UNIVERSITY OF WATERLOO
Faculty of Engineering
Department of Electrical and Computer Engineering

React and Redux Rendering Performance Report

Bespoke Metrics, Inc.
55 York Street, Suite 1203
Toronto, Ontario, Canada

Prepared by
Chang Bok Lee
ID 20361001
userid cb3lee
3A, Computer Engineering
23 December 2017

Chang Bok Lee
23 Sheppard Ave East, Unit 706
North York, Ontario
M2N 0C8

December 23, 2017

Vincent Gaudet, chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir:

This report "React Rendering Performance Report", was prepared for my 3B work report. This report is intended for the WKRPT 301 course and it was written during my work term at Bespoke Metric, where I worked as a full stack web developer for the company's main product, COMPASS.

Bespoke Metrics provides data control and analysis services. The current main product, COMPASS, is focused on managing risks between subcontractors and general contractors in major construction projects.

The software engineering department of Bespoke Metrics is mainly separated into two teams; Front-end and Back-end. The Front-end team, in which I was employed, is managed by Jason Bellamy and primarily focused on developing the COMPASS application user interface. The front-end of COMPASS application is designed as a single page web application.

I would like to thank Mr. Bellamy for his guidance around the web application ecosystem. He helped me to understand React and Redux. Without his help, I would not be writing about React Performance in this report. I also would like to thank my mentor Ms. M Heo for reviewing my report. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Chang Bok Lee
ID 20361001

Contributions

The Engineering Team of Bespoke Metrics Inc. was relatively small in size. It consisted of four full time employees and six coops including myself. There were nine other full time employees and one coop student in the finance team. The company was less than a year old when I started my term. The company was in the process of finalizing the terms and agreements with its first major client. At the stage, the company's first product, COMPASS, was successfully launched and being used on a daily basis.

COMPASS is a web application targeting to help construction general contractors to manage their risks over hiring subcontractors. The product was roughly divided into three parts; general contractor mode, subcontractor mode, and admin mode. Subcontractors are required to fill in the questionnaire electronically and the answers are aggregated to generate quantifiable credit ratings. The credit rating of each subcontractor then would be presented to general contractors. The engineering team was divided into two teams; backend and frontend. The backend team was responsible for building a scalable server architecture providing RESTful API for the frontend team to use. The frontend team was focused on creating a single-page application. All coops were asked to join either team. I chose to join the frontend team.

The frontend is a web application built with React, Redux, and many other libraries. My main tasks were focused on adding more features to the COMPASS frontend. Along with the other coops, I successfully launched the very first admin page and added features to it. Due to the size and the stage of the company, it was natural to see bugs in production and I contributed to fix the bugs promptly. In the beginning of the work term, I prepared a presentation for some of the other coops to get familiar with the development tools such as *git* and issue trackers. The pages and the features for the data administrators were the major milestones that I accomplished. I implemented a dynamic form generator class that produces a form from a schema retrieved from the backend API. The form generator is capable of handling nested questions by recursively traveling down the tree and preparing a closure. If a triggering form action, such as choosing a radio button that has nested questions attached to it, happens, then the closure will populate the nested questions. The implementation required deeper understanding in *React*, *JSX,* and *Redux*. I was also able to contribute to refactoring the codebase to be more scalable and expandable using the knowledge that I acquire while implementing the features.

The relationship between this report and my job is that the report involves different techniques to optimize the rendering speed of the application. The application contains large lists, either in a table format or a carousel card format, that needs to be updated based on the server responses and the user interactions in real time. Although *React* has generally high performance, it can bottleneck in the situations. This report analyses different technique to improve the performance of a *React* and *Redux* application. The manager of the frontend team, Mr. Bellamy, asked me if I could prepare a report for *React* and *Redux*. This report is an expanded version of what I reported to him. After the submission of this report, a final copy will be delivered to Mr. Bellamy. It would be grateful if the gains from this report make a better performing *React* application. Writing this report helped me understand what and how to measure the performance of a web based application. Also, I gained a deeper understanding of what is happening under the hood of *React* which is extremely valuable for my future developments.

I am glad that I was able to understand the whole COMPASS product's stack. I am satisfied what I learned from Bespoke Metrics In the broader scheme of things, the work that I performed improved the only product keeping clients satisfied.

Summary

The main purpose of this report is to analyze and compare different optimization techniques of *React* and *Redux* for COMPASS. These techniques will help to improve the quality and the performance of the application. This report provides a reference point to the future refactoring of the codebase. This report is intended for readers with knowledge of *React* and *Redux*.

