0.1 Summary

In this thesis we have worked on improving the performance of JavaScript applications, written in the front-end framework React. We designed and implemented *React-compiler*, a compiler written in JavaScript for compiling applications into more performant JavaScript code. *React-compiler* enables React developers to obtain better performing applications, without rewriting them in a different front-end framework. This thesis is a continuation of the preliminary work in our ninth semester project. In our ninth semester project, we built a proof of concept compiler to investigate whether it was possible to compile React applications to vanilla JavaScript. Additionally, we proposed a methodology for benchmarking front-end frameworks as there was no standard in the industry for comparison. In our ninth semester project, we concluded that it was possible to build a compiler for React, however, with mixed results.

In this thesis, we chose to focus on improving the performance of our proof of concept, while addressing the flaws found in our methodology. We structured our work in 3 iterations and ran benchmarks at the end of each iteration to measure the performance of the optimisations created in the iteration. We use an open source benchmarking tool, named js-benchmark-framework, but we refer to it by its author's name, Krausest. Krausest measures how fast implementations written in different frameworks perform operations on the DOM of a web page in the browser. It also measures startup time and memory consumption.

React-compiler removes the React runtime from the compiled application entirely, thus reducing its total file size, which improves load times, and reduces memory consumption. React uses a process at runtime to reduce the number of DOM operations being performed when the application is updated. This process is called reconciliation and relies on a virtual DOM, which is an in memory representation of the DOM shown to the user. The reconciliation process improves the performance of the applications at a cost of increased memory usage. It also increases the total size of the application, because of all the features required in the React runtime.

React-compiler takes inspiration from another framework, named Svelte, and tracks any dependecies found throughout an application. *React-compiler* will then generate DOM update functions, which only updates DOM nodes whose dependencies have been modified. This allows *React-compiler* to reduce the amount of DOM operations required to update the DOM when a change occurs. However, because the dependencies are pre-computed, *React-compiler* can update the DOM more efficiently than React, because it can update the DOM directly rather than having to perform a "diffing" process.

During development of *React-compiler*, we encountered various challenges. For instance, we found a correlation between performance issues, discovered after iteration 2, and the amount of CPU slowdown used in the individual benchmarks. Disabling CPU slowdown showed, that the compiled applications performed as expected. We theorised that the slower CPU speeds were unable to handle the large number of DOM operations being performed in the benchmarks, which caused the browser to overload. Our conclusion was that we needed to implement a scheduler for the compiled applications at runtime to prioritise DOM operations, especially on slower CPU speeds. This was based on the performance of React, which uses a scheduler at runtime, and managed to perform better than the compiled applications both with and without CPU slowdown. We have not implemented a scheduler in this project, however, the implementation in iteration 3 resulted in better performance than React in the majority of benchmarks. These results did not show any performance issues related to CPU slowdown despite the lack of a scheduler.

The compiled applications outperform React by 28% - 1740% in 5 out of 10 benchmarks in the DOM operation category of the Krausest benchmarking tool. In the 5 benchmarks where *React-compiler* is worse, it is 6% - 40% slower. Furthermore, when compiling an application with *React-compiler*, the total file size of the application is reduced by up to 87%.

React-compiler does have some limitations in terms of which React features it is capable of compiling such as React hooks, portals, and context. Multiple return expressions in a React component's render method are also not supported. Using third party libraries such as Redux or MobX, does not work if the library requires its input to be a React component, since the compiled application has removed React entirely.

React-compiler

Compiling React Applications to Improve Performance

Group pt104f20

Kristoffer Magill Nash

Niclas Jon Sommer

June 12, 2020

Aalborg University

Software, 10. semester

Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
React-compiler - Compiling React Applications to Improve Performance

Theme:
Master Thesis

Project Period:
Spring Semester 2020

Project Group:
pt104f20

Participant(s):
Kristoffer Magill Nash
Niclas Jon Sommer

Supervisor(s):
Lone Leth Thomsen and Bent Thomsen

Copies: Online on Digital Exam

Page Numbers: 127

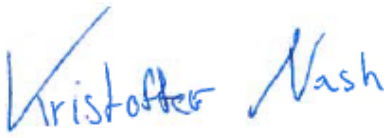
Date of Completion: June 12, 2020

Abstract:

In this project, we design and implement *React-compiler*, a compiler written in JavaScript for compiling applications written in React, a front-end framework for the web, to more performant JavaScript code. The compiled applications outperform React by up to 28% - 1740% in 5 out of 10 benchmarks in the Krausest benchmarking tool. In the worst performing benchmark we are 40% worse than React. The total file size of the application is reduced by up to 87% when compiled in our benchmarks. These results are achieved, in part by, removing the React runtime, and minimising DOM operations by computing dependencies of JSX elements at compile time. *React-compiler* does have some limitations in terms of which React features it is capable of compiling such as React hooks, portals, context. It also does not support third party libraries without modification to the library itself.

Preface

This report assumes that the reader has a Computer Science related degree or is knowledgable in the field. Furthermore, the report, assumes a general understanding of the JavaScript programming language. This report only considers JavaScript front-end frameworks.



Kristoffer Magill Nash

Niclas Jon Sommer

List of Figures

Contents

Chapter 1

Introduction

This report is based on our previous work involving the development of a methodology for comparing front-end frameworks and the compilation of React applications [1]. This report will reuse many concepts as well as knowledge provided in said report and it is, therefore, advantageous to have read it. A summary of our previous work is added for convenience in section 1.1, which covers the major contributions, however, for more in-depth explanations of various concepts, one must refer to our previous report [1].

1.1 Background And Related Work

We investigated front-end frameworks in terms of performance, with the goal of the project being to compile React applications to vanilla JavaScript [1]. To assist with this investigation, we chose to focus on two frameworks based on their distinct approach to optimising DOM operations on a web page. In addition to this, we developed a methodology for comparing the two frameworks as well as a number of other popular frameworks such as Vue.js and Angular. Using the knowledge gained from researching the two frameworks, we built a Babel plugin which can compile React applications into vanilla JavaScript applications. Using our developed methodology, we then compared our implementation against both Svelte, React, and vanilla JavaScript implementations to see how they compared in terms of performance. This section will provide the background information needed to understand our 9th semester report. A shortened explanation of the two frameworks is provided in section 1.1.1 and section 1.1.2. For a more detailed description of the frameworks, sections of the same names can be found in our 9th semester project [1]. In section 1.1.3, we describe the methodology and results of our tests, and in section 1.1.4, we describe our implementation.

1.1.1 React

React, is developed by a dedicated team at Facebook and uses a component based approach to the development of UIs [2]. Applications written in React are structured using components [3]. Components are JavaScript classes or functions that describe how the UI should appear. UI elements such as buttons or input fields are created as components. Furthermore, components are composable, meaning a component may include one or more other components in its output.

An example of what a React component could look like can be seen in Code 1.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment(event) {
    this.setState({ count: this.state.count + 1 });
  }

  componentDidMount() {
    console.log('component mounted!');
  }

  render() {
    return (
      <div id="container">
        <h1>Hello {this.props.value}</h1>
        <h2>Count: {this.state.count}</h2>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

ReactDOM.render(
  <App value={'World!'}/>,
  document.getElementById('root')
);
```

Code 1: A class component in React.

React has two distinct methods of creating components. Class components, as depicted in

Code 1, and function components, which are JavaScript functions that return JSX, similar to the contents of the `render` method in Code 1 on line 15. The `render` method is called by React to render HTML in the browser [4]. A React application is instantiated by calling the `ReactDOM.render` method as seen on line 26. The first parameter is the React component to render and the second is a reference to the DOM node to use as the parent for the application [3].

JSX and Babel

JSX is a syntax extension introduced by React, it allows developers to write markup and logic in the same file in a convenient format to improve productivity [5]. JSX is similar to XML/HTML syntax and is used by preprocessors, such as Babel. Babel is a toolchain used to convert JavaScript versions, newer than ES5, into a backward-compatible version of JavaScript, such that it can be used in older browsers or environments [6]. JSX consists of JSXElements which are declared using the XML tag syntax, where the name inside $< NAME >$ denotes the type of HTML element or component that should be rendered in the DOM [5]. JSX is structured as trees of JSXElements. A JSXElement can have children, which are JSXElements nested inside another JSXElement. Inversely, a JSXElement's parent is the enclosing JSXElement. JSXElements can have attributes as seen on line 17 in Code 1, which are used to control the behavior of either the HTML element or component that the JSXElement represents.

React uses Babel to transform JSX syntax into `React.createElement` calls [5]. JSX is shorthand for `React.createElement`, a function used to build HTML elements. The example in Code 2 show how Babel transpiles JSX to `React.createElement` call. The JSX needs to be transpiled into `React.createElement` function calls because JSX is not supported by browsers directly since it is not part of the JavaScript standard.

```
1  <div id="container">Hello World</div>
2
3  // is transpiled into
4
5  React.createElement(
6    "div",
7    { id: "container" },
8    "Hello World"
9  )
```

Code 2: Example of JSX transformation to createElement call.

A JSXExpression is denoted with and it can contain any valid JavaScript expression [5]. An example of this can be seen in Code 1 on line 18. In this case, the value stored in the `props` object's `value` property is returned from the JSXExpression and displayed in the div element. JSX can be used as an expression in JavaScript, so JSXExpression can also contain JSX inside [5]. The `props` object is used to pass values from the parent component to a child component. React populates the `props` object with the attributes provided in the JSX so they can be used inside the component. The `value` property on line 18 contains the value of the attribute with the same name declared on line 27.

State and Lifecycle

One of the features of React is that it automatically updates the DOM when the state of the application changes, which is also referred to as rerendering [7]. React only tracks certain changes to a component's props or state object [3]. The props object is controlled by the parent, since the values inside the object is passed to the component through the JSXElement's attributes. The state object is internal to the React component and is updated by calling the `setState` method as seen on line 8 in Code 1. The first parameter is an object, which is merged with the current state object and a rerender is triggered by React automatically. React also exposes an API for executing code in different phases of the rerendering process called lifecycle methods [7]. One of these, `componentDidMount`, can be seen on line 11 in Code 1. React calls this method when the component has been mounted into the DOM. There are additional lifecycle methods, a more detailed description of them can be found in our 9th semester project [1].

Events

React also needs to handle user interaction of a webpage. This is done by adding a function called an event listener to reserved attributes on JSXElements. For instance, to handle button clicks, we can set the `onClick` attribute to a function as seen on line 20 in Code 1. In this case, we use the `increment` method on the component to increment the `count` property in the component's state. Similar attributes exist for events such as mouseover and keypress.

Reconciliation

As mentioned in section 1.1.1, whenever the state of a component changes, the React runtime will update the DOM to match the new state of the application [4]. The DOM represents what the user can currently see. React calls the `render` method inside the components to determine what the new DOM looks like [4, 8]. Updating the DOM is expensive, so React tries to minimize the number of changes made to the DOM during rerendering to improve performance [4, 8]. This is called the reconciliation process in React [4, 8]. React use a virtual DOM which is a representation of the DOM that is held in memory and, whenever a component's state changes, that change is applied to the virtual DOM [4, 8]. Operating on the virtual DOM is much faster than the actual DOM because the browser does not need to display it to the user [4, 8]. During rerendering, React will compute the new version of the virtual DOM by calling the `render` methods of all components in the application. When React has updated all components, it will compute the difference between the new virtual DOM and the old one which represents the current DOM shown to the user. This process is called "diffing" [4, 8]. The result of the diffing process is a list of DOM operations which have to be performed in the DOM to bring it up to date with the new virtual DOM. This way, React can compute the minimal number of DOM operations needed to bring the DOM up to date, rather than updating everything [4, 8].

1.1.2 Svelte

Svelte implements a compiler which replaces the virtual DOM by precomputing the required DOM operations, resulting in better runtime performance compared to React [9]. Similar to React, Svelte is also a component based framework and also introduces its own syntax [9]. Svelte is written in *.svelte* files and the syntax closely resemble standard HTML files. One key difference between React and Svelte is how Svelte handles styling within the component, whereas React relies on CSS to style components in an additional file. An example of a Svelte component can be seen in Code 3.

A component consists of three parts. The `<script>` section, which contains the JavaScript, the `<style>` section, which contains any styling for the component, and the HTML section which contains the HTML markup. Similar to React, Svelte also allows for inserting expressions into the markup, as seen on line 9. The name of the file is the name of the component.

```
1  <script>
2    let name = 'world';
3    let count = 0
4
5    const incrementCount = () => {
6      count++
7    }
8
9    $: doubled = count * 2;
10 </script>
11
12 <style></style>
13
14 <h1>Hello {name}!</h1>
15 <div>{count}</div>
16 <p>count doubled is {doubled}</p>
17 <button on:click={incrementCount}>increment</button>
18 <input bind:value={name} >
```

Code 3: A basic svelte component

State and Lifecycle

Svelte also uses the notion of state within its components. However, as opposed to React, where properties are updated through the function `setstate`, Svelte can directly update the value of a variable and still trigger a rerender [10]. An Example of this is the `incrementCount` function on line 5 in Code 3. The `count` variable is used on line 16, which will be displayed to the user. Svelte keeps track of where variables are used in the DOM so that it can update the DOM nodes when the variables change [9].

Svelte introduces the **$** syntax to handle computed values as seen on line 9 in Code 3 [9]. That is, values whose value relies on other variables. This syntax allows for automatically updating a value whenever the variable(s) it depends on is updated. The value `doubled` is declared to be dependant on the count variable. This means whenever the user clicks the button, which increments the count, the doubled variable is also updated.

In addition to state, Svelte also has lifecycle events that dictate how a component should behave when mounting and unmounting into the DOM [10]. Svelte has four lifecycle events that handle different stages of a component's lifecycle — `onMount`, similar to React's `componentDidMount`, `onDestroy`, similar to React's `componentWillUnmount`, `beforeUpdate`, and `afterUpdated`. The two former of the four have similar use cases to React, such as starting and clearing timers or fetching data from sources when the component is mounted. The two latter lifecycle events are used to schedule work to happen either

immediately before or after the DOM has been updated [9].

Events

Svelte uses event listeners to handle interactions such as button presses similar to React. The button on line 18 in Code 3 has `incrementCount` assigned as the event listener for the click event on the button. Svelte also allows the binding of variables to the value of the input field as seen on line 19. This automatically updates the `name` variable as the user types into the input field. This feature reduces the number of event listeners that a developer needs to write if the event listener only needs to update the value of a variable.

The Svelte Compiler

Svelte ships with its own compiler instead of using a transpiler like Babel [9]. The *.svelte* files are not supported by browsers and the syntax is non-standard which means the entire Svelte file must be compiled to JavaScript. Similar to how React must compile JSX, Svelte compiles the entirety of the file to JavaScript. Svelte pre-computes all values and dependencies during compilation which means it does not require an algorithm or runtime to update the DOM [9, 11]. This means Svelte can automatically update values and, thus, the DOM whenever a value changes, as has been described in the previous sections. The inspiration for this approach to front-end frameworks, which in recent years has used runtimes and virtual DOM's, comes from spreadsheets [11]. At compile time, the Svelte compiler will generate a dependency graph for the state of a component [11]. It maps out all the values and their dependencies and generates functions that specifically update those values depending on the various states of the component. The Svelte compiler moves the work done by the React reconciliation process to compile time which results in a smaller overhead at runtime.

1.1.3 Methodology

To investigate and benchmark React and Svelte, we developed a methodology on how to compare the frameworks. The methodology consisted of benchmarks in two categories — micro- and macrobenchmarks. For each category, we used an existing tool in order to run the benchmarks. For the microbenchmarks, we used Krausest [12], and for the macrobenchmarks, we used Google Lighthouse [13]. The microbenchmarks primarily consisted of measuring the performance of DOM operations in terms of duration, memory usage, and start-up metrics of frameworks, while the macrobenchmarks measured various metrics relating to the

interactiveness of a webpage written in a framework. Krausest uses a web driver to automate the benchmarking process. Each implementation in the Krausest tool must implement a set of features which are used for running the benchmarks. Krausest is framework agnostic, so it can benchmark any framework that can be run in the browser like JavaScript. It is fast to add additional implementations to the Krausest benchmarks. The result of running the Krausest benchmarks is a table containing the runtime of each implementation for each benchmark and the relative performance of each implementation to the others.

We measure the performance of a framework in 3 categories — (1) DOM operations, (2) Memory usage, and (3) Startup metrics. We focus mainly on the performance in the DOM operations category and have, therefore, listed the benchmarks used below:

- **Create rows:** Insert a 1.000 rows of DOM nodes into a table in the DOM.
- **Replace all rows:** Replaces 1.000 rows of DOM nodes with 1.000 new rows in a table.
- **Partial update:** Update every 10th row in a table with 1.000 rows
- **Select row:** Highlights a selected row by clicking on it.
- **Swap rows:** Swap the 2nd and 2nd to last rows in a table of 1.000 rows
- **Remove row:** Removes one row from a table of 1.000 rows
- **Create many rows:** Same as create rows but with 10.000 rows.
- **Append rows:** Append 1.000 rows to a table of 10.000 rows.
- **Clear rows:** Remove all rows from a table.

The benchmarks used in the remaining categories are present in our result tables and will be introduced when used.

The microbenchmarks showed that Svelte outperformed React in terms of performance. Svelte performed similarly to a JavaScript implementation without any framework. However, these performance differences were not apparent in the macrobenchmarks. The results showed negligible performance difference between React and Svelte, although Svelte still came out on top. This seemed to suggest that the performance of the DOM operations generated by a framework does not have as big of an impact on a real use case compared to the synthetic microbenchmarks. It is also possible that our methodology for macrobenchmarking was incorrect. The tool used for macrobenchmarks, Google Lighthouse, did not interact with the rendered web page by, for instance, clicking a button which means that only the

loading of the application and initial render was benchmarked. Svelte had an advantage over React in this case, since it generated smaller applications in terms of total file size.

1.1.4 Proof of Concept

Once the methodology for comparing frameworks had been created, we moved on to building a proof of concept (PoC) compiler which can compile React applications into standard JavaScript (also known as vanilla JavaScript). The goal of the PoC was to test whether compiling a React application was possible or not. Furthermore, we were interested in seeing the potential performance-related impact of compiling React to vanilla JavaScript and removing the virtual DOM in favour of a compilation process similar to Svelte. We took advantage of an existing compiler, Babel, which is already part of most React applications and used to compile JSX into vanilla JavaScript. This allowed us to implement the PoC as a Babel plugin which meant we did not have to implement a compiler from scratch.

The plugin transforms React components and JSX by modifying the abstract syntax tree (AST). React components are converted to a new component type called `Component`. The component type uses the same API as React, which allows us to reuse existing API calls in the source code. React components can be created using either a function or a class. React does not distinguish the two approaches except for state and lifecycle methods, which are only available to class components. The PoC converts the function components into class components by inserting the body of the function component into a new class components `render` method. This is done so that the PoC does not have to handle different types of components when converting to the new component type. The JSX in the application is transformed into a JavaScript object tree structure, which is then passed to a helper function, `createElement`, to perform the DOM operations. The DOM nodes created by a component are removed during a rerender before creating new ones to replace them. This was done for the simplicity of the PoC rather than for performance reasons. We expected the compiled application to perform worse in our micro benchmarks as a result of this design.

Having built the plugin, we could apply our devised methodology, and compare the compiled React implementation to React, Svelte, and vanilla JavaScript implementations. The results of the benchmarks were promising, with our implementation outperforming React in some categories, despite the lack of focus on performance optimisations and a naive approach to component rerendering. However, it did not perform as well as Svelte. Furthermore, we were unable to run the macrobenchmarks on the compiled React applications because of a type checking error in the state management library, Redux. The compiled application failed this type check at runtime since all React components had been converted to the com-

ponent type. In order to run the macrobenchmarks, we implemented a new version of the React application that did not use Redux for state management. However, this work was performed between the report hand-in and the examination, meaning it is not included in the 9th semester project [1]. Running the macrobenchmarks on the compiled React application showed performance improvements compared to the original React application. We suspect that these improvements are primarily caused by the reduced size of the application similar to Svelte.

1.2 Project Objective

After the completion of [1], we were left with a number of avenues to pursue, were we to continue working on that project. As stated in section 1.1, we had created a methodology for comparing front-end frameworks and built a Babel plugin, which allowed us to compile React applications to vanilla JavaScript. Therefore, in order to gain a better understanding on which aspects of the previous report to pursue, we created a mindmap, which can be seen in fig. 1.1, and listed the various branches of the previous project, we could continue to work on and extend.

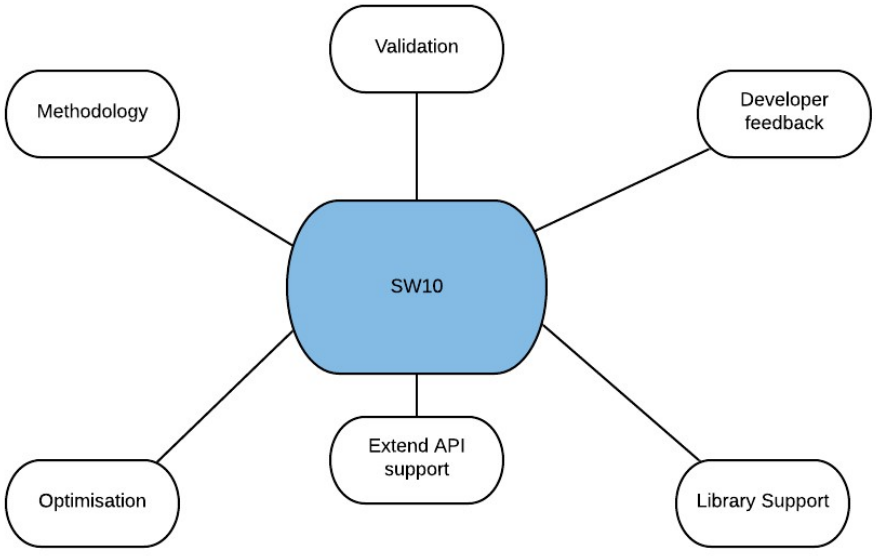


Figure 1.1: Project Mindmap.

There are six branches we can pursue:

- **Methodology:** Continuing the progress made on developing a methodology for benchmarking front-end frameworks. Extending the capabilites of Krausest either in the form of new implementations or benchmarks.
- **Optimisation of DOM operations:** Further development of the PoC from [1] by extending it to improve performance.
- **Extend API support:** Extend the amount of functionality the PoC can compile with, for instance, Hooks and other unsupported React features.
- **Library support:** Add support for commonly used libraries such as Mobx and Redux, thus, allowing us to run the macrobenchmarks from [1] on the compiled applications.
- **Validation:** Introduce a validation tool which verifies the HTML output of the compiled application by comparing it to the original applications HTML.
- **Developer feedback:** Build a UI for displaying error messages to users during compilation.

We consider all branches to have a place in a final product, however, we do not have the resources to work on all of them and make significant progress. Instead, we will limit the number of branches to the most essential, at this time, and work on those. Of the six branches, we have decided to postpone *Extend API support*, *Library support*, *developer feedback*, and *validation* to future work. The main reason for doing this, is that we assess that these branches add the least value to the project at this stage in development. We will return to these branches and describe our thoughts surrounding them in section 6.3. The remainder of this section will be dedicated to describing the two branches we have chosen to pursue in greater detail as well as the reasons for choosing to continue working on them.

We have decided to focus on *methodology* and *optimisation of DOM operations*. The primary reason for working on [1] and this project comes down to our passion for developing our PoC compiler. However, in order to properly evaluate our progress on the PoC, we require a methodology to properly compare it to other frameworks. While we would like to focus 100% of our attention on the further development of the PoC, we cannot justify doing so given our criticism of our methodology in [1]. In this project, we will take a new approach to the methodology, as we found during [1], that our split attention between the PoC and methodology, did not result in a satisfactory methodology. Instead of attempting to extend our methodology beyond [1], we have elected to cut the macrobenchmarks, in part due to the lack of support for libraries such as Mobx and Redux, as well as, the conclusion that the results they provided did not give much insight into the performance differences of frameworks. Furthermore, we assess that our methodology is not something we can make

a great deal of progress on in short time, and, as stated, our focus will primarily be on the compiler. Therefore, we will continue to use our existing methodology but attempt to highlight some of the inherent biases within Krausest and create implementations or benchmarks that address these.

The PoC from [1] showed a lot of promise in terms of results, however, the main drawback of it was the inability to reuse DOM nodes. In its current iteration, when, for instance, swapping two DOM nodes, the PoC will remove all DOM nodes, and then recreate them again with the necessary changes. Other frameworks will simply swap the two DOM nodes. This resulted in some impressive results in benchmarks such as "create rows", however, when it came to benchmarks where it was more efficient to reuse DOM nodes, it fell short. This project will focus on implementing strategies and features that allow the proof of concept to reuse DOM nodes, such that we can gain better performance in our microbenchmarks. Futhermore, we want to use Svelte as an inspiration to replace the reconciliation process from React, with a form of dependency tracking at compile time to optimise the DOM operations generated by our compiler.

1.2.1 Project structure

We will structure our report as follows:

1. A chapter dedicated to methodology that describes how we will address some of the issues in [1].
2. A series of iterations, each comprising a chapter, which describe a performance enhancing feature added to the proof of concept. Each chapter will consist of a design section, an implementation section, and a results section that determines next steps and the impact of the feature(s) that was implemeted.
3. A chapter dedicated to discussion of the the project, our choices, and potential future work.

We define the problem definition for this project as: *To what degree can we improve the performance of the PoC in the microbenchmarks by optimising the its output?*

Chapter 2

Methodology

In this chapter we look at extending our microbenchmarks with addtional implementations, which tests other aspects of the frameworks than those tested by Krausest.

2.1 Methodology Improvements

In section 1.2, we wrote about how we chose to shelve our macrobenchmarks, seeing as they made use of 3rd party libraries, something we do not support. Instead, we decided to focus on our microbenchmarks and improve upon them. One of the main issues we had with the existing microbenchmarks was the inherent bias in some of the benchmarks. Specifically, we want to investigate the impact of how DOM nodes are generated and the effect of keyed/non-keyed modes. We want to investigate how the frameworks compare when we remove keyed and non-keyed modes from the equation.

2.1.1 Keyed and non-keyed mode

Keyed/non-keyed mode refers to whether there is a binding between data and DOM nodes when dynamically generating DOM nodes at runtime. In keyed mode, a unique identifier is used to associate a given data point to a specific DOM node. In non-keyed mode DOM no key is used. React uses keyed mode, and will throw a warning if a developer does not specify a key. It is possible to use the index as a key, if you do not have a unique identifier in your data, however, this essentially puts it into non-keyed mode. If you use the index as the key when mapping over data, and that data is changed by either adding or removing items,

the order of the data is changed, thus, referencing an index will not provide the same data as before the change. Using an identifier as a key, on the other hand, will always yield the same data as there is a direct mapping between the DOM node and the key. If the keyed mode finds an element with a key that does not exist in the DOM, it creates a new DOM node and inserts it into the DOM. Similarly, if a key which currently exists in the DOM is not present in the new data array, it is removed from the DOM.

In a non-keyed implementation, we keep track of the DOM nodes currently in the DOM. During a rerender, we iterate through each DOM node and update its value based on the element in the data entry with the same index, thus overwriting the data previously associated with that DOM node. This means that no DOM nodes are moved, as is the case in keyed mode. If there is more DOM nodes than elements in the data array, we can remove the unused DOM nodes. New DOM nodes are created if there are too few.

2.1.2 Pre-generated rows

The Krausest tool only benchmarks the keyed/non-keyed implementation of a framework. The Krausest tool benchmarks are, therefore, not representative of the performance of a framework as a whole. The benchmarks do not measure how the frameworks handle implementations that do not utilse a keyed/non-keyed mode. Rather, Krausest measures how fast the frameworks are at displaying data as rows in a table where each row consists of 9 DOM nodes. Additionally, the frameworks dynamically generate DOM nodes based on state using either a keyed or non-keyed mode implementation.

In order to benchmark frameworks without keyed/non-keyed mode, we create a new type of implementation called Pre-generated rows. These implementations are designed to force the frameworks to not use their keyed/non-keyed implementation. This is done by hardcoding each row and its data into the source file instead of generating it dynamically. The code in Code 4 shows what it would look like in React.

We do not run all benchmarks in the Krausest tool on the pre-generated rows implementations, since they depend on different numbers of rows. Therefore, it is not possible to have all the functionality work simultaneously since the number of rows is hardcoded and, thus, we have to alter the source code of the implementation in order to get specific benchmarks to work. We have elected to focus on "create rows", "partial update", and "swap rows", as these are benchmarks where we suspect keyed/non-keyed implementations have a significant impact on performance. "Partial update" updates every 10th row and the "swap rows" benchmark updates 2 rows. Using keyed/non-keyed mode allows the frameworks to only

```
// Keyed mode
{this.state.data.map(el => <div key={el.id}>{el.value}</div>)}

// hardcoded
<div>{this.state.data[0].value}</div>
<div>{this.state.data[1].value}</div>
<div>{this.state.data[2].value}</div>
<div>{this.state.data[3].value}</div>
```

Code 4: Example of dynamically generating DOM nodes and hardcoding them in React

update the affected rows in each benchmarks, rather than updating all 1.000 rows.

The example in Code 4 use a div element in each row. If we wanted to change this to another HTML structure, we would need to rewrite every line to the new structure. To help us quickly generate these pre-generated rows implementation, we create a tool to assist us.

Pre-generated rows tool

The tool is comprised of 3 main parts — (1) It accepts an input HTML file containing the HTML tree structure we want to hardcode into the implementation. (2) a modified React directory where the implementation has been altered to allow for hardcoded values. (3) a directory builder which clones the modified React directory, takes the input HTML file and generates the necessary hardcoded data, and inserts it into the render function of the implementation.

In order to display the data from the state of the React component, we need to add placeholders in the HTML input file. An example of this is shown in Code 5.

