Optimizing Web UI Framework Rendering Performance

Duan Li

April 20, 2019

McGill University

School of Computer Science

1 Introduction

Nowadays web technologies are evolving at a blazing speed. With the appearances of new technologies, it's possible for web pages to have richer user interface. Started as just a place for information storage, web page gradually became a full-fledged client side application. Web technologies are no longer limited to web - they can used to provide cloud application services, to build mobile application as well as desktop application.

One of the most important web technologies introduced these years is web framework. It's interesting that just several years before, there were plenty frameworkless web applications in the market, however, most of the web developers would use a web framework to develop application now.

Web frameworks make web app development easier by providing ready-to-use boilerplate project template, enforcing well-known MV* architecture pattern, handling application logic irrelevant details like data binding and DOM manipulation properly so developers can focus on the business logic. The simpleness allows people with relatively less CS background to develop decent web applications within one month of training.

As web apps are growing bigger and got used in mobiles which have limited computing power, the performance of application is one of the most important factors that determine the success of the application. Performance, obviously, is an important evaluation factor of web frameworks.

We are interested in the techniques web frameworks use to improve performance. In this report, we explain various performance optimization techniques for web, evaluate and examine them for possible weak points. Finally, we propose new method.

1.1 Web Framework Overview

Web framework is a software framework that is designed to support the development of web applications[1]. There are two types of web frameworks: the front end framework and the back end framework. In the report, the term "web framework" refers to front end web framework.

Web frameworks provide interfaces for developers to develop client side web application. Developer needs to specify the UI elements, the data (model) that the UI elements depend on, and event handler functions that instruct how the UI elements should react to user events. The framework connects these pieces together, invokes the event handler functions upon events, and update the UI elements upon data changes.

Most web frameworks hide DOM operation from users, so developer can work on a higher level of abstraction without worrying about low level DOM operations. By seperating application logic and DOM operations, the code is better modularized and more maintainable.

Web frameworks are usually opinionated. They specify the standard approach to modularize project elements. Projects based on the same web framework are usually structured in the same way. The standardization and modularization help new members get on board a project faster, improving cooperation efficiency.

There are many web frameworks in the market, Angular.js, Angular 2, React, Vue, etc. This field is under active development these years.

1.2 Scope and Purpose

The purpose is to explore techniques that can be used to improve the framework performance.

Web framework performance consists of two parts: initialization time performance and update time performance. We will focus on the latter.

We define a web framework has better update time performance if the web application based on it reacts to events faster. We quantify this by introducing UI update time, the time required for UI update after an event that modifies the data. Since DOM operations are expensive, they dominate the UI update time. The problem can be transformed to minimizing DOM operations for each UI update.

We built a simplified web framework E.js and integrated it with various UI update methods. We developed a benchmarking system to evaluate the performance of different implementations. Finally we proposed an UI update algorithm VDOM-CDHSI that aimed to improve performance by trying to solve the inappropriate matching problem most virtual DOM based UI update algorithms are suffering from.

2 Evaluation Method

In this section, we explain the mechanism to evaluate our framework's update time performance.

We wrote a todo list application with our framework, and built 3 benchmarks on top of it: "Add 100 Todos", "Delete 99 Todos" and "Reorder 99 Todos". The benchmark can be initiated via button click.

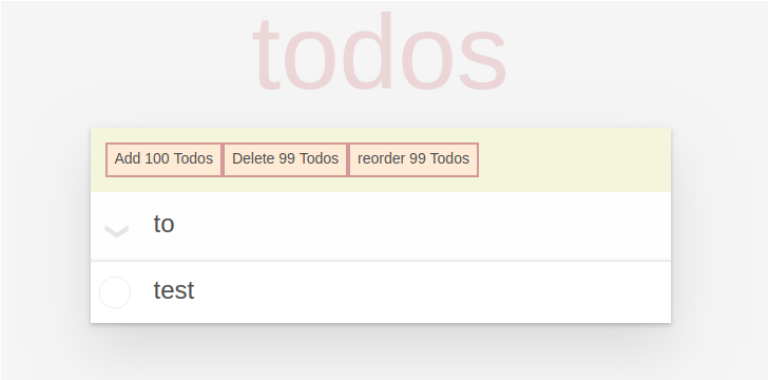
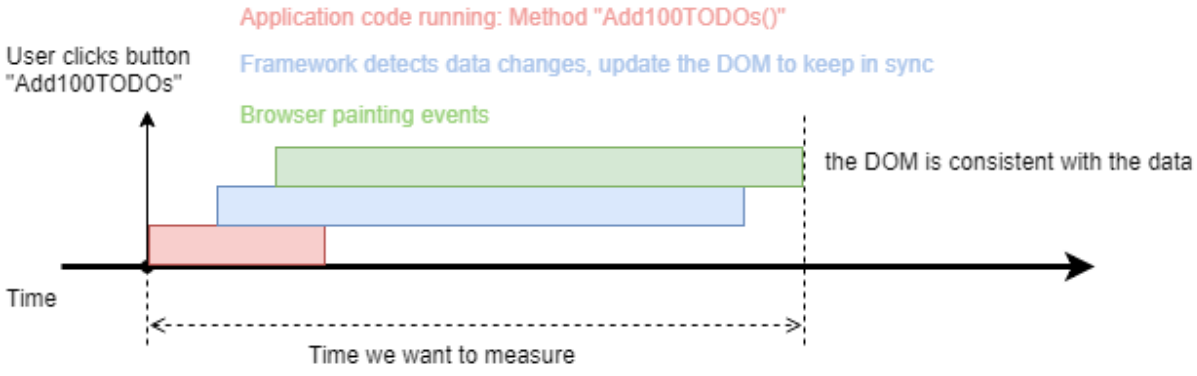


Figure 1. Todo app with 3 benchmarks

For each implementation, we want to measure its UI update time on the three benchmarks.