The first section introduces the company and the product. More importantly, it contains a brief overview of the technical stack. The following section explains the test plans and defines the test cases.

There are two major sections of the body of this report. Section 3 investigates and analyzes the *React* performance optimization scenarios. The second section analyzes the performance and the test results of *Redux*.

In conclusion, *React* needs to be broken down into smaller layers when there are user interactions on the component. And *Redux* could be optimized using general programming techniques such as normalizing. In both cases, it is key to understand where the user interaction happens.

The recommendation in this report is to list detected bottlenecks in COMPASS product and solutions to the problems.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Bespoke Metrics

Bespoke Metrics Inc. is a start-up based in the downtown Toronto. The company founded to deliver values to its clients by providing tailored data analytics. The current focus of the company is in the construction industries. A general contractor is company that manages the whole construction project. They vary in size; from one man with one truck to multi-billion dollars giants such as EllisDon and PCL Construction. General contractors (usually) hires subcontractors to do the actual construction while they manage the different subcontractors and coordinate the different stages of the construction. Every construction project, from renovating homes to building airports, comes with risks. Especially, in a mega project like building a shopping mall or renovating the Union station, a small failure from a subcontractor could cause a serious damage to the whole project, a catastrophic effect to the schedule, or a significant financial loss. As an example, if a subcontractor for roofing job could not finish its job by the deadline, then the concrete cannot be poured to form the base of the building as it might rain in the next few days. Therefore, controlling and understanding the reliability of subcontractors are one of the most important factors to conduct a successful project. Bespoke Metrics provides a tailored data analytic service, COMPASS, to the general contractors that can quantify the reliability of each subcontractor. It is similar to credit ratings in financial and insurance industries. COMPASS is publicly accessible but requires a credential.

1.2 COMPASS

COMPASS is a contract risk management service. The main target clients are general contractors who hires multiple subcontractors to accomplish a construction project. Generally, such construction project takes many subcontractors and COMPASS helps managing the risks of each subcontractor. The current COMPASS product has three different modes; general contractor, subcontractor, and administrator. The subcontractors are required to access COMPASS and fill out the provided questionnaire. The questionnaire asks about the company's information such as health and safety information, and financial information. All the input from the questionnaire then would be piped to a model that aggregates all the information and generate a credit rating. The credit rating then would be visible to the general contractor. The engineering team is separated

into a frontend team and a backend team. The frontend team is responsible for creating a user friendly single page web application. And the backend team is responsible for a scalable REST API. This report focuses on the frontend.

1.3 Frontend stack

Single-page Application

The frontend is the GUI of the COMPASS service. It is a single page web application that gets delivered to its users by CDN such as CloudFront by AWS. All of the required assets and the files that are needed to be served are bundled by *Webpack.* A single page web application is an app that works inside of a browser and does not require page reloading during use. [**1**] On the initial access to the page, the browser will load the entire application and the entire application does not require page reload to navigate to other pages. Its initial load could be slower due to the bundle size that contains entire application. However, this is becoming a smaller problem as the general internet speed is getting faster. Also, there are optimizations such as server side rendering are present to improve the initial loading speed. A single page app removes the page reloading upon navigating to other features making it feels natural to use.

React and Redux

The frontend of COMPASS uses *React*. *React* is a JavaScript framework library that helps creating user interfaces. It has gained popularity over the past couple years. In *React*, an HTML DOM is managed as a *component*. A *component* contains states and properties that can be used to render the DOM. A component gets re-rendered whenever there is an update to its state or properties. A DOM rendering is relatively expensive compared to computing a JavaScript object. *React* detects the state changes at the object level then update only the DOMs that are affected. It is naturally faster than manipulating the DOM itself and the DOM carries the required information as a custom attribute like *data-foobar.* A benefit of using a *component* is that it is reusable. In fact, there are thousands of open-sourced *React components* available online. It also allows you to write the HTML markups for each component directly inside of a JavaScript file.