```
// HTML input file
<div>VALUE</div>

// Translate to
<div>{this.state.data[0].value}</div>
<div>{this.state.data[1].value}</div>
<div>{this.state.data[2].value}</div>
<div>{this.state.data[3].value}</div>
```

Code 5: Example of using placeholder values in the HTML input file to access the data stored in the state of the React component.

In addition to replacing the placeholders, the tool will also duplicate the HTML input with an

increasing index based on a "count" value provided as an argument. Some of the benchmarks in the Krausest tool use 1.000 rows while others use 10.000. We can use the tool to generate multiple implementation with different row counts to support the different benchmarks in the Krausest tool. However, the tool does not work for all benchmarks, since we have not been able to implement all the required features without generating rows dynamically.

Because we have implemented the tool by modifying the original React directory, we can continue to use the webdriver in Krausest to build and run the implementations. Furthermore, this functionality is extendable to Svelte and vanillaJS by creating a new directory containing the framework specific code. This enables us to generate pre-generated rows implementations for other frameworks as well.

Chapter 3

Iteration 1

3.1 Design

For this iteration, we focus on a new compiler named *React-compiler* based on our PoC [1]. This version of the compiler will aim to improve the performance of the compiled application compared to the PoC. Additionally, we implement the compiler as a stand-alone JavaScript application as opposed to a Babel plugin which was the case for the PoC. This decision is discussed in detail in section 6.2.

The goals for this iteration are:

- Design and implement a new compiler structure
- Improve performance of the compiled applications compared to the PoC

We introduce a React component in Code 6. We utilise this component throughout this section in our compilation examples. The reader may find it useful to keep the example handy while reading this section.

3.1.1 State of the PoC

The PoC compiles React components by [1]:

- Transforming function components to class components
- Transforming JSX into JavaScript objects

```
class Main extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0,
            data: [{value: 1}, {value: 2}]
        };
    }

    increment() {
        this.setState({
            count: this.state.count + 1
        });
    }

    render() {
        return (
            <div className={this.state.count}>
                <div>Testing Static</div>
                <Container text="Testing Functional">
                    <Header title="Testing Variables"/>
                </Container>
                <div>
                    <div>{this.props.value}</div>
                    {this.state.data.map(el => <div><div>{el.value}</div></div>)}
                    <div className="box">{this.state.count}</div>
                    <input type="button" value="Increment" onClick={() => this.increment()}/>
                </div>
            </div>
        );
    }
}
```

Code 6: A React class component example.

- Using a helper function, `createElement`, to create DOM nodes from the objects at runtime

A JSX expression always has a single root node that all other JSX elements are children of [5]. The PoC utilises this heuristic during rerendering [1]. The PoC removes the parent node from the DOM when a component needs to be rerendered. Removing a parent node also removes its children from the DOM. As a result, all DOM nodes have to be recreated, which is done by calling the render method of the component [1]. Rerendering a component in the compiled application is, therefore, slower compared to the original React application in most cases, as shown in the results of [1]. The PoC compiled version of the component seen in Code 6, can be seen in Code 7.

```
class Main extends Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0,
            data: [{ value: 1 }, { value: 2 }]
        };
    }

    increment() {
        this.setState({
        c   ount: this.state.count + 1
        });
    }

    render(root) {
        let clothes_slipped_farmer = createElement({
            type: "div",
            parent: root,
            className: this.state.count,
            children: [{
                type: "div",
                children: [{
                type: "text",
                    value: "Testing Static"
                }]
            }, {
                id: "believed_beside_process",
                type: "component",
                componentName: "Container",
                props: {
                text: "Testing Functional",
```

```
33                  },
34                  init: (props, parent) => new Container(props).init(parent)
35              }, {
36                  type: "div",
37                  children: [{
38                      type: "div",
39                      children: [{
40                          type: "propAccess",
41                          expression: this.props.value
42                      }]
43                  }, {
44                  type: "expression",
45                  init: parent => this.state.data.map(el => function () {
46                      let court_upward_so = createElement({
47                      type: "div",
48                      parent: parent,
49                      children: [{
50                          type: "div",
51                          attributes: [],
52                          eventListeners: [],
53                          innerHTML: el.value,
54                          children: []
55                      }]
56                      });
57                  }())
58                  }
59              ]
60          });
61          this.container = clothes_slipped_farmer;
62      }
63  }
```

Code 7: The code generated by the PoC when compiling the component seen in Code 6. The JSX inside the render method has been replaced with an object representation. Some of the elements have been removed for brevity.

The `createElement` function uses the `document.createElement` JavaScript API for creating DOM nodes based on objects that represent HTML elements, such as divs [1]. Components are instantiated by calling the `init` function, as seen on line 34. This creates a new instance of the class, which then creates its own DOM nodes internally. JSXExpressions can contain any valid JavaScript expression and the returned value, if any, is rendered into the DOM [5]. JSXExpressions can, therefore, contain JSX inside, which is the case in the React component in Code 6 on line 25. The PoC handles this by converting the JSX to `createElement` calls similar to components [1]. The PoC does perform limited analysis on JSXExpressions at compile time to be able to handle certain cases. A JSXExpression has

three different object representations in the PoC:

- Props access: The expression matches the expression `this.props.p` where p is a valid property name.
- Children render: The expression matches the expression `this.props.children`.
- Expression: Any other expression not matched by the above.

This distinction is made so that we can handle the first 2 cases differently. Components can be assigned to a prop and `this.props.children` [5]. When a component is used in JSX, it can contain nested JSX elements as well, as seen on line 21 in Code 6 where the `Container` component contains a single child, the `Header` component. These children are accessed inside the component using `this.props.children`. Because the children can either be HTML elements or other components, we must look at the type at runtime, in order to properly create DOM nodes. This is done inside the `createElement` function. All other JSXExpressions, including those that contain JSX like the expression on line 25 in Code 6, are treated as a generic JavaScript expression by wrapping them in a function as seen on line 45 in Code 7. Wrapping an expression in a function means we can treat all JSXExpressions as a function, making it easier to generate the JavaScript code compared to handling each type of JSXExpression individually. Technically, a property in the state of the component could be a JSXElement, however, this was not used in the applications in the benchmarks, so it is not supported in the PoC [1]. We would need to add some form of typechecking at runtime to support it, since we would need to handle it differently than a regular JavaScript value.

The PoC stores the root node in `this.container` property on the component, so it can be removed during rerender [1]. The implementation does not need to keep track of all the DOM nodes that it creates, since they will never be updated before being removed. In order to support DOM node reuse, it must keep track of which DOM nodes need to be updated every rerender [1].

3.1.2 Tracking and updating DOM nodes

There are several different approaches to tracking and updating DOM nodes as we discuss in the following sections. The challenge is to access a specific DOM node in the DOM tree and update its attributes and contents during a rerender. Some DOM nodes are static, an example of this can be seen in Code 6 on line 19.

The JavaScript API for creating DOM elements, `document.createElement`, returns a reference to the new DOM node [14]. This reference can be used to manipulate the DOM node. JavaScript also has API functions for searching for DOM nodes in the DOM of a web page [14]. `document.getElementById` can be used to find a DOM node based on its Id attribute, provided one exists. Ids are assumed to be unique, so only a single node is

returned. Functions also exist for finding DOM nodes based on their class attribute and their tag name (HTML element type) [14]. Front-end frameworks like React and Svelte use this DOM API internally to manipulate the DOM [15, 16]. In the following sections, we look at how these frameworks translate their components into DOM operations.

3.1.3 Reusing DOM nodes in React

React uses a virtual DOM implementation for updating DOM nodes as mentioned in section 1.1.1. The entire process is called reconciliation [4]. The reconciliation implementation is generalised to any DOM-like tree structure [17]. This allows the code to be reused in different environments, known as host environments in React [17]. One such environment is the DOM in a browser. React applications must import the ReactDOM library in order to support the browser host environment. The ReactDOM library contains a react renderer for the browser environment [17]. A react renderer serves as an interpreter of the generic DOM operations generated by the reconciliation process for a specific host environment. For instance, the `createInstance` operation is translated to a `document.createElement` call in the ReactDOM renderer [17]. Other renderers also exist. The developers at Facebook maintain React Native, which is a renderer for native Android and iOS apps [18]. Third party renderers have also been created for other environments such as 3D graphics with WebGL [19]. The `createInstance` operation supports a return value. Whatever value is returned will be stored internally and passed around to other operations such as `prepareUpdate` [17]. ReactDOM renderer returns the reference to the created DOM nodes so it can be used in other operations. This approach allows React to reuse DOM nodes during rerender, since it keeps a reference to the DOM node that the virtual DOM element represents [17].

3.1.4 Reusing DOM nodes in Svelte

Since Svelte does not use a virtual DOM, it takes a different, but somewhat similar approach to keep references to DOM nodes. When Svelte compiles an application, it generates a `create_fragment` function for each component [16]. The `create_fragment` function for the example component in Code 6 can be seen in Code 8 [1].

```
1 function create_fragment(ctx) {
2     let div4;
3     let div0;
4     let t1;
5     // other declarations removed for brevity
6     let button;
```

[1] Since the example component is written in React, we have created an equivalent Svelte version which the compiled code example is based on.

```
7       let current;
8       let dispose;
9
10      const container = new Container({
11              props: {
12                  text: "Testing Functional",
13                  $$slots: { default: [create_default_slot] },
14                  $$scope: { ctx }
15              }
16          });
17
18      return {
19          c() {
20              div4 = element("div");
21              div0 = element("div");
22              div0.textContent = "Testing Static";
23              t1 = space();
24              create_component(container.$$.fragment);
25              // other elements removed for brevity
26              button = element("button");
27              button.textContent = "Increment";
28              attr(div2, "classname", "box");
29              attr(div4, "classname", /*count*/ ctx[0]);
30          },
31          m(target, anchor, remount) {
32              insert(target, div4, anchor);
33              append(div4, t1);
34              mount_component(container, div4, null);
35              // other appends removed for brevity
36              current = true;
37              if (remount) dispose();
38              dispose = listen(button, "click", /*handleClick*/ ctx[2]);
39          },
40          p(ctx, [dirty]) {/* Contents remove for brevity */},
41          i(local) {/* Contents remove for brevity */},
42          o(local) {/* Contents remove for brevity */},
43          d(detaching) {/* Contents remove for brevity */}
44      };
45  }
```

Code 8: Code output from the Svelte compiler for an equivalent component to Code 6. Some code has been remove for brevity. The Svelte compiler shortens some function names, for instance, *created* becomes *c* and *mount* becomes *m*.

The purpose of the `create_fragment` function is to encapsulate the DOM operations needed for a given Svelte component [16]. Svelte determines at compile time, which DOM

nodes are needed and how they need to be changed during a rerender. The first lines in Code 8 declare local variables to contain references to each DOM node that is created. Since they are declared in the scope of the function, they can be accessed from any nested function. The `create_fragment` function returns an object, which serves as an interface for creating, mounting, updating and deleting the DOM nodes [16]. It effectively hides the specific DOM operations behind a set of generalized operations similar to React. However, whereas the React renderer interface operates on a single DOM node, Svelte operates on all DOM nodes for the given component. Furthermore, Svelte uses its own functions for manipulating DOM nodes. For instance, the `element` function calls `document.createElement` internally.

3.1.5 Reusing DOM nodes in Solid

Both React and Svelte create each DOM node individually, however, it is possible to create a tree of DOM nodes by using an HTML string as seen in Code 9.

```
1 const parent = document.createElement('div');
2 parent.innerHTML = '<div><div>Hello World</div></div>';
```

Code 9: Using a HTML string to create multiple DOM nodes in a single operation.

In Code 9 we use `parent` as a placeholder to contain the DOM nodes created on line 2 by setting the `innerHTML` to the HTML string. This technique is used by another front-end framework named Solid [20]. Solid is best described as a combination of principles from both React and Svelte. It is written in JavaScript files and uses the JSX syntax like React. Similar to Svelte, it uses a compiler, or rather a Babel plugin, to generate DOM operations instead of using a virtual DOM. Solid performs better than both React and Svelte according to the Krausest tool and achieves its performance by using HTML strings to create all DOM nodes for a component in a single operation as seen in Code 10 [20, 21].

On line 1 in Code 10, the `template` function is called with an HTML string representing the DOM nodes for the component. This function is provided by Solid but creates an HTML template element internally [20]. A template element and its contents, unlike other HTML elements, is not visible in the browser [22]. The template element is used to contain the DOM nodes created by the HTML string in Solid [20]. The DOM nodes are then cloned on line 4 and inserted into the DOM on line 11. Cloning a DOM node and its children is faster than inserting the HTML string directly [20]. Solid also reuses DOM nodes to improve performance. However, references are not returned for all the DOM nodes in the template function call, since the nodes are created with the HTML string in a single operation. The `cloneNode` call on line 4 does return a reference, but only to the outer-

```
1  const _tmpl$ = template(`<div><div><div>Hello </div></div></div>`);
2
3  const HelloMessage = props => {
4    const _el$ = _tmpl$.cloneNode(true).
5          _el$2 = _el$.firstChild,
6          _el$3 = _el$2.nextSibling;
7    insert(_el$3, () => props.name, null);
8    return _el$;
9  };
10
11 render(
12   () => createComponent(HelloMessage, { name: "World" }),
13   document.getElementById("hello-example")
14 );
```

Code 10: Compiled code of a component in Solid using HTML string and template to insert multiple DOM nodes in a single operation.

most element, meaning Solid must obtain references through other means. This is achieved by calling `firstChild` and `nextSibling` as seen on lines 5-6 which is a part of the JavaScript DOM API [23].

3.1.6 Reusing DOM nodes in *React-compiler*

In this section, we focus on the design challenges related to the different approaches for DOM reuse introduced above.

The Solid approach is the best performing in the Krausets benchmarks [21]. It is, therefore, the most desirable for us to implement. However, the Solid approach does complicate the code generation process. We need to generate the HTML string based on the JSX of a component. We can utilize the `DocumentFragment` object in JavaScript, which effectively enables us to build an in-memory DOM representation of the HTML in the JSX [24]. We could then turn the DOM into a string and use it in a template element similar to Solid. However, not everything can be represented in an HTML string, such as event listeners and components. Any `JSXExpression` that is not a constant must be evaluated outside of the HTML string. The result is missing nodes in the DOM generated by the HTML string, since the dynamic nodes have to be added afterwards. We, therefore, have to keep track of the locations of the missing nodes. This is not a problem in the Svelte approach because each node is inserted in order. One potential solution is to use the HTML slot element to indicate a placeholder for another node, which could then be replaced after insertion [25]. The HTML slot element is not rendered in the browser unless it contains DOM nodes, which enable us to use it as a placehoder node [25]. To obtain a reference to the node, we would need to traverse the DOM nodes with `firstChild` and `nextSibling` the same way that Solid

does.
We must, therefore, decide if it is worth implementing a similar approach to Solid at the cost of extra development time, or if we should use the Svelte approach as inspiration. To determine which approach to implement, we compare the performance of `document.createElement` and HTML strings using templates. For comparison, we use a website called JSPerf [26]. The JSPerf website lets us create JavaScript tests and measure the number of operations each test case performs every second [26]. We create four test cases. Two of the test cases create a single div 10.000 times using either `document.createElement` or an HTML template. We call these the small test cases because we use them to represent a single DOM insertion. The small tests cases should help to identify the overhead associated with both approaches on a single element. The other 2 test cases create 100 divs 10.000 times and are denoted with the name large. The idea is to compare the performance difference for both small and large DOM trees. The results can be seen in table 3.1 and the code is available in [27].

	Small	Large
CreateElement	153	1,71
Template	107	4,8

Table 3.1: Results of the JSPerf performance comparison. Measured in operations per second.

The results show that `document.createElement` is approximately 50% faster than HTML templating using `cloneNode` for the small test case. However, `document.createElement` is more than twice as slow in the large test case compared to HTML strings. The results indicate that there is performance to be gained by using the Solid approach, but it is not more performant in all cases. We decide to move forward with the Svelte approach for this iteration, because the implementation performs well and requires less development time, enabling us to iterate faster in the project.

3.1.7 Designing *React-compiler*

With *React-compiler* we move from a Babel plugin to a multi-pass standalone compiler, which we detail in section 3.2. We use the Svelte approach as inspiration to replace the PoC `createElement` helper function, effectively generating the DOM operations directly at compile time rather than deferring it to runtime. It is desirable to generate the DOM operations at compile time, because it reduces the amount of work performed at runtime, which improves performance.
The PoC modifies the `render` method of the component to call the `createElement` function, which is responsible for inserting the DOM nodes into the DOM. DOM nodes are

removed on every rerender so the code in the `render` method does not need to take any existing DOM nodes into account. This does not work when reusing DOM nodes since the creation and insertion operations would be performed on every rerender, which would create duplicate DOM nodes in the DOM. Instead, we must split the work in the `render` method into two new methods: `mount` and `update`. The `mount` method contains all creation and insertion operations and the `update` method contains all the modification operations. Each DOM node will be assigned to a property on the component class instance with a unique name, such that it can be referenced in all methods inside the component. Furthermore, we need to generate the DOM operations for unmounting each DOM node, now that we no longer remove them each rerender as discussed in section 3.1.1. The `mount` method is called once when the component is instantiated. The `update` method is called on every rerender instead of the `render` method.
The next code snippets show how JSXElements are transformed into DOM operations.
The code in Code 11 shows the DOM operations for the first two JSXElements in the React component from Code 6.

```
mount() {
    this.el_1 = document.createElement('div');
    this.root.appendChild(this.el_1);
    this.el_2 = document.createElement('div');
    this.el_1.appendChild(this.el_2);
    this.el_3 = document.createTextNode('Testing Staic');
    this.el_2.appendChild(this.el_3);
    this.update();
}

update() {
    this.el_1.className = this.state.count;
}

unmount() {
    // el_1 is the parent of all other elements
    // so they will also be removed
    this.el_1.remove();
}
```

Code 11: DOM operations for the first 2 elements in the React component in Code 6.

The create and insert operations are generated in depth-first order of the JSXElements in the `render` method of the component. This ensures that the operations associated with a JSXElement, and its children, are executed before any JSXElements below it, such that the

DOM nodes appear in the correct order in the DOM. The `className` attribute is deferred to the `update` method since its value is a JSXExpression which should be evaluated on every rerender.
Components can be created by calling their constructor as seen on line 2 in Code 12. The component mounts itself internally by calling its own `mount` method. Component props are provided on instantiation and updated on every rerender as seen on lines 20-33.

The children prop represents the nested JSXElements inside the component, which can be seen on lines 4-14 in Code 12. The children prop is an array of functions. Each function contains the mount and update operations for the children, in this case, the component `Header`. The children of a component are related to a specific instance, which is why the PoC had to handle these children at runtime as discussed in section 3.1.1. In *React-compiler*, we pass a create function to the component instead, which means the component can treat each child as a function regardless of the JSXElement type. This has the same benefits as the function wrapping of JSXExpressions, meaning we can control each child with the same interface. Inside the `Container` component, we replace `this.props.children` with the code seen in Code 13. The value of the `this` keyword is bound to the parent component when the child function is called with the `bind` function in JavaScript [28].

The child function also adds a function to the `this.children` property as seen on line 11 in Code 12. This function is called in the `update` method, which removes the child from the DOM. The child is then recreated by calling the code in Code 13 again. This means that we currently do not reuse JSXElements passed as children. The children are passed as props, so they can potentially change every time the props of a component are updated. In order to reuse the children, we must implement a way to determine which children to add, update, and remove from the DOM during rerendering. We return to this in section 5.2.1.
The last case we look at is JSXExpressions, that contain JSX. One such case can be seen on line 25 in Code 6. The JSX inside can be thought of as a nested component "one off" with no name and class or function declaration. By treating the JSX as a component, we can generate the DOM operations for the JSX and replace it with the DOM operations. We assign the DOM node references to local variables instead of properties on the component, so that they can only be used inside the expression itself. The generated output for line 25 can be seen in Code 14.

The JSXExpression is wrapped in a function call, such that each expression can be treated as the same type as we discussed earlier. We push an unmount function to the class property `tempList`, which is an array, similarly to how children are handled. During rerendering, we call each unmount function in the `tempList` which removes the existing DOM nodes

```
mount() {
    this.component_8 = new Container({
        text: "Testing Functional",
        children: [function (parent) {
          this.component_9 = new Header({
            title: "Testing Variables"
          }).init(parent);
          this.component_9.receiveNewProps({
            title: "Testing Variables"
          });
          this.children.push(() => {
            this.component_9.unmount();
          });
        }]
    }).init(this.$el_6);
    this.update();
}

update() {
    this.component_8.receiveNewProps({
      text: "Testing Functional",
      children: [function (parent) {
        this.component_9 = new Header({
          title: "Testing Variables"
        }).init(parent);
        this.component_9.receiveNewProps({
          title: "Testing Variables"
        });
        this.children.push(() => {
          this.component_9.unmount();
        });
      }]
    });
}

unmount() {
    this.component_8.unmount();
}
```

Code 12: Mount and update methods for the Container and Header components found in Code 6.

```
this.props.children.forEach(child => child.bind(this)(this.el_1));
```

Code 13: The code replacing the this.props.children expression inside the Container component.

```
(parent => {
    let tempList = this.tempList;
    this.state.data.map(el => function () {
        let el_1 = document.createElement('div');
        parent.before(el_1);
        tempList.push(el_1);
        let el_2 = document.createElement('div');
        el_1.appendChild(el_2);
        el_2.innerHTML = el.value;
    }());
})(this.el_14);
```

Code 14: Code generation for JSXExpression containing JSXElements inside.

from the DOM. Calling the wrapped JSXExpression again will then create new DOM nodes based on the new data. We expect this to affect the results of the Krausest benchmarks since it heavily relies on keyed/non-keyed implementations. We expect that recreating DOM nodes on every rerender is less performant than reusing them in a keyed/non-keyed implementation. We return to this decision in section 5.1.4.

3.2 Implementation

The PoC was implemented as a plugin for Babel [1]. This resulted in limited control of the compilation process since the Babel plugin API restricts a plugin to only perform AST analysis and modification. The primary limitation in the PoC implementation was that it was limited to a single pass over the AST. Babel attempts to optimise the compilation time, by only traversing the AST once and calling the visitors of the plugins sequentially. The single-pass restriction led to more complex visitors in the PoC since the visitors now had to serve multiple purposes.

For *React-compiler*, we split the visitors from the PoC into separate passes as seen in fig. 3.1.

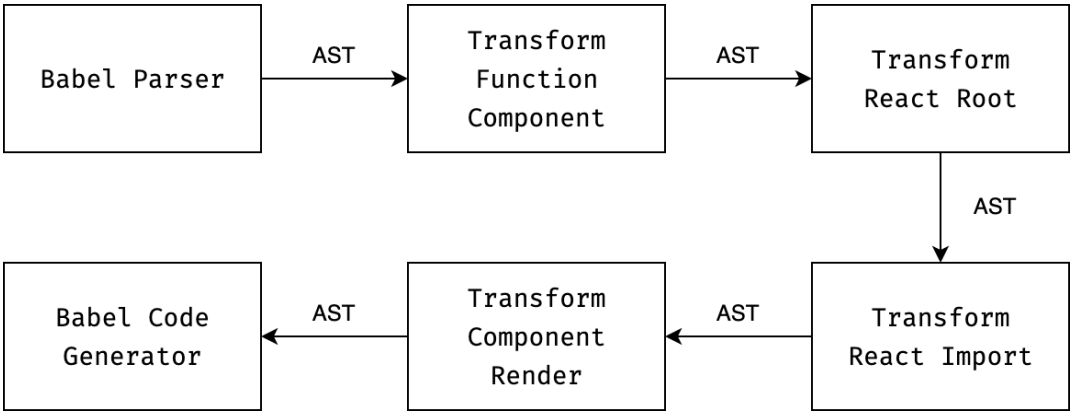


Figure 3.1: Compiler multipass overview.

1. We use the Babel parser library to parse the source code into the AST
2. The Transform Function Component visitor is responsible for finding React function components and transforming them into equivalent class components.
3. The ReactDOM.render, or as we refer to it, application root, is transformed by the Transform React Root visitor.
4. All JavaScript files containing a component, have an import statement for the React framework to use the React.Component class. These import statements are replaced with an import of our runtime library containing our Component class.
5. The final visitor is Transform Component Render, which contains the code for transforming the JSX inside the render method of a component into the DOM operations inside the mount, update and unmount methods.
6. The last step of the compiler is to call the Babel code generation library to generate the compiled code based on the AST.

The Transform Function Component, Transform React Root, and Transform React Import visitors use code discussed in [1]. Section 3.2.1 details the changes needed in the Transform Component Render visitor to implement the design dicussed in section 3.1.

3.2.1 Generating DOM operations for JSX

The specific operations that should be generated depend on the type that a JSXElement represents. For HTML elements such as divs, inputs, and images, we can use the DOM API in JavaScript. The `document.createElement` function does not support the creation of HTML elements with attributes, so they must be added in a separate call [29]. The code for generating DOM operations for HTML elements can be seen in Code 15.

We use a counter to generate the unique property names for each DOM node as seen on line

```
if (el.type === 'html') {
    const elVariable = `el_${idCounter++}`;
    const ast = templateStatement(
        `${declarePrefix}${elVariable} = document.createElement('${el.htmlType}')`
    )();
    actions.mountActions.push(ast);

    if (el.attributes) {
        el.attributes.forEach(atr =>  {
            const ast = templateStatement(`
                ${elVariablePrefix}${elVariable}.setAttribute('${atr.name}', VALUE)`
            )({VALUE: atr.value});

            // If the value is dependent on state or props we move it to
            // the update method instead of mount
            if (atr.isStatic) {
                actions.mountActions.push(ast);
            } else {
                actions.updateActions.push(ast);
            }
        });
    }

    const mountAst = templateStatement(
        `${parentVar}.appendChild(${elVariablePrefix}${elVariable})`
    )();
    actions.mountActions.push(mountAst);

    if (el.children) {
        el.children.forEach(
            child => DOMFactory(child, false, elVariable, component, actions, options)
        );
    }
}
```

Code 15: Partial code for HTML element section of the DOMFactory function.

2. The `templateStatement` function takes a JavaScript statement in the form of a string as input and generates the AST for that statement using the Babel template library. We use the term, action, for these generated JavaScript statements, because each statement represent an action peformed on the DOM. We use a JavaScript template string, so we can insert values of variables into the string. The `declarePrefix` variable, for instance, is used to control if the return value of `document.createElement` is stored in a local variable with the `let` keyword or as a class property using `this` as discussed in section 3.1.7. This approach lets us reuse the same code for both JSXElements and JSXExpressions that contain JSX (We use DOMExpressions as a shorthand for these JSXExpressions).
The code seen in Code 15 is part of a function called `DOMFactory`. This function is responsible for generating the DOM operations for all JSXElements based on their types. The `DOMFactory` function takes an object named `actions` as one of its parameters. This object contains 3 arrays of actions representing the actions related to the `mount`, `update` and `unmount` methods. The action generated on lines 3-5 is added to the `mountActions` array on line 6.
Each attribute is handled by generating an action using the `setAttribute` function on the DOM node stored in the variable. The `elVariablePrefix` variable is used to control the prefix for accessing the DOM node. If it is stored as a class property, it must be prefixed with the `this` keyword. `elVariablePrefix` is empty for local variables. The Babel template library supports AST injection into a template using a placeholder in all capital letters. In this case of the `templateStatement` on line 3 we declare the placeholder `VALUE`, which is then replaced with the AST stored in the object representing the attribute. We can, therefore, avoid converting the AST representing the value of an attribute into a JavaScript value, just to generate the AST for said value again. The action is added to the `updateActions` array if an attributes value is defined by a JSXExpression, since we have evaluated it on every rerender as discussed in section 3.1.7. If the value is defined as a string literal, we can add it to the `mount` methods action array, such that it is never evaluated again.
The children of the HTML element are handled on lines 29-33. Since each child is in itself a JSXElement, we can call the `DOMFactory` function recursively to handle each child. We pass in the actions object, so each DOM operation is added to the same actions array.
The code for handling components can be seen in Code 16. A component can be created by calling its constructor and the `init` method as seen on lines 28-30 in Code 16.

The children of a component are passed into the component as a prop. However, we must generate a function for each child which contains its `mount`, `update`, and `unmount` as discussed in section 3.1.7. This is done on lines 5-12 by calling the `DOMFactory` function with a new actions object, so that the actions generated are not added to actions of the parent component. The `templateExpression` function seen on line 14 is similar to the

```
if (el.type === 'component') {
    const componentVariable = `component_${idCounter++}`;
    if (el.children.length) {
        const childActionsList = el.children.map((child) => {
            const childActions = DOMFactory(
                child,
                true,
                'parent',
                el.componentName,
                actions,
                options
            );

            return templateExpression(`function(parent) {
                MOUNTACTIONS
                UPDATEACTIONS
                this.children.push(() => {UNMOUNTACTIONS});
            }`)({...childActions});
        });

        // Add the children functions as the actual prop
        el.props.push({
            name: 'children',
            value: t.arrayExpression(childActionsList)
        });
    }

    const mountAction = templateStatement(
        `${declarePrefix}${componentVariable} = new ${el.componentName}(PROPS).init(${parentVar});`
    )({PROPS: el.propsAst});
    actions.mountActions.push(mountAction);

    const updateAction = templateStatement(
        `${elVariablePrefix}${componentVariable}.receiveNewProps(PROPS);`
    )({PROPS: el.propsAst});
    actions.updateActions.push(updateAction);

    const unmountAction = templateStatement(
        `${elVariablePrefix}${componentVariable}.unmount();`
    )();
    actions.unmountActions.push(unmountAction);

}
```

Code 16: Partial code for component section of the DOMFactory function.

`templateStatement` function introduced earlier. The difference is in the type of the root AST node generated, which is an expression node rather than a statement node. JavaScript object properties cannot have a statement as a value.
The remaining JSXElement types, such as text and DOMExpressions, follow a similar approach, where an AST is generated using the `templateStatement` and `templateExpression` functions, which are added to the action arrays. The `mount`, `update` and `unmount` methods can be generated for the component once all JSXElements have been handled. The code for generating the mount method can be seen in Code 17.