We define UI update time with an example.



The benchmark "Add 100 Todos" starts when user clicks the button "Add100Todos". The event handler method attached to the button, `Add100Todos()`, will be triggered. The method will create 100 Todo items and add them to the Todo list one by one. So the data *todoList* will change 100 times. The framework will soon detect the data changes, and perform UI update to keep the DOM in sync with the data.

As shown in the figure, UI update time consists of 3 part:

Application Time: the time used to run event handler. (Red Part)

Framework Time: the time framework used perform UI update algorithm. In UI update algorithm, framework decides how to manipulate the DOM and issues DOM commands. (Blue Part)

Browser Time: the time for browser to paint the DOM. (Green Part)

UI update time can be calculated by:

$$BroswerTime.end.timestamp - userClickEvent.timestamp$$

A framework has good update time performance if its UI update time is small.

How to measure UI update time is included as appendix in 8.1.

3 Web Framework Design

We designed a naive web framework E.js based on a open source web framework aoy.js[11]. E.js provides APIs for user to create E.js UI Component, allows user to bind custom function as event handler to UI Component, create data model and bind it with UI Component.

The full requirement specification for E.js is included as appendix in 8.2.

3.1 Framework API

Following is a user guide of E.js. More example programs are available in GitHub.

Component

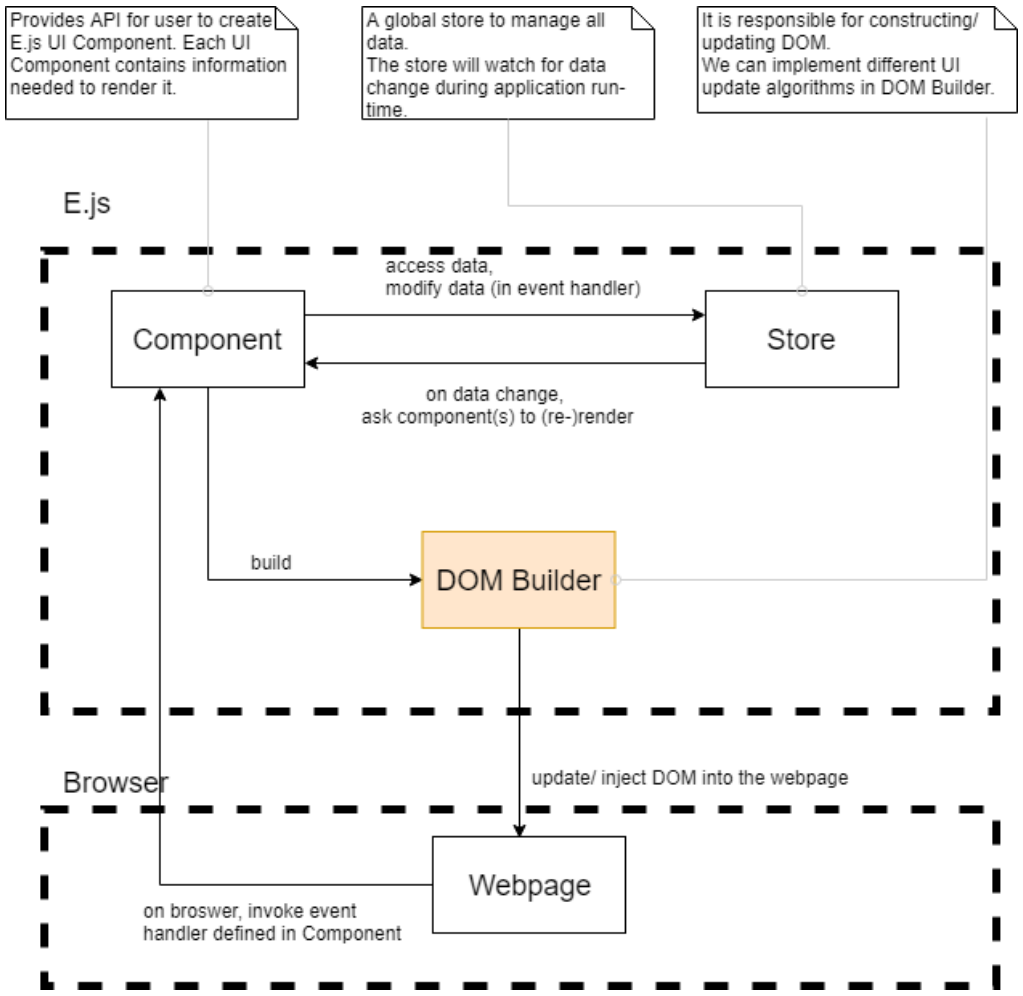
```
// create UI component
let component = new Component({
  // specify where to inject the component,
  // e.g. the component could be injected in document.body
  position: document.body,
  render: () =>
    new ElementNode({
      // a valid html tag, e.g. "Input", "p","h2", etc
      tag: "div",
      // properties for this element, the generated DOM element
      // will have the same properties
      properties: {
      // event handling achieved with property
      onclick: () => console.log("this is div"),
      id: "12" },
      // children must be viewTemplate instance (TextNode or ElementNode)
      children: [
        new TextNode("A div"),
        new ElementNode({
          tag: "input",
          properties: {
            oninput: e => console.log(e.target.value),
            placeholder: "placeholder",
            type: "text"
          }
        }),
        new TextNode("string after input")
      ]
    })
});
```