The other major library that COMPASS uses is *Redux. Redux* is a predictable state container for JavaScript apps. It is inevitable that a *React component* needs to access some other component's states and properties. This could be done through having a parent component then sharing the

data with the siblings. However, it is difficult to manage chains of dependencies and passing the properties around gets tedious very quickly. There are different design patterns to make the app state manageable including MVC, Flux and Redux. The famous MVC did not scale well for Facebook's huge codebase. The main problem for them was the bidirectional communication. [**3**] Facebook introduced Flux which is an unidirectional data flow architecture. The idea of Flux is to have *stores* for all the states and the properties required for the app. Updating the stores is only available through dispatching an action. Flux enforces unidirectional data flow and eliminates complex state managements. In *Redux*, the library reduces the number of the stores to one. To use *Redux*, reducers have to be provided to the store. Each reducer is a pure function. Since a pure function cannot depend on reading any hidden value outside of the function scope, such as another field in the store or any other global variable, *Redux* can support helpful features like time-travel debugging. [**4**]

1.4 Scope

This report will take a step into the practical usage of *React* and analyze its performance. Although React is a high performant library, its performance could only be maximized when the implementation is in lined with the library. The result of performance tests in different methods of updating the component state will help us to analyze how to use the library properly. Also, this report will cover how *Redux* can be optimized to have a better performance with *React.*

2. Test Outline

2.1 Test Environment

The testing environment is setup on the author's personal laptop. The result is subject to change based on the machine spec and the settings. Therefore, the results will focus on the relative differences rather than actual number.

NPM

NPM (Node Package Manager) is used to manage all the dependencies of the testing environment. It supports installing a specific version of a library by executing `npm install <library_name>@<semver>`

Webpack

Webpack is used to bundle the testing components. The minifying feature to reduce the bundle size is unused since the bundle size is out of the scope of this report. Also, none of the performance optimization plugins are included to keep the variables minimum. The webpack configuration is available in Appendix A of this report.

ECMAScript 6 (ES6)

The test environment uses ES6 syntax to reflect COMPASS's ES6 syntax. ES6 introduced a set of syntactic sugars for the developers to use. The test environment uses features like arrow function, class, and destructing assignments. This report will focus on the impact of the rest parameter and destructing assignments. A transpiler, *babel,* is used to support ES6.

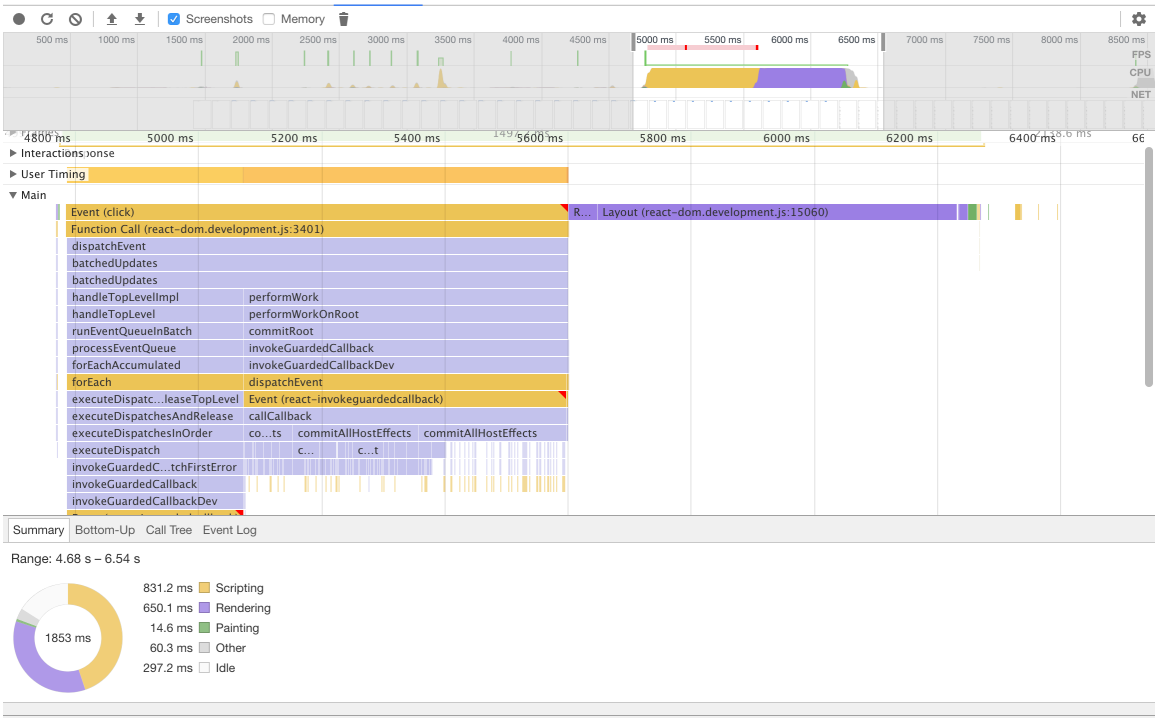

Figure 1. The screenshot of the performance tool of Chrome

Chrome Developer Tools

Chrome has a collection of powerful tools helpful for the developers. This report especially focuses on the *performance* tab of the developer tools suite. The performance tab supports profiling of the application allowing the detailed information to be collected. It is also useful to detect bottlenecks throughout the applications. Figure 1 represents a typical view and where the measured data collected from. The data of interest is the scripting time and the rendering time.

