.

7.2. Improvement Investigation of React

With improved analysis comes the opportunity to identify and investigate potential enhancements for React. This involves finding new optimisations that could be integrated into the library. For instance, examining advanced state management solutions or improved methods for handling component updates could yield significant performance gains.

Another approach is to propose further improvements for the rendering engine, which could involve collaboration with the React team. Potential for this include leveraging new language features or optimising the rendering engine to take advantage of several modern features in JavaScript engines. These could be integrated into the codebase, offering the possibility of further enhancing the runtime performance of React.

Moreover, investigating community-driven improvements and experimental features from the broader React ecosystem can also provide valuable insights. Engaging with ongoing research and development efforts can help in adopting best practices and cutting-edge techniques, ensuring that React continues to evolve in line with the latest technological advancements.

7.3. Serverside React

In this work, we focused on client-side rendered React web applications. However, React also has the capability to render on the server side, which offers a whole new perspective of code analysis.

When using React on the server side, the client does not have to handle the React implementation, as ready HTML/JavaScript code are directly sent from the server. This approach requires the server to have a JavaScript engine and significantly more processing capabilities, because of handling more users simultaneously. Consequently, application security becomes much more critical with this approach. With code running on the server, there is a broader range of potential vulnerabilities.

Investigating the performance and security aspects of server-side rendering with React is of high interest. Performance analysis could reveal how server-side rendering affects load times, server load, and overall user experience. Security analysis would focus on identifying and mitigating the increased risk of vulnerabilities.

7.4. Own opinion

React is the most widely used web application framework. This is in high contrast regarding that reliable information gathering was a quite difficult task. With the widespread use of React, only very little official information about the actual engine code is publicly available. With React still maintaining two different documentation websites and only a deprecation hint in one of them, gathering important information is clumsy. Also, documentation of the internal code is not available. To leverage the open-source approach of Meta, i strongly recommend making internal documentation of the public (if available). Another option i advise is to create a detailed version of it. This would help authors, who want or already participate in the development to get a better grasp of the codebase.

Besides that, the development Team did an astounding job with the improvement of the Rendering algorithm over the years. Doubling the performance of the component rendering is big. After research, the logic of the new algorithm seems very suitable for the JavaScript engine use case.

To further employ better code quality in the code, i recommend adding a static code quality analysis tool into the CI/CD pipelines of React. By using these tools, developers can systematically identify and address issues in their codebase, leading to more robust and maintainable software. This proactive approach to code quality management is especially important in large projects where the complexity of the code can easily lead to hidden issues that are difficult to detect through manual reviews alone. Incorporating static code analysis tools into the development workflow ensures that code quality is continuously monitored and improved, ultimately resulting in a more reliable and efficient software product.

For me, i love the open principle of React and the way the library offers just a handful of functions to develop modern web applications. This seems like a more low-level approach of developing web apps, which personally suits me very well.

Appendix A.

Detailed File Information

A.1. Python scripts

A.1.1. Static dependency analyser

```
1  import os
2
3  def analyze_imports(package_path: str):
4      react_deps_from_direct_imports = dict()
5
6      for package in os.listdir(package_path):
7          react_deps_from_direct_imports[package] = list()
8
9          full_package_path = package_path + "/" + package
10
11         for entry in os.scandir(full_package_path):
12             if not entry.is_file():
13                 continue
14
15             if not entry.name.endswith(".js"):
16                 continue
17
18             if entry.path.__contains__("node_modules"):
19                 continue
20
21             print("Scanning " + entry.name)
22
23             js_file_handle = open(entry.path)
24             js_file_content = js_file_handle.readlines()
25
26             for line in js_file_content:
```

```
27
28          # Filter out lines that do not contain require or import
29          if not (line.__contains__("require") or line.startswith("
              import")):
30              continue
31
32          # Filter out lines that do not contain react
33          # This filters imports to non-production dependencies
34          if not line.__contains__("react-"):
35              continue
36
37          # Code time dependency was found, add it to the list
38          react_deps_from_direct_imports[package] += line
39
40      return react_deps_from_direct_imports
```

Listing A.1: This python method extract peer dependencies of many node-modules which exist in the same directory.