Data

User can use plain JavaScript object as data model. For simplicity, data is not kept inside each component, it will be stored in a global store. User can access the data with `store.data`.

3.2 Architecture Structure

The following diagram explains E.js's architecture.



4 Related Work

All web frameworks try to modify and adjust the DOM, instead of re-mounting every DOM node on every change. As DOM-element are heavy, the goal is to minimize the cost of DOM manipulations by avoiding unnecessary DOM operations, modifying and reusing old DOM node when appropriate.

A web framework needs to figure out what DOM elements are actually affected and how to update them. It tries to generate a series of DOM operations t that:

1. After applying operations in t in order on the DOM tree, the DOM tree is in sync with the model.
2. The total cost of operations should be minimized.

4.1 Angular.js

Angular.js, one of the most popular web frameworks in the past, uses a "dirty checking" approach[4].

Angular.js uses watchers, basically listeners attached to the scope objects, to synchronize the view with the model.

In the following Angular.js code, two watchers will be created, one for `isMine` variable, one for `myVar` variable.

```
<div ng-show='isMine'>
{{myVar}}
</div>
```

For each watcher, a `watcher` function will be created in the Angular.js's digest cycle. The digest cycle is basically a function that repeatedly checks for changes in the watchers. The `watcher` function executes every time the digest cycle runs, comparing the watcher's old value with it's new value.

A digest cycle can be initiated by data change (probably due to user event). As the digest cycle itself may change object value, digest cycles can be fired for multiple times. It will stop when all watchers stop changing.

With this approach, the performance of UI update depends on the number of watchers. Less experienced programmer may write bad application code that contains many unnecessary watchers, slowing down the application significantly. Furthermore, most of the time Angular.js cannot detect all changes with one digest cycle, running multiple cycles is time-consuming.

4.2 Backbone

Backbone adopts a KVO (key value observation) approach. It uses intelligent data structure for model.

Whenever a UI action causes an attribute of a model to change, the model triggers a "change" event. All the views that display the model's state can be notified of the change, so that they are able to respond accordingly, re-rendering themselves with the new information.[10]

One disadvantage of this approach is that developer has to learn Backbone Model API, he cannot simply use JavaScript object as view model.

4.3 React

React introduces a notion called virtual DOM.

After each event that results in data change, the framework wants to figure out the DOM that are actually affected, and only perform necessary DOM operations. Backbone uses some intelligent data structure for model that knows which DOM nodes depend on it. This adds complexity in data model. Developers usually have to use APIs provided by framework to create data model, this increases learning cost.

React is more developer-friendly in the sense that developers can use normal JavaScript object as data model. With dumb data, React has to compare the DOM node properties with the data model value to figure out the outdated nodes. Querying and comparing DOM nodes are expensive, so React keeps a light weight representation of the DOM in memory, that is virtual DOM. Virtual DOM contains all information required for figuring out the affected nodes, but the data structure is optimized for comparison and manipulation. Additionally the virtual DOM is kept in memory, information retrieval is fast.

On every event, React generates a new virtual DOM tree and compares it with the previous one to find out the affected DOM nodes. It can then perform DOM operations on the affected nodes.

5 Experimenting Different Methods

This section describes 4 UI update methods experimented on E.js. The UI update algorithm is contained in the DOM Builder module; other modules basically remain untouched. Framework API is not modified, so the same application code can be used for all implementations.

5.1 Rebuild Everything

The most naive approach is to rebuild the whole DOM tree on every event. This is our baseline implementation.

5.2 Throttle

The browser's refresh rate is 60fps, so each frame lives for 16 ms. There is no need to rebuild more than once in a frame. The Throttle approach limits the DOM builder for one rebuild operation every 16ms.

We uses `window.requestAnimationFrame` when building the UI. The method takes a function as argument, it informs the browser to call the function before next repaint. We pass `DOMBuilder.build` to `window.requestAnimationFrame`. In this way, we let browser schedule the `build` function, so the UI rebuild can fit into the browser's rendering pipeline. Without `requestAnimationFrame`, it's possible that the `build` function will interrupt the browser's painting job, resulting a frame miss.

5.3 Snabbdom

The third version still throttles the `DOMBuilder.build`. Additionally, in `DOMBuilder.build`, we implement a UI update algorithm based on a popular open source virtual DOM library Snabbdom[5], to replace the full rebuild implementation. Snabbdom is almost identical to the React virtual DOM explained before.