2.2 Test Plan

The report starts with the pure *React* optimization. It will render a list of elements (rows) that is clickable. The element will change its color once it is clicked. The change of color is simply a toggling between on/off. The test subjects are as follows:

- case (a) Rendering time with the state of the element is stored in the parent list
- case (b) Rendering time with the state of the element is stored in each element
- case (c) Impact of using …(spread) operator
- case (d) Impact of using `shouldComponentUpdate()` function to pick from the states

In the next part, an analysis of the performance when using *React* with *Redux*. COMPASS barely uses *React* without *Redux*. The tests are similar to the pure *React* tests.

- case (e) Rendering time with the reducer updating the whole list
- case (f) Rendering time with having a sub-reducer for each item in the list
- case (g) Optimizing with `mapStateToProps()`

All of the testing codes are publicly available at https://github.com/donkeysmash/wkrpt2 .

3. React

3.1 Lifecycle

React allows componentization of a DOM element. The created components could easily be used in other component, making it possible to compose an application at a small piece by piece. Each *component* has a lifecycle from its creation to the removal. The chart on Figure 2 shows the change of lifecycle state when an operation is applied to a component. The black node in the centre of the figure represent “normal” state. Every time there is a change in the component’s props or state, component would go through the life cycles on the right side of the black node. A

component is simply a JavaScript object; hence the computation is relatively cheaper compared to directly working with the DOM. According to the official *React* document re-rendering of a DOM element could be asynchronous. It is part of an optimization that *React* does out of the box. *React* tries to minimize the number of times that `ReactDOM.render()` gets called. [5] Some high speed real time operations would be batched together (e.g. burst of clicks or repeated keystrokes) and re-render the DOM only once. Naturally, it is important to pay attention to the states and the properties as they dictate how components are being rendered.

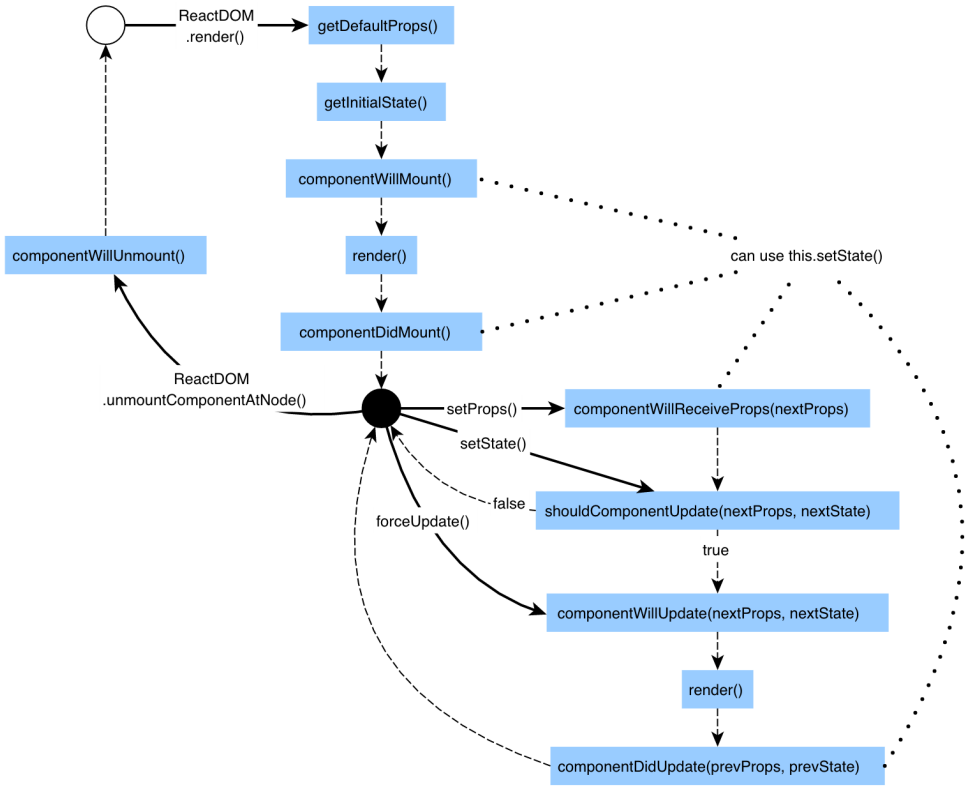

Figure 2. React Component Lifecycle [2]