```
1 const mountMethod = componentMountMethodTemplate({
2     RENDERCONTENT: renderContent,
3     ACTIONS: actions.mountActions
4 });
```

Code 17: Code for creating the mount method in a component using the actions generated by the DOMFactory function.

The `componentMountMethodTemplate` generates a class method AST containing the actions from the `mountActions` array. It also adds the contents of the `render` method at the beginning of the method. The `render` method of a React component often contains local variables, which are used in the JSXElements. These statements must be included to ensure that these variables are available to the actions inside the `mount` method as well. The `update` and `unmount` method are generated identically.

3.3 Results

This section outlines the results of the work done in iteration 1, as well as, presents what could be improved upon or changed in the following iteration.

3.3.1 Test setup

Since this project is a continuation of [1], the test setup will have similarities to that of [1], in order to properly compare previous and new results. However, as mentioned in section 1.2, we discarded macrobenchmarks and are, therefore, focusing on the microbenchmarks. To this end, we have made some extensions to the original microbenchmarks from [1], which were outlined in section 2.1, and are elaborated upon in section 3.3.2.
Since the completion of [1], the hardware, on which the benchmarks are run, has been upgraded. All tests are run on a desktop PC with the following specs:

- **OS:** Windows 10

- **CPU:** Ryzen 7 3700X
- **Memory/RAM:** 32gb 3600MHz DDR4
- **Motherboard:** Gigabyte Aorus Elite X570
- **GPU:** MSI Nvidia GTX 1070
- **Google Chrome:** 83
- **Nodejs:** 12.16

Additionally, we run the benchmarks on the same frameworks and versions as in [1]. They are provided in table 3.2.

Versions	Keyed	Non-keyed	Modified
Vanillajs	x	x	
Vanillajs1	x		
Svelte-v3.5.1	x	x	
Svelte-v3.12.1	x	x	x
React-v16.8.6	x	x	
React-v16.11.0	x	x	x
React-hooks-v16.8.6	x		
React-hooks-v16.11.0	x		x
React-redux-hooks-v16.8.6 + 7.1.0	x		
React-redux-hooks-v16.11.0 + 7.1.0	x		x
React-redux-v16.8.6 + 7.1.0	x		
React-redux-v16.11.0 + 7.1.0	x		x
React-mobX-v16.4.1 + 5.0.3	x		
React-mobX-v16.11.0 + 5.0.3	x		x

Table 3.2: Micro benchmark implementations. The *x* indicates that an implementation exists of that type for the given framework version.

Each version in table 3.2 can have a keyed, non-keyed, or modified implementation and, in some cases, some versions will have multiple. The keyed or non-keyed implementations differ in the method used for creating relationships between data and DOM nodes, which is described in section 2.1. Modified refers to whether the version of the framework has been updated to a newer version. Some implementations have received updates since [1], however, in order to be able to compare the results, we have elected not to bump the versions again. If we were to bump the versions, we would introduce an additional variable, besides the hardware change, that could impact results.
The benchmarks run on the implementations consist of the entire repertoire of benchmarks in the Krausest [12] tool, as was the case in [1]. Furthermore, we have extended the imple-

mentations to allow us to account for the ideas presented in section 2.1.
The objective for this iteration was to reuse DOM nodes to improve performance of the compiled React application. However, we did not implement DOM node reuse in our DOM-Expressions since we do not have a non-keyed/keyed mode implementation for this iteration. Therefore, we do not expect performance improvements in the Krausest microbenchmarks, as Krausest relies on DOMExpressions. On the other hand, we expect to see a performance improvement in the benchmarks, when comparing the PoC and *React-compiler*, since *React-compiler* reuses the DOM nodes. The [1] results show that reusing DOM nodes is more performant than recreating them on every rerender. Therefore, we expect to close the gap on React.

3.3.2 Pre-generated rows

In order to directly test the rendering times of the frameworks, when not using keyed or non-keyed mode, we have created a new implementation called pre-generated rows, as described in section 2.1. This implementation has hardcoded DOM nodes into the source code, as opposed to creating them dynamically. As it is React we are building a compiler for, we have elected to only benchmark a React implementation, in order to see the impact of keyed and non-keyed on the rendering process. While including an implementation for Svelte would provide additional information on the performance of Svelte, we argue that the results would not aid us in the development of *React-compiler*. The pre-generated rows implementation is almost identical in functionality to the React-v16.11.0 version, with the difference being that each DOM node is hardcoded into the `render` method of the component, rather than generating it based on component state. The pre-generated rows implementation removes the use of a keyed/non-keyed mode, which allows *React-compiler* to reuse DOM nodes and thus improve performance. The results can be seen in fig. 3.2.
The "create rows" benchmark in fig. 3.2 shows that the compiled version performans significantly better than the original React application. However, this result is misleading, because the compiled application inserts the rows on startup rather than by the click of a button as is the case of the React application. The compiled application is outperformed by React in both "partial update" and "swap rows". We expect the compiled application to perform on-par with the original React version, because we can utilise the DOM node reuse functionality of *React-compiler*. These results indicate that either the generated code is slow, or that something in the browser is slowing down the code. It is possible that the browser is overloaded by the more than 1.000 DOM operations performed by the `update` method generated by *React-compiler*. We assess that the time required to profile the performance of the browser is greater than the time required to investigate whether our code is the issue. We, therefore, want to eliminate our code as the problem, before investing more time into

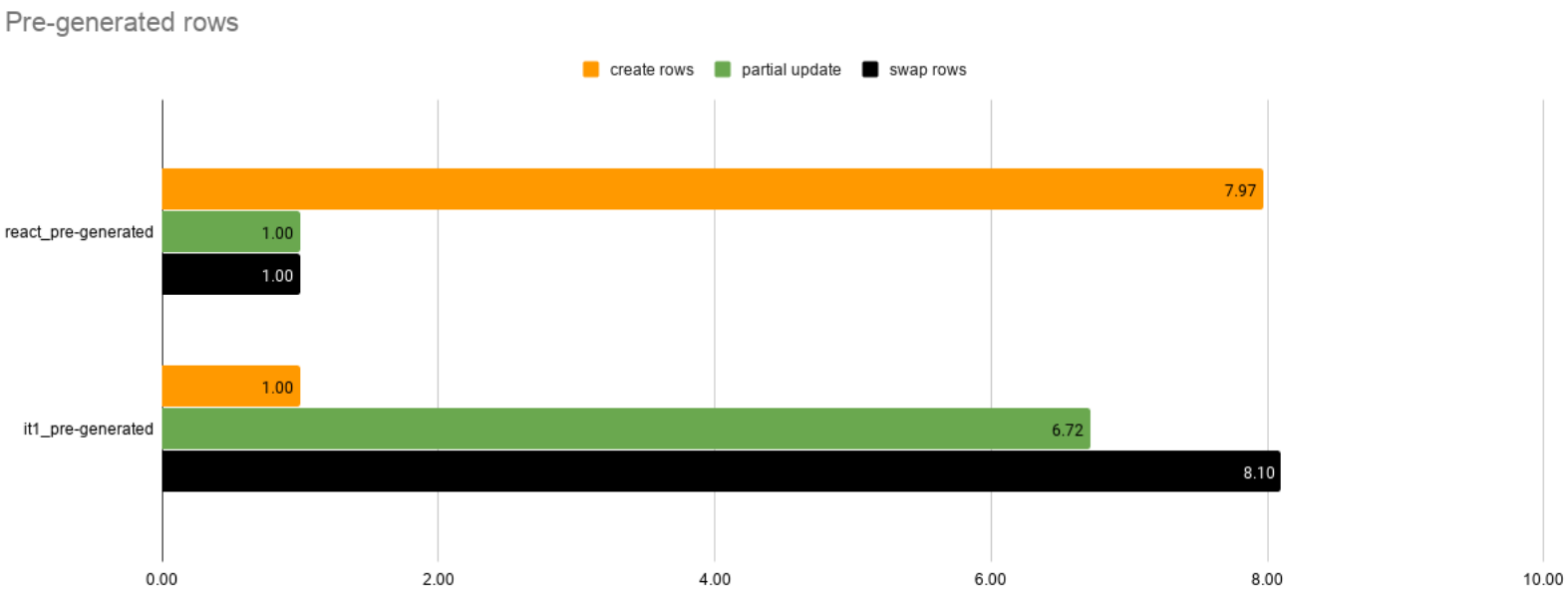


Figure 3.2: Pre-generated rows. The values indicate relative performance, where 1.0 is the fastest.

the behaviour of the browser. Before we can determine the cause of this issue, we must first investigate whether this issue is also present within our DOMExpressions, as they do not support DOM reuse.

3.3.3 Microbenchmarks

We are unable to run the benchmarks on the compiled output of *React-compiler* from this iteration. The reason for this is how the benchmarks are run on the frameworks. Krausest [12] makes use of XPath, which is a path expression that indicates the position of a DOM node by following a path or number of steps [30]. For instance, when the benchmarks are run, they navigate the webpage, produced by a framework, using the XPath. In the benchmark for "create 1000 rows", the webdriver, which emulates user behaviour, is directed to click the "create 1000 rows" button. Thereafter, it is instructed to verify that a table with 1000 elements has been created using the following XPath: `//tbody/tr[1000]/td[2]/a`. It starts from the initial table node, and then steps 1 level deeper, into the `<tbody>` element. Then it steps into one of the 1000 `<tr>` elements and so on. If it does not find an element, the benchmark fails. The problem with our implementation is our DOMExpression implementation. We use a HTML template element as the parent for DOMExpressions such that the DOMExpression elements are inserted into the correct place in the DOM. In the case of the microbenchmarks it inserts a template element as the first child of the table body (tbody) in the DOM. This invalidates the XPaths, meaning they do not locate the elements they are supposed to find and, therefore, fail. It is possible to change the XPaths within the benchmarks to properly navigate our output. Therefore, instead of attempting to fix the issue by compromising the integrity of our benchmarks, we have elected to postpone presenting the results of iteration 1 and address the issue in chapter 4. We argue that fixing the benchmarks to make our implementation work is not the correct course of action, but rather to fix our implementation such that the benchmarks can be run. Therefore, a goal of the next iteration must be to change the DOMExpression implementation, such that it no longer requires the template element, thus allowing XPaths to function as intended.
The XPath issue is only related to implementations using *React-compiler*, which means we can still present the results of running the microbenchmarks, on the new hardware as seen in appendix A for vanillaJS, Svelte, React and the PoC. The values are given in both actual values, the metric is indicated in the benchmark title, and relative performance to the best performing implementation in each benchmark. To summarise the results, the vanillaJS implementations outperform both Svelte and React. Svelte closely follows vanillaJS across the board, and React is the worst performer, of the three, in almost all cases and by a large margin (up to 40 times worse). One example is the "swap rows" benchmark shown in fig. 3.3, where all React implementations apart from the non-keyed versions perform worse than both

vanillaJS and Svelte. Furthermore, in the memory and startup categories, vanillaJS is also the best performer, followed by Svelte, with React in last place. However, the difference in performance is smaller within these categories, compared to the DOM operation category. In general, our PoC performed worse than vanillaJS, Svelte, and React in most categories, as was expected. The PoC does not reuse DOM nodes, meaning whenever a component changes state, it rerenders the entire component, as opposed to only the elements or values that changed [1]. This means that in benchmarks such as "swap rows", the PoC will remove all DOM nodes and recreate them, whereas, other implementations swap the DOM nodes associated with the two rows.

However, there are signs of performance gains in various categories/benchmarks. In the startup category, the PoC is the best performing implementation in the "script bootup" time benchmark, as well as, outperforming React in general in the other benchmarks of the same category. Furthermore, in the memory category, the PoC outperforms React and Svelte implementations in the benchmark "memory usage after creating 1000 rows", shown in fig. 3.4. In general, the results suggest that while the PoC did not see performance increases in terms of the speed at which it manipulates the DOM, it did improve on the memory consumption and startup metrics, when compared to the non-compiled versions of React. We attribute the performance improvement to the reduce total byte size of the compiled application. The React applications in the microbenchmarks, all take up more than 150KB compared to the 11KB of the PoC, which results in faster load times depending on how fast the browser can load the source code.

It is worth noting that the PoC performs worse compared to the results seen in [1]. The PoC outperformed all keyed React implementations in the "replace all rows" benchmark in [1], which is not the case in appendix A. We do not believe the change of hardware to have affected the relative performance of the different implementations. We use a newer version of Chrome in our microbenchmarks than in [1]. Contributors to the Krausest tool have also seen performance degradations caused by newer versions of the Chrome browser [31, 32], which leads us to believe that Chrome is the reason that some implementations perform better and others perform worse compared to [1]. Therefore, the results presented in appendix A, are the new baseline benchmarks for this project.

With all this in mind, the objective of chapter 4 will be to address the issues presented in section 3.3.3. Specifically, change the implementation of DOMExpressions such that we can run the microbenchmarks on the compiled application of iteration 1. Furthermore, we must return to the performance issues regarding pre-generated rows and verify whether DOMExpressions are affected.

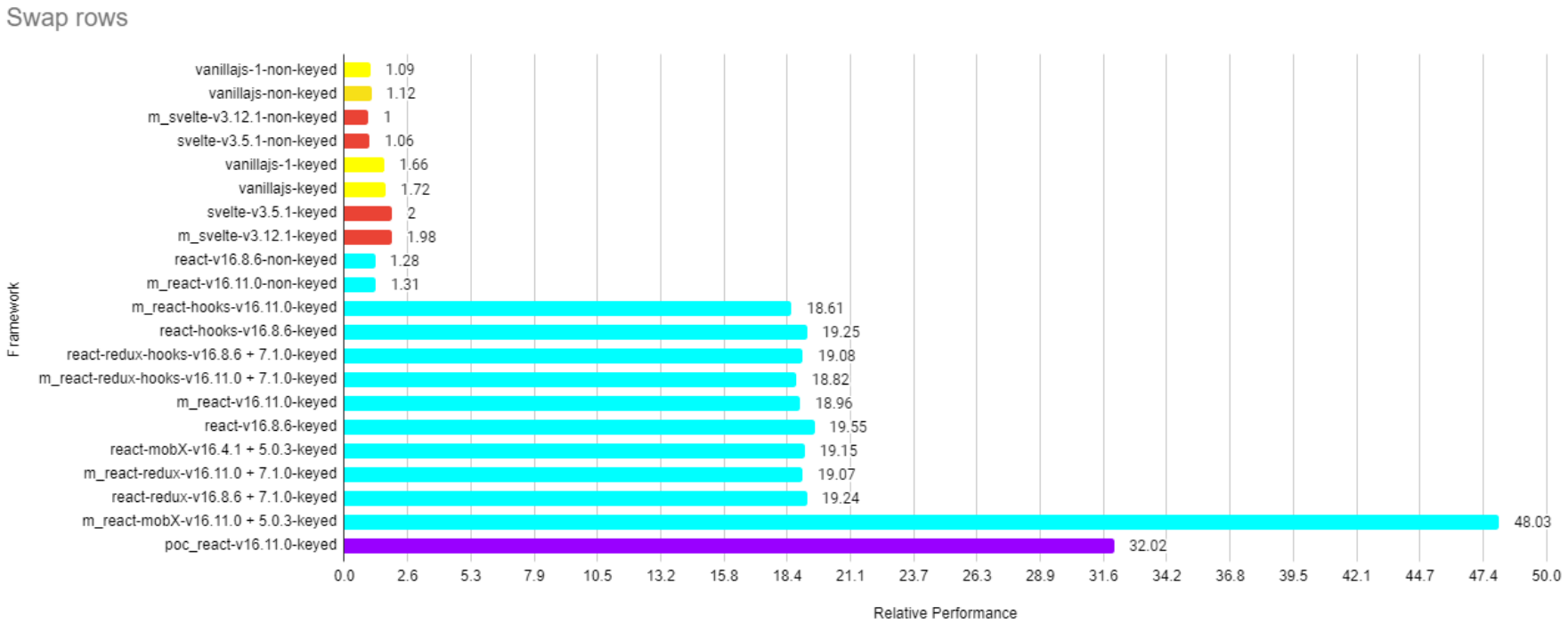


Figure 3.3: Swap rows benchmark in the DOM manipulation category of Krausest. Measured in relative performance. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

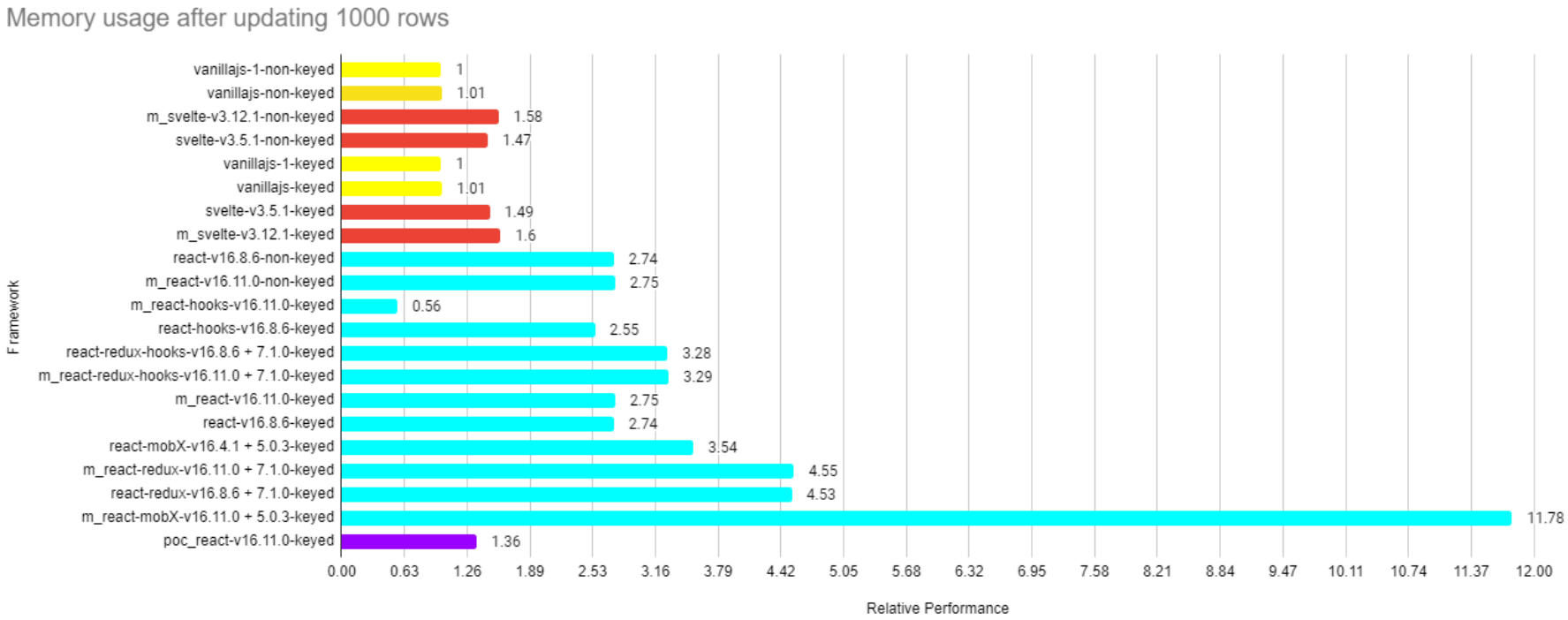


Figure 3.4: Memory usage after updating 1000 rows. Measured in relative performance. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

Chapter 4

Iteration 2

This chapter covers the design and implementation of the second iteration of *React-compiler*. This iteration focuses on improving upon the employed updating technique, and extending it to allow for updating specific parts of a component, instead of the whole component, when a rerender is triggered. Furthermore, we must design and implement a new approach to handling DOMExpressions as the microbenchmarks in section 3.3 showed the approach introduced in section 3.1.7 broke the Krausest tool used for the microbenchmarks
The objectives for this iteration are:

1. Design and implement a new update method, which only performs the DOM operations necessary, based on the change in component state or props.
2. Rework the DOMExpressions design and implementation so it works with the microbenchmarks

4.1 Design

The `update` method introduced in chapter 3 contains the actions for updating all dynamic parts of a component, which means that everything is updated on every rerender. However, it is unlikely that everything need to be updated every time a component is rerendered. Consider the React component from Iteration 1 in Code 6. The state of the component is updated when the button on line 27 is clicked, which triggers a rerender. The React reconciliation process determines which DOM nodes need to be updated as discussed in section 1.1.1. In the case of Code 6 only the expressions on line 18 and 26 needs to be updated. A compiled version using *React-compiler* from chapter 3 instead calls the `update` method on the component, resulting in unnecessary DOM operations. Essentially, we need to replicate the behavior of the reconciliation process in React, so we can determine which DOM operations must be performed when state and props change in a component.

We introduce a state diagram for a React component seen in fig. 4.1, which shows the different states a component can be in and the transitions between each of them. Of interest are the transitions from the `Mounted` state to the `shouldComponentUpdate` state. A component and the DOM are considered up-to-date when in the `Mounted` state. User interaction can then trigger a rerender by, for instance, clicking a button as seen in Code 6.

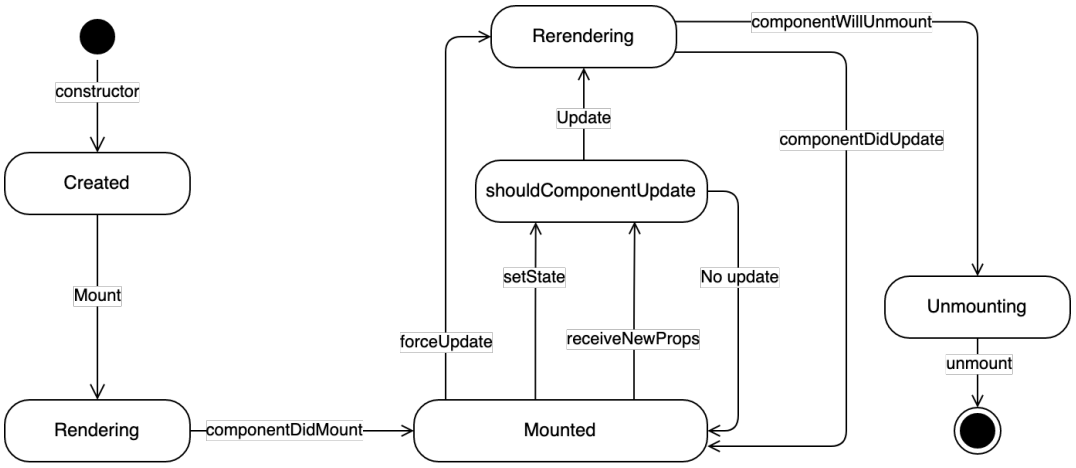


Figure 4.1: State diagram of the lifetime of a React component. The diagram is based on the offical documentation [7].

The `shouldComponentUpdate` state represents a React API method with the same name. Its return value determines whether a component is rerendered or not. By default, it always returns true, but it can be overwritten inside a component to change its behavior to block some or all rerenders of a component triggered by updating the state or props. `forceUpdate` circumvents the `shouldComponentUpdate` method and forces a component to rerender regardless of any changes to state or props. It is worth pointing out that only the state can be updated inside the component. Props are passed into a component from its parent as discussed in section 1.1.1. React does, therefore, not provide an API method for updating the props inside a component [7]. It is for this reason that the `receiveNewProps` transition does not represent a React API method, but rather the external update made by React. In the case of *React-compiler* we use `receiveNewProps` as the name of the method on the `Component` class to update props as was seen in section 3.1.7. Based on fig. 4.1 we can see that the only ways a component is updated is by calling `forceUpdate`, `setState`, and `receiveNewProps`.

To minimize the DOM operations performed on a rerender, we must generate new methods to replace the `update` method in a component. These methods should only contain DOM operations relevant for the state and props that have changed. Thus, we need to know which props and state have changed when a update is triggered. The `forceUpdate` method does

not modify the state or the props of a component, but contents of the original `render` method must be reevaluated since it may contain values used in the JSX. Props are always provided as a complete object, so we can use a single method to handle rerenders triggered by updating props. We designed the `receiveNewProps` method to receive props that are always provided as a complete object because this simplifies the `shouldComponentUpdate` API method which relies on the new and old props. We can use a single method to handle rerenders triggered by updating props as they are always updated together. In section 4.1.1 we discuss how to track JSX dependent on props and how to generate the `propsUpdate` method.
The first argument to `setState` is either a function generating the new state object or an object to be merged with the current state. We can determine which properties in the state object change for each call to `setState` in a component by looking at the argument provided to each `setState` invocation. Each call may modify a disjoint set of properties in the state object, meaning we need to create a dedicated method for every `setState` call because the affected JSXElements are likely to be different between each call. We discuss this in detail in section 4.2.1.
So we call a different method inside the component based on which API triggered the update:

- `forceUpdate` requires everything inside the `render` method to be reevaluated. We can use the existing `update` method from iteration 1 since it updates all DOM nodes in the component.
- We need a new method which only updates the DOM nodes which depend on the props. Since all props are updated together, we do not need to track which props have changed.
- Each `setState` method updates the component's state, but not all of the properties on the state object is updated by every call to `setState`. This means we need to generate an update method for each `setState`, which only contains DOM operations that depend on the parts of the state that has changed.

In the next sections we look at tracking props and state dependencies in the JSX.

4.1.1 Component props dependency tracking

We can inspect the JSX at compile time to find any uses of component props. We add information of props uses to the object representation of a JSXElement. Consider the JSX example in Code 18.

We see that the `className` attribute uses a value on the props object. However, it is worth recalling that each JSXElement can generate multiple actions. For instance, each attribute in

```
<div id="container" className={this.props.name}></div>
```

Code 18: Example of a JSXElement with multiple attributes where only 1 use component props.

the div element will be a separate action. We do not need to update all attributes when props change since only one of them uses the props value. To support this behavior, we add a new property to each attribute on the element object as seen in Code 19.

```
{
    name: attribute.name.name,
    value: attribute.value.expression,
    isStatic: false,
    useProps: true,
};
```

Code 19: Updated object representation for JSXElement attribute with useProps flag.

During action generation for each element, we can use these flags to determine if a given action should be added to the `propsUpdate` method. In section 3.2.1 we used an actions object with an array of actions for each method: `mount`, `update`, and `unmount`. We can extend this object with another array for the `propsUpdate` method. When the `receiveNewProps` method is called from the components parent, we can call the `propsUpdate` method instead of the `update` method to only perform the DOM operations, which are dependent on the props.
This approach has the possibility of resulting in different behavior of the compiled application compared to the original. Consider the JSX example in Code 20.