A.2. Configuration Files

A.2.1. Standard package.json

```
1  {
2    "name": "json",
3    "version": "1.0.0",
4    "main": "index.js",
5    "scripts": {
6      "test": "echo␣\"Error: no test specified\"␣&&␣exit␣1"
7    },
8    "author": "",
9    "license": "ISC",
10   "description": ""
11 }
```

Listing A.2: This package.json provides an overview of a node project.

A.2.2. Reacts 18.3.1 package.json

```
1  {
2    "private": true,
```

```
3    "workspaces": [
4      "packages/*"
5    ],
6    "devDependencies": { ... },
7    "devEngines": {
8      "node": "^12.17.0 || 13.x || 14.x || 15.x || 16.x || 17.x"
9    },
10   "jest": {
11     "testRegex": "/scripts/jest/dont-run-jest-directly\\.js$"
12   },
13   "scripts": {
14     "build": "node ./scripts/rollup/build.js",
15     "build-combined": "node ./scripts/rollup/build-all-release-
         channels.js",
16     "build-for-devtools": "cross-env RELEASE_CHANNEL=experimental 
         yarn build react/index,react/jsx,react-dom/index,react-dom/
         unstable_testing,react-is,react-debug-tools,scheduler,react
         -test-renderer,react-refresh,react-art --type=NODE && cp -r
          ./build/node_modules build/oss-experimental/",
17     "build-for-devtools-dev": "yarn build-for-devtools --type=
         NODE_DEV",
18     "build-for-devtools-prod": "yarn build-for-devtools --type=
         NODE_PROD",
19     ...
20     "test": "node ./scripts/jest/jest-cli.js",
21     ...
22   },
23   "resolutions": {
24     "react-is": "npm:react-is"
25   }
26 }
```

Listing A.3: React 18 main package.json.