To recap, Snabbdom keeps a light weight representation of the real DOM, called virtual DOM. Upon event, it generates a new virtual DOM tree, compares it with the old one to find out the affected nodes, then perform necessary DOM operations.

During comparison, Snabbdom needs to map nodes in the new virtual DOM tree to the old tree. To make two different nodes the same, whether to delete and create a new one, or to update the old one, depends on if the two nodes are "matched". For example, on a button click event, if the only change is a css style on the button (e.g. the button becomes *pressed*), it should map the new node for the button to the old node. As updating the old DOM node's css style is certainly faster than removing the old DOM node and inserting a new one.

Figuring out a proper matching function tricky. For two objects, deep comparison of value with tolerance on minor difference is time consuming. Unfortunately efficiency is crucial for UI update algorithms. Browser has a refresh rate of 60 fps, so 16ms per frame. Browser has some house keeping tasks to do, so the UI update algorithm has to finish in 10 ms. It's infeasible to do precious computation on deep value similarity of a huge DOM tree. Snabbdom uses a simple heuristic based matching rule: match two nodes if they have the same tag, class, id (if available).

Snabbdom traverses the tree to match the nodes, it performs DOM operation once a possible matched pair is found, i.e. it does not wait for matching to finish before generating the DOM operations. Both nodes matching and DOM manipulations are done in one traversal of the tree. The detail implementation is included in 8.3.

5.4 VDOM-CDHSI

5.4.1 Introduction

Finally, we came up with our own UI Update algorithm VDOM-CDHSI. As the name suggests, is based on Virtual DOM.

Essentially, the Virtual DOM UI update problem is a hierarchical change detection problem, a problem of finding a "minimum-cost edit script" that transforms one data tree to another. Specifically, the data tree are ordered, in which each node has a label and value. The algorithm basically takes two Virtual DOM trees, and finds out a list of DOM operations that:

- Applying DOM operations in order on the old tree will transform it to the new tree.
- The total cost of DOM operations should be as small as possible, in other words, the total time required for the browser to perform these DOM operations should be as small as possible.

Additionally, the algorithm should be efficient, for reason explained before.

The minimum tree edit distance is a NP hard problem. The state of art generic algorithm[7] has a complexity in the order of $O(n^3)$ where n is the number of elements in the tree. React and snabbdom implements a heuristic $O(n)$ algorithm.

React and snabbdom's algorithm is efficient, however, it does not always generate the a good edit script (the list of DOM operations) as it cannot map nodes properly sometimes due to its simple matching rule.

We proposed an experimental solution: VDOM-CDHSI that supports more sophisticated matching. It's based on a generic tree edit script algorithm[9], with some modification so it can better adapt to the web framework problem domain.

5.4.2 Detailed Algorithm

Operations

There is a fixed set of supported operations that can be applied to a tree node.

- INSERT(x, y, k): insert a new leaf x as a child of y, just before the child k.
- DELETE(x): delete leaf node x.
- UPDATE(x, y): update node x so it has the same value as y.
- MOVE(x, y, k): move subtree rooted at x to child of y, just before child k.

Operation has cost. In VDOM-CDHSI, we use a simplified cost model: $cost(INSERT) = cost(DELETE) = cost(MOVE)$. $cost(UPDATE)$ is given by a distance function `compare`, that evaluates how different the old value is from the new value. The `compare` takes two nodes as argument and returns a number in the range [0, 2]. The distance function will be compatible with the cost model. For small difference where UPDATE + MOVE is preferred, the `compare` will return a number smaller than 1. Otherwise, DELETE + INSERT will be chosen.

Two Sub-Problem

VDOM-CDHSI divides the change detection problem into two sub-problems:

- Finding a "good" matching between the nodes of the two trees;
- Finding a minimum edit script for the two trees given a computed matching.

Finding Matching

In this part, we want to find a matching between the nodes in the old Virtual DOM tree $T1$ and new Virtual DOM tree $T2$. This mapping will be used to generate the edit script later.

The matching criteria depends on the domain being considered. In VDOM-CDHSI, the `generateEditScript` algorithm is deterministic. To evaluate the quality of a matching M, we can consider the total cost of edit script returned by `generateEditScript(M)`. The goal is to find out a matching such that it minimizes the cost of minimum edit script computed based the matching. Additionally, we want to make sure the algorithm is efficient and can finish in 10ms.

In VDOM-CDHSI, we match leaf nodes and internal nodes based on different rules.

- For leaf node:
  - Match (x, y) if $label(x) = label(y)$ and $compare(value(x), value(y)) <= f$.
  - *compare* is a function to measure how different the two nodes are. VDOM-CDHSI uses a very simple *compare* function. For TextNode, it compares the text string length without looking into the content. For ElementNode, it decides based on the number of different properties.

- For internal nodes:
  - Match (x, y) if $label(x) = label(y)$ and $matchedChildrenCnt(x, y)/max(|x|, |y|) > t$.
  - For internal nodes, VDOM-CDHSI matches them if they have many common children. $|x|$ denotes the number of children of x.