```
<div>{new Date.toISOString()}</div>
```

Code 20: Example of an expression which returns a new value every time a rerender is triggered.

The inner value of the div is an expression which calls the JavaScript `Date` object. The returned value is the current time of the day. This JSXElement is located inside the `render` method of a component in a React application. The expression is evaluated every time a rerender is triggered. However, it is only evaluated once during mounting in our design because the `update` method is not called again, now that we use the `propsUpdate and \cverb` setState| methods for updating during rerendering. The time displayed in the div in Code 20 will, therefore, never update, which is a problem. A similar situation can occur if an expression has side effects when invoked. We could attempt to analyse each expression for side effects, but we argue that this is outside the scope of this project and could be a thesis on its own, as proven by existing work on the topic [33, 34].

Furthermore, it is our opinion that writing JSXExpressions with side effects is a bad practice because it can lead to unintentional behavior because the developer is not in control of the rerendering. If the code in Code 20 is intended to display the current time, it would only work if the component is regularly update, effectively relying implicitly on React to trigger a rerender, which it does not do for side effects of JSXExpressions. Finally, the goal of *React-compiler* is to support the applications in our microbenchmarks, which do not have the problem described above. The `propsUpdate` method includes, like the other methods, the content of the original render method above the JSX, so it is possible to rewrite the example above to the code shown in Code 21 and it behaves indentical to the original React application.

```
function render() {
    const now = new Date.toISOString();
    return <div>{now}</div>
}
```

Code 21: Refactored code for correctly using date object in JavaScript without changing behavior when compiled.

4.1.2 Component state dependency tracking

Handling state updates in a component requires more work compared to props. We can use a similar approach to finding uses of state in the JSX as we do for props. However, we need to include additional information on the elements regarding which properties on the state object are used. Since a call to `setState` does not necessarily update all properties on the state object, we only want to perform the DOM operations affected by the state change. We can determine which properties on the state object changes in a `setState` call by looking at the first parameter to the function which is an object containing the changes to the state. The code in Code 22 shows the extended element object for handling state.

```
{
    name: attribute.name.name,
    value: attribute.value.expression,
    isStatic: false,
    useProps: true,
    useState: true,
    stateKeys: ['value']
};
```

Code 22: Updated object representation for JSXElement attribute with state information.

The `stateKeys` array on line 7 contains each property on the state object that is used in the JSXElement. With this information, we can determine during action generation if an action should be performed after a `setState` call. For it to work, we also need information about each `setState` call inside the component. We create a symbol table for component information, which we call the component table. This lets us store information each component in the application which can be used in the other parts of *React-compiler*.Each entry in the table has the following information:

- The name of the component
- An array of properties on the state object found in the component
- An array of `setState` method calls and the properties modified on the state object by the call. Each call is given a unique name to use during action generation.

During action generation, we look up the `setState` calls and generate a method on the component for each call. We can sort the actions into the correct method by comparing the property names in the `stateKeys` array of the element with those found in the `setState` entry of the component table. The details are discussed in more detail in section 4.2.1. Finally, we must replace the `setState` call with the new method generated to replace it.

These changes mean that we no longer call the `update` method during rerender. Instead, we call either the `propsUpdate` method or one of the `setState` methods, which limits the DOM operations to those which depend on the values that have changed.

4.1.3 Reworking DOMExpressions

In chapter 3 we used a template element as the parent provided as the first argmuent to the compiled DOMExpression function seen in Code 14. We found that the template element caused the XPath used in the Krausest tool's webdriver to break, so the benchmarks could not be completed. We need to rework the design so we do not need a template element in the DOM for DOMExpressions.

Furthermore, In Code 14 we used a component property `tempList` to track entries generated inside DOMExpressions. A single array worked because we always updated all DOMExpressions in the `update` method. However, now that we split updating into props and state methods, we can no longer guarantee that all DOMExpressions are updated on every rerender. Consider the example in Code 23 with two DOMExpressions.

The DOMExpression on line 2 is updated inside the `propsUpdate` method, where the other DOMExpression on line 3 is not. The `tempList` property is cleared when a DOMExpression is updated, so this results in missing DOM nodes from the second DOMExpression because they share the same array. We fix the problem by using an array for each DOM-

```
<div>
    {this.props.data.map(item => <div>{item}</div>)}
    {[1,2,3,4].map(item => <div>{item}</div>)}
</div>
```

Code 23: Example of two DOMExpressions using different data to iterate over. The compiled application would result in both using the same tempList to track DOM nodes.

Expression. We can move all of the code related to the DOMExpression inside the arrow function as seen in Code 24.

```
(parent => {
    if (this.tempList_el_20 && this.tempList_el_20.length) {
        this.tempList_el_20.forEach(el => {
            el(); // function call unmounts the DOM nodes in the element
        });
        this.tempList_el_20 = [];
    }

    let tempList = [];
    let previousEl = null;
    let useAppend = false;
    this.props.data.map(el => function () {
        let el_14 = document.createElement('div');

        if (useAppend) {
            previousEl ? previousEl.appendChild(el_14) : parent.appendChild(el_14);
        } else {
            previousEl ? previousEl.after(el_14) : parent.after(el_14);
        }
        el_14.innerHTML = el;
        tempList.push(() => el_14.remove());
        previousEl = el_14;
    }());
    this.tempList_el_20 = tempList;
})(this.el_13);
```

Code 24: New compiler output for DOMExpressions using the first DOMExpression from Code 23.

Lines 2-7 clears the existing DOM nodes from the DOM before the new ones are created. In iteration 1 we used a template element as a pointer for where the DOM nodes in a DOM-Expression should be inserted. We need to find another way to keep track of where to insert DOM nodes, since the template element was the reason we could not run the microbenchmarks. We can remove the template element if we keep track of the element that was in-

serted before the DOMExpression, since the first element in the DOMExpression should be inserted directly after the previous element. We introduce two new variables `previousEl` and `useAppend` on lines 10-11. The `previousEl` variable holds a reference to the DOM node that was inserted, so it can be used to insert the next DOM node. The `parent` variable is used for the first DOM node where `previousEL` is null. This means we only need the parent of the first element in the DOMExpression. After we insert one of the element from the DOMExpression, we update the `previousEL` variable to reference that element on line 22 such that the next element in the DOMExpression is also inserted correctly. The `useAppend` boolean is used to indicate whether the DOM node should be inserted as a sibling or a child of `previousEl`. The template element in iteration 1 was designed in such a way that each DOM node in the DOMExpression should always be inserted into the DOM as a sibling. However, now that we rely on the previous element in the JSX for inserting the first DOM node, we need to handle the case where the first DOM node is the first child of the parent node. Consider the example in Code 25.

```
1  // This JSX
2  <div>
3      {[1,2,3].map(el => <p>{el}</p>)}
4  </div>
5  // Generates the following HTML
6  <div>
7      <p>1</p>
8      <p>2</p>
9      <p>3</p>
10 </div>
```

Code 25: Example of a DOMExpression, where the element inserted before the DOMExpression is the parent element of the DOMExpression itself.

The div on line 1 is the parent of the DOMExpression and its the last DOM node inserted into the DOM before the DOMExpression. This means that the first DOM node in the DOMExpression should be inserted as a child of the div. We use the `useAppend` to control this by setting it to true. When we insert the second DOM node from the DOMExpression, we should no longer insert it as a child or it would become nested inside the first DOM node on line 7 in Code 25.
These changes ensure that all DOMExpressions can be updated independently and that the Krausest tool can run the microbenchmarks.

4.2 Implementation

In this section, we detail the changes made to *React-compiler* to accommodate the design changes outlined in section 4.1. We include an updated visitor overview which can be seen in fig. 4.2.

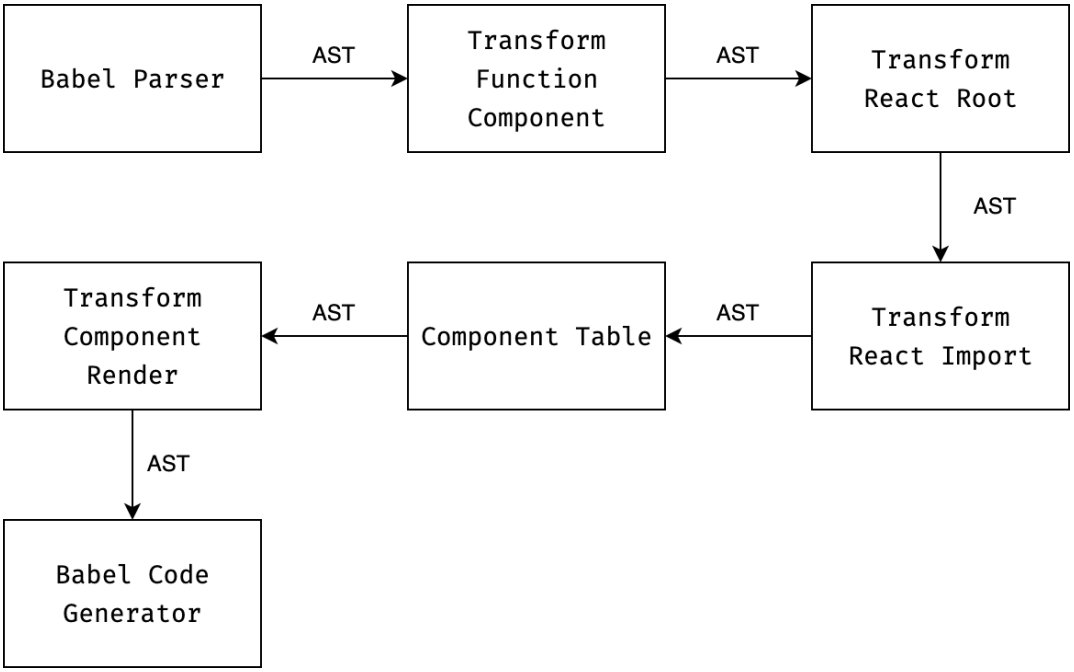


Figure 4.2: Revised Compiler overview.

We introduce a new visitor, Component Table Visitor, which is responsible for generating the component table entry for each component in the application. We place it after the Transform Function Component visitor, so we do not need to handle function components. Technically, we do not need to visit function components since they cannot use state in React as discussed in section 1.1.1. Nonetheless, we place the Component Table Visitor such that all function components have been transformed into class components beforehand. We need the component table in the updated Transform Component Render visitor, so the table must be completed before the visitor is called.
The implementation details of the new Component Table visitor are described in section 4.2.1. In section 4.2.2, we discuss the updated action generation process to support the new `propsUpdate` and `setState` methods.

4.2.1 Component table visitor

The component table is generated by its own visitor to separate concerns and to keep each visitor as simple as possible. The component table is stored as an array where each element is an object representing a component. The name of the component is retrieved by reading the name of the class component in the AST. We create a helper function for finding expressions that use state or props. The function for detecting state access can be seen in Code 26.

```
1 function isExpressionStateAccess(astNode) {
2     return t.isMemberExpression(astNode)
3         && t.isThisExpression(astNode.object.object)
4         && astNode.object.property.name === 'state';
5 }
```

Code 26: The isExpressionStateAccess function used to test if an AST sub tree represents the code this.state.[property].

The function `isExpressionStateAccess` follows a similar pattern to those provided in the Babel type library. The library contains functions for testing if an AST node is a specific type such as `isMemberExpression` as seen on line 2 in Code 26. We have created helper functions built on top of those provided by Babel, so we can reuse the same code throughout the compiler. We have a similar approach to finding `setState` calls in components. The code in the Component Table visitor responsible for handling `setState` calls can be seen in Code 27.

On line 1 we use the visitor to find `CallExpression` AST nodes. We then test the call expression on line 3 with the helper function `isSetStateCall`. We use the name of the class method where the `setState` call is located, as part of the new `setState` method name as seen on line 8. This helps us read the compiled output and know what the method was generated from. The `setState` call is replaced by the new method such that it is called at runtime instead. The first argument to `setState` contains the new state object to be merged with the current state. On lines 10-18, we iterate through each property on the object and the `setState` information to the component table entry.

4.2.2 Action generation

In section 3.2.1 we used the `DOMFactory` function as seen in Code 15 and 16 to generate the actions based on the type of element provided. We refactor the function in this iteration to make it easier to maintain and separate the code for generating actions for each element type. We introduce a new function named `componentDOMGenerator`, which can be seen in Code 28.

```
CallExpression(subPath) {
    // Find this.setState calls
    if (isSetStateCall(subpath.node.callee) {
        // Get either the name of the class method or the property name if its an arrow function
        const methodName = getClassMethodParent(subPath)
            ? getClassMethodParent(subPath).node.key.name
            : getClassPropertyParent(subPath).node.key.name;
        const newStateMethod = `setState_${methodName}_${idCounter++}`;

        if (t.isObjectExpression(subPath.node.arguments[0])) {
            // Get key of each property in the object
            const keys = subPath.node.arguments[0].properties.map(p => p.key.name);
            componentDetails.setStates.push({
                name: newStateMethod,
                method: methodName,
                keys
            });
        } else {
            console.warn('setState called without an object');
        }

        const ast = templateStatement(`this.${newStateMethod}(ARGUMENTS)`)({
            ARGUMENTS: subPath.node.arguments
        });

        subPath.replaceWith(ast);
    }
}
```

Code 27: Snippet from the Component Table Visitor.

```
1  function componentDOMGenerator(element, context, actions, options) {
2      const elementGenerator = elementTypeGeneratorMapping[element.type];
3      let elementActions = elementGenerator(element, context, componentDOMGenerator, options);
4
5      elementActions.forEach(action => {
6          if (action.isUnmount) {
7              actions.unmountActions.push(action.actionAST);
8          } else if (action.isStatic) {
9              actions.mountActions.push(action.actionAST);
10         } else {
11             actions.updateActions.push(action.actionAST);
12
13             if (action.useProps) {
14                 actions.propsActions.push(action.actionAST);
15             }
16
17             if (action.useState) {
18                 // Add the action to action lists for each key it uses
19                 action.stateKeys.forEach(key => {
20                     if (actions.stateActions[key]) {
21                         actions.stateActions[key].push(action.actionAST);
22                     } else {
23                         // Create the array if this is the first time it's seen
24                         actions.stateActions[key] = [action.actionAST];
25                     }
26                 });
27             }
28         }
29     });
30
31     return actions;
32 }
```

Code 28: The componentDOMGenerator function, which replaces the DOMFactory used in chapter 3.

We use a dictionary as a lookup table for an elements type and the function used to generate actions for it. The element type is used as the key and the value is the function to be used. On line 2 in Code 28 we perform a lookup in the table to get the correct action generator function. This design replaces the long if-else-if chain that was used in the `DOMFactory` function from Code 15 and 16. Most of the code inside each if block has been moved into its own function. For instance, the code in Code 16 regarding action generation for components has been moved into a function called `componentActionGenerator`. Every action generator function accepts the same parameters and returns a list of actions as seen on line 3 in Code 28. On lines 5-29 we sort each action into the methods to be created in the component: `mount`, `unmount`, `update`, `propsUpdate`, and `setStates`. This was

previously done inside the if blocks in the `DOMFactory` function, which meant some code was duplicated and made it harder to make changes.
We use an object of arrays to sort actions that use state as seen on lines 20-25. Each state object property has an array such that any action using said property, is added to the array. An action can, therefore, exist in multiple arrays if it depends on multiple state properties. The Component Render Visitor uses the state actions object to generate the `setState` methods as seen in Code 29.

```
1  const comp = componentTable.find(c => c.name === componentName);
2  comp.setStates.forEach(setState => {
3      const relevantActions = [];
4      setState.keys.forEach(key => {
5          if (actions.stateActions[key]) {
6              relevantActions.push(...actions.stateActions[key]);
7          }
8      });
9      const stateUpdateMethod = componentStateUpdateMethodTemplate(setState.name, {
10         RENDERCONTENT: renderContent,
11         BODY: relevantActions
12     });
13
14     subpath.insertAfter(stateUpdateMethod);
15 });
```

Code 29: Part of the Transform Component Render visitor responsible for creating the setState methods based on the actions.

Using the entry in the component table, we iterate through each `setState` call in the component on line 2. We then use the `stateKeys` array to get all relevant actions from the state actions object by using the property name as the key on lines 5-7. One downside to this approach is that new `setState` methods can contain duplicate code, which increases the size of the compiled application. However, the reduction in file size by removing the React runtime should outweigh the increase in size casued by the new `setState` methods.

4.3 Results

One of the objectives of iteration 2 has been to address the issues, which were presented in section 3.3. Therefore, this section outlines the results of the work done in both iteration 1 and iteration 2, as well as, presents what could be improved upon or changed in chapter 5. We expect to see performance improvements in the DOM operations category, but at a cost of more memory consumption and longer startup times.

4.3.1 Iteration 1 and 2

The tables B.1 and B.2 show the microbenchmarks results from chapter 3 with the compiled results for iteration 1 and 2. The performance issues we found in pre-generated rows in section 3.3.2 are also present in the DOMExpression implementation for "swap rows" as seen in fig. 4.3.
In fig. 4.3 we see that both the iteration 1 and 2 compiled applications are up to 30 times slower than the other implementations. This is similar to the performance of the PoC we saw in section 3.3.3.
In general, the DOM operation results have not seen much improvement over the course of both iteration 1 and 2, despite changes such as introducing DOM reuse. It makes sense that the changes made in iteration 1 and 2 do not improve performance significantly, since the microbenchmarks rely on the DOMExpressions which do not benefit from the changes. The minor improvement that we do see, may be attributed to the fact that some of the code executed at runtime in the PoC, has been moved to compile time in *React-compiler*.
Furthermore, when looking at the memory and startup categories, we have seen performance degradations in some benchmarks. However, this makes sense as *React-compiler*, in general, generates more code and performs more actions when updating the DOM compared to the PoC which would remove the DOM nodes and rebuild them in all cases. For instance, in fig. 4.5, the performance has decreased by around 26% in the "Run memory" benchmark when comparing it1 and it2 to our PoC. It should be noted, that the percentage is calculated based off of the actual values in appendix B. The relative performance only tells us how well the implementations perform relative to the best performer in that benchmark. In order to find the percentage difference between two implementations, we must compare their actual values. Any further percentages displayed in this report are devised using the same method. We assume this performance decrease is due to the compiler tracking references to all DOM nodes, as opposed to the PoC where references were not tracked.
Now that we have verified that the performance issues with DOMExpressions as we did with pre-generated rows, we can focus on investigating what the potential causes are. We expect to see the best results in pre-generated rows, since we have focused on DOM node reuse in *React-compiler*.
The results of compiling the pre-generated rows implementation with the iteration 2 version of *React-compiler* yielded similar results as seen in section 3.3.2. In iteration 1, we theorised that the performance issues were either related to the compiled code being slow, or that the browser environment was overloaded by the number of DOM operations executed in the update methods. If the code generated by *React-compiler*, is the culprit, then we would see a difference between the performance in the microbenchmarks using DOMExpressions compared to pre-generated rows. However, when inspecting the benchmarks, we can see that

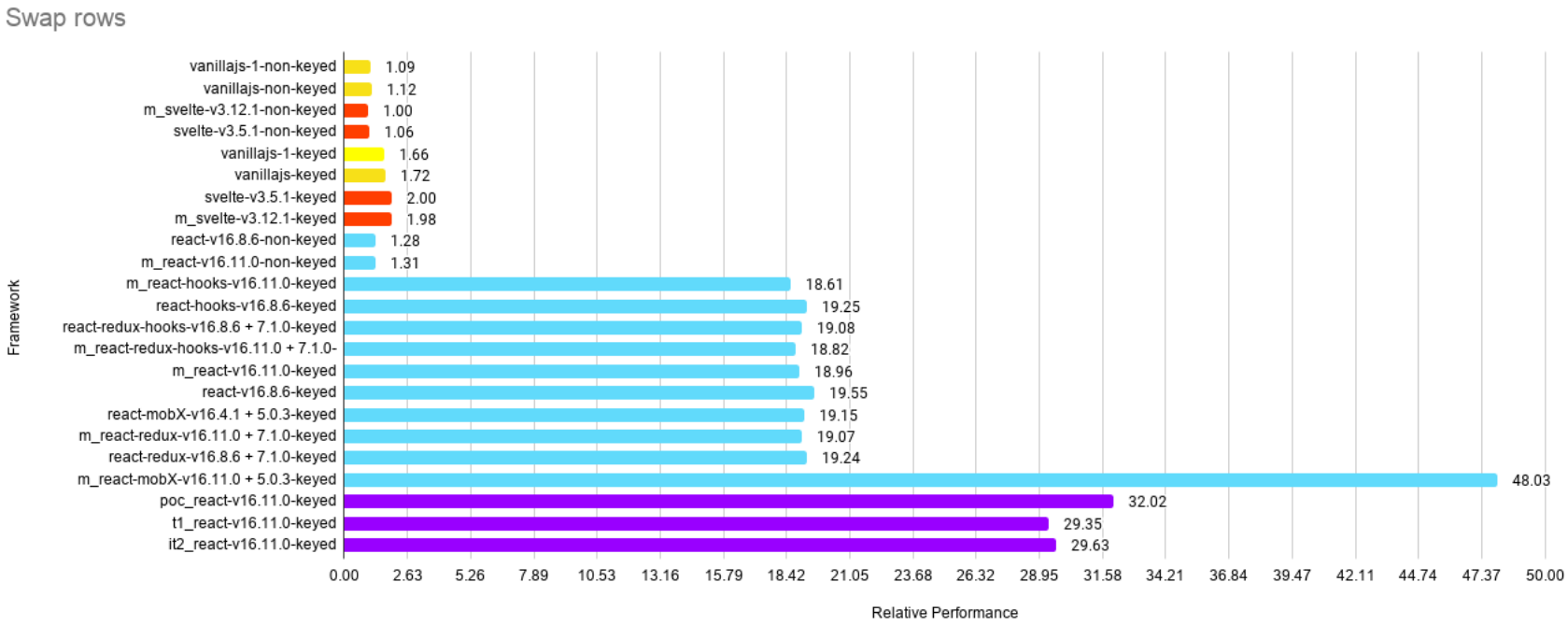


Figure 4.3: Swap rows. The values indicate relative performance, where 1.0 is the fastest. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

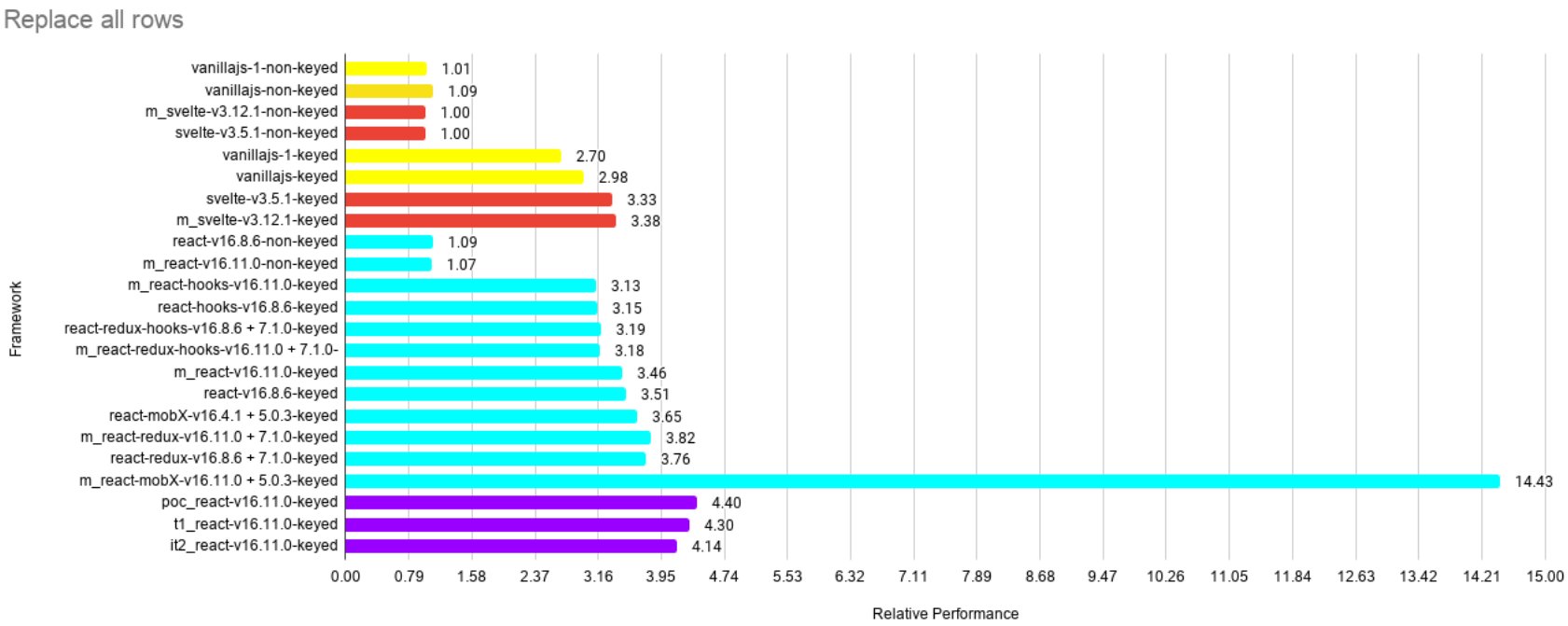


Figure 4.4: Replace all rows. The values indicate relative performance, where 1.0 is the fastest. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

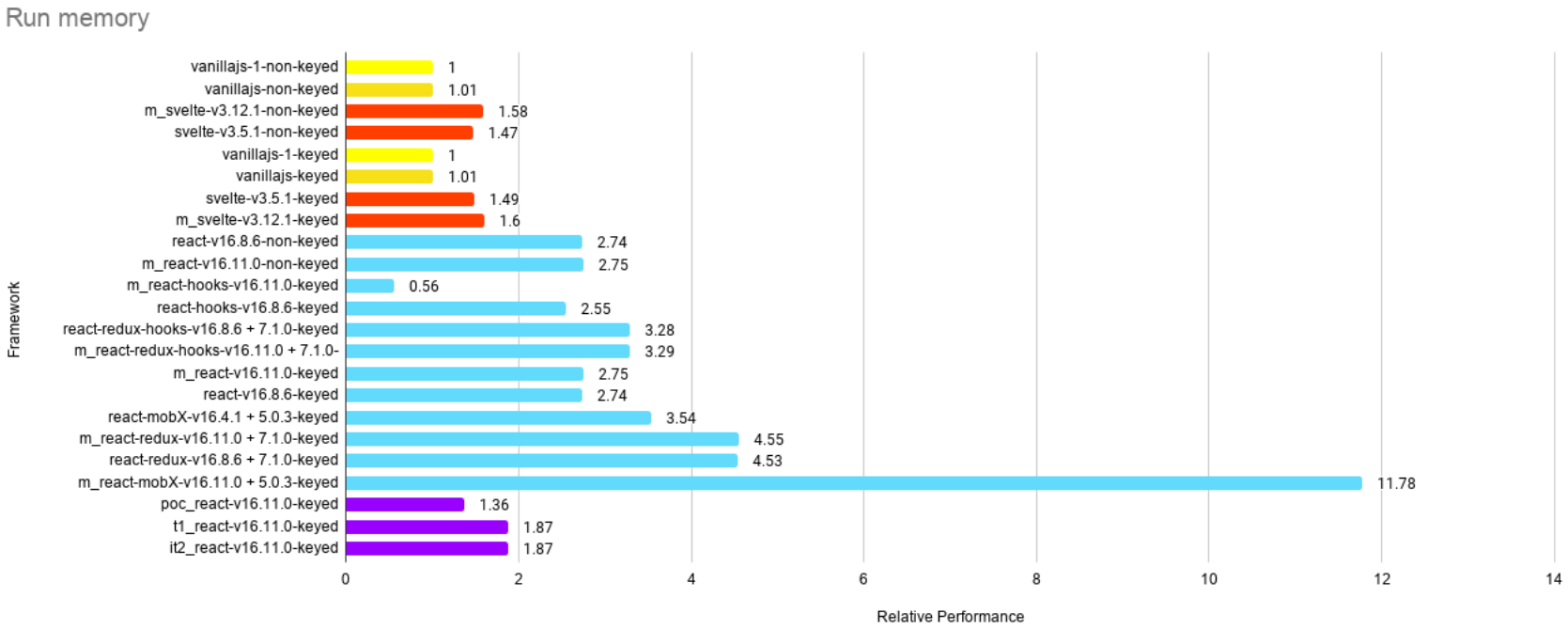


Figure 4.5: Memory usage after adding 1k rows. The values indicate relative performance, where 1.0 is the best. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

it is the same benchmarks that perform poorly in both cases. We expect the DOMExpressions code to be slightly slower in "partial update" than in the "create 1.000 rows" benchmark, since the code removes the existing rows and recreates them. This is also the case in the PoC. Instead we see, that the performance in "select row" and "partial update" follow the same pattern across DOMExpressions and pre-generated rows for both the PoC and *React-compiler*.

Some of the microbenchmarks in Krausest use CPU slowdown, to simulate a slower computer in a benchmark. The specific amount of slowdown varies between each benchmark from 0 to 16x. Interestingly, the two worst performing benchmarks for both the PoC and *React-compiler* are "partial update" and "select row" which have the highest CPU slowdown of 16x. The "append rows" bench uses 2x CPU slowdown where we also see the PoC and *React-compiler* perform poorly compared to the other implementations. It stands to reason that the CPU slowdown may have a significant impact on the performance of our implementations, whereas the React, Svelte and vanillaJS are less affected. To test this theory, we disable CPU slowdown on the microbenchmarks, to see the performance difference between *React-compiler* and React. The results of disabling CPU slowdown on the microbenchmarks can be seen in table 4.1. We have also included the results of the same benchmarks with the original CPU slowdown for comparison.

We have used the actual performance values in table 4.1 to be able to better compare the effect of CPU slowdown on the results. React still outperforms the compiled applications. However, when looking at the actual values for both it1 and it2, we can see that the "partial update", "select row" and "swap rows" are approximately 10 ms slower than "create rows". These results are what we expect, since the DOMExpressions recreate the DOM nodes on every rerender as discussed above. We do not expect it1 and it2 applications using DOMExpressions to perform better than the results seen in table 4.1. The React application utilises a keyed mode implementation, which suggests there is significant performance to be gained by implementing a similar approach in *React-compiler*. Running the "partial update" benchmark with 16x CPU slowdown results in an actual runtime of 5.926 ms and 5.525 ms for it1 and it2, respectively. Disabling CPU slowdown results in 194 ms runtime for both versions of *React-compiler*. Comparing these shows that the 16x CPU slowdown version is 30 times slower for it1 and 28 times slower for it2. Compare this to the React application which is 19 times slower with 16x CPU slowdown in the "partial update" benchmark. This suggests that the CPU affects *React-compiler* more than React. Furthermore, the results without slowdown show that the generated code, while slower than React, is not the reason for the poor performance results in the microbenchmarks.

We now perform the same test on the pre-generated rows benchmark by disabling CPU and compare the actual results. The results of the test can be seen in table 4.2.

Once again, React outperforms the compiled application. However, it is only approximately

	Create rows	*Partial update*	*Select row*	*Swap rows*	*Append Rows*
react-v16.11.0-keyed	177.6ms	15.5ms	5.3ms	118.1ms	182.4ms
it1_react-v16.11.0-keyed	181.0ms	193.8ms	192.9ms	188.6ms	415.5ms
it2_react-v16.11.0-keyed	182.6ms	194.0ms	193.1ms	188.3ms	424.4ms
no-cpu-slowdown_react-v16.11.0-keyed	186.2ms	296.5ms	91.5ms	574.1ms	399.6ms
no-cpu-slowdown_it1_react-v16.11.0-keyed	181.2ms	5926.5ms	5505.9ms	861.9ms	828.6ms
no-cpu-slowdown_it2_react-v16.11.0-keyed	185.6ms	5525.5ms	5751.5ms	870.1ms	833.8ms

Table 4.1: The microbenchmark results with and without CPU slowdown.

	Create rows	Partial update
pre-generated-rows-compilable-v16.8.6-keyed	32.5ms	276.7ms
it2_pre-generated-rows-compilable-v16.8.6-keyed	4.1ms	1859.2ms
no-cpu-slowdown_pre-generated-rows-compilable-v16.8.6-keyed	32.3ms	23.4ms
no-cpu-slowdown_it2_pre-generated-rows-compilable-v16.8.6-keyed	15.2ms	62.3ms

Table 4.2: Pre-generated rows with and without CPU slowdown.

3 times as slow compared to the results in table 4.1, where the compiled application is 12 times slower. This result makes sense since the pre-generated rows application does not use DOMExpressions, meaning it can take advantage of DOM node reuse. Furthermore, the CPU slowdown has the same effect on performance in pre-generated rows as we saw in the other microbenchmarks.
The implementations in the pre-generated rows do not take advantage of the dependency tracking feature of *React-compiler* introduced in this iteration. We can see why by inspecting the code for "partial update" in Code 30.