A.3. React Sourcecode Snippets

A.3.1. Type definition of a Fiber

```
65 // A Fiber is work on a Component that needs to be done or was
     done. There can
```

```
66 // be more than one per component.
67 export type Fiber = {|
68   // These first fields are conceptually members of an Instance.
       This used to
69   // be split into a separate type and intersected with the other
       Fiber fields,
70   // but until Flow fixes its intersection bugs, we've merged them
        into a
71   // single type.
72
73   // An Instance is shared between all versions of a component. We
        can easily
74   // break this out into a separate object to avoid copying so
       much to the
75   // alternate versions of the tree. We put this on a single
       object for now to
76   // minimize the number of objects created during the initial
       render.
77
78   // Tag identifying the type of fiber.
79   tag: WorkTag,
80
81   // Unique identifier of this child.
82   key: null | string,
83
84   // The value of element.type which is used to preserve the
       identity during
85   // reconciliation of this child.
86   elementType: any,
87
88   // The resolved function/class/ associated with this fiber.
89   type: any,
90
91   // The local state associated with this fiber.
92   stateNode: any,
93
94   // Conceptual aliases
95   // parent : Instance -> return The parent happens to be the same
        as the
96   // return fiber since we've merged the fiber and instance.
```

```
97
98   // Remaining fields belong to Fiber
99
100  // The Fiber to return to after finishing processing this one.
101  // This is effectively the parent, but there can be multiple
        parents (two)
102  // so this is only the parent of the thing we're currently
        processing.
103  // It is conceptually the same as the return address of a stack
        frame.
104  return: Fiber | null,
105
106  // Singly Linked List Tree Structure.
107  child: Fiber | null,
108  sibling: Fiber | null,
109  index: number,
110
111  // The ref last used to attach this node.
112  // I'll avoid adding an owner field for prod and model that as
        functions.
113  ref:
114    | null
115    | (((handle: mixed) => void) & {_stringRef: ?string, ...})
116    | RefObject,
117
118  // Input is the data coming into process this fiber. Arguments.
        Props.
119  pendingProps: any, // This type will be more specific once we
        overload the tag.
120  memoizedProps: any, // The props used to create the output.
121
122  // A queue of state updates and callbacks.
123  updateQueue: mixed,
124
125  // The state used to create the output
126  memoizedState: any,
127
128  // Dependencies (contexts, events) for this fiber, if it has any
129  dependencies: Dependencies | null,
130
```

```
131   // Bitfield that describes properties about the fiber and its
         subtree. E.g.
132   // the ConcurrentMode flag indicates whether the subtree should
         be async-by-
133   // default. When a fiber is created, it inherits the mode of its
134   // parent. Additional flags can be set at creation time, but
         after that the
135   // value should remain unchanged throughout the fiber's lifetime
         , particularly
136   // before its child fibers are created.
137   mode: TypeOfMode,
138
139   // Effect
140   flags: Flags,
141   subtreeFlags: Flags,
142   deletions: Array<Fiber> | null,
143
144   // Singly linked list fast path to the next fiber with side-
         effects.
145   nextEffect: Fiber | null,
146
147   // The first and last fiber with side-effect within this subtree
         . This allows
148   // us to reuse a slice of the linked list when we reuse the work
          done within
149   // this fiber.
150   firstEffect: Fiber | null,
151   lastEffect: Fiber | null,
152
153   lanes: Lanes,
154   childLanes: Lanes,
155
156   // This is a pooled version of a Fiber. Every fiber that gets
         updated will
157   // eventually have a pair. There are cases when we can clean up
         pairs to save
158   // memory if we need to.
159   alternate: Fiber | null,
160
```

```
161   // Time spent rendering this Fiber and its descendants for the
          current update.
162   // This tells us how well the tree makes use of sCU for
          memoization.
163   // It is reset to 0 each time we render and only updated when we
          don't bailout.
164   // This field is only set when the enableProfilerTimer flag is
          enabled.
165   actualDuration?: number,
166
167   // If the Fiber is currently active in the "render" phase,
168   // This marks the time at which the work began.
169   // This field is only set when the enableProfilerTimer flag is
          enabled.
170   actualStartTime?: number,
171
172   // Duration of the most recent render time for this Fiber.
173   // This value is not updated when we bailout for memoization
          purposes.
174   // This field is only set when the enableProfilerTimer flag is
          enabled.
175   selfBaseDuration?: number,
176
177   // Sum of base times for all descendants of this Fiber.
178   // This value bubbles up during the "complete" phase.
179   // This field is only set when the enableProfilerTimer flag is
          enabled.
180   treeBaseDuration?: number,
181
182   // Conceptual aliases
183   // workInProgress : Fiber ->  alternate The alternate used for
          reuse happens
184   // to be the same as work in progress.
185   // __DEV__ only
186
187   _debugSource?: Source | null,
188   _debugOwner?: Fiber | null,
189   _debugIsCurrentlyTiming?: boolean,
190   _debugNeedsRemount?: boolean,
191
```

```
192   // Used to verify that the order of hooks does not change
        between renders.
193   _debugHookTypes?: Array<HookType> | null,
194 |};
```

Code A.4: packages/react-reconciler/src/ReactInternalTypes.js -
Type definition of Fiber.