3.2 Optimizing `state` and `props`

As it is shown in the life cycle chart, only `setProps()` and `setState()` trigger `render()`. Therefore, it is important to call the functions when it is necessary to update the DOM. In this section, the first two test cases are discussed; case (a) and (b). To share a state between components in *React*, it is necessary to have a shared parent. The shared parent contains the state that needs to be shared and distributes the state as a property to the children components. The first test case (a) is when the operation of a child component is shared as the parent's state.

The test is conducted as follows:

i. Generate a list of elements. To test the impact of having multiple elements, the length of the list ranges from 1 to 10,000.
ii. Record the initial rendering and scripting time.
iii. Update (click) one of the elements to toggle the state.
iv. Record the rendering time and the scripting time.
v. Update the same element to toggle it back to original state.
vi. Record the rendering time and the scripting time.

The values are average of 3 different attempts. The detail measurement data is available in Appendix B on Table B-1.

Table 3-1. Summary of the test case (a)

Length of the list	initial load (ms)	Toggle on (first click)		Toggle off (second click)		Toggling average (ms)
		scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)	
1	15.7	3.2	0.23	2.4	0.3	3.1
100	21.2	10.7	0.6	10.2	0.5	11
10000	1291.9	360.5	20.7	342	16.7	370.0

The case where the length is 10,000 suffers from a delay. An important point to note here is that toggling also has a high delay especially when the length is high. In the test case (a), `render()` is getting called whenever there is any update in the list. This means that the resources have been wasted re-rendering the untouched elements in the list. To improve the performance, the element components should store its information in its own state. This case is demonstrated with test case (b). Test case (b) has exactly same steps with the above. The code structure, on the other hand, has another layer (element) under the list component. The element layer contains its own state.

The results are as follows in Table 3-2:

Table 3-2 Summary of the test case (b)

Length of the list	initial load (ms)	Toggle on (first click)		Toggle off (second click)		Toggling average (ms)
		scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)	
1	10.5	4.8	0.27	3.2	0.2	4.2
100	34.6	7.8	0.5	4.1	0.9	6.65
10000	1659.9	184.6	14.9	117.5	7.07	162.0

The initial load time increased when there are many elements in the list. One of the factors is that the overhead from the extra *React* component layer. It is unnatural that the initial load time is lower for the case (b). However, there are many uncontrolled variables that could affect the test result. Especially, the high cost of boilerplates of a *React* application makes it difficult to measure

any meaningful difference when there is no repetition like a list or a tree. The main focus is when there are 10 000 elements in the list. Unfortunately, it increased the initial load time by 28 %. The average time to toggle to finish, however, reduced by 56 %. In fact, it was clearly noticeable when toggling the state of the list element. These local tests are subjective to a number of other variables such as the spec of the testing machine or the different versions of various libraries. It still provides a valuable insight with the relative comparison. A highly user interactive component that updates its state frequently could possibly be segmented down to smaller component to improve general user experience. If a component is relatively static and stateless, then it would give a better experience to the end user.

3.3 Impact of spread (…) operator and `shouldComponentUpdate()`

Throughout the COMPASS application, the components often use the following pattern.

```
const Wrapper = ({ arg0, arg1: Component, ...rest }) =>
  <Foo bar={arg0}>
    <Component {...rest} />
  </Foo>;
```

Figure 3 Usage of spread operator in React component

This pattern is especially useful to build a layout with subcomponents. Another use case is to protect a component to only be accessible if the user is authorized (i.e. logged in). In the snippet above, the *arg0* prop is safely consumed by *Foo* and the *arg1* prop is turned into a component *Component*. The *arg0* and *arg1* are not passed on to the <*Component*>. All the other props are passed via the *...rest* object making this component highly flexible. [**6**] For example, if *onClick* is provided to the *...rest,* then the component will trigger the provided function when it is clicked. Similarly, the same component can take *onTouchStart* to have a different behaviour on a touch device.