Inspired by React VDOM, VDOM-CDHSI only compares nodes at the same depth to speed up the comparison. The assumption does not degrade the result much, as in web applications, it's rare to have node move up or down in depth.

Generating the Edit Script

Given an old Virtual DOM tree $T1$, a new Virtual DOM tree $T2$, and a partial matching M between their nodes, generates a minimum cost edit script E that conforms to M and transforms $T1$ to $T2$.

The algorithm consists of 5 phases:

1. The Update Phase: For any node pair $(oldVNode, newVNode)$ in matching M such that $val(oldVNode) != val(newVNode)$, perform $UPDATE(oldVNode, newVNode)$.

2. The Align Phase: A node pair $(oldVNode, newVNode)$ has misaligned children if:

   - u and v are children of $oldVNode$ and u' and v' are children of $newVNode$.
   - (u, u') and (v, v') are node pairs in M.
   - Node u is to the left of v in $T1$ and node u' is to the right of v' in $T2$.

   For all node pair $(oldVNode, newVNode)$ in M, check all pairs of children of the $oldVNode$ and $newVNode$. Perform MOVE to reorder misaligned children.

3. The Insert Phase: For any unmatched node $newVNode$ in $T2$. Build newVNode and perform INSERT to add the new node as a child of $parent(newVNode)$'s partner. At the end of this phase, all node in $T2$ has been matched, but there are still unmatched nodes in $T1$.

4. The Move Phase: Look for node pairs $(oldVNode, newVNode)$ in M such that $(parent(oldVNode)$ is not in M. Since after the Insert Phase, all nodes in $T2$ are matched, so $parent(newVNode)$ has a partner p in $T1$. Perform MOVE operation so oldVNode becomes a child of p.

5. The Delete Phase: DELETE all remaining unmatched nodes in $T1$.

In actual implementation , the first four phases (the update, insert, align and move phases) can be combined into one breadth-first search on $T2$. Then we do a post-order traversal of $T1$ for the delete phase.

6 Result

We evaluated the 4 implementations on the 3 benchmarks: ”Add 100 Todos”, “Delete 99 Todos” and “Reorder 99 Todos”. The Benchmark Runner will run each benchmark 10 times to compute the median.

Following is the experiment environment detail and result:

CPU	Intel® Core™ i7-4710MQ CPU @ 2.50GHz × 8
OS	Ubuntu 18.04 LTS
OS-type	64-bit
Browser	Chrome, Version 73.0.3683.86 (Official Build) (64-bit)

As shown in the graphs, for all the benchmarks, snabbdom performs the best. The result is not a surprise as snabbdom is one of most popular open source Virtual DOM library and the virtual DOM used by Vue.js.

The benchmark “Add 100 Todos” is the easiest for all implementations, and the “Reorder 99 Todos” is the hardest one. For Baseline and Throttle, the UI update time for “Delete” benchmark is slightly longer than the first benchmark ("Add" benchmark). The UI update time difference (between benchmark 2 and benchmark 1) is 80 ms (1300 - 1221) for Baseline, and 118 ms (1053 - 935) for Throttle. The “Reorder” benchmark takes much longer, with an increase of 500ms. We believe the significant time increase is due to the increase of average DOM elements per frame in the webpage. For benchmark 1, we add todo item into an empty todo list one by one until there are 100 items in the 100 todo items. If we average the number

of todo items for all frames, a frame has 50 todo items. For benchmark 2, we delete todo item from a todo list that has 100 todo items initially one by one until there is only one left. On average, a frame has 50.5 todo items. For benchmark 3, the number of todo items remain the same during reordering. On average, a frame has 100 todo items. The more todo items, the more DOM elements in the webpage. Both Baseline and Throttle rebuild the whole DOM tree, intuitively, the more the DOM elements, the longer the update time.

Meanwhile, we notice that even though Snabbdom and CDHSI also spend more time on the "Reorder" benchmark, the increase is much smaller. Especially for Snabbdom, the time difference between "Reorder" and "Delete" is only 15 ms (626 - 611). This shows that Snabbdom successfully reuses some old DOM nodes and avoids some unnecessary DOM manipulations during "Reorder" benchmark.

Throttle outperforms the Baseline by batching some changes in one UI update. We print out the data change events and DOM Builder build events for Benchmark 1, as shown in the following Figure 2 and Figure 3. As the log suggests, Baseline rebuilds the DOM whenever a new todo is added. In contrast, Throttle rebuilds once for every 1 - 3 new todos. The variation in number of todos added per rebuild probably contributes to the larger standard deviation in Throttle processing times.

While CDHSI provides significant improvement on Throttle, it does not perform as well as Snabbdom, especially for benchmark 2 and 3. For benchmark 1, the performance difference is relatively trivial (75 ms). The difference is probably due to the way CDHSI and Snabbdom handles matching and DOM manipulation. As explained before, Snabddom only does one traversal of the tree, matching nodes and manipulating nodes at the same time. Differently, CDHSI splits the two problems, first performs a tree traversal to generate a matching, then computes the edit script and manipulates the DOM based on the matching. For the second part, it will do two tree traversals, one breadth-first search for UPDATE, ALIGN, INSERT, MOVE, and one post-order traversal for DELETE. Intuitively, CDHSI will perform worse than Snabddom if they come up with a similar matching.