```
1 update() => {
2     const data = this.state.data;
3     for (let i = 0; i < data.length; i += 10) {
4         const item = data[i];
5         data[i] = { id: item.id, label: item.label + " !!!" };
6     }
7     this.forceUpdate();
8 };
```

Code 30: The method used in the React "partial update" benchmark.

The code does not use the `setState` React API at all, but instead uses the forceUpdate API on line 7. This means that none of the improvements in iteration 2 *React-compiler* for state changes are used in the compiled version. Calling the `forceUpdate` method means, that the update method is called in the compiled application, which performs unnecessary DOM operations, since it is designed to update all computed values. The code in Code 30 manipulates the state of the component directly, which is not the intended approach in React. The pre-generated rows implementations need to be refactored to use `setState` instead of `forceUpdate` such that the a `setState` method is called in the compiled application rather than the `update` method. Inspecting the code in Code 30 also reveals a design flaw in our dependency tracking design. The data property of the state object is an array and the state dependency tracking, will not look at a specific index when tracking dependencies, which means that if we were to rewrite the code in Code 30 to use the setState API, the compiled code, would update all DOM nodes that depend on the data array, rather than just the ones that changed. This limitation of *React-compiler* reduces the possible performance improvements in the pre-generated rows benchmarks. In order, to properly benefit from the improvements made to *React-compiler* in this iteration, we would need to rewrite the state object to use a separate property for each row of data. An example of this can be seen in Code 31.

We have identified some limitations of the current version of *React-compiler*, which, if ad-

```
this.state = {
    0: {id: 0, label: 'value 0'},
    1: {id: 1, label: 'value 1'},
    2: {id: 2, label: 'value 2'}
}
```

Code 31: Example of converting the data array into object properties instead to work with the dependency tracking in this iteration.

dressed, could lead to better performance in the pre-generated rows. However, the CPU slowdown is biggest contributor to the poor performance we see in both the microbenchmarks and pre-generated rows. To fix this problem, we must identify what mechanisms other frameworks like React use to overcome overloading of slower CPU's. The React Core Team introduces a new architeture named fiber with the release of React version 16 [8, 35]. This new architeture changed the way that React structures it work. Specifically, it introduced a scheduler to help keep the UI responsive while updating the DOM [35]. The React Scheduler assigns a priority to each DOM operation, such as low, normal and immediate. JavaScript is single threaded [36]. Any work done on the main thread can block the DOM from updating, which results in unresponsive UI [35]. The purpose of the React Scheduler is to perform as many DOM operations as possible without blocking the main thread long enough to impact the responsiveness of the UI. The scheduler will prioritise the most important DOM operations such as button clicks. However, each priority has an associated timeout, which ensures that all DOM operations are eventually performed, even if higher priority operations are queued. The results of this implementation is that the UI remains responsive throughout the rerendering process in React. When *React-compiler* triggers a rerender, it will queue all DOM operations on the main thread. This means that the main thread is blocked untill all the operations have been performed. We suspect that this is the root cause of our performance issues. This approach is not a problem if the CPU is sufficiently fast since it is capable of performing all DOM operations without leaving the UI unresponsive. However, with CPU slowdown, this is no longer the case. Implementing a scheduler for *React-compiler* should result in better performance.

4.3.2 Goals for iteration 3

As mentioned previously, we suspect the cause of our poor results to be due to the lack of scheduling of DOM operations. We theorise that in order to gain any significant performance increases in the Krausest benchmarks, we have to implement a scheduler. However, due to the inherent time limitations that comes with a semester project, we do not have time to both acquire the knowledge required to build a scheduler *and* implement it. While we could attempt to implement a naive queue system for DOM operations, that implementation would

have to be completely reworked later and replaced with a proper scheduler. We assess that this solution is not viable and have, therefore, elected to use the remaining time of this project on a third iteration which focuses on optimising and rewriting existing functionality of *React-compiler*, as well as, implementing the foundation of a UI for propogating errors when attempting to compile applications. The rewrite of the codebase will focus on exposing a single API for all elements such as components and DOMExpressions. The UI will focus on validating and providing error messages for invalid HTML elements, however, the idea is that this UI will be extendable to other error handling. This feature falls under the developer feedback category, which we initially postponed to future work in section 1.2. Furthermore, as the benchmarks are a test of keyed/non-keyed modes, as our results suggest, we will dedicate some time to implementing a non-keyed version of *React-compiler* in hopes of providing a performance increase despite the lack of a scheduler.

Chapter 5

Iteration 3

This chapter outlines the design and implementation of 4 features — (1) a common interface for all elements in *React-compiler*, (2) a validation module that can determine whether an HTML element is valid, (3) a module which can print error messages to the terminal when errors occur, and (4) an implementation of a non-keyed mode in *React-compiler*
1-3 describe "nice to have" features to *React-compiler*, while 4 focuses on performance optimisations in the microbenchmarks.

5.1 Design

One of the complexities we have encountered while developing *React-compiler* is the different interfaces of generated code. Components are implemented as a class that requires the `new` keyword to create an instance. Other elements are wrapped in functions such as DOMExpressions and component's children as discussed in section 3.1.7. The internal logic of *React-compiler* needs to handle each type of element differently as a result. Furthermore, it means that a change to an element interface may require changes throughout the compiler. Effectively, there is a tight coupling between each element in the compiler and their specific interfaces. In this iteration, we look at designing and implementing a common interface for all element types in the compiler, which hides their specific implementation details, such that changes can be made without requiring modifications in other parts of the compiler.

5.1.1 Common Element Interface

We focus on creating a minimal interface, that only contains functionality currently used in the compiler. This way we avoid adding features, which seem to be useful, but may never be used. We can use the categories of actions used in the `componentDOMGenerator` discussed in section 4.2.2 as a starting point. Each action is sorted into a list based on its

purpose. Currently, we have the following action lists:

- `Mount`: Actions that create and insert a DOM node into the DOM or instantiate a component.
- `Update`: Actions that modify the value or attribute of a DOM. Any computed value is considered an update action, as explained in section 3.2.
- `PropsUpdate`: If an action uses the props object of a component it is included in this list. These actions are also present in the Update list.
- `SetStates`: Each entry contains a set of actions that depend on the state of a component. Similar to propsUpdate.
- `Unmount`: Actions that unmount components and removes DOM nodes from the DOM. Used for clean up.

The `Update` and `SetState` action lists are intended for internal use because they are only triggered by code inside the component. The `Update` action list is used in the `update` method which is called by the mount method. Similarly, the `SetState` actions are used for the `setState` methods replacing the React API method with the same name as discussed in section 4.1. Since the state of a component is internal, it should not be modifiable outside of the component explicitly through an interface.
We could create an interface with `Mount`, `PropsUpdate`, and `Unmount`, but if we look at the `Component` class in Code 32, we can see that not all React API methods are supported by these 3 methods.

Specifically the `forceUpdate` method on line 19 in Code 32. This React API method is used to trigger a rerender without updating the props or state of a component as we discussed in section 4.1. So we need to add the `forceUpdate` method to our interface to support the React API. The `init` and `receiveNewProps` methods on line 3 and 33 respectively, are not React API methods. We use the `init` method to initialize and mount the component, which is the equivalent to the `mount` method in the proposed interface. In fact, the `init` method calls the `mount` method as seen on line 7 in Code 32. `receiveNewProps` is equivalent to the `propsUpdate` interface method. The remaining methods in the `Component` class are React lifecycle methods, which can be used inside the component by overriding them. They are defined in the `Component` class to avoid errors if a lifecycle method is called from a component that has not implemented its version of the lifecycle method.
We need to decide how to implement our interface. One option is to create a new class that conforms to the new interface. Another option is to use the factory function pattern [37]. This pattern consists of a function that returns an object which conforms to the API [37]. If we use a class, we need to use the `new` keyword, whereas the factory function is a normal function invocation. Furthermore, we would need to create a class declaration for every component, DOMExpression, etc, which leads to a lot of additional code. The factory

```
class Component {
    constructor(props) { this.props = props || {}; }
    init(rootEl, isSibling = false) {
        this.root = rootEl;
        this.isSibling = isSibling;
        this.children = [];
        this.mount();
        this.componentDidMount();
        return this;
    }
    setState(state) {
        const oldState = this.state;
        this.state = { ...oldState,...state  };
        if (this.shouldComponentUpdate(this.props, state)) {
            this.update();
            this.componentDidUpdate(this.props, oldState, null);
        }
    }
    forceUpdate(callback) {
        const oldState = this.state;
        this.update();
        this.componentDidUpdate(this.props, oldState, null);

        if (callback) {
            callback();
        }
    }
    shouldComponentUpdate(nextProps, nextState) { return true; }
    componentDidMount() {}
    componentDidUpdate(prevProps, prevState, snapshot) {}
    componentWillUnmount() {}
    unmount() { if (this.container) this.container.remove(); }
    receiveNewProps(props) {
        this.props = props;
        if (this.propsUpdate) {
            this.propsUpdate();
        } else {
            this.update();
        }
    }
}
```

Code 32: The Component class used by *React-compiler* to replace the React Component class.

function can be placed inline which means we can declare it and use it in place of the current implementation. Additionally, the `Component` class from Code 32 does not contain a lot of logic, so we can convert all class components to the factory pattern without having to re-implement a lot of code from the `Component` class. Thus we settle on a factory function pattern which looks like the function in Code 33.

```
function facotry(props) {
    // init code here such as function and variables
    return {
        mount(root, isSibling = false) {},
        update(props) {},
        forceUpdate() {},
        unmount() {}
    };
}
```

Code 33: General pattern for the common interface using the factory function pattern.

The implementation details are detailed in section 5.2.1.

5.1.2 Validating elements

In [1], we do not validate HTML elements during compilation apart from checking the type of the element. In this iteration we design and implement a module which validates whether an HTML element is valid i.e. it follows the specification provided by MDN [38].
The code shown in Code 34, shows the current validation performed before the HTML element is processed. This code is found in the JSXTreeVisitor, used in the Transform Component Render phase in fig. 4.2.

```
function convertJSXElement(JSXElement, parentId, path) {
    const elType = JSXElement.openingElement.name.name;

    if (supportedHTMLElements.includes(elType)) {
        return convertHTMLElement(JSXElement, parentId, path);
    } else if (/[A-Z]/.test(elType[0])) {
        return convertComponentElement(JSXElement, parentId, path);
    } else {
        console.error(`Invalid or unsupported element: ${elType}`);
        return;
    }
}
```

Code 34: The convertJSXElement function.

In the function `convertJSXElement`, on line 4, the conditional statement checks whether

the `supportedHTMLElements` list contains the `elType` of the element we are looking at, which is the type of the HTML element. If that statement evaluates to true, it will proceed to convert the HTML into an object representation shown in Code 35.

```
let element = {
        id: elId,
        type: 'html',
        htmlType: elType,
        parent: parentId || null,
        attributes: [],
        eventListeners: [],
        children: [],
        isStatic: true
    };
```

Code 35: Object representation of an HTML element.

The object contains information about the HTML element, as well as, any children it may have. As the visitor proceeds through the AST, it will add the attributes and event listeners found in the JSX to the `attributes` and `eventListeners` properties. This design originates from the PoC in [1].
While this solution works, it is vulnerable to errors and can lead to runtime errors if we attempt to create invalid HTML elements. Instead, we can redesign the current method in order to verify the entirety of the element object, rather than just the type of the element. We create a function, which takes the element object as input, shown in Code 35, once it contains all information. This allows us to verify the element object by adding a function call in the `convertHTMLElement` function, which is called in Code 34, and shown in Code 36.
On line 23, we can validate the element and display any errors that may occur to the user. We define errors as being invalid attributes and event listeners i.e those which are not valid for a specific HTML element, as well as, duplicated attributes or event listeners.

5.1.3 Displaying error messages

We can create an interface which can be used to display error messages to the user when an HTML element is invalid. This serves as a basis on which we can extend towards more developer feedback functionality in future iterations of *React-compiler*.
The interface is built for the terminal, similar to how other compilers produce their error messages through the terminal. Instead of only pointing to where the error occurs with a line number, we are able to take it one step further by displaying the code snippet and indicating which part of the code is an error. In order to do this, we use two tools — babel code-frame [39] and Ink [40].
The former, babel code-frame, is a babel package which allows us to produce code snippets

```
function convertHTMLElement(JSXElement, parentId, path) {
    const elId = generateId();
    const elType = JSXElement.openingElement.name.name;

    let element = {
        id: elId,
        type: 'html',
        htmlType: elType,
        parent: parentId || null,
        attributes: [],
        eventListeners: [],
        children: [],
        isStatic: true
    };

    JSXElement.openingElement.attributes.forEach((attr, index) =>
    convertHTMLElementAttribute(element, attr,
    path.get(`openingElement.attributes.${index}`)));

    JSXElement.children.forEach((child, index) =>
    convertJSXChild(element, child, path.get(`children.${index}`)));

    //call validation function

    return element;
}
```

Code 36: The convertHTMLElement function which will contain the validation call.

with syntax highlighting, as well as, arrows pointing towards a specific line and column of the snippet, as depicted in fig. 5.1. The column refers to the depth or n'th character in that line.

```
  1 | class Foo {
> 2 |   constructor()
    |               ^
  3 | }
```

Figure 5.1: Example of the use of babel code-frame.

The latter, Ink, is a custom React renderer for building command line interfaces (CLI). We need a custom React renderer, because we cannot use HTML elements in the terminal since it is not a browser environment. We discussed how react handles different environments and renderers in section 3.1.3.
There are multiple methods of developing CLI's in various languages, which require a number of libraries that each handle different functionality, such as interaction, styling, and orienting content. However, Ink wraps all such functionality within a single library and allows one to write CLI applications similar to writing web applications in React and JavaScript. It utilises much of the functionality provided by React, such as both functional and class components, as well as, Hooks. Furthermore, one can style and orient the content of the output using JSX syntax and custom components that are built for the terminal. This means that instead of using multiple libraries in order to build the CLI, we can use Ink and use our knowledge from React to speed up the process of developing the CLI.
An example of a application built in Ink is shown in fig. 5.2.

Figure 5.2: Snapshot of a Counter application in Ink.

The code required to build the application in fig. 5.2 is shown in Code 37.
The structure of the application is similar to a React application using Hooks. The `Counter` function returns JSX, which is then rendered in the terminal. The `Color` component on line 14, is a custom component provided by Ink. It is used styling for strings and rendering

```
const Counter = () => {
    const [counter, setCounter] = useState(0);

    useEffect(() => {
        const timer = setInterval(() => {
            setCounter(previousCounter => previousCounter + 1);
        }, 100);

        return () => {
            clearInterval(timer);
        };
    }, []);

    return <Color green>{counter} tests passed</Color>;
};

render(<Counter />);
```

Code 37: Code for the Counter example.

coloured text in the terminal.

5.1.4 DOM Reuse in DOMExpressions

Implementing DOM node reuse in DOMExpressions is required to improve the performance of the compiled applications of *React-compiler* in the microbenchmarks. The DOM node reuse implemented in chapter 3 does not work for DOMExpressions, because it is not capable of determining which DOM nodes to create, update and remove during rerender, since it depends on the expression in the DOMExpression itself. Consider the DOMExpression in Code 38.

```
{this.state.data.map(el => <div>{el.value}</div>)}
```

Code 38: DOMExpression that creates a DOM node for every entry in the data array of the component's state.

The DOMExpression uses the component's state to generate DOM nodes. In order to reuse the DOM nodes created by the DOMExpression, we need to know what DOM nodes were created last render. For instance, if the data array contains two elements initially then we have 2 DOM nodes in the DOM. If we trigger a rerender by adding a third, we need to add another DOM node to the DOM. Inversely, we need to remove a DOM node if an element is removed from the array. We also need to ensure that each DOM node displays the correct value, such that each element's value is displayed in the same order as the data array. The job of a keyed/non-keyed mode is to handle updating these DOM nodes, such that they always

match the data they were generated from, as is explained in section 2.1.1.

In this iteration, we design and implement a non-keyed mode, because it is the best overall performer in the microbenchmarks, as shown in section 3.3. It is also the simplest to implement, since we do not need to deal with keys. It is worth noting that there are cases where a keyed implementation is better. For instance, if an element has been added at the beginning of the data array, then every DOM node is updated in a non-keyed mode, while a keyed implementation will detect that all but the first element has just been moved. Deciding which mode to use is a tradeoff between the cost of performing unnecessary DOM operations in non-keyed mode and the cost of computing what has been moved, updated and removed in a keyed mode.

In algorithm 1 we present the algorithm we have designed for our non-keyed implementation.

```
1  elements = []
2  position = 0
3  for item in DOMExpression do
4      if elements[position]! = null then
5          elements[position].update()
6      else
7          element = newElement()
8          elements[position] = element
9          element.mount()
10     position += 1
11 if elements.length − 1 >= position then
12     i = position
13     while i < elements.length do
14         elements[i].unmount()
15         elements[i] = null
16         i += 1
```

Algorithm 1: Non-keyed mode for *React-compiler*.

On line 1 we define an array, `elements`, which we use for storing the references to the elements in the DOM. Each entry is the common element interface object we designed in section 5.1.1, which enables us to perform `update` and `remove` on each element in the DOMExpression. The `postion` variable on line 2 is used to track the current position in the elements array. As we evalute the DOMExpression, we must keep track of which elements we have reused. The for loop on line 3 is controlled by the expression of the DOMExpression. In the example Code 38, the expression would be `this.state.data.map`. The

non-keyed implementation must work with any DOMExpression, so we cannot assume how the expression in a DOMExpression works. For instance, it must also work for DOMExpression, which do not iterate through data in the component's state. This is why we use the `position` variable to track the postion in the elements list. When the JSX is replaced inside the DOMExpression, we insert the code on lines 4 - 10 inside the for loop on line 3 to handle each item in the DOMExpression. Incrementing the `position` variable on line 10 lets us track our progress without havin direct control of the expression itself. On line 4 we check if an element exists for the current position, so we can reuse it. A new element is created on line 7 and inserted into the elements array on line 8. We defer the details regarding element creation to the implementation in section 5.2.5. Once the expression has been evaluated, we must check if there is any elements in the DOM that was not reused. On line 11 we check if the current position is at the end of the elements array. If we still have elements left, we loop through each remaining element on line 13 and remove it from the DOM and the array. This ensure that unused DOM nodes are not left in the DOM.

5.2 Implementation

The implementation of the proposed design in section 5.1, is split into 5 parts following the order in section 5.1.

5.2.1 Common Element Interface

Implementing the common element interface designed in section 5.1.1 requires us to transform all class components into factory functions. This transformation involves the following steps:

- Transforming class methods into local functions of the factory function.
- Moving properties on the class to local variables of the factory function.
- Moving the content of the class constructor into the body of the factory function.

We use a Babel template similar to those used in section 3.2.1 to generate the AST for the factory function. The template can be seen in Code 39.

We use this template as the scaffolding for each of the class components that need to be converted. We modify the Component Render Visitor introduced in section 3.2, to replace the entire class declaration instead of only replacing the render method. We extend the visitor to visit every class method and property, so we can copy the AST to the factory function. We use an array to store the ASTs. If we attempted to insert the method and property ASTs directly into our factory function template, we would get a syntax error from Babel. Class methods and properties have different declaration syntax than local variables

```
function COMPONENTNAME(props) {
    let container;
    let children = [];
    let state;

    // React API Methods

    // DOM Nodes
    DECLARATIONS

    // Constructor Content
    CONSTRUCTORCONTENT

    // Component Methods
    COMPONENTMETHODS

    // SetState Methods
    SETSTATES

    return {

        mount(root, isSibling = false) {
            componentMount(root, isSibling);
        },

        update(props) {
            componentPropsUpdate(props);
        },

        forceUpdate() {
            componentUpdate();
        },

        unmount() {
            componentUnmount();
        }
    };
}
```

Code 39: Template used for creating factory functions for components in *React-compiler*. Capitalised words are replaced by Babel with AST's provided as input.

and function declarations. So, we need to convert class methods to function declarations and class properties to local variables. An example of this transformation can be seen in Code 40.

```
// Class method
print(text) { console.log(text); }
// to
function print(text) { console.log(text); }

//Class property
Type = 'component';
// to
let Type = 'component';
```

Code 40: Example of the transformation from class method to function declaration and class property to local variable.

We also need to convert any references to the class methods and properties, because these references use the `this` keyword to access methods and properties on the class, which will not work in the function factory. We create a visitor to find uses of `this` and remove them as seen in Code 41.

```
    funcPath.traverse({
        // this.state.value --> state.value
        MemberExpression(MEPath) {
            if (t.isThisExpression(MEPath.node.object)) {
                MEPath.replaceWith(MEPath.node.property);
            }
        }
    });
```

Code 41: Visitor used to remove uses this in an AST.

This approach can cause errors at runtime. Accessing a local variable before it is defined is not allowed in JavaScript, which means that we need to ensure that all converted class methods and properties are inserted before any of them are referenced. Class properties do not have to deal with this, because they are properties on an object and not a variable. JavaScript is function scoped, which means that every function creates its own scope. Variables in JavaScript can be declared using `var`, `let`, and `const`[1] [41, 42]. Variables declared with `var` are implicitly moved to the beginning of its enclosing scope, which is known as

[1]Const variables cannot be reassigned, however, the value can still be mutated i.e adding an item to an array stored in a const variable.

hoisting [43]. `let` and `const` were introduced with the ES2015 version of JavaScript and they are not hoisted [42] . This is important because variables declared with `var` can be referenced before they are declared because the declaration is implicitly hoisted at runtime [43, 41]. We could solve the potential reference errors by using `var` to declare all the converted class properties. However, we prefer to use the newer declaration syntax [44]. We can replicate the behaviour of `var`, by explicitly moving all declarations to the beginning of the factory function, so that we avoid any reference errors. Function declarations are also hoisted in JavaScript, so we do not need to deal with the order of function declarations [43]. The template in Code 39 reflects this design. Only React API methods and *React-compiler* specific variables are declared above the converted class methods and properties.

The final step is to copy the content of the class constructor to the factory function. The factory function itself serves as the constructor in the new design, so we can move the statements found in the constructor to the root of the factory function. React class components take a single parameter, which is the props object for that component. The factory function also takes the props of a component as its first parameter so we safely move the content of the constructor as long as we ensure that the name of the first parameter of the constructor and all its references matches the name given to the parameter of the factory function. The initial value of a component's state is commonly defined in the constructor, but it is optional. The component state is declared as a variable on line 4 in Code 39 such that the state variable is always declared even when no initial state is defined. We, therefore, have to handle the initial state declaration differently from any other class property, since it has already been declared. The state declaration has to be converted into an assignment, such that we do not get a duplicate variable declaration errors at runtime.

Next, we look at changing the action generation code introduced in section 4.2.2 to reflect the new interface. We only discuss a subset of these changes rather than systematically walking through every change, since most of the changes consist of renaming functions on components.

We designed component children to be passed into a component as a function in section 3.2.1. Each child's function encapsulated the actions required to mount and update the child in the DOM. During rerender, we removed the child and recreated it using the child's function. This design was created to simplify the code generation by avoiding the need to handle the specific element type of each child. This is the problem that the common element interface is intended to solve. Each child's function is replaced with a factory function, which hides the specific element type and exposes the common element interface. This means we can call mount, update, and unmount on children without knowing if they are a component, HTML element, DOMExpression, etc. The component children factory function array can be seen in Code 42 on line 2.

```
// component creation
let el = Container({
    text: "Hello World!",
    children: [function () {
        // declarations
        return {
            mount(root, isSibling = false) {
                el_9 = Header({
                    title: "Testing Variables"
                });
                el_9.mount(root);
            },

            update() {
                el_9.update({
                    title: "Testing Variables"
                });
            },

            unmount() {
                el_9.unmount();
            }

        };
    }()]
});

// inside component on mount
children.forEach(child => child.mount());
```

Code 42: Component children mounting using the common element interface.

We previously used the children array to contain the unmount function for each child which was called in the propsUpdate method as seen in Code 43.

```
1  // old pattern
2  propsUpdate() {
3      if (this.children.length) {
4          this.children.forEach(fn => fn());
5          this.children = [];
6      }
7
8      // rest of propsUpdate
9  }
10
11 // new pattern
12 propsUpdate() {
13     if (children.length) {
14         children.forEach(child => child.update());
15     }
16
17     // rest of propsUpdate
18 }
```

Code 43: Component children updating using the common element interface versus the previous method.

In our new design, we use the children array to contain the common element interface object for each child, such that we can call the `update` function in `propsUpdate` instead as seen on line 12.
The actions generated by the `componentDOMGenerator` for DOMExpressions are inserted directly into the DOMExpression itself, replacing the JSX it represents as seen in Code 44.

In section 4.1.3 we detailed the use of arrays for storing unmount functions for elements generated by DOMExpressions similar to the way children were handled. With the common element interface, we replace the JSX with the factory function declaration for said JSX, which allows us to encapsulate the specific element details of each element in a DOMExpression in the common element interface. This change can be seen in Code 45.

We now have an interface for reusing the DOM nodes created inside DOMExpressions, which we use for creating the non-keyed implementation designed in section 5.1.4. The implementation details can be seen in section 5.2.5.

```
// JSXExpressionContainer
{this.state.data.map(el => <div><div>{el.value}</div></div>)}

// Compiles to
(parent => {
    if (this.tempList_el_30 && this.tempList_el_30.length) {
        this.tempList_el_30.forEach(el => el());
        this.tempList_el_30 = [];
    }

    let tempList = [];
    let previousEl = null;
    let useAppend = true;
    // JSXExpression
    this.state.data.map(item => function () {
        // JSX replaced with actions
        let el_14 = document.createElement('div');

        if (useAppend) {
          previousEl ? previousEl.appendChild(el_14) : parent.appendChild(el_14);
        } else {
          previousEl ? previousEl.after(el_14) : parent.after(el_14);
        }

        tempList.push(() => el_14.remove());
        previousEl = el_14;
        let el_15 = document.createElement('div');
        el_14.appendChild(el_15);
        el_15.innerHTML = el.value;
    }());
    this.tempList_el_30 = tempList;
})(this.el_27);
```

Code 44: Example of compiling a DOMExpression with *React-compiler* with iteration 2 version.