In this section, the performance impact of using the spread operators is analyzed. As it was mentioned above, changing the `props` will trigger re-rendering of the component. When any of the `props` changes, the <*Wrapper*> component will be updated and re-rendered. This re-rendering is usually undesired because *...rest* is for passing down to the children rather than actually updating the `props` of the component. However, *React* has no way of knowing that the `props` in the *...rest* is not being used for the <*Wrapper*> component. One of the possible

solutions is to add `shouldComponentUpdate()` to the component with a spread operator in its `props`. If a component does not need to update upon state or prop changes, returning *false* from `shouldComponentUpdate()` skips the whole re-rendering process. [**6**] This may cause an overhead. The test case (c) and (d) experiments with the situation to provide a better insight on the performance.

The test case (c) will measure the scripting and rendering time when a value gets updated that belongs to *...rest* `props`. Then a `shouldComponentUpdate()` function will be added to measure the changes in performance as the test case (d). The number of the subcomponents are set to 5000. The results are as follows:

Table 3-3 Summary of the test case (c) and (d)

Test case	Initial load (ms)	Unrelated Prop change (ms)	Related Prop change (ms)
c	1369	625	553
d	1401	331	300

The unrelated prop change means that the prop change has no visible effect. The related prop change updates one of the subcomponents visually. Generally, the performance increases when the component has an explicit `shouldComponentUpdate()`. This result shows that the spread operator should not be used without having `shouldComponentUpdate()` in a component with many subcomponents. In general, if there is a bottleneck with rendering, consider checking re-rendering without any visual effect using the React Dev Tools. [**6**] The React Dev Tools allow you to see what is re-rendered by showing colored border around the updated DOM elements.

4. Redux

It becomes complicated to manage the states and the props if they are shared across components. A centralized *store* that talks to an arbitrary component often resolves the problem. There are a number of solutions available from the *React* community such as MVC, Flux or Redux. Redux is popular in the web developer community. From a recent survey, almost 80 % of the respondents answered positively about *Redux*. [**7**] The COMPASS frontend product also uses *Redux* to manage its states.

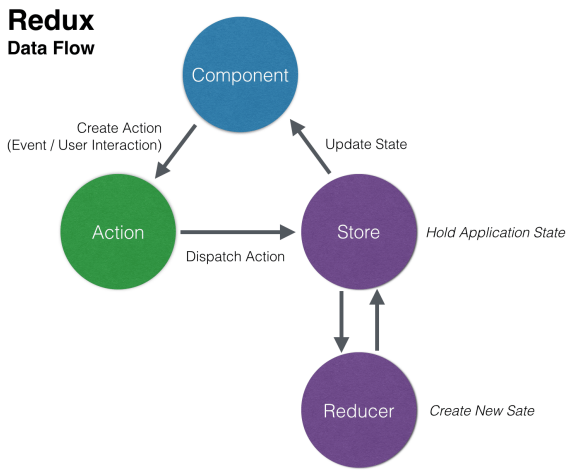

Figure 4. Redux Data Flow [8]

In this section, the performance optimization with *Redux* is discussed. As it is shown in Figure 3, the only way to update the store is by dispatching an action. Then, the reducers will update the store using the action. The state is directly accessible to any arbitrary *React* component that is passed to `connect(mapStateToProps, mapDispatchToProps)(component);`. The `connect` function is part of the *react-redux* library and it "connects" the *React* component to the *Redux* store. Hence, the mapping function/object *mapStateToProps* and *mapDispatchToProps* are passed into the first arguments. As the variable name says, it maps the states from the *Redux* store as the props of the connected component. A *React* component gets updated when there is a change in its *props*. Naturally, anything connected to the props affects the performance of the component. The first section analyzes the performance difference with various styles of reducers. In the second section, the `mapStateToProps` function is tested as a mean of optimizing the performance.

4.1 Reducer optimization

A typical reducer takes two arguments; *state* and *action*. It is a pure function that returns same outputs if the inputs are the same. The output of a reducer is the new state. When *Redux* is being used with *React*, the *Redux* state store is connected to a component. And the component gets a slice of the store as `props` of the component. Updating the state store will trigger the connected component to update. The store is only being updated by the returned values (new state) of the reducers. The case (e) and (f) analyze the performance difference with the changes in the reducers.

```
function itemsReducer(state = _init, action) {
  switch (action.type) {
    case 'TOGGLE':
      return state.map((item) =>
        action.id === item.id ?
          { ...item, isToggled: !item.isToggled} :
          item
      );
  }
  // more codes
}
```

Figure 5 The reducer for case (e)

The snippet above shows the reducer used for the case (e). The same steps as for the case (a) and (b) were taken to measure the data. The summary of results are as follows:

Table 4-1. Summary of the test case (e)

Length of the list	initial load (ms)	Toggle on (first click)		Toggle off (second click)		Toggling average (ms)
		scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)	
1	19.4	3.5	0.2	3.5	0.3	3.8
100	23.2	6.7	0.3	7.2	0.4	7.3
10000	1742.6	187.2	18.1	211.2	16.3	216.4

Certainly, adding *Redux* to the *React* components added slight overhead. The initial load time increased in all three cases. The rendering times, on the other hand, decreased especially in the high count of rows situation. Although the rendering time has decreased, the reducer structure still could be improved. The new state returned by this reducer has created a new array. The component has no way knowing that only one of the items has been updated. Hence, it will undesirably re-render the entire list. One of the possible improvements is to normalize the array in the state store. Instead of storing the object in the form of array, the object is stored as a key-value format. Each *item* is accessible in O(1) time. Also, to support the normalized state, the reducer has to be updated. The reducer used for the test case (f) is shown as Figure 6. This should improve the performance of the reducer. And the results are as follows:

Table 4-2. Summary of the test case (f)

Length of the list	initial load (ms)	Toggle on (first click)		Toggle off (second click)		Toggling average (ms)
		scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)	
1	18.4	3.8	0.2	3.1	0.4	3.8
100	27.1	5.8	0.3	8.2	0.4	7.4
10000	1655.1	102.2	17.1	96.2	14.5	115

A small number of rows does not show the impact clearly. The improvements are clearly observable with the length of 10000. However, this still undesirably re-renders entire list. One of the possible solutions is analyzed in the next section.

```
function itemsReducer(state = _init, action) {
  return {
    ids: ids(state.ids, action),
    items: items(state.items, action)
  };
}

// function ids is simply there to have a list of IDs
function ids ....

function items(state = _init, action) {
  switch (action.type) {
    case 'TOGGLE':
      const item = state[action.id];
      return {
        ...state,
        [action.id]: {
          ...item,
          isToggled: !item.isToggled
        }
      }
// more codes
```

Figure 6 Normalised reducer function

4.2 Using `mapStateToProps()` to optimize

The goal of this optimization is to stop undesirable updates in some components. It is achievable by using `mapStateToProps()` and adding a layer to the list component. This final test, case (g), is a combination of the ideas from the case (b) and the case (f). At this point, the test suite has a normalised state, a list component, and an item component. The list component is connected to the *Redux* store. For the case (g), the item is connected to the store. And the mapStateToProps() function is shown on Figure 7.

```
const mapStateToProps = (state, props) => ({
  item: state.items[props.id]
});
```

Figure 7 mapStateToProps function

This means that the parent list component does not have to know any detail about the item. The parent list is receiving only the IDs of each item from the *ids* state. Then, the item component receives the corresponding item mapped by the function shown in Figure 7. The implementation ensures that only the item will be updated once it is clicked. The results are as follows:

Table 4-3. Summary of the test case (g)

Length of the list	initial load (ms)	Toggle on (first click)		Toggle off (second click)		Toggling average (ms)
		scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)	
1	23.8	2.1	0.4	2.6	0.4	2.8
100	24.6	4.8	1.2	4.2	1.1	5.7
10000	1843.4	4.4	2.1	4.5	1.3	6.2

It is evident that this implementation has a superior performance compared to other test cases. Basically, the operation of changing one of the elements in a large list is done in a constant time.

5. Conclusions

The *React* framework has a high performance but a poor implementation will affect the performance. Similar to the other frameworks, *React* has multiple ways of using the provided functions. From the investigation and the analysis, it is concluded that generally more layer added closer to the user interaction promises a better performance. If there are a list of components to render, then the implementation should be modular. An update in the state or the props of the component should only affect the component in interest.

On top of the *React* component optimisation, the tests revealed that the reducer has a significant role for the *Redux* performance. It makes more sense in terms of performance to have a normalised state rather than a plain array. A plain array is likely to have O(n) access time since the index is seldom the id of the item of interest.

The spread operator causes delay in rendering in a specific case. It should be restrained from using on a highly interactive component. The *shouldComponentUpdate* function could remove

the delay but in a limited situation. Utilizing the *shouldComponentUpdate* function in a component is generally a good idea for performance.