A.3.2. v18.6.2 Lane Definition

```
33 export const TotalLanes = 31;
34
35 export const NoLanes: Lanes = /*                        */ 0
     b0000000000000000000000000000000;
36 export const NoLane: Lane = /*                          */ 0
     b0000000000000000000000000000000;
37
38 export const SyncLane: Lane = /*                        */ 0
     b0000000000000000000000000000001;
39
40 export const InputContinuousHydrationLane: Lane = /*    */ 0
     b0000000000000000000000000000010;
41 export const InputContinuousLane: Lane = /*             */ 0
     b0000000000000000000000000000100;
42
43 export const DefaultHydrationLane: Lane = /*            */ 0
     b0000000000000000000000000001000;
44 export const DefaultLane: Lane = /*                     */ 0
     b0000000000000000000000000010000;
45
46 const TransitionHydrationLane: Lane = /*                */ 0
     b0000000000000000000000000100000;
47 const TransitionLanes: Lanes = /*                       */ 0
     b0000000001111111111111111000000;
48 const TransitionLane1: Lane = /*                        */ 0
     b0000000000000000000000001000000;
49 const TransitionLane2: Lane = /*                        */ 0
     b0000000000000000000000010000000;
50 const TransitionLane3: Lane = /*                        */ 0
     b0000000000000000000000100000000;
```

```
51 const TransitionLane4: Lane = /*                        */ 0
     b0000000000000000000001000000000;
52 const TransitionLane5: Lane = /*                        */ 0
     b0000000000000000000010000000000;
53 const TransitionLane6: Lane = /*                        */ 0
     b0000000000000000000100000000000;
54 const TransitionLane7: Lane = /*                        */ 0
     b0000000000000000001000000000000;
55 const TransitionLane8: Lane = /*                        */ 0
     b0000000000000000010000000000000;
56 const TransitionLane9: Lane = /*                        */ 0
     b0000000000000000100000000000000;
57 const TransitionLane10: Lane = /*                       */ 0
     b0000000000000001000000000000000;
58 const TransitionLane11: Lane = /*                       */ 0
     b0000000000000010000000000000000;
59 const TransitionLane12: Lane = /*                       */ 0
     b0000000000000100000000000000000;
60 const TransitionLane13: Lane = /*                       */ 0
     b0000000000001000000000000000000;
61 const TransitionLane14: Lane = /*                       */ 0
     b0000000000010000000000000000000;
62 const TransitionLane15: Lane = /*                       */ 0
     b0000000000100000000000000000000;
63 const TransitionLane16: Lane = /*                       */ 0
     b0000000001000000000000000000000;
64
65 const RetryLanes: Lanes = /*                            */ 0
     b0000111110000000000000000000000;
66 const RetryLane1: Lane = /*                             */ 0
     b0000000010000000000000000000000;
67 const RetryLane2: Lane = /*                             */ 0
     b0000000100000000000000000000000;
68 const RetryLane3: Lane = /*                             */ 0
     b0000001000000000000000000000000;
69 const RetryLane4: Lane = /*                             */ 0
     b0000010000000000000000000000000;
70 const RetryLane5: Lane = /*                             */ 0
     b0000100000000000000000000000000;
71
```

```
72 export const SomeRetryLane: Lane = RetryLane1;
73
74 export const SelectiveHydrationLane: Lane = /*              */ 0
       b0001000000000000000000000000000;
75
76 const NonIdleLanes: Lanes = /*                              */ 0
       b0001111111111111111111111111111;
77
78 export const IdleHydrationLane: Lane = /*                   */ 0
       b0010000000000000000000000000000;
79 export const IdleLane: Lane = /*                            */ 0
       b0100000000000000000000000000000;
80
81 export const OffscreenLane: Lane = /*                       */ 0
       b1000000000000000000000000000000;
```

Listing A.5: v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Numerical lane prioritisation definitions.