```
// JSXExpressionContainer
{this.state.data.map(el => <div><div>{el.value}</div></div>)}

// Compiles to
(parent => {
    let tempList = [];
    let previousEl = null;
    let useAppend = true;
    // JSXExpression
    this.state.data.map(item => function () {
        // JSX replaced with factory function
        function node(props) {
            // factory function content removed for brevity
        }

        let element = node({
        value: el.value
        });
        tempList.push(element);
        element.mount(previousEl || parent, !useAppend);
    }());
    this.tempList_el_30 = tempList;
})(this.el_27);
```

Code 45: Example of compiling a DOMExpression with *React-compiler* with iteration 3 version. The factory function is named node, since the JSX from the original code is not a component.

5.2.2 Validating HTML elements

In order to validate HTML elements, we first have to establish what constitutes a valid HTML element. This is a somewhat tedious task as it, essentially, requires us to go through the MDN reference [38] and map out all valid event event listeners and attributes a given HTML element can have. The result is a dictionary with entries as depicted in Code 46.

```
1  const htmlElements = {
2      'a': {
3          isSupported: true,
4          validAttrs: globalAttributes.concat(
5              ['download', 'href', 'hreflang', 'ping',
6              'referrerpolicy', 'rel', 'target', 'type']
7              ),
8          validEvents: globalEvents
9      },
10     ...
11 }
```

Code 46: Example of an entry in the HTML element dictionary.

Prior to HTML5, each element would support a set amount of attributes and event listeners which would vary from element to element. With the introduction of HTML5, the reference was changed such that HTML elements support what is referred to as global attributes and events listeners. This means all HTML elements support these global attributes/event listeners and can have additional attributes/event listeners which are unique to that element. Due to this change, we make two lists, as indicated on line 4 and 8 in Code 46, which contain the global attributes and event listeners and then concatenate any additional attributes/event listeners, an element supports, to the lists. On line 3, we indicate whether the element is supported by *React-compiler*, as there are some elements we do not support, such as canvas and WebGl, because they use a different JavaScript API than the other HTML elements. This information is stored for *all* HTML elements.

The logic for validating an element is split into 3 separate functions — `checkEl`, `validateEl`, and `isDuplicate`. `checkEl` is the initial entry point when validating an element and is shown in Code 47.

The `checkEl` function accepts an element as a parameter with a similar structure to that of the element shown in Code 35 in section 5.1. On line 3, we check whether the `htmlType` of the element i.e. which element it is, is a key in our dictionary of HTML elements. If this is the case, we pass the valid attributes and event listeners for that particular element, as well as, the element and whether the element is supported to the `validateEl` function.

The code for the `validateEl` function is shown in Code 48.

```
1  function checkEl(el) {
2
3      if (el.htmlType in data.htmlElements) {
4          let attrs = data.htmlElements[el.htmlType].validAttrs;
5          let isSupported = data.htmlElements[el.htmlType].isSupported;
6          let events = data.htmlElements[el.htmlType].validEvents;
7
8          return validateEl(el, attrs, isSupported, events);
9
10     } else {
11         return console.log('Not a valid HTML element')
12     }
13 }
```

Code 47: The checkEl function.

The responsibility of the `validateEl` function is to go through the element and its attributes and event listeners to check whether they are valid in terms of the aforementioned dictionary of HTML elements. In order to do this, we first create an object, on line 2, which is the return value of the function. This object houses any information or errors found with the element. Thereafter, on line 13, we check whether it is an element supported by *React-compiler*. If this is the case, on lines 18 and 23, we proceed to check the attributes and event listeners of the element and use the `isDuplicate` function to check whether any attributes are duplicated, as this would be invalid. Furthermore, we iterate over the attributes and event listeners, on lines 32 and 34, and check whether they are in the list of valid attributes or event listeners for that element. Any duplicate or invalid attributes or event listeners will be added to the object, on line 2. Lastly, the function will return the object to the caller.

5.2.3 Building codeframes

The purpose of `buildCodeFrame` is to be called from within the function in Code 35, in section 5.1. It will then parse the necessary information to the validation component and use the returned object to build codeFrames [39].
An example of what the input to `buildCodeFrame` could look like is shown in Code 49.
The babel AST is decorated with source code locations, which means we have access to information such as the line and column of each attribute and event listener. This information allows us to display errors such as the image shown in fig. 5.1. The task of the `buildCodeFrame` is to take an element, after validation, identify which attributes or event listeners from the code snippet, supplied on lines 39-41 in Code 49, are invalid and build a codeFrame pointing out which are invalid.

```
function validateEl(el, validAttrs, isSupported, validEvents) {
    let errmsg = {
        type: el.htmlType,
        attrIsValid: false,
        elIsValid: false,
        invalidAttrs: [],
        invalidEvents: [],
        containsDuplicateAttrs: false,
        containsDuplicateEvents: false,
        isSupported: true,
    };

    if (!isSupported) {
        errmsg.isSupported = false;
        return errmsg;
    }

    const duplicateAttr = isDuplicate(el.attributes);
    if (duplicateAttr.length > 0) {
        errmsg.containsDuplicateAttrs = true;
        errmsg.invalidAttrs.push(duplicateAttr);
        errmsg.invalidAttrs = errmsg.invalidAttrs.flat();
    }

    const duplicateEL = isDuplicate(el.eventListeners);
    if (duplicateEL.length > 0) {
        errmsg.containsDuplicateEvents = true;
        errmsg.invalidEvents.push(duplicateEL);
        errmsg.invalidEvents = errmsg.invalidEvents.flat();
    }

    el.attributes.forEach((attr) => isValidAttribute(attr));

    el.eventListeners.forEach((event) => isValidEventlistener(event));

    return errmsg;
}
```

Code 48: The validateEl function.

```
    element: {
        id: 123,
        type: 'html',
        htmlType: 'div',
        attributes: [ {
                            name: 'id',
                            codeSnippet: {
                                start: {line: 1, column: 6},
                                end: {line: 1, column: 7}
                                }
                            },
                        {
                            name: 'class',
                            codeSnippet: {
                            }
                        },
                        {
                            name: 'tool',
                            codeSnippet: {
                            }
                        }
                ],
        eventListeners: [ {
                            name: 'click',
                            codeSnippet: {
                                start: {line: 1, column: 42},
                                end: {line: 1, column: 46}
                                }
                            },
                            {
                            name: 'stick',
                            codeSnippet: {
                                }
                            }
                        ],
        children: [],
        codeSnippet: {
            fileLocation: './app.js',
            code: '<div id="test" class="big" tool="yes"
            click={} stick={}>{this.state.count}</div>',
            location: { start: { line: 1, column: 18 } },

        }
    }
```

Code 49: Example of input to buildCodeFrame.

The code for `buildCodeFrame` is displayed in Code 50, however, we have removed some of its contents for brevity.
On line 3, we define the object, which the function returns to be used in the Ink component. Once the element has been validated, on line 13, we check whether there were any invalid attributes and iterate over them in the remaining code. For each invalid attribute, we match it to the attributes in the intial input element and build a codeFrame using the line and column information of that attribute. The same process is repeated for the event listeners. The codeFrames are built using the babel code-frame [39] function `codeFrameColumns`. It accepts a string representing a code snippet, and an object indicating the line and column of the error that occurs. The codeFrames are then added to the object on line 3 and returned from the function. The end result of running the `buildCodeFrame` function on our example input in Code 49 is shown in fig. 5.3.

```
> 1 | <div id="test" class="big" tool="yes" click={} stick={}>{this.state.count}</div>
    |                            ^
> 1 | <div id="test" class="big" tool="yes" click={} stick={}>{this.state.count}</div>
    |                                                ^
```

Figure 5.3: Output of buildCodeFrames.

5.2.4 Ink interface

The ink interface consists of a singular component named BCF. The return object from `buildCodeFrame` is passed in as props. We will not display the code for the component as there is nothing significant within it to discuss. However, an image of the end product is shown in fig. 5.4.

```
File: ./app.js  Element ID: 123
Start: 1:18     Element Type: div
End: 1:90

> 1 | <div id="test" class="big" tool="yes" click={} stick={}>{this.state.count}</div>
    |                            ^
> 1 | <div id="test" class="big" tool="yes" click={} stick={}>{this.state.count}</div>
    |                                                ^
```

Figure 5.4: Output in terminal after validating an element.

It displays the output after validating the element shown in Code 49. We display the file in which the error occurs, the location of where the code snippet starts and ends, as well as, the type of the element and its id. Furthermore, in the case that the element is unsupported or invalid, an error message stating this is displayed in place of the codeFrames displayed in fig. 5.4. If an element is supported, but has invalid attributes or event listeners, the errors are displayed as shown in fig. 5.4.

```
function buildCodeFrame(codeFrameData) {
    //store codeframes
    let codeFrames = {
        ...
    };

    //map out required info for the validation function
    const validationEl = {
        ...
    }

    //validate the element
    const validatedEl = validate.checkEl(validationEl);

    //check whether there are any invalid attributes
    if (validatedEl.attrIsValid === false) {
        validatedEl.invalidAttrs.forEach(attr => {
            codeFrameData.element.attributes.forEach(attr2 => {
                if (attr2.name === attr) {
                    codeFrames.codeFrame.push(createAttibuteCodeFrame(attr, attr2));
                }
            });
        });
    }

    //check whether there are any invalid events
    if (validatedEl.elIsValid === false) {
        validatedEl.invalidEvents.forEach(event => {
            codeFrameData.element.eventListeners.forEach(event2 => {
                if (event2.name === event) {
                    codeFrames.codeFrame.push(createEventlistenerCodeFrame(event, event2));
                }
            });
        });
    }

    return codeFrames;

}
```

Code 50: The buildCodeFrame function.

5.2.5 Non-keyed Mode

A major part of implementing our non-keyed mode has been completed in section 5.2 by providing the necessary interface for reusing existing DOM nodes. The code in Code 51 shows the compiled output with the non-keyed implementation of the DOMExpression from Code 45.

We use the factory function named `node` on line 12 to create a new instance of an element. We can then use the common element interface to mount, update and unmount the element in the non-keyed implementation. The `tempList_1` variable is used as the elements array from algorithm 1 introduced in section 5.1.4. The variable is delcared in the component itself, rather than inside the DOMExpression, so it can be referenced in other update functions, such as a `setState` function. We keep the `previousEl` and `useAppend` to control insertion of DOM nodes as discussed in section 4.1.3. Lines 16 to 30 are equivalent to those seen inside the for loop on line 3 in algorithm 1, but we have added the specific code for calling the factory function `node` on line 22. We use the `slice` function to create a new array without the unused elements on line 38 in Code 51.

```
// JSXExpressionContainer
{this.state.data.map(el => <div><div>{el.value}</div></div>)}

// Compiles to
(parent => {
    let position = 0;
    let previousEl = null;
    let useAppend = false;
    // JSXExpression
    this.state.data.map(item => function () {
        // JSX replaced with factory function
        function node(props) {
            // factory function content removed for brevity
        }

        if (tempList_1[position]) {
            tempList_1[position].update({
                value: el.value
            });
            previousEl = tempList_1[position].getContainer();
        } else {
            let element = node({
                value: el.value
            });
            tempList_1.push(element);
            element.mount(previousEl || parent, !useAppend);
            previousEl = element.getContainer();
        }

        position += 1;
    }());

    if (tempList_1.length - 1 >= position) {
        for (let i = position; i < tempList_1.length; i++) {
            tempList_1[i].unmount();
        }