6. Recommendations

Throughout the COMPASS application, some bottlenecks have been detected. The following list is for the upcoming refactoring sprint.

i. *QuestionFactory* – use *shouldComponentUpdate* or remove the spread operator
ii. *ProtectedRoute* – use *shouldComponentUpdate* or remove the spread operator
iii. *AdminList* – normalise the state
iv. *WidgetFactory* – update *mapStateToProps* to have smaller slice of the state
v. Some of the question *Fields* compoents – add another layer to handle user input

There are a handful more of places where we need to refactor the current structure. However, the impact would be near minimum and the gain is too small for the limited resources.

Although it is out of scope of this report, a possibly valuable library called *reselect* is found during the research. It provides selectors with memoization for the *mapStateToProps* function. A further research on the library is recommended.

Glossary

GUI: Graphical User Interface

CDN: Content Delivery Network

DOM: Document Object Model

AWS: Amazon Web Services

MVC: Model-View-Controller

GC: General Contractor

Component: React Component

State: React Component state

Props: React Component properties

Reducer: Redux reducers are pure functions that takes state and action as arguments and returns a new state.

Store: Redux store is a representation of collection of application states.

References

[1] Neoteric. Single-Page Application vs. Multiple-page application. [Online]. https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

[2] T. McGinnis, An Introduction to Life Cycle Events in React. [Online] https://tylermcginnis.com/an-introduction-to-life-cycle-events-in-react-js

[3] A. Salihefendic, Flux vs. MVC (Design Patterns). [Online] https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns-57b28c0f71b7

[4] A. Alexander, "The Definition of Pure Function," in *Functional Programming Simplified,* CreateSpace Independent Publishing, 2017, pp. 31-32

[5] Facebook Inc, Optimizing Performance. [Online] https://reactjs.org/docs/optimizing-performance.html

[6] Facebook Inc, JSX in Depth. [Online] https://reactjs.org/docs/jsx-in-depth.html

[7] The State of JavaScript 2017, State Management Tools – Results. [Online] https://stateofjs.com/2017/state-management/results/

[8] Coding The Smart Way, Redux – Introduction To State Management With React [Online] https://codingthesmartway.com/learn-redux-introduction-to-state-management-with-react/

Appendix A webpack.config.js

```
const path = require('path');
const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');
module.exports = {
    entry: "./src/index.js",
    devtool: "source-map",
    plugins: [new CleanWebpackPlugin(["build"]), new webpack.DefinePlugin({
        "process.env.NODE_ENV": JSON.stringify("development")
    })],
    module: {
        loaders: [{
            test: /\.js$/,
            exclude: path.join(process.cwd(), "node_modules"),
            include: path.join(process.cwd(), "src/"),
            loaders: [{
                loader: "babel-loader"
            }]
        }]
    },
    output: {
        path: path.resolve(process.cwd(), "build"),
        filename: "application.js"
    }
};
```

Appendix B Test Results in Detail

Table B-1. Test case (a)

Length of the list	Trial	initial load (ms)	Toggle on (first click)		Toggle off (second click)	
			scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)
1	#1	18.4	2.8	0.2	1.8	0.2
	#2	11.9	3.6	0.3	3.2	0.4
	#3	16.8	3.3	0.2	2.3	0.3
	Average	15.7	3.2	0.23	2.4	0.3
100	#1	22.1	14	0.5	8.5	0.6
	#2	20.1	8	0.7	10.3	0.5
	#3	21.4	10.1	0.6	11.9	0.4
	Average	21.2	10.7	0.6	10.2	0.5
10000	#1	1374.6	451.1	22	326.9	16
	#2	1211.7	313.5	19	313.1	17.1
	#3	1289.4	317	21	386	17
	Average	1291.9	360.5	20.7	342	16.7

Table B-2. Test case (b)

Length of the list	Trial	initial load (ms)	Toggle on (first click)		Toggle off (second click)	
			scripting (ms)	rendering (ms)	scripting (ms)	rendering (ms)
1	#1	13.3	6.9	0.3	3.7	0.2
	#2	8.5	3.1	0.2	1.9	0.2
	#3	9.8	4.4	0.3	3.9	0.2
	Average	10.5	4.8	0.27	3.2	0.2
100	#1	38.3	6.5	0.3	3.3	0.3
	#2	31.7	10	0.7	5.2	1.7
	#3	33.9	6.9	0.4	3.8	0.8
	Average	34.6	7.8	0.5	4.1	0.9
10000	#1	1672	151.6	11.1	110.9	6.9
	#2	1676.7	221.2	18.6	119.5	7.8
	#3	1631	181	15	122	6.5
	Average	1659.9	184.6	14.9	117.5	7.07