List of Figures

1.1. File structure of a typical Node project called 'minimal-project'. 7
1.2. Strongly simplified stack representation after each function call. 8
1.3. Strongly simplified growing stack after each function call. 9
1.4. Singly Linked List Structure. A singly linked list consists of a sequence of elements, where each element points to the next element in the sequence. 10
1.5. Comparison of Various Time Complexities in Algorithms. The plot illustrates the growth rates of linear (n), quadratic (n^2), cubic (n^3), logarithmic ($\log n$), and linearithmic ($n \log n$) time complexities with respect to input size n. As n increases, each complexity demonstrates its characteristic rate of growth in the number of computational steps. 12

3.1. Overview of the react folder structure. 19
3.2. Simplified component of React 18.3.1, applicable for the web application. 20
3.3. This is a simple tree-like structure of a vDOM. 22
3.4. Console printout from listing 3.1, Safari Browser. 23
3.5. Theoretical maximum frames per second according to rendering time. 24
3.6. Initial rendering process of a React v15 application. 26
3.7. Call Stack sequence of creating a Fiber, with the node being a React Class Component. 35
3.8. Illustration of the official prioritization code example; Indices stand for priority; the lower the higher priority. 38

4.1. Project Anatomy of the Testing Project "domruntime". 42
4.2. Illustration of component relationships in a hierarchical structure. Component 1 includes Component 2, which in turn triggers Component 3. Component 3 further triggers component n, represented by a dashed arrow. 46
4.3. Resulting Rendering timings in version 15.6.2. 48
4.4. Resulting Rendering timings in version 18.3.1. 49

List of Tables

1.1. Evolution of ECMAScript versions and their release years [TC36 24]. 2
1.2. Time Complexity of Binary Trees. The table shows the time complexity of various methods for binary trees, including size, root, and height [Good 14, p. 333ff.]. . . 12

4.1. Technical Specifications of the Testing Lab. 42
4.2. Performance comparison between React versions 15.6.2 and 18.3.1 across various metrics. 49
4.3. Comparison of Algorithm 1 and Algorithm 2 based on various metrics. 50

List of Listings

1.1. Example of sequentially executed functions. 8
1.2. Recursive function example in JavaScript. 8

3.1. An example application which prints a React Fiber Root object to the console. . 22
3.2. Minimal approach on how a react v15 application is created. 25
3.3. react-sourcecode/v15.6.2/src/renderers/dom/client/ReactMount.js - Implementation of the `render()` method. 25
3.4. packages/react-reconciler/src/ReactInternalTypes.js - Definition of a *FiberRoot*. 30
3.5. v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Type Definition of Lane, Lanes and LaneMap. 31
3.6. packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Definition of React UI Updates and the subsequent Queues. 31
3.7. packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Defintion of the React UpdateQueue. 31
3.8. v18.3.1/packages/react-reconciler/src/ReactFiberRoot.old.js - Function implementation of `initializeUpdateQueue()`. 36
3.9. v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Binary number definitions of Lane priorities. 37
3.10. v18.3.1/packages/react-reconciler/src/ReactFiberClassUpdateQueue.old.js - Function parameters of `processUpdateQueue()`. 39

4.1. Recursive implementation of building the test interface. 45
4.2. Measuring the time taken to require React and ReactDOM. 47

A.1. This python method extract peer dependencies of many node-modules which exist in the same directory. 61
A.2. This package.json provides an overview of a node project. 62
A.3. React 18 main package.json. 62
A.4. packages/react-reconciler/src/ReactInternalTypes.js - Type definition of Fiber. 63
A.5. v18.3.1/packages/react-reconciler/src/ReactFiberLane.old.js - Numerical lane prioritisation definitions. 68

Bibliography

[Acke 17] Ackermann, Philip. *JavaScript: das umfassende Handbuch*. *Rheinwerk Computing*, Rheinwerk Verlag GmbH, Bonn, 1. auflage 2016, 1. korrigierter nachdruck Ed., 2017.

[Angu 24] Angular. *Angular*. https://github.com/angular/angular, Accessed at 2024-11-07.

[C Do 16] C. Dodds, Kent. *Why, What, and How of React Fiber with Dan Abramov and Andrew Clark*. https://www.youtube.com/watch?v=crM1iRVGpGQ, Accessed at 2024-02-06.