There is still improvement space for Snabddom, it has problem matching nodes properly sometimes and it may generate a edit script that's far from the optimal solution. With only one tree traversal, it's relatively hard to improve the matching feature in Snabddom. The complexity of CDHSI allows more fine tuned matching. CDHSI introduced the idea of operation cost, distance (similarity) between nodes. With a proper cost function, CDHSI probably can figure a better matching than Snabddom, resulting in a better edit script.

Apparently the current cost function in CDHSI is not ideal. The considerable performance difference between Snabddom and CDHSI in Benchmark 2 and 3 suggests some problem in matching of two list. After some research, we found that in all the three benchmarks, CDHSI matches the ith todo item in the new list to the ith todo item in the old list. This matching works for the first benchmark as we are only adding new todo item at the end of the todo list. For benchmark 2, we remove todo items every time from the top of the list, so the ideal matching is to map the ith todo item in new list to the $i+1$ th todo item in the old list. Similarly, in benchmark 3, the mapping is wrong.

```
[STORE INFO] todos is modified to  ▶[{…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(2) [{…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(3) [{…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(4) [{…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(5) [{…}, {…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(6) [{…}, {…}, {…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to  ▶(8) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
```

Figure 2. Baseline Log

```
[STORE INFO] todos is modified to
  (88) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}]
[STORE INFO] todos is modified to
  (89) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}, {…}]
[STORE INFO] todos is modified to
  (90) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to
  (91) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
[STORE INFO] todos is modified to
  (92) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}, {…}, {…}, {…}, {…}]
[STORE INFO] todos is modified to
  (93) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
▶  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
   {…}, {…}, {…}, {…}, {…}, {…}]
② [DOM BUILDER] building component.
```

Figure 3. Throttle Log

7 Summary and Future Work

We have motivated the run-time performance problem of web frameworks. We have built a naive web frameworks and implemented 4 UI reconciliation algorithms on it: the "Rebuild All" solution, the Throttle solution, a popular open source Virtual DOM library snabbdom, and our own solution VDOM-CDHSI based on a generic tree minimum edit script algorithm. We evaluated the four implementations on 3 benchmarks.

Our UI reconciliation algorithm VDOM-CDHSI is an attempt to lower total cost of DOM manipulations by providing better matching. However, the matching is not ideal at this point due to a lack of proper cost function. Further studying in the cost for various DOM operations is needed.

Additionally, to get a relatively overall and unbiased performance evaluation of frameworks, we need to improve our benchmarking system. More benchmarks should be added. For now we only have three benchmarks that are all related to list operation, we should have more varied benchmarks. The benchmarking system should be configured to support different browsers and different version of a browser. As different browsers, or even different version of a same browser, may perform differently due to some browser optimizations.

Finally, Virtual DOM is not the only way to do UI reconciliation. There are many other approaches for this problem that worths investigation: "intelligent" data-structures, Memorized DOM, etc.

8 Appendix

8.1 Benchmarking System Implementation

As explained in 2, web framework update time performance is quantified by UI update time. This section explains how to measure the UI update time.

8.1.1 window.setTimeout

A intuitive solution would be:

1. Keep track of the initial click event time T_0.
2. At the end of event handler method, runs a for-loop, in each iteration:
   (a) Check if the current DOM is consistent with the data (if the UI update is done).

Since JavaScript is single-threaded, we do not want the for-loop keeps running nonstop, blocking other execution. The `check` can be performed once in a while. We implement this using `window.setTimeout(check, x)`, so `check` once in time x.

This method gives us a rough estimate of performance that's far from ideal.

Unreliable setTimeOut: `setTimeOut` does not guarantee the `check` method will be performed once in time x. It will put the `check` method in JavaScript execution queue every time x. If there are functions waiting in the queue, there could be delay.

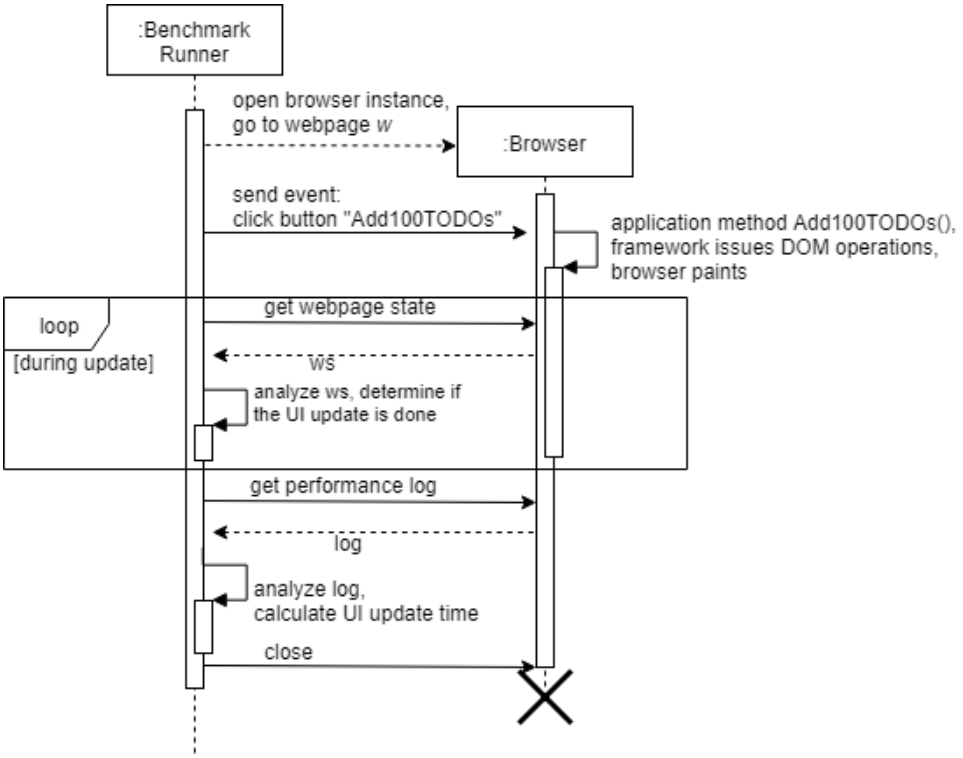
Degraded Performance: Running the `check` method increases the UI update time, as the the time for `check` method is also included. Evaluation code should have as little influence on the application performance as possible.