        tempList_1 = current.slice(0, position);
    }
})(this.el_27);
```

Code 51: Example of compiling a DOMExpression with *React-compiler* with iteration 3 version non-keyed mode. The factory function is named node, since the JSX from the original code is not a component.

5.3 Results

The objective for iteration 3 was to refactor the codebase towards a more uniform API for elements, as well as, building a foundation for displaying errors during compilation to users. Furthermore, we implemented a non-keyed implementation of *React-compiler* in order to investigate whether the performance in the Krausest benchmarks would improve. This section will focus on the latter objective of iteration 3, as the two former objectives do not involve any performance optimisations.

The microbenchmarks do show a bias towards keyed/non-keyed implementations, as shown in section 3.3 and section 4.3, meaning we do expect to see improvements due to the ability to reuse DOM nodes in DOMExpressions, which is what is benchmarked by Krausest. However, we also expect the issues presented in section 3.3 and section 4.3 to be present in the non-keyed implementation and are, therefore, uncertain as to the degree of improvement we can expect to see.

The results of running the microbenchmarks can be seen in appendix C. Suprisingly, we do not see the peformance issues in the it3 results as we did in section 4.3. In fact, the non-keyed implementation outperforms all React implementations and keyed Svelte implementations, which is a larger improvement than we expected. The "partial update" benchmark results can be seen in fig. 5.5.

This was one of the benchmarks using 16x CPU slowdown and we can see the poor results from iteration 1 and 2. However, it is also clear that the CPU slowdown does not affect the iteration 3 version of *React-compiler*, which performs similarly to the React non-keyed implementations. In fact, the it3 version has seen a 1755% increase in performance, when compared to it2, meaning the that it takes it2 17.5 times as long to to update every 10th row in a table of 1.000 rows. These results seem to indicate that the changes we have made in the this iteration has solved the performance issues from the previous iterations. Additionally, it seems that the performance gained from the iteration 3 implementation does not come with an increased cost to memory consumption, as shown in fig. 5.6. In fact, this implementation outperforms *all* of our previous iterations in both the memory and startup categories.

We did perform a single run of the microbenchmarks after having completed the common element interface detailed in section 5.2.1, which showed no impact on the performance when comparing the factory function code to the class based code. The major change between iteration 2 and 3 is the DOM reuse for DOMExpressions in the compiled applications of *React-compiler*. We did not expect this change to improve the results to the extent that it did. We already implemented DOM node reuse in the pre-generated rows implementations, which also had performance issues, so it seems unlikely that this change alone is the reason for the improvements. Our theory from section 4.3 was that performance issues was cause by overloading the browser due to the number of DOM operations. We still think that the perfor-

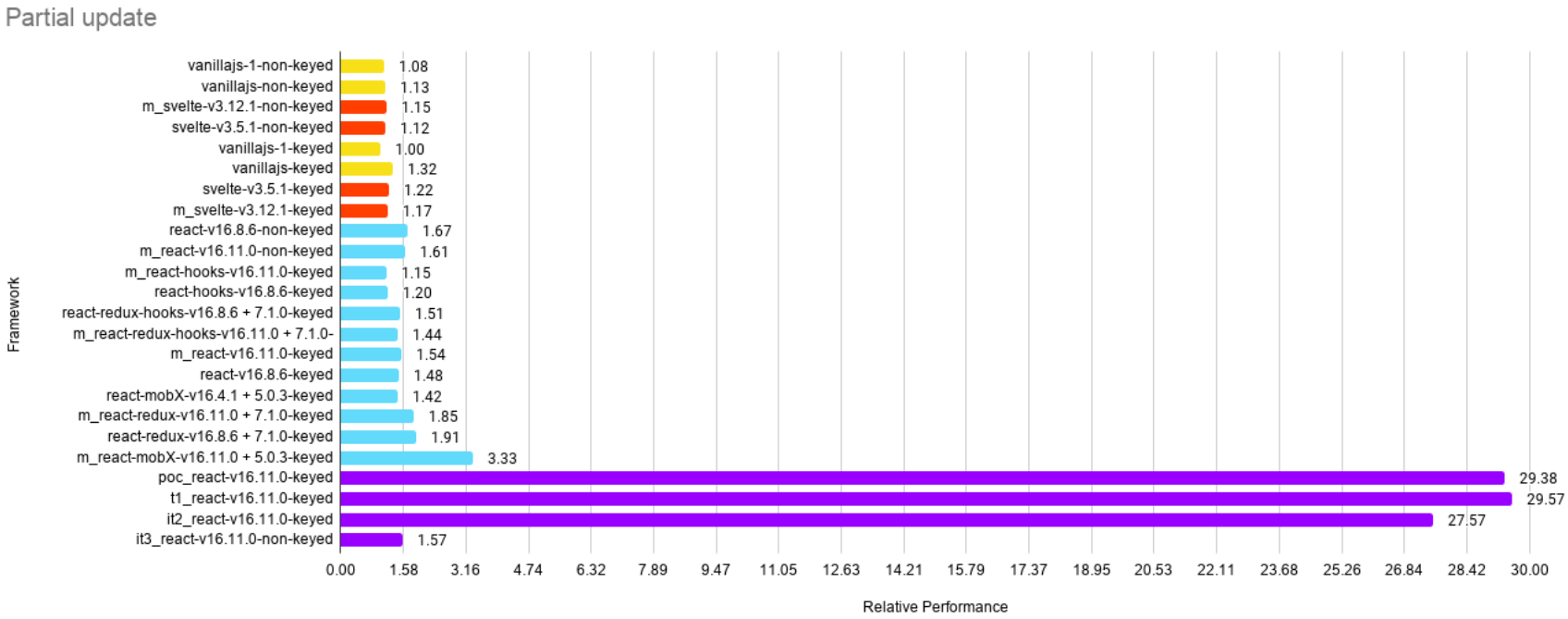


Figure 5.5: Update every 10th row of 1000 rows. The values indicate relative performance, where 1.0 is the best. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

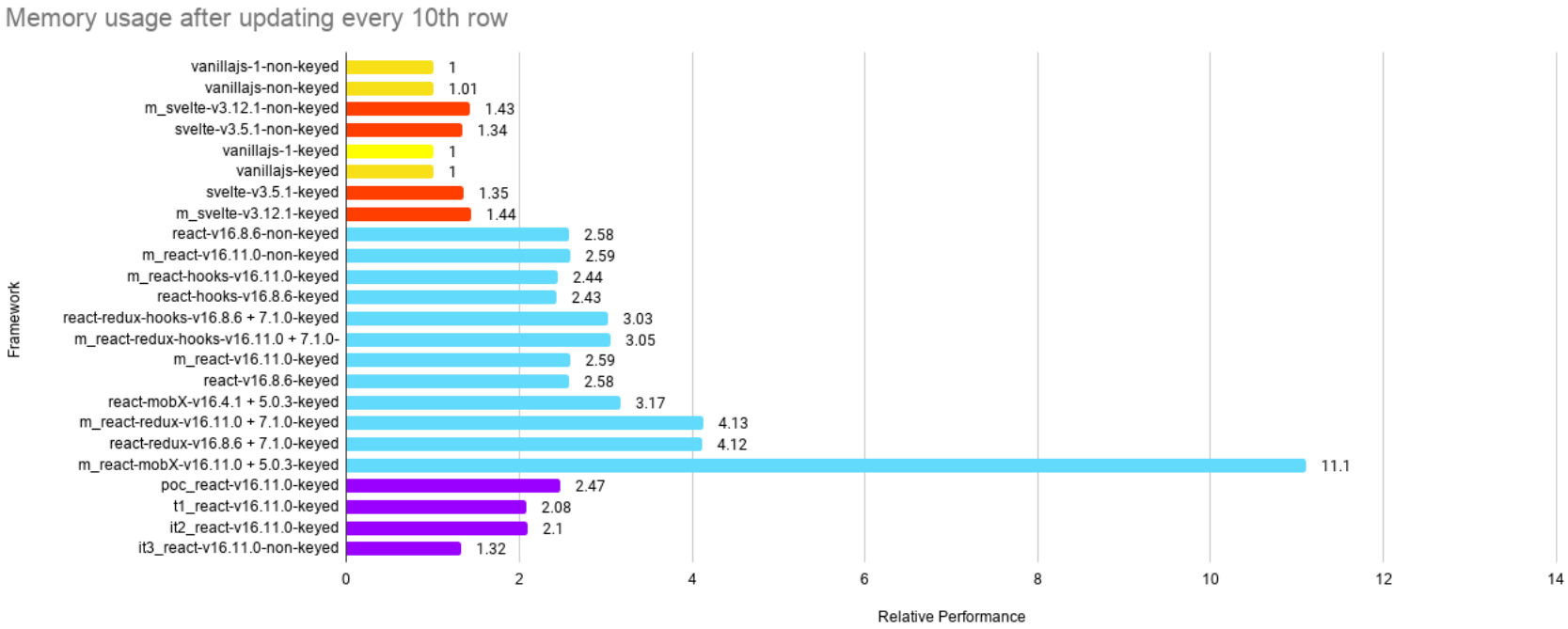


Figure 5.6: Memory usage after updating every 10th row of 1000 rows. The values indicate relative performance, where 1.0 is the best. Svelte implementations are coloured red, React are blue, and vanillaJS are yellow. Our implementations are purple.

mance issues in section 4.3 are cause by overloading the browser, because the performance was related to the CPU slowdown. However, it is unlikely that the browser is unable to handle the number of DOM operations triggered by the applications, because the pre-generated rows implementations perform the same number of DOM operations as the non-keyed mode from this iteration, since they are both using the `componentDOMGenerator` for generating the actions in the `update` method. It is unclear to us at this time, what exactly causes the performance issues in section 4.3, while it does appear to be related to the speed of the CPU, it does not appear to relate to the number of DOM operations generated by *React-compiler*. We discuss this further in section 6.2.

This project set out to improve the performance of the PoC from [1]. With the conclusion of this iteration, we have managed to exceed our own expectations by outperforming React in most of the microbenchmarks. When we started [1], we posed the question of whether it was possible to compile a React application to more performant JavaScript code. We argue, with the results of iteration 3, we have succeeded and shown that it is possible to improve the performance of React applications, without rewriting them in a new framework, like Svelte.

Chapter 6

Discussion

In this chapter, we discuss the project and its contents. Specifically, we discuss the decisions we have made and their impact. Furthermore, we return to various subjects mentioned througout the report, such as the notion of scheduling. Lastly, we discuss next steps for the project and which direction one could take *React-compiler* after the completion of this project.
To begin our discussion, it is relevant to look back at where we started with this project. Initially, we created a mindmap in order to distinguish between the important aspects of the project and decide which to pursue. The mindmap is shown in fig. 6.1.

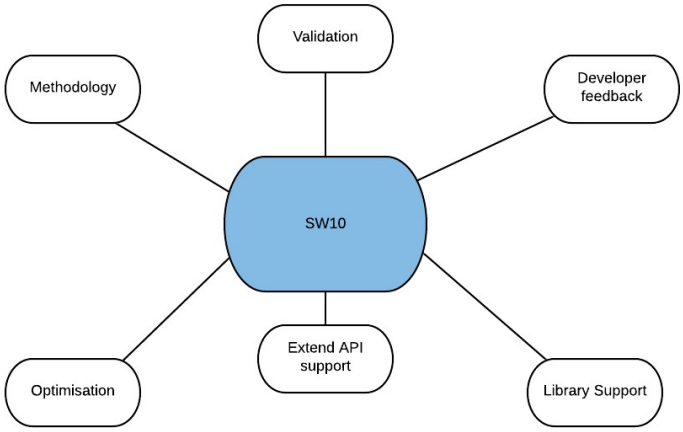


Figure 6.1: Project Mindmap.

When we ended [1], we had a lot of aspirations for *React-compiler*, however, we also knew we would not have time to complete all of them within the timeframe of this semester.

Therefore, we prioritised items from the mindmap in order to produce the most value as discussed in chapter 1. Now that we are at the end of this project, it is relevant to ask, did we make good choices, and what are the opportunities to move forward?
To structure our discussion, we have divided our thoughts into two areas of focus, as outlined in chapter 1, based on fig. 6.1:

- **Methodology:** Add additional implementations to compare the performance of keyed/non-keyed to other approaches.
- **Performance Optimisations:** Expand on the PoC from [1] to improve the performance of the compiled output.

In sections 6.1 and 6.2, we describe our progress within each category, discuss the choices we have made, and comment on our process. Finally, we look at potential directions for future projects in section 6.3.

6.1 Methodology

In this section, we discuss our decisions regarding our methodology. Specifically, we discuss — (1) the exclusion of macrobenchmarks, (2) the creation of the pre-generated rows implementations, (3) the prioritisation of methodology.

6.1.1 Excluding Macrobenchmarks

In section 1.2, we detailed the issues we had with our methodology from [1] and, therefore, set out to improve upon it. We wrote off the existing macrobenchmarks for two main reasons. Firstly, we concluded in [1], that the results from our macrobenchmarks did not provide valuable insight into the performance of the frameworks. Secondly, the implementations in the macrobenchmarks utilised third party libraries, which are not supported by the PoC. To support the macrobenchmarks in *React-compiler*, would require us to dedicate time to implement support for multiple libraries which would only enable us to compile additional applications, rather than spending time on improving the performance of *React-compiler*. We assess that this is still the correct decision. We argue that the macrobenchmarks would not have enabled us to improve the performance of the compiled applications further since we have not attempted all potential performance optimisations identified by the microbenchmarks.

6.1.2 Creating Pre-generated Rows Tool

The implementations in our microbenchmarks dynamically generate rows based on data using either keyed or non-keyed implementation as detailed in chapter 2. In chapter 2 we created the pre-generated rows implementations, which have the rows hardcoded into the

code rather than generating them dynamically. We described the design and implementation of a tool for creating pre-generated rows implementations. The Krausest tool [12] used for our microbenchmarks only benchmarks a subset of a framework's capabilities. The pre-generated rows implementations are designed to challenge the inherent biases within Krausest. The results of the pre-generated rows implementations helped us identify performance issues with the application compiled with both iteration 1 and 2 versions of *React-compiler* as discussed in section 4.3. While we did not see the expected performance of the compiled applications, we showed that React performed better in pre-generated rows compared to the microbenchmarks. We, therefore, argue, that the pre-generated rows implementations did highlight the shortcomings of Krausest and how it does not give a full representation of how a framework performs overall.

6.1.3 Prioritisation of Methodology

It is clear from this report, that our focus has not been on the methodology. Most of our resources have been dedicated to developing *React-compiler* and, therefore, the methodology has been prioritised lower. This is both a result of our interest mainly being on developing *React-compiler*, but also due to the lack of a widely accepted standard for benchmarking front-end frameworks in the industry, which is investigated in [1]. We have established that Krausest is not an accurate representation of the performance of a framework, however, there is no alternative that benchmarks the same aspects as Krausest. The issue with this field of research is that there is no accepted methodology for benchmarking front-end frameworks and, thus, it has been up to us to determine the correct method for our project. We have found that developing a complete methodology, which takes all or, at least more, aspects of a framework's performance into account, compared to Krausest, is an extensive task. *React-compiler* is our priority and it is not feasible for us to develop a compiler while also developing the correct and complete method for evaluating its performance in our given timeframe.

6.2 Performance Optimisations

In this project, we transitioned our PoC, designed as a Babel plugin, into *React-compiler*, a standalone compiler. *React-compiler* is built on top of Babel, to avoid implementing a JavaScript parser, AST, and code generator from scratch. In [1] we ran into limitations with the PoC, because it was implemented as a Babel plugin, which limits our control of the compilation process. Specifically, we had to implement the PoC plugin as a single-pass compiler, which resulted in more complex code, as discussed in section 3.1.7. Using Babel as a foundation has provided us with some "nice to have" tools, such as the Babel

code-frame library used in chapter 5. We are confident this decision was correct, seeing as the development process was made significantly easier. Furthermore, we gained access to additional tools and by using Babel as the foundation, we could focus on optimisation efforts instead of compiler basics.

As part of *React-compiler* we implemented the following performance optimisations:

- **DOM Node Reuse:** In chapter 3 we designed and implemented DOM node reuse for most cases with the exception of DOMExpressions.
- **Dependency Tracking of Props and State:** Reduce the number of DOM operations performed during rerender by tracking component props and state use in the JSX and only updating relevant DOM nodes as detailed in chapter 4.
- **Non-keyed Mode:** Create a non-keyed implementation to allow DOM node reuse in DOMExpressions as detailed in section 5.1.1.

In the following subsections, we will outline and discuss our decisions for the aforementioned optimisations.

6.2.1 DOM Node Reuse

We decided to implement DOM node reuse at the beginning of this project because it was not supported in the PoC and both Svelte and React reuse DOM nodes. Furthermore, we expected that updating DOM nodes was more performant that recreating them on every rerender as we did in the PoC. This is further supported by our results in section 3.3 which indicate that *all* other frameworks, that use DOM node reuse, perform better than the PoC. We expected our pre-generated rows implementations to show better performance when reusing DOM nodes in *React-compiler* compared to the PoC. The pre-generated rows implementations are specifically designed to benchmark the performance of DOM node reuse without relying on a keyed/non-keyed implementation. However, the performance difference was overshadowed by other performance issues, which meant we were unable to determine the performance impact of reusing DOM nodes.

6.2.2 Dependency Tracking of Props and State

We were inspired by Svelte to create dependency tracking of a component's props and state in order to improve performance by reducing the number of DOM operations being performed during rerender. Dependency tracking is highlighted by the authors of Svelte [45], as one of the primary ways that the framework achieves its performance. In the results of iteration 1, we saw that Svelte is only beaten by the vanillaJS implementations, which motivated us to implement a similar approach. While React optimises its DOM operations at runtime using the Reconciliation process outlined in section 1.1.1, it adds overhead to rerendering which reduces performance. Svelte does this at compile time, which is what inspired

the dependency tracking in *React-compiler*. However, we did not see any performance improvements in our benchmarks, which we determined was caused by the way the React applications in the microbenchmarks were implemented as discussed in section 4.3. The applications would have to be refactored in order to benefit from the dependency tracking implemented in chapter 4.

6.2.3 Non-keyed Mode

In chapter 5 we achieved DOM node reuse for the DOMExpressions by implementing a non-keyed mode. We expected this to result in improved performance in the microbenchmarks now that our compiled application functions similarly to the other frameworks. This optimisation resulted in the greatest performance improvement to *React-compiler*. *React-compiler* outperforms *all* React versions and is on par with Svelte implementations in some benchmarks. We also alleviated the performance issues from iteration 1 and 2 related to CPU throttling.

6.2.4 Evaluation

We are confident implementing DOM node reuse was the right call, as it serves as a foundation for our implementation of a non-keyed mode. Furthermore, our results show that reusing DOM nodes is more performant than recreating them as we did in the PoC. We still think that dependency tracking is a good idea and that we still expect it to improve performance. If our goal is to perform better in the Krausest benchmarks, implementing dependency tracking was not the most efficient decision, as the results do not show an improvement. However, if our goal is to create a more complete compiler, we assess it to be the correct decision. This further supports the fact that Krausest does not capture the overall performance of a framework, but a narrow subset. The decision, essentially, boils down to two things — (1) do we optimise *React-compiler* to perform better in Krausest, which we know does not give a full picture of a framework's performance, or (2) optimise *React-compiler* to perform better in general use cases of a framework, which is not captured in the Krausest benchmarks.
It is clear from the results of iteration 3, that the benchmarks in Krausest benefit significantly from implementing a performant keyed/non-keyed mode. However, it is unlikely that other applications, such as the real-world applications used in [1] for macrobenchmarks, would benefit as much from a non-keyed implementation as we have seen in Krausest. While real-world applications do utilise keyed/non-keyed mode, they also use other aspects of a framework, which means that an improvement in DOMExpressions only partially contributes of the overall performance of the applications. If we had implemented our non-keyed mode as part of iteration 1, we would likely have had more time to work on benchmarking other aspects of a framework's capabilities. The results would have made it clear, that there was

limited performance to be gained, in the microbenchmarks, by focusing on optimising DOM-Expressions further in the project. While we could have saved time by implementing non-keyed mode in iteration 1, the specific implementation of *React-compiler* in iteration 1 and 2 identified issues such as CPU slowdown and scheduling in general.
We assess that we would not have identified these issues had we not proceeded the way we did. This issue is a bottleneck, that we did not anticipate to encounter, and the knowledge of this issue can prevent problems if work was to continue on *React-compiler*. Our theory in section 4.3 was that the browser running with the slower CPU speeds was overloaded by the number of DOM operations generated by the compiled applications. The results in section 5.3 suggest that this is not the case, since the it3 implementation did not have these issues. It is clear that we have issues with CPU slowdown in some of our implementations, but it is unlikely that performing too many DOM operations is the cause. Our assessment is that the it3 implementation has not fixed these issues, but instead that it is not affected in the same way as it1 and it2. We need to dig deeper into the performance of the browser during execution of the benchmarks in order to determine what causes the performance issues. Ultimately, in terms of achieving short term results, implementing non-keyed in iteration 1 could have allowed us to shift our focus towards improving our methodology and broadening the aspects of the frameworks we benchmark.

6.3 Future Work

In section 1.2, we presented 6 potential avenues we could pursue in this project. Ultimately, we postponed 4 of them to future work as we wanted to focus on performance optimisations and methodology.
In iteration 3, we decided to begin development on developer feedback, which was initially discarded. While this is not a major part of this project, we deemed it to be a good addition to *React-compiler* as good error messages save a lot of debugging time when working with a compiler. It is easy for us, as the authors of *React-compiler*, to find the causes of errors, however, that is not the case for other potential users. Nonetheless, it is a vital part of *React-compiler* if it is to be used by React developers. Some compilers have cryptic error messages, which can be a hurdle for developers when working with a compiler. Therefore, a project working on extending *React-compiler* could focus on the usability of the compiler from the perspective of other React developers.
During development of *React-compiler*, we experimented with a tool, which would validate if two applications had identical HTML outputs. This was done to verify whether the output of *React-compiler* modified the HTML of the original React application, as this should not be the case. In section 3.3 we saw that the DOMExpression implementation we had implemented, broke the Krausest webdriver. The problem was that the HTML output of

React-compiler was different than the original React applications HTML output. However, we did not find any further use for it in this project, due to the limited complexity of the applications used, which made it easy to validate the output by hand. We argue that in the future, a tool that validates the HTML output could be useful when testing *React-compiler* on larger and more complex applications, where the HTML tree is significantly larger, making it tedious to validate by hand. Nevertheless, as the tool was not used, we decided against presenting it in this report.

The current state of *React-compiler* improves the performance of the compiled applications in the microbenchmarks. Essentially, we have created a compiler which allows React developers to obtain more performant code without rewriting the application in another framework. We argue this is a valuable contribution because, while front-end frameworks like React and Svelte share many concepts, it still requires a time investment to become familiar with a new framework. For instance, both React and Svelte use components to structure applications, but use a different syntax for creating components. While *React-compiler* has achieved better performance than React, it is still limited in its use-cases. For instance, we do not support the use of external libraries nor do we support all HTML elements. The next steps should involve extending the support of *React-compiler*, which is one of the branches of our mindmap in fig. 6.1, without degrading the performance we have obtained. This would enable React developers to use *React-compiler* on real world applications, where an increase in performance could have monetary benefits. Furthermore, it also means that front-end developers may not disregard React as their framework of choice due to its performance compared to other frameworks such as Svelte.

Throughout this report, we have commented on the biases within our microbenchmarks. We added the pre-generated rows implementations in order to benchmark other aspects of the frameworks than keyed/non-keyed mode. However, we were unsuccessful in capturing what we hoped with the pre-generated rows implementations due to the performance issues with the compiled applications in iteration 1 and 2. A future project based on our work could involve identifying better ways of benchmarking more functionality of a framework, which extends what Krausest benchmarks. This project, along with [1], has shown that working on a methodology for benchmarking front-end frameworks, more extensively than Krausest, is a significant undertaking. A large amount of our research in [1], was dedicated to investigating and developing a methodology for benchmarking front-end frameworks. Furthermore, we concluded in [1] that expanding our methodology would be a project in and of itself, which was the reason for not focusing on it in this project. Even with the extensions made in this project, the methodology is far from complete, we still only benchmark a subset of what a framework is capable of and the developers of Krausest encounter many issues in maintaining the tool itself [12, 31, 32].

Chapter 7

Conclusion

In this project we have designed and implemented *React-compiler*, a compiler written in JavaScript for compiling applications written in React, a front-end framework for the web, into more performant JavaScript code. *React-compiler* enables React developers to obtain better performing applications, without rewriting them in a different front-end framework.

The main contribution of this project is *React-compiler*, which has the following features:

- Removes the React runtime entirely, thus reducing the total file size of the compiled application, improving load times, and reducing memory consumption.
- Replaces the React Reconciliation algorithm, by computing the dependencies of JSX elements in React components at compile time.
- Tracks dependencies of JSX elements to generate DOM update functions, which only updates DOM nodes whose dependencies have been modified.

The compiled applications outperform React by 28% - 1740% in 5 out of 10 benchmarks in the DOM operation category of the Krausest benchmarking tool [12]. In the 5 benchmarks where *React-compiler* is worse, it is 6% - 40% slower. Furthermore, when compiling an application with *React-compiler*, the total file size of the application is reduced by up to 87% in our benchmarks.

The current state of *React-compiler* does have some limitations in terms of which React features it is capable of compiling such as React hooks, portals, context. Multiple return expressions in a React component's render method are also not supported. Using third party libraries such as Redux or MobX, does not work if the library requires its input to be a React component, since the compiled application has removed React entirely.

This project is a continuation of the preliminary work in [1]. In [1] we built a proof of concept compiler to investigate whether it was possible to compile React applications to vanilla JavaScript. Additionally, we proposed a methodology for benchmarking front-end frame-

works as there was no standard in the industry for comparison. [1] concluded that it was possible to build a compiler for React, however, with mixed results. The goal of this project was to expand on the proof of concept compiler to improve the performance of the compiled applications. During development, we encountered a number of challenges with the performance of the compiled applications. In iteration 1, we could not run our microbenchmarks due to how we had implemented DOMExpressions. The cause for this was that the benchmarking tool could not locate the correct elements using XPath. In iteration 2, we fixed the XPath issue, however, the microbenchmark results did not show the performance we had expected to achieve with the optimisations made in the first two iterations. We found a correlation between the performance issues and the amount of CPU slowdown used in the individual benchmarks. Disabling the CPU slowdown showed, that the compiled applications performed as expected. We theorised that the slower CPU was unable to handle the large number of DOM operations being performed in the benchmarks, which caused the browser to overload.

Our conclusion was that we needed to implement a scheduler for the compiled applications at runtime to prioritise DOM operations, especially on slower CPU's. This was based on the performance of React, which uses a scheduler at runtime, and managed to perform better than the compiled applications both with and without CPU slowdown. We have not implemented a scheduler in this project, however, in iteration 3 we created a non-keyed implementation that resulted in better performance than React in the majority of benchmarks. These results did not show any performance issues related to CPU slowdown despite the lack of a scheduler.

We argue that *React-compiler* shows that it is possible to improve the performance of the proof of concept compiler, and, in some cases, exceed the performance of the original React application.

Appendix A

Microbenchmark Results Iteration 1

	Create rows (ms)	*Replace all rows (ms)*	*Partial update (ms)*	*Select row (ms)*	*Swap rows (ms)*	*Remove row (ms)*
vanillajs-1-non-keyed	130.7 (1.14)	47.4 (1.01)	215.9 (1.08)	25.8 (1.00)	31.9 (1.09)	45.5 (1.11)
vanillajs-non-keyed	133.4 (1.16)	51.2 (1.09)	225.9 (1.13)	26.1 (1.01)	32.8 (1.12)	46.3 (1.13)
m_svelte-v3.12.1-non-keyed	157.9 (1.37)	47.1 (1.00)	230.5 (1.15)	35.4 (1.37)	29.4 (1.00)	41.0 (1.00)
svelte-v3.5.1-non-keyed	149.8 (1.30)	47.0 (1.00)	225.4 (1.12)	37.0 (1.43)	31.2 (1.06)	41.2 (1.01)
vanillajs-1-keyed	114.9 (1.00)	126.8 (2.70)	200.4 (1.00)	26.7 (1.04)	48.8 (1.66)	45.4 (1.11)
vanillajs-keyed	120.9 (1.05)	139.9 (2.98)	265.2 (1.32)	38.8 (1.51)	50.6 (1.72)	46.9 (1.14)
svelte-v3.5.1-keyed	146.8 (1.28)	156.4 (3.33)	244.3 (1.22)	38.3 (1.48)	58.8 (2.00)	46.6 (1.14)
m_svelte-v3.12.1-keyed	150.0 (1.31)	158.7 (3.38)	234.2 (1.17)	38.3 (1.49)	58.0 (1.98)	46.3 (1.13)
react-v16.8.6-non-keyed	191.3 (1.66)	51.0 (1.09)	333.8 (1.67)	98.5 (3.82)	37.5 (1.28)	47.6 (1.16)
m_react-v16.11.0-non-keyed	194.1 (1.69)	50.2 (1.07)	323.5 (1.61)	106.6 (4.13)	38.6 (1.31)	48.6 (1.19)
m_react-hooks-v16.11.0-keyed	170.1 (1.48)	146.8 (3.13)	230.6 (1.15)	44.5 (1.73)	546.6 (18.61)	46.2 (1.13)
react-hooks-v16.8.6-keyed	169.6 (1.48)	148.0 (3.15)	241.4 (1.20)	44.0 (1.71)	565.1 (19.25)	46.2 (1.13)
react-redux-hooks-v16.8.6 + 7.1.0-keyed	180.0 (1.57)	150.0 (3.19)	303.1 (1.51)	47.2 (1.83)	560.3 (19.08)	46.8 (1.14)
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	175.5 (1.53)	149.5 (3.18)	289.3 (1.44)	51.5 (2.00)	552.5 (18.82)	46.8 (1.14)
m_react-v16.11.0-keyed	189.2 (1.65)	162.6 (3.46)	308.6 (1.54)	91.5 (3.55)	556.8 (18.96)	47.5 (1.16)
react-v16.8.6-keyed	186.2 (1.62)	165.0 (3.51)	296.5 (1.48)	93.3 (3.62)	574.1 (19.55)	48.4 (1.18)
react-mobX-v16.4.1 + 5.0.3-keyed	220.2 (1.92)	171.5 (3.65)	284.3 (1.42)	72.1 (2.80)	562.3 (19.15)	49.6 (1.21)
m_react-redux-v16.11.0 + 7.1.0-keyed	223.5 (1.95)	179.2 (3.82)	371.1 (1.85)	51.4 (1.99)	560.0 (19.07)	58.1 (1.42)