[Clar 17] Clark, Andrew. *Web Archive: React v16.0*. https://web.archive.org/web/20171229235000/http://reactjs.org/blog/2017/09/26/react-v16.0.html, Accessed at 2024-05-06.

[Docs 24] NPM Docs. *workspaces*. https://docs.npmjs.com/cli/v10/using-npm/workspaces, Accessed at 2024-06-21.

[Fowl 03] Fowler, Martin. *Patterns of enterprise application architecture*. *The Addison-Wesley signature series*, Addison-Wesley, Boston, 2003.

[Gamm 95] E. Gamma, Ed. *Design patterns: elements of reusable object-oriented software*. *Addison-Wesley professional computing series*, Addison-Wesley, Reading, Mass, 1995.

[Good 14] Goodrich, Michael T. and Tamassia, Roberto and Goldwasser, Michael H. *Data structures and algorithms in Java*. Wiley, Hoboken, NJ, 6. ed Ed., 2014.

[Hofm 00] Hofmeister, Christine and Nord, Robert L. and Soni, Dilip. *Applied software architecture*. *Addison-Wesley object technology series*, Addison-Wesley, Reading (Mass.) Harlow (G. B.) Menlo Park (Calif.), 2000.

[Inte 24] ECMA International. *Home - Ecma International*. https://ecma-international.org, Accessed at 2024-06-13.

[JSCo 13] JSConf. *[JSConfUS 2013] Tom Occhino and Jordan Walke: JS Apps at Facebook*. https://www.youtube.com/watch?v=GW0rj4sNH2w, Accessed at 2024-05-06.

[Lin 17] Lin, Clark. *Lin Clark - A Cartoon Intro to Fiber - React Conf 2017*. https://www.youtube.com/watch?v=ZCuYPiUIONs, Accessed at 2024-04-06.

[Meta 24] Meta. *react Frontend Framework Repository*. https://github.com/facebook/react, Accessed at 2024-04-20.

[Sips 06] Sipser, Michael. *Introduction to the theory of computation*. Thomson Course Technology, Boston, 2nd ed Ed., 2006.

[Spri 23] Springer, Sebastian. *React: das umfassende Handbuch*. *Rheinwerk Computing*, Rheinwerk Verlag, Bonn, 2., aktualisierte und erweiterte auflage Ed., 2023.

[Swei 22] Sweigart, Al. *The recursive book of recursion: ace the coding interview with Python and JavaScript*. No Starch Press, San Francisco, CA, 2022.

[TC36 24] TC36. *ECMAScript® 2025 Language Specification*. https://tc39.es/ecma262/, Accessed at 2024-12-06.

[Team 17] React Team. *Release v15.6.2*. https://github.com/facebook/react/releases/tag/v15.6.2, Accessed at 2024-01-07.

[Team 20] React Team. *Reconciliation - React*. https://legacy.reactjs.org/docs/reconciliation.html, Accessed at 2024-01-07.

[Team 24a] Babel Dev. Team. *What is Babel?* https://babeljs.io/docs/, Accessed at 2024-06-14.

[Team 24b] React Team. *Describing the UI*. https://react.dev/learn/describing-the-ui, Accessed at 2024-06-21.

[vuej 24] vuejs. *core*. https://github.com/vuejs/core, Accessed at 2024-11-07.

[Walk 24] Walke, Jordan. *Jordan Walke | LinkedIn*. https://www.linkedin.com/in/jordan-walke-1250b634/, Accessed at 2024-11-07.

[WIRF 20] WIRFS-BROCK, ALLEN and EICH, BRENDAN. “JavaScript: The First 20 Years”. Vol. Proceedings of the ACM on Programming Languages, No. HOPL, p. 189, Dec. 2020.

[York 15] York, Richard. *Web development with jQuery*. Wiley Online Library, Indianapolis, 2015.