Fault in Design: In the design, we run the `check` once in time x, thus the result can be off by a max value of x. Larger the x, less precise the result. However, with a small x, `check` will be executed more frequently and make up a significant part of UI update time, ending up with a large influence on the result.

8.1.2 Selenium WebDriver

We decided to follow a Selenium based approach used in a open source benchmark project for JavaScript frameworks[2].

Selenium is a suite of tools that automates browsers. Selenium comes with a Browser Driver that can send event to browser, monitor current webpage state, read raw browser performance log entries. The Selenium based benchmarking system will be completely separate from the application code.



The Benchmark Runner is an independent process. As illustrated in the diagram, the Benchmark Runner will open the browser as a new process, instructs it to load the web page that contains the benchmarks. When the web page is loaded properly, the Benchmark Runner will send event to start the benchmark. For the "Add 100 Todos" benchmark, it would be a click event on button "Add 100 Todos".

The browser receives the event and passes it to the web application, like mentioned before, the event handler `Add100Todos()` will be triggered, data will be modified, framework will issue DOM operations to keep the DOM in sync with the data, browser will start painting.

While browser is working, Benchmark Runner queries webpage state to determine if the update is done. Given that the Todo list is empty initially, we can assume the UI update is done when we detect there are 100 Todo items in the list. Therefore, the query for "Add 100 Todos" would be "check if the 100th Todo item exists in DOM".

After the UI update is done, the Benchmark Runner will fetch all the raw performance log entries from the browser's timeline. The performance log entries includes entries for user events, the beginning and end of a application function call, as well as that of browser rendering events. All entries include a type and a timestamp.

The Benchmark Runner will analyze the performance log entries, do some basic validation to make sure the evaluation procedure is correct, and find out the initial click event entry e_1 and the end of final browser paint entry e_2. The UI update time can be computed by the $e_2.timestamp - e_1.timestamp$.

The benefit of this approach is obvious. Since Benchmark Runner and the web application are two separate process, there is no timing code or `check` method in the application process, leaving the application performance relatively uninfluenced. Additionally, there are a lot helpful information we can extract from the performance log entries. In addition to the process time, we can compute the number of paint events, which could be helpful for our framework performance analysis.

8.2 E.js Requirement Specification

Following is the detailed requirement specification for E.js.

No Extra compiler: All applications built on E.js should be compilable by normal JavaScript (ES6) compiler. The framework API should only use valid JS syntax.

UI Template: E.js should allow user to specify UI template for a UI component.

Data: E.js should allow user to specify data (model) for UI component, the data could be used in templates. Data could be shared by multiple components.

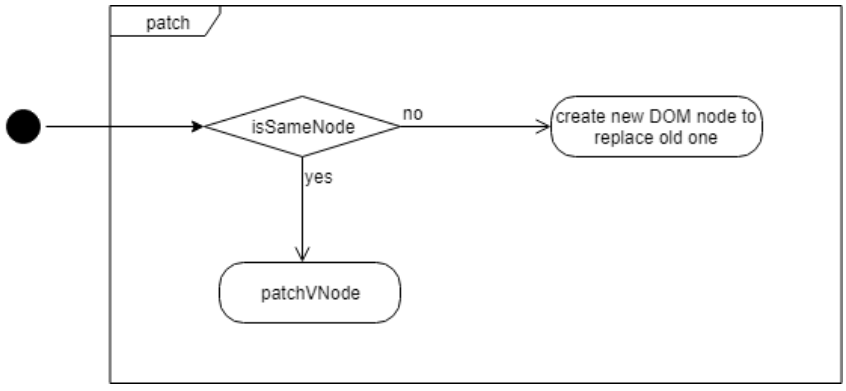
Custom Method: E.js should allow user to define custom methods, and bind the custom method to browser event(s).

Component Hierarchy: While having multiple components in the same level is supported, parent-child component is not supported. The same can be achieved with a giant component.