react-redux-v16.8.6 + 7.1.0-keyed	210.2 (1.83)	176.6 (3.76)	381.9 (1.91)	66.2 (2.57)	565.0 (19.24)	57.6 (1.41)
m_react-mobX-v16.11.0 + 5.0.3-keyed	747.3 (6.50)	677.9 (14.43)	667.7 (3.33)	332.7 (12.90)	1,410.3 (48.03)	61.8 (1.51)
poc_react-v16.11.0-keyed	216.2 (1.88)	206.4 (4.40)	5,889.0 (29.38)	5,985.7 (232.11)	940.3 (32.02)	200.0 (4.88)

Table A.1: Benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Create many rows (ms)	*Append rows (ms)*	*Clear rows (ms)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,197.4 (1.00)	272.6 (1.00)	157.6 (1.02)	1.05
vanillajs-non-keyed	1,213.8 (1.01)	275.4 (1.01)	158.9 (1.03)	1.08
m_svelte-v3.12.1-non-keyed	1,552.0 (1.30)	342.8 (1.26)	217.6 (1.41)	1.2
svelte-v3.5.1-non-keyed	1,528.9 (1.28)	333.6 (1.23)	221.8 (1.44)	1.2
vanillajs-1-keyed	1,201.8 (1.00)	272.1 (1.00)	154.2 (1.00)	1.2
vanillajs-keyed	1,235.5 (1.03)	285.2 (1.05)	166.3 (1.08)	1.34
svelte-v3.5.1-keyed	1,494.7 (1.25)	336.8 (1.24)	227.0 (1.47)	1.51
m_svelte-v3.12.1-keyed	1,530.5 (1.28)	347.0 (1.27)	221.4 (1.44)	1.51
react-v16.8.6-non-keyed	1,952.2 (1.63)	415.1 (1.53)	221.5 (1.44)	1.58
m_react-v16.11.0-non-keyed	1,920.7 (1.60)	419.6 (1.54)	232.7 (1.51)	1.6
m_react-hooks-v16.11.0-keyed	1,721.7 (1.44)	362.5 (1.33)	219.4 (1.42)	2.01
react-hooks-v16.8.6-keyed	1,733.1 (1.45)	363.9 (1.34)	207.1 (1.34)	2.01
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1,762.4 (1.47)	353.6 (1.30)	222.4 (1.44)	2.11
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1,793.4 (1.50)	348.7 (1.28)	231.3 (1.50)	2.12
m_react-v16.11.0-keyed	1,917.1 (1.60)	405.4 (1.49)	223.7 (1.45)	2.37
react-v16.8.6-keyed	1,937.2 (1.62)	399.6 (1.47)	219.0 (1.42)	2.37
react-mobX-v16.4.1 + 5.0.3-keyed	2,007.9 (1.68)	463.5 (1.70)	305.2 (1.98)	2.49
m_react-redux-v16.11.0 + 7.1.0-keyed	2,075.3 (1.73)	464.0 (1.70)	258.4 (1.68)	2.49
react-redux-v16.8.6 + 7.1.0-keyed	2,086.4 (1.74)	458.9 (1.69)	233.2 (1.51)	2.52
m_react-mobX-v16.11.0 + 5.0.3-keyed	44,458.7 (37.13)	1,655.0 (6.08)	514.1 (3.33)	8.45
poc_react-v16.11.0-keyed	2,015.5 (1.68)	930.5 (3.42)	181.3 (1.18)	7.31

Table A.2: Coninued benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Consistently interactive (ms)	*Script bootup time (ms)*	*Total kilobyte weight (Kb)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,876.1 (1.00)	16.0 (1.00)	144.2 (1.00)	1
vanillajs-non-keyed	1,875.6 (1.00)	16.0 (1.00)	147.0 (1.02)	1.01
m_svelte-v3.12.1-non-keyed	1,875.6 (1.00)	16.0 (1.00)	145.3 (1.01)	1
svelte-v3.5.1-non-keyed	1,876.2 (1.00)	16.0 (1.00)	145.2 (1.01)	1
vanillajs-1-keyed	1,875.7 (1.00)	16.0 (1.00)	143.6 (1.00)	1
vanillajs-keyed	1,875.9 (1.00)	16.0 (1.00)	149.5 (1.04)	1.01
svelte-v3.5.1-keyed	1,905.6 (1.02)	16.0 (1.00)	145.8 (1.01)	1.01
m_svelte-v3.12.1-keyed	1,876.1 (1.00)	16.0 (1.00)	147.5 (1.03)	1.01
react-v16.8.6-non-keyed	2,502.3 (1.33)	16.0 (1.00)	261.1 (1.82)	1.34
m_react-v16.11.0-non-keyed	2,652.4 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
m_react-hooks-v16.11.0-keyed	2,553.4 (1.36)	16.0 (1.00)	270.8 (1.89)	1.37
react-hooks-v16.8.6-keyed	2,500.9 (1.33)	16.0 (1.00)	260.1 (1.81)	1.34
react-redux-hooks-v16.8.6 + 7.1.0-keyed	2,654.6 (1.42)	16.0 (1.00)	277.7 (1.93)	1.4
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	2,702.1 (1.44)	16.0 (1.00)	288.5 (2.01)	1.43
m_react-v16.11.0-keyed	2,653.9 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
react-v16.8.6-keyed	2,502.1 (1.33)	16.0 (1.00)	261.8 (1.82)	1.34
react-mobX-v16.4.1 + 5.0.3-keyed	2,851.3 (1.52)	64.0 (4.00)	313.6 (2.18)	2.37
m_react-redux-v16.11.0 + 7.1.0-keyed	2,701.9 (1.44)	59.7 (3.73)	290.9 (2.03)	2.22
react-redux-v16.8.6 + 7.1.0-keyed	2,656.7 (1.42)	59.1 (3.69)	280.1 (1.95)	2.17
m_react-mobX-v16.11.0 + 5.0.3-keyed	2,965.9 (1.58)	73.2 (4.57)	341.8 (2.38)	2.58
poc_react-v16.11.0-keyed	1,905.6 (1.02)	16.0 (1.00)	149.3 (1.04)	1.02

Table A.3: Benchmark results for Startup metrics. Measured in milliseconds (ms) and kilobytes (Kb) depending on the benchmark. The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Ready memory (MBs)	*Run memory (MBs)*	*Update each 10th row (MBs)*	*Replace 1k rows (MBs)*	*Creating/Clearing 1k rows (MBs)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	1.8 (1.00)	2.4 (1.05)	1.01
vanillajs-non-keyed	1.1 (1.00)	1.6 (1.01)	2.0 (1.01)	1.8 (1.03)	2.4 (1.04)	1.02
m_svelte-v3.12.1-non-keyed	1.1 (1.02)	2.5 (1.58)	2.8 (1.43)	2.6 (1.45)	2.4 (1.05)	1.29
svelte-v3.5.1-non-keyed	1.1 (1.02)	2.3 (1.47)	2.7 (1.34)	2.5 (1.42)	2.3 (1.00)	1.23
vanillajs-1-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	2.3 (1.30)	2.4 (1.05)	1.06
vanillajs-keyed	1.1 (1.01)	1.6 (1.01)	2.0 (1.00)	2.3 (1.32)	2.3 (1.02)	1.07
svelte-v3.5.1-keyed	1.1 (1.02)	2.4 (1.49)	2.7 (1.35)	2.9 (1.66)	2.3 (1.00)	1.28
m_svelte-v3.12.1-keyed	1.1 (1.02)	2.5 (1.60)	2.9 (1.44)	3.1 (1.77)	2.4 (1.06)	1.35
react-v16.8.6-non-keyed	1.3 (1.23)	4.4 (2.74)	5.1 (2.58)	6.5 (3.69)	3.7 (1.61)	2.2
m_react-v16.11.0-non-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.5 (3.72)	3.7 (1.63)	2.21
m_react-hooks-v16.11.0-keyed	1.4 (1.28)	4.1 (2.56)	4.9 (2.44)	5.0 (2.87)	3.1 (1.37)	1.99
react-hooks-v16.8.6-keyed	1.4 (1.27)	4.1 (2.55)	4.8 (2.43)	5.1 (2.92)	3.1 (1.35)	1.99
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1.4 (1.31)	5.2 (3.28)	6.0 (3.03)	6.7 (3.82)	3.5 (1.53)	2.38
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1.4 (1.35)	5.2 (3.29)	6.1 (3.05)	6.6 (3.73)	3.5 (1.52)	2.38
m_react-v16.11.0-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.6 (3.78)	3.7 (1.62)	2.22
react-v16.8.6-keyed	1.3 (1.25)	4.4 (2.74)	5.1 (2.58)	6.6 (3.77)	3.7 (1.61)	2.22
react-mobX-v16.4.1 + 5.0.3-keyed	1.5 (1.42)	5.6 (3.54)	6.3 (3.17)	10.2 (5.83)	5.1 (2.24)	2.91
m_react-redux-v16.11.0 + 7.1.0-keyed	1.5 (1.38)	7.2 (4.55)	8.2 (4.13)	8.3 (4.72)	3.3 (1.43)	2.81
react-redux-v16.8.6 + 7.1.0-keyed	1.5 (1.37)	7.2 (4.53)	8.2 (4.12)	8.4 (4.77)	3.3 (1.44)	2.81
m_react-mobX-v16.11.0 + 5.0.3-keyed	1.6 (1.48)	18.7 (11.78)	22.0 (11.10)	26.3 (14.96)	6.9 (3.01)	6.14
poc_react-v16.11.0-keyed	1.1 (1.03)	2.2 (1.36)	4.9 (2.47)	4.3 (2.46)	5.1 (2.24)	1.8

Table A.4: Benchmark results for Memory consumption. Measured in megabytes per second (MBs). The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

Appendix B

Microbenchmark Results Iteration 2

	Create rows (ms)	*Replace all rows (ms)*	*Partial update (ms)*	*Select row (ms)*	*Swap rows (ms)*	*Remove row (ms)*
vanillajs-1-non-keyed	130.7 (1.14)	47.4 (1.01)	215.9 (1.08)	25.8 (1.00)	31.9 (1.09)	45.5 (1.11)
vanillajs-non-keyed	133.4 (1.16)	51.2 (1.09)	225.9 (1.13)	26.1 (1.01)	32.8 (1.12)	46.3 (1.13)
m_svelte-v3.12.1-non-keyed	157.9 (1.37)	47.1 (1.00)	230.5 (1.15)	35.4 (1.37)	29.4 (1.00)	41.0 (1.00)
svelte-v3.5.1-non-keyed	149.8 (1.30)	47.0 (1.00)	225.4 (1.12)	37.0 (1.43)	31.2 (1.06)	41.2 (1.01)
vanillajs-1-keyed	114.9 (1.00)	126.8 (2.70)	200.4 (1.00)	26.7 (1.04)	48.8 (1.66)	45.4 (1.11)
vanillajs-keyed	120.9 (1.05)	139.9 (2.98)	265.2 (1.32)	38.8 (1.51)	50.6 (1.72)	46.9 (1.14)
svelte-v3.5.1-keyed	146.8 (1.28)	156.4 (3.33)	244.3 (1.22)	38.3 (1.48)	58.8 (2.00)	46.6 (1.14)
m_svelte-v3.12.1-keyed	150.0 (1.31)	158.7 (3.38)	234.2 (1.17)	38.3 (1.49)	58.0 (1.98)	46.3 (1.13)
react-v16.8.6-non-keyed	191.3 (1.66)	51.0 (1.09)	333.8 (1.67)	98.5 (3.82)	37.5 (1.28)	47.6 (1.16)
m_react-v16.11.0-non-keyed	194.1 (1.69)	50.2 (1.07)	323.5 (1.61)	106.6 (4.13)	38.6 (1.31)	48.6 (1.19)
m_react-hooks-v16.11.0-keyed	170.1 (1.48)	146.8 (3.13)	230.6 (1.15)	44.5 (1.73)	546.6 (18.61)	46.2 (1.13)
react-hooks-v16.8.6-keyed	169.6 (1.48)	148.0 (3.15)	241.4 (1.20)	44.0 (1.71)	565.1 (19.25)	46.2 (1.13)
react-redux-hooks-v16.8.6 + 7.1.0-keyed	180.0 (1.57)	150.0 (3.19)	303.1 (1.51)	47.2 (1.83)	560.3 (19.08)	46.8 (1.14)
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	175.5 (1.53)	149.5 (3.18)	289.3 (1.44)	51.5 (2.00)	552.5 (18.82)	46.8 (1.14)
m_react-v16.11.0-keyed	189.2 (1.65)	162.6 (3.46)	308.6 (1.54)	91.5 (3.55)	556.8 (18.96)	47.5 (1.16)
react-v16.8.6-keyed	186.2 (1.62)	165.0 (3.51)	296.5 (1.48)	93.3 (3.62)	574.1 (19.55)	48.4 (1.18)
react-mobX-v16.4.1 + 5.0.3-keyed	220.2 (1.92)	171.5 (3.65)	284.3 (1.42)	72.1 (2.80)	562.3 (19.15)	49.6 (1.21)
m_react-redux-v16.11.0 + 7.1.0-keyed	223.5 (1.95)	179.2 (3.82)	371.1 (1.85)	51.4 (1.99)	560.0 (19.07)	58.1 (1.42)
react-redux-v16.8.6 + 7.1.0-keyed	210.2 (1.83)	176.6 (3.76)	381.9 (1.91)	66.2 (2.57)	565.0 (19.24)	57.6 (1.41)
m_react-mobX-v16.11.0 + 5.0.3-keyed	747.3 (6.50)	677.9 (14.43)	667.7 (3.33)	332.7 (12.90)	1,410.3 (48.03)	61.8 (1.51)
poc_react-v16.11.0-keyed	216.2 (1.88)	206.4 (4.40)	5,889.0 (29.38)	5,985.7 (232.11)	940.3 (32.02)	200.0 (4.88)
it1_react-v16.11.0-keyed	176.7 (1.54)	202.1 (4.30)	5.925,8 (29.57)	6,396.0 (248.02)	862.8 (29.35)	186.4 (4.55)
it2_react-v16.11.0-keyed	185.6 (1.62)	194.7 (4.14)	5,525.5 (27.57)	5,751.5 (223.03)	870.1 (29.63)	202.9 (4.95)

Table B.1: Benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Create many rows (ms)	*Append rows (ms)*	*Clear rows (ms)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,197.4 (1.00)	272.6 (1.00)	157.6 (1.02)	1.05
vanillajs-non-keyed	1,213.8 (1.01)	275.4 (1.01)	158.9 (1.03)	1.08
m_svelte-v3.12.1-non-keyed	1,552.0 (1.30)	342.8 (1.26)	217.6 (1.41)	1.2
svelte-v3.5.1-non-keyed	1,528.9 (1.28)	333.6 (1.23)	221.8 (1.44)	1.2
vanillajs-1-keyed	1,201.8 (1.00)	272.1 (1.00)	154.2 (1.00)	1.2
vanillajs-keyed	1,235.5 (1.03)	285.2 (1.05)	166.3 (1.08)	1.34
svelte-v3.5.1-keyed	1,494.7 (1.25)	336.8 (1.24)	227.0 (1.47)	1.51
m_svelte-v3.12.1-keyed	1,530.5 (1.28)	347.0 (1.27)	221.4 (1.44)	1.51
react-v16.8.6-non-keyed	1,952.2 (1.63)	415.1 (1.53)	221.5 (1.44)	1.58
m_react-v16.11.0-non-keyed	1,920.7 (1.60)	419.6 (1.54)	232.7 (1.51)	1.6
m_react-hooks-v16.11.0-keyed	1,721.7 (1.44)	362.5 (1.33)	219.4 (1.42)	2.01
react-hooks-v16.8.6-keyed	1,733.1 (1.45)	363.9 (1.34)	207.1 (1.34)	2.01
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1,762.4 (1.47)	353.6 (1.30)	222.4 (1.44)	2.11
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1,793.4 (1.50)	348.7 (1.28)	231.3 (1.50)	2.12
m_react-v16.11.0-keyed	1,917.1 (1.60)	405.4 (1.49)	223.7 (1.45)	2.37
react-v16.8.6-keyed	1,937.2 (1.62)	399.6 (1.47)	219.0 (1.42)	2.37
react-mobX-v16.4.1 + 5.0.3-keyed	2,007.9 (1.68)	463.5 (1.70)	305.2 (1.98)	2.49
m_react-redux-v16.11.0 + 7.1.0-keyed	2,075.3 (1.73)	464.0 (1.70)	258.4 (1.68)	2.49
react-redux-v16.8.6 + 7.1.0-keyed	2,086.4 (1.74)	458.9 (1.69)	233.2 (1.51)	2.52
m_react-mobX-v16.11.0 + 5.0.3-keyed	44,458.7 (37.13)	1,655.0 (6.08)	514.1 (3.33)	8.45
poc_react-v16.11.0-keyed	2,015.5 (1.68)	930.5 (3.42)	181.3 (1.18)	7.31
it1_react-v16.11.0-keyed	3,722.7 (3.11)	854.9 (3.14)	213.4 (1.38)	7.95
it2_react-v16.11.0-keyed	1,934.7 (1.62)	833.8 (3.06)	196.7 (1.28)	6.96

Table B.2: Continued benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Consistently interactive (ms)	*Script bootup time (ms)*	*Total kilobyte weight (Kb)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,876.1 (1.00)	16.0 (1.00)	144.2 (1.00)	1
vanillajs-non-keyed	1,875.6 (1.00)	16.0 (1.00)	147.0 (1.02)	1.01
m_svelte-v3.12.1-non-keyed	1,875.6 (1.00)	16.0 (1.00)	145.3 (1.01)	1
svelte-v3.5.1-non-keyed	1,876.2 (1.00)	16.0 (1.00)	145.2 (1.01)	1
vanillajs-1-keyed	1,875.7 (1.00)	16.0 (1.00)	143.6 (1.00)	1
vanillajs-keyed	1,875.9 (1.00)	16.0 (1.00)	149.5 (1.04)	1.01
svelte-v3.5.1-keyed	1,905.6 (1.02)	16.0 (1.00)	145.8 (1.01)	1.01
m_svelte-v3.12.1-keyed	1,876.1 (1.00)	16.0 (1.00)	147.5 (1.03)	1.01
react-v16.8.6-non-keyed	2,502.3 (1.33)	16.0 (1.00)	261.1 (1.82)	1.34
m_react-v16.11.0-non-keyed	2,652.4 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
m_react-hooks-v16.11.0-keyed	2,553.4 (1.36)	16.0 (1.00)	270.8 (1.89)	1.37
react-hooks-v16.8.6-keyed	2,500.9 (1.33)	16.0 (1.00)	260.1 (1.81)	1.34
react-redux-hooks-v16.8.6 + 7.1.0-keyed	2,654.6 (1.42)	16.0 (1.00)	277.7 (1.93)	1.4
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	2,702.1 (1.44)	16.0 (1.00)	288.5 (2.01)	1.43
m_react-v16.11.0-keyed	2,653.9 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
react-v16.8.6-keyed	2,502.1 (1.33)	16.0 (1.00)	261.8 (1.82)	1.34
react-mobX-v16.4.1 + 5.0.3-keyed	2,851.3 (1.52)	64.0 (4.00)	313.6 (2.18)	2.37
m_react-redux-v16.11.0 + 7.1.0-keyed	2,701.9 (1.44)	59.7 (3.73)	290.9 (2.03)	2.22
react-redux-v16.8.6 + 7.1.0-keyed	2,656.7 (1.42)	59.1 (3.69)	280.1 (1.95)	2.17
m_react-mobX-v16.11.0 + 5.0.3-keyed	2,965.9 (1.58)	73.2 (4.57)	341.8 (2.38)	2.58
poc_react-v16.11.0-keyed	1,905.6 (1.02)	16.0 (1.00)	149.3 (1.04)	1.02
it1_react-v16.11.0-keyed	2,032.6 (1.08)	16.0 (1.00)	154.9 (1.08)	1.05
it2_react-v16.11.0-keyed	2,030.9 (1.08)	16.0 (1.00)	154.9 (1.08)	1.05

Table B.3: Benchmark results for Startup metrics. Measured in milliseconds (ms) and kilobytes (Kb) depending on the benchmark. The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Ready memory (MBs)	*Run memory (MBs)*	*Update each 10th row (MBs)*	*Replace 1k rows (MBs)*	*Creating/Clearing 1k rows (MBs)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	1.8 (1.00)	2.4 (1.05)	1.01
vanillajs-non-keyed	1.1 (1.00)	1.6 (1.01)	2.0 (1.01)	1.8 (1.03)	2.4 (1.04)	1.02
m_svelte-v3.12.1-non-keyed	1.1 (1.02)	2.5 (1.58)	2.8 (1.43)	2.6 (1.45)	2.4 (1.05)	1.29
svelte-v3.5.1-non-keyed	1.1 (1.02)	2.3 (1.47)	2.7 (1.34)	2.5 (1.42)	2.3 (1.00)	1.23
vanillajs-1-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	2.3 (1.30)	2.4 (1.05)	1.06
vanillajs-keyed	1.1 (1.01)	1.6 (1.01)	2.0 (1.00)	2.3 (1.32)	2.3 (1.02)	1.07
svelte-v3.5.1-keyed	1.1 (1.02)	2.4 (1.49)	2.7 (1.35)	2.9 (1.66)	2.3 (1.00)	1.28
m_svelte-v3.12.1-keyed	1.1 (1.02)	2.5 (1.60)	2.9 (1.44)	3.1 (1.77)	2.4 (1.06)	1.35
react-v16.8.6-non-keyed	1.3 (1.23)	4.4 (2.74)	5.1 (2.58)	6.5 (3.69)	3.7 (1.61)	2.2
m_react-v16.11.0-non-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.5 (3.72)	3.7 (1.63)	2.21
m_react-hooks-v16.11.0-keyed	1.4 (1.28)	4.1 (2.56)	4.9 (2.44)	5.0 (2.87)	3.1 (1.37)	1.99
react-hooks-v16.8.6-keyed	1.4 (1.27)	4.1 (2.55)	4.8 (2.43)	5.1 (2.92)	3.1 (1.35)	1.99
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1.4 (1.31)	5.2 (3.28)	6.0 (3.03)	6.7 (3.82)	3.5 (1.53)	2.38
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1.4 (1.35)	5.2 (3.29)	6.1 (3.05)	6.6 (3.73)	3.5 (1.52)	2.38
m_react-v16.11.0-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.6 (3.78)	3.7 (1.62)	2.22
react-v16.8.6-keyed	1.3 (1.25)	4.4 (2.74)	5.1 (2.58)	6.6 (3.77)	3.7 (1.61)	2.22
react-mobX-v16.4.1 + 5.0.3-keyed	1.5 (1.42)	5.6 (3.54)	6.3 (3.17)	10.2 (5.83)	5.1 (2.24)	2.91
m_react-redux-v16.11.0 + 7.1.0-keyed	1.5 (1.38)	7.2 (4.55)	8.2 (4.13)	8.3 (4.72)	3.3 (1.43)	2.81
react-redux-v16.8.6 + 7.1.0-keyed	1.5 (1.37)	7.2 (4.53)	8.2 (4.12)	8.4 (4.77)	3.3 (1.44)	2.81
m_react-mobX-v16.11.0 + 5.0.3-keyed	1.6 (1.48)	18.7 (11.78)	22.0 (11.10)	26.3 (14.96)	6.9 (3.01)	6.14
poc_react-v16.11.0-keyed	1.1 (1.03)	2.2 (1.36)	4.9 (2.47)	4.3 (2.46)	5.1 (2.24)	1.8
it1_react-v16.11.0-keyed	1.1 (1.04)	3.0 (1.87)	4.2 (2.10)	3.5 (2.00)	2.3 (1.03)	1.53
it2_react-v16.11.0-keyed	1.1 (1.04)	3.0 (1.87)	4.2 (2.09)	3.5 (2.00)	2.3 (1.03)	1.53

Table B.4: Benchmark results for Memory consumption. Measured in megabytes per second (MBs). The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

Appendix C

Microbenchmark Results Iteration 3

	Create rows (ms)	*Replace all rows (ms)*	*Partial update (ms)*	*Select row (ms)*	*Swap rows (ms)*	*Remove row (ms)*
vanillajs-1-non-keyed	130.7 (1.14)	47.4 (1.01)	215.9 (1.08)	25.8 (1.00)	31.9 (1.09)	45.5 (1.11)
vanillajs-non-keyed	133.4 (1.16)	51.2 (1.09)	225.9 (1.13)	26.1 (1.01)	32.8 (1.12)	46.3 (1.13)
m_svelte-v3.12.1-non-keyed	157.9 (1.37)	47.1 (1.00)	230.5 (1.15)	35.4 (1.37)	29.4 (1.00)	41.0 (1.00)
svelte-v3.5.1-non-keyed	149.8 (1.30)	47.0 (1.00)	225.4 (1.12)	37.0 (1.43)	31.2 (1.06)	41.2 (1.01)
vanillajs-1-keyed	114.9 (1.00)	126.8 (2.70)	200.4 (1.00)	26.7 (1.04)	48.8 (1.66)	45.4 (1.11)
vanillajs-keyed	120.9 (1.05)	139.9 (2.98)	265.2 (1.32)	38.8 (1.51)	50.6 (1.72)	46.9 (1.14)
svelte-v3.5.1-keyed	146.8 (1.28)	156.4 (3.33)	244.3 (1.22)	38.3 (1.48)	58.8 (2.00)	46.6 (1.14)
m_svelte-v3.12.1-keyed	150.0 (1.31)	158.7 (3.38)	234.2 (1.17)	38.3 (1.49)	58.0 (1.98)	46.3 (1.13)
react-v16.8.6-non-keyed	191.3 (1.66)	51.0 (1.09)	333.8 (1.67)	98.5 (3.82)	37.5 (1.28)	47.6 (1.16)
m_react-v16.11.0-non-keyed	194.1 (1.69)	50.2 (1.07)	323.5 (1.61)	106.6 (4.13)	38.6 (1.31)	48.6 (1.19)
m_react-hooks-v16.11.0-keyed	170.1 (1.48)	146.8 (3.13)	230.6 (1.15)	44.5 (1.73)	546.6 (18.61)	46.2 (1.13)
react-hooks-v16.8.6-keyed	169.6 (1.48)	148.0 (3.15)	241.4 (1.20)	44.0 (1.71)	565.1 (19.25)	46.2 (1.13)
react-redux-hooks-v16.8.6 + 7.1.0-keyed	180.0 (1.57)	150.0 (3.19)	303.1 (1.51)	47.2 (1.83)	560.3 (19.08)	46.8 (1.14)
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	175.5 (1.53)	149.5 (3.18)	289.3 (1.44)	51.5 (2.00)	552.5 (18.82)	46.8 (1.14)
m_react-v16.11.0-keyed	189.2 (1.65)	162.6 (3.46)	308.6 (1.54)	91.5 (3.55)	556.8 (18.96)	47.5 (1.16)
react-v16.8.6-keyed	186.2 (1.62)	165.0 (3.51)	296.5 (1.48)	93.3 (3.62)	574.1 (19.55)	48.4 (1.18)
react-mobX-v16.4.1 + 5.0.3-keyed	220.2 (1.92)	171.5 (3.65)	284.3 (1.42)	72.1 (2.80)	562.3 (19.15)	49.6 (1.21)
m_react-redux-v16.11.0 + 7.1.0-keyed	223.5 (1.95)	179.2 (3.82)	371.1 (1.85)	51.4 (1.99)	560.0 (19.07)	58.1 (1.42)
react-redux-v16.8.6 + 7.1.0-keyed	210.2 (1.83)	176.6 (3.76)	381.9 (1.91)	66.2 (2.57)	565.0 (19.24)	57.6 (1.41)
m_react-mobX-v16.11.0 + 5.0.3-keyed	747.3 (6.50)	677.9 (14.43)	667.7 (3.33)	332.7 (12.90)	1,410.3 (48.03)	61.8 (1.51)
poc_react-v16.11.0-keyed	216.2 (1.88)	206.4 (4.40)	5,889.0 (29.38)	5,985.7 (232.11)	940.3 (32.02)	200.0 (4.88)
it1_react-v16.11.0-keyed	176.7 (1.54)	202.1 (4.30)	5.925,8 (29.57)	6,396.0 (248.02)	862.8 (29.35)	186.4 (4.55)
it2_react-v16.11.0-keyed	185.6 (1.62)	194.7 (4.14)	5,525.5 (27.57)	5,751.5 (223.03)	870.1 (29.63)	202.9 (4.95)
it3_react-v16.11.0-non-keyed	145.2 (1.26)	65.1 (1.38)	314.7 (1.57)	130.5 (5.06)	31.2 (1.06)	66.7 (1.63)

Table C.1: Benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Create many rows (ms)	*Append rows (ms)*	*Clear rows (ms)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,197.4 (1.00)	272.6 (1.00)	157.6 (1.02)	1.05
vanillajs-non-keyed	1,213.8 (1.01)	275.4 (1.01)	158.9 (1.03)	1.08
m_svelte-v3.12.1-non-keyed	1,552.0 (1.30)	342.8 (1.26)	217.6 (1.41)	1.2
svelte-v3.5.1-non-keyed	1,528.9 (1.28)	333.6 (1.23)	221.8 (1.44)	1.2
vanillajs-1-keyed	1,201.8 (1.00)	272.1 (1.00)	154.2 (1.00)	1.2
vanillajs-keyed	1,235.5 (1.03)	285.2 (1.05)	166.3 (1.08)	1.34
svelte-v3.5.1-keyed	1,494.7 (1.25)	336.8 (1.24)	227.0 (1.47)	1.51
m_svelte-v3.12.1-keyed	1,530.5 (1.28)	347.0 (1.27)	221.4 (1.44)	1.51
react-v16.8.6-non-keyed	1,952.2 (1.63)	415.1 (1.53)	221.5 (1.44)	1.58
m_react-v16.11.0-non-keyed	1,920.7 (1.60)	419.6 (1.54)	232.7 (1.51)	1.6
m_react-hooks-v16.11.0-keyed	1,721.7 (1.44)	362.5 (1.33)	219.4 (1.42)	2.01
react-hooks-v16.8.6-keyed	1,733.1 (1.45)	363.9 (1.34)	207.1 (1.34)	2.01
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1,762.4 (1.47)	353.6 (1.30)	222.4 (1.44)	2.11
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1,793.4 (1.50)	348.7 (1.28)	231.3 (1.50)	2.12
m_react-v16.11.0-keyed	1,917.1 (1.60)	405.4 (1.49)	223.7 (1.45)	2.37
react-v16.8.6-keyed	1,937.2 (1.62)	399.6 (1.47)	219.0 (1.42)	2.37
react-mobX-v16.4.1 + 5.0.3-keyed	2,007.9 (1.68)	463.5 (1.70)	305.2 (1.98)	2.49
m_react-redux-v16.11.0 + 7.1.0-keyed	2,075.3 (1.73)	464.0 (1.70)	258.4 (1.68)	2.49
react-redux-v16.8.6 + 7.1.0-keyed	2,086.4 (1.74)	458.9 (1.69)	233.2 (1.51)	2.52
m_react-mobX-v16.11.0 + 5.0.3-keyed	44,458.7 (37.13)	1,655.0 (6.08)	514.1 (3.33)	8.45
poc_react-v16.11.0-keyed	2,015.5 (1.68)	930.5 (3.42)	181.3 (1.18)	7.31
it1_react-v16.11.0-keyed	3,722.7 (3.11)	854.9 (3.14)	213.4 (1.38)	7.95
it2_react-v16.11.0-keyed	1,934.7 (1.62)	833.8 (3.06)	196.7 (1.28)	6.96
it3_react-v16.11.0-non-keyed	2,596.8 (2.17)	502.3 (1.85)	175.4 (1.14)	1.69

Table C.2: Benchmark results for DOM operations. Measured milliseconds (ms). The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Consistently interactive (ms)	*Script bootup time (ms)*	*Total kilobyte weight (Kb)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1,876.1 (1.00)	16.0 (1.00)	144.2 (1.00)	1
vanillajs-non-keyed	1,875.6 (1.00)	16.0 (1.00)	147.0 (1.02)	1.01
m_svelte-v3.12.1-non-keyed	1,875.6 (1.00)	16.0 (1.00)	145.3 (1.01)	1
svelte-v3.5.1-non-keyed	1,876.2 (1.00)	16.0 (1.00)	145.2 (1.01)	1
vanillajs-1-keyed	1,875.7 (1.00)	16.0 (1.00)	143.6 (1.00)	1
vanillajs-keyed	1,875.9 (1.00)	16.0 (1.00)	149.5 (1.04)	1.01
svelte-v3.5.1-keyed	1,905.6 (1.02)	16.0 (1.00)	145.8 (1.01)	1.01
m_svelte-v3.12.1-keyed	1,876.1 (1.00)	16.0 (1.00)	147.5 (1.03)	1.01
react-v16.8.6-non-keyed	2,502.3 (1.33)	16.0 (1.00)	261.1 (1.82)	1.34
m_react-v16.11.0-non-keyed	2,652.4 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
m_react-hooks-v16.11.0-keyed	2,553.4 (1.36)	16.0 (1.00)	270.8 (1.89)	1.37
react-hooks-v16.8.6-keyed	2,500.9 (1.33)	16.0 (1.00)	260.1 (1.81)	1.34
react-redux-hooks-v16.8.6 + 7.1.0-keyed	2,654.6 (1.42)	16.0 (1.00)	277.7 (1.93)	1.4
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	2,702.1 (1.44)	16.0 (1.00)	288.5 (2.01)	1.43
m_react-v16.11.0-keyed	2,653.9 (1.41)	16.0 (1.00)	271.8 (1.89)	1.39
react-v16.8.6-keyed	2,502.1 (1.33)	16.0 (1.00)	261.8 (1.82)	1.34
react-mobX-v16.4.1 + 5.0.3-keyed	2,851.3 (1.52)	64.0 (4.00)	313.6 (2.18)	2.37
m_react-redux-v16.11.0 + 7.1.0-keyed	2,701.9 (1.44)	59.7 (3.73)	290.9 (2.03)	2.22
react-redux-v16.8.6 + 7.1.0-keyed	2,656.7 (1.42)	59.1 (3.69)	280.1 (1.95)	2.17
m_react-mobX-v16.11.0 + 5.0.3-keyed	2,965.9 (1.58)	73.2 (4.57)	341.8 (2.38)	2.58
poc_react-v16.11.0-keyed	1,905.6 (1.02)	16.0 (1.00)	149.3 (1.04)	1.02
it1_react-v16.11.0-keyed	2,032.6 (1.08)	16.0 (1.00)	154.9 (1.08)	1.05
it2_react-v16.11.0-keyed	2,030.9 (1.08)	16.0 (1.00)	154.9 (1.08)	1.05
it3_react-v16.11.0-non-keyed	2,027.6 (1.08)	16.0 (1.00)	164.7 (1.15)	1.07

Table C.3: Benchmark results for Startup metrics. Measured in milliseconds (ms) and kilobytes (Kb) depending on the benchmark. The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

	Ready memory (MBs)	*Run memory (MBs)*	*Update each 10th row (MBs)*	*Replace 1k rows (MBs)*	*Creating/Clearing 1k rows (MBs)*	*Slowdown geometric mean*
vanillajs-1-non-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	1.8 (1.00)	2.4 (1.05)	1.01
vanillajs-non-keyed	1.1 (1.00)	1.6 (1.01)	2.0 (1.01)	1.8 (1.03)	2.4 (1.04)	1.02
m_svelte-v3.12.1-non-keyed	1.1 (1.02)	2.5 (1.58)	2.8 (1.43)	2.6 (1.45)	2.4 (1.05)	1.29
svelte-v3.5.1-non-keyed	1.1 (1.02)	2.3 (1.47)	2.7 (1.34)	2.5 (1.42)	2.3 (1.00)	1.23
vanillajs-1-keyed	1.1 (1.00)	1.6 (1.00)	2.0 (1.00)	2.3 (1.30)	2.4 (1.05)	1.06
vanillajs-keyed	1.1 (1.01)	1.6 (1.01)	2.0 (1.00)	2.3 (1.32)	2.3 (1.02)	1.07
svelte-v3.5.1-keyed	1.1 (1.02)	2.4 (1.49)	2.7 (1.35)	2.9 (1.66)	2.3 (1.00)	1.28
m_svelte-v3.12.1-keyed	1.1 (1.02)	2.5 (1.60)	2.9 (1.44)	3.1 (1.77)	2.4 (1.06)	1.35
react-v16.8.6-non-keyed	1.3 (1.23)	4.4 (2.74)	5.1 (2.58)	6.5 (3.69)	3.7 (1.61)	2.2
m_react-v16.11.0-non-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.5 (3.72)	3.7 (1.63)	2.21
m_react-hooks-v16.11.0-keyed	1.4 (1.28)	4.1 (2.56)	4.9 (2.44)	5.0 (2.87)	3.1 (1.37)	1.99
react-hooks-v16.8.6-keyed	1.4 (1.27)	4.1 (2.55)	4.8 (2.43)	5.1 (2.92)	3.1 (1.35)	1.99
react-redux-hooks-v16.8.6 + 7.1.0-keyed	1.4 (1.31)	5.2 (3.28)	6.0 (3.03)	6.7 (3.82)	3.5 (1.53)	2.38
m_react-redux-hooks-v16.11.0 + 7.1.0-keyed	1.4 (1.35)	5.2 (3.29)	6.1 (3.05)	6.6 (3.73)	3.5 (1.52)	2.38
m_react-v16.11.0-keyed	1.3 (1.23)	4.4 (2.75)	5.1 (2.59)	6.6 (3.78)	3.7 (1.62)	2.22
react-v16.8.6-keyed	1.3 (1.25)	4.4 (2.74)	5.1 (2.58)	6.6 (3.77)	3.7 (1.61)	2.22
react-mobX-v16.4.1 + 5.0.3-keyed	1.5 (1.42)	5.6 (3.54)	6.3 (3.17)	10.2 (5.83)	5.1 (2.24)	2.91
m_react-redux-v16.11.0 + 7.1.0-keyed	1.5 (1.38)	7.2 (4.55)	8.2 (4.13)	8.3 (4.72)	3.3 (1.43)	2.81
react-redux-v16.8.6 + 7.1.0-keyed	1.5 (1.37)	7.2 (4.53)	8.2 (4.12)	8.4 (4.77)	3.3 (1.44)	2.81
m_react-mobX-v16.11.0 + 5.0.3-keyed	1.6 (1.48)	18.7 (11.78)	22.0 (11.10)	26.3 (14.96)	6.9 (3.01)	6.14
poc_react-v16.11.0-keyed	1.1 (1.03)	2.2 (1.36)	4.9 (2.47)	4.3 (2.46)	5.1 (2.24)	1.8
it1_react-v16.11.0-keyed	1.1 (1.04)	3.0 (1.87)	4.2 (2.10)	3.5 (2.00)	2.3 (1.03)	1.53
it2_react-v16.11.0-keyed	1.1 (1.04)	3.0 (1.87)	4.2 (2.09)	3.5 (2.00)	2.3 (1.03)	1.53
it3_react-v16.11.0-non-keyed	1.1 (1.03)	2.3 (1.45)	2.6 (1.32)	2.3 (1.33)	2.3 (1.01)	1.22

Table C.4: Benchmark results for Memory consumption. Measured in megabytes per second (MBs). The unit is indicated in the benchmark title. The relative performance can be seen in the parenthesis, where 1.0 is the best.

Bibliography

[1] Kristoffer Magill Nash and Niclas Jon Sommer. *Compiling React Applications to Improve Performance: Analysing and Benchmarking Front-end Frameworks*. https://projekter.aau.dk/projekter/da/studentthesis/compiling-react-applications-to-improve-performance(c24dac3f-4dd1-4064-bd05-f3b56e2dea05).html. 2020.

[2] Facebook. *React - A JavaScript library for building user interfaces*. https://reactjs.org/. 2020.

[3] Facebook. *React - Components and Props*. https://reactjs.org/docs/components-and-props.html/. 2020.

[4] React. *Reconciliation*. https://reactjs.org/docs/reconciliation.html. 2020.

[5] React Core Team. *JSX in Depth*. https://reactjs.org/docs/jsx-in-depth.html. 2020.

[6] Babel. *Babel*. https://babeljs.io/. 2020.

[7] Facebook. *React Component API Reference*. https://reactjs.org/docs/react-component.html. 2020.

[8] Max Koretskyi. *Inside Fiber: in-depth overview of the new reconciliation algorithm in React*. https://indepth.dev/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react/. 2020.

[9] Rick Harris. *Svelte 3: Rethinking reactivity*. https://svelte.dev/blog/svelte-3-rethinking-reactivity. 2019.

[10] Svelte. *Lifecycle / beforeUpdate and afterUpdate*. https://svelte.dev/tutorial/update. 2020.

[11] Rich Harris. *Rethinking reactivity*. YGLF - Code Camp 2019. 2019.

[12] krausest. *A comparison of the perfomance of a few popular javascript frameworks*. https://github.com/krausest/js-framework-benchmark. 2020.

[13] Google. *Lighthouse*. https://developers.google.com/web/tools/lighthouse/. 2020.

[14] MDN. *Document API*. https://developer.mozilla.org/en-US/docs/Web/API/Document. 2020.

[15] React Core Team. *React Source Code*. https://github.com/facebook/react/. 2020.

[16] Rich Harris and Svelte contributors. *Svelte Source Code*. https://github.com/sveltejs/svelte. 2020.

[17] Sophie Alpert. *Building a Custom React Renderer*. https://www.youtube.com/watch?v=CGpMlWVcHok. 2019.

[18] React Core Team. *React Native*. https://reactnative.dev/. 2020.

[19] Contributors. *React Three Fiber*. https://github.com/react-spring/react-three-fiber. 2020.

[20] Ryan Solid. *Solid A declarative, efficient, and flexible JavaScript library for building user interfaces*. https://github.com/ryansolid/solid. 2020.

[21] Ryan Solid. *Results for js web frameworks benchmark round 8*. https://stefankrause.net/js-frameworks-benchmark8/table.html. 2018.

[22] MDN. *template: The Content Template element*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template. 2020.

[23] MDN. *Node*. https://developer.mozilla.org/en-US/docs/Web/API/Node. 2020.

[24] MDN. *DocumentFragment*. https://developer.mozilla.org/en-US/docs/Web/API/DocumentFragment. 2020.

[25] MDN. *Slot*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/slot. 2020.

[26] JSPerf. *JSPerf*. https://jsperf.com/. 2020.

[27] Niclas Sommer and Kristoffer Nash. *HTMLStringVCreateElement*. https://jsperf.com/htmlstringvcreateelement/1. 2020.

[28] MDN. *Function.prototype.bind()*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind. 2020.

[29] MDN. *Document.createElement()*. https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement. 2020.

[30] MDN. *XPath*. https://developer.mozilla.org/en-US/docs/Web/XPath. 2019.

[31] Github. *Github Issue*. https://github.com/krausest/js-framework-benchmark/issues/741. 2020.

[32] Github. *Github Issue*. https://github.com/krausest/js-framework-benchmark/issues/683. 2020.

[33] Jens Nicolay et al. “Detecting function purity in JavaScript”. eng. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 101–110. ISBN: 9781467375290.

[34] Webpack Contributors. *Tree Shaking*. https://webpack.js.org/guides/tree-shaking/. 2020.

[35] Philipp Spiess. *Scheduling in React*. https://philippspiess.com/scheduling-in-react/. 2020.

[36] Francesco. *JavaScript main thread. Dissected.* https://medium.com/@francesco_rizzi/javascript-main-thread-dissected-43c85fce7e23. 2017.

[37] Eric Elliott. *JavaScript Factory Functions with ES6+*. https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1. 2017.

[38] Mozilla. *HTML elements reference*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element. 2020.

[39] Babel. *Babel code-frame*. https://babeljs.io/docs/en/babel-code-frame. 2020.

[40] vadimdemedes. *React for interactive command-line apps*. https://github.com/vadimdemedes/ink. 2020.

[41] MDN. *var*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var. 2020.

[42] MDN. *let*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let. 2020.

[43] MDN. *Hoisting*. https://developer.mozilla.org/en-US/docs/Glossary/Hoisting. 2020.

[44] Eric Elliott. *JavaScript ES6+: var, let, or const?* https://medium.com/javascript-scene/javascript-es6-var-let-or-const-ba58b8dcde75. 2020.

[45] Svelte. *Cybernetically enhanced web apps*. https://svelte.dev/. 2020.