Only Element in HTML namespace: Elements in other namespace like SVG, MathML are not supported.

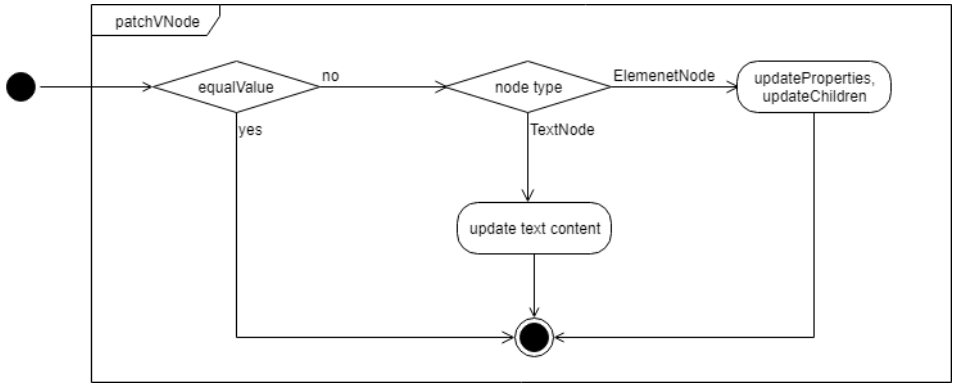
8.3 Snabbdom Implementation Detail

Snabbdom started with a method `patch`. `patch` takes two arguments, the old virtual DOM tree and the new virtual DOM tree. It compares two virtual DOM trees, and figures out the actual differences.

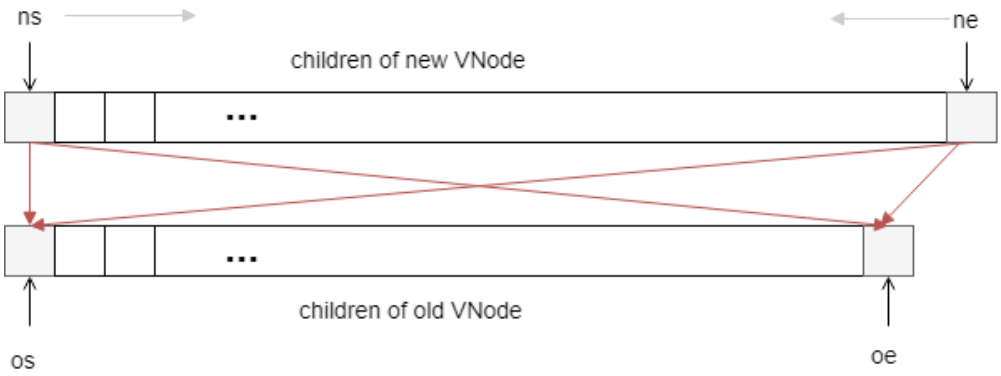


In `patch(newVNode, oldVNode)`, it first compares if the two VNodes (virtual DOM

Node) are the same node. Here the same node does not imply that the two VNodes have the same value, we will match two nodes as long as they are of the same type (both TextNodes or both ElementNodes) and same tag (if they are ElementNodes). If the two VNodes are not the same, it will build a new DOM node and replace the old DOM node with it. Otherwise, it will call the reuse the old DOM and update its value using `patchVNode(newVNode, oldVNode)`.



In `patchVNode(newVNode, oldVNode)`, it will compare the value of the two VNodes. For VNodes with same value, it can return without doing update. Otherwise, it will update based on the node type. For TextNode, since TextNode is always a leaf node, it can simply update the text content and return. For ElementNode, in addition to updating the value of the current node, it also needs to update the its children.



In `updateChildren(newVNode, oldVNode)` compares two children list. It uses 4 pointers (*ns*, *ne*, *os*, *oe*), 2 for each children list, one pointer to the start node and one pointing end node. It checks 4 nodes pairs in order to see if they are the same nodes: (*ns*, *os*), (*ne*, *oe*), (*ns*, *oe*), (*ne*, *os*). Like in patch, being "same nodes" does not imply the two nodes have the same value, it just mean the two nodes match. If none of the 4 pairs is a match, it will create map for all the nodes, and try to find if there exist a node in the other list that has the same identifier. The identifier could be computed with a combination of object type, tag, class and id.

If a match (*childNew*,childOld) is found, it moves the pointers towards the middle of the list and calls `patchVNode(childNew, childOld)`. When two pointers in a children list

meet, all children have been visited, so the list traversal will stop. If there are unvisited nodes in the other children list, it will remove/ add DOM nodes correspondingly.

We believe that the way it compares nodes is largely based on heuristic. As in web application, for most events, the list items remain in the same order. Reorder event is not frequent. The purpose of comparing the start with the end is probably to catch the cases that the start node is moved to the end. Probably this kind of reordering is more common.

9 References

[1] Web Framework

[2] JS framework benchmark

[3] Selenium

[4] Angular.js Digest cylce, Angular.js Digest cylce 2

[5] snabbdom

[6] requestAnimationFrame

[7] generic tree edit distance algos

[8] React reconciliation

[9] Change Detection in Hierarchically Structured Information, Sudarshan S. Chawathe

[10] Backbone

[11] aoy.js