MSc thesis

Master's Programme in Computer Science

A performance comparison of rendering strategies in open source web frontend frameworks

Risto Ollila

April 12, 2021

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. T. Mikkonen

Examiner(s)

Prof. T. Mikkonen, Dr. N. Mäkitalo

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Risto Ollila

Työn nimi — Arbetets titel — Title

A performance comparison of rendering strategies in open source web frontend frameworks

Ohjaajat — Handledare — Supervisors

Prof. T. Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	April 12, 2021	51 pages, 14 appendice pages

Tiivistelmä — Referat — Abstract

The Web has become the world's most important application distribution platform, with web pages increasingly containing not static documents, but dynamic, script-driven content. Script-based rendering relies on imperative browser APIs which become unwieldy to use as an application's complexity grows. An increasingly common solution is to use libraries and frameworks which provide an abstraction over rendering and enable a less error-prone declarative programming model.

The details of how web frontend frameworks implement rendering vary widely and can potentially have significant consequences for application performance. Frameworks' rendering strategies are typically invisible to the application developer, and may consequently be poorly understood despite their potential impact.

In this thesis, we review rendering strategies used in a number of influential and popular web frontend frameworks. By studying their implementation details, we discover ways to categorize and estimate rendering strategies' performance based on input sizes in update loops. To verify and measure the effects of these differences, we implement a number of benchmarks that measure different aspects of rendering.

In our benchmarks, we discover significant performance differences ranging up to an order of magnitude under some conditions. Additionally, we confirm that categorizing rendering strategies based on input sizes of update loops is an effective way to estimate their relative performance. The best performing rendering strategies are found to be ones which minimize input sizes in update loops using techniques such as compile-time optimization and reactive programming models.

ACM Computing Classification System (CCS)
Information systems → World Wide Web → Web applications
General and reference → Cross-computing tools and techniques → Performance
General and reference → Cross-computing tools and techniques → Measurement

Avainsanat — Nyckelord — Keywords

web frontend frameworks, web application rendering performance, react, vue, svelte, blazor, angular

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

From its humble beginnings as an application for viewing simple hyperlinked documents containing primarily text, the browser of today is a full-fledged application platform with an ever increasing number of capabilities [31]. The journey has been in many ways unplanned, with incremental additions to existing technologies being used in unexpected ways to create content not necessarily originally envisaged by stewards of web standards. Nothing exemplifies this better than JavaScript: initially conceived at Netscape in 1995 as a minor scripting language [39], JavaScript today is used to develop everything from enterprise software to VR games and space shuttle GUIs [18].

Modern browsers offer a multitude of ways for implementing applications [34]. Browser extensions such as Flash and Silverlight have largely been superseded by native browser capabilities, many of which are accessible to scripts in the browser's JavaScript execution context. These client-side Web APIs can be used alongside static content and enable adding dynamic content incrementally to otherwise static web pages.

On static web pages, any changes to the document being viewed require page navigation, where the current page is discarded and a new one fetched from the server [24]. With techniques such as DHTML [34] and AJAX [24], it is possible to dynamically modify the document, whether to add small bits of dynamic content or to replace page navigation entirely [21]. These techniques, which we will collectively call script-based rendering, can be used even if a the content on a page is primarily static, blurring the line between a web page and a web application. The performance characteristics of a web page, therefore, have less to do with its content than they do with the way it is rendered.

For an application to be perceived as responsive, many user interactions require almost instantaneous feedback. One standard proposed by Miller suggests that such actions should take no more than 100ms [23]. 50 years later, web pages may struggle to achieve this due to the need to load resources and re-render the entire page on most user interactions. Script-based rendering, on the other hand, can fetch resources asynchronously while incrementally updating the user interface [24], or avoid resource loading entirely if the interaction can be handled locally. Responsiveness then depends primarily on the cost of script execution and speed of the browser's rendering process.

Runtime costs of script-based rendering depend not only on the complexity of the applica-

tion, but on the tools used to implement it. Script-based rendering relies on browser APIs which are error-prone to use in complex applications (chapter 3). As a result, we have in recent years witnessed a rapid growth in popularity* of a new generation of JavaScript libraries and frameworks which provide an abstraction over browser APIs and greatly simplify application development. When frameworks are used to manage rendering, they may impose costs on performance depending on how they implement rendering. This potentially makes the choice of a framework and its particular rendering strategy of primary importance when developing browser applications. A poorly performing rendering strategy might hinder or entirely prevent the goal of achieving responsiveness, which is the purpose of using script-based rendering in the first place.

In this thesis, we review a selection of open source web frontend frameworks and investigate their rendering strategies with the aim of understanding their performance characteristics. We will attempt to answer the following research questions:

- RQ1: What rendering strategies are used in web frontend frameworks?
- RQ2: Which factors affect the performance of each strategy?
- RQ3: What are the measured performance differences between strategies?

To our knowledge, no previous work exists that attempts to systematically study script-based rendering strategies in the context of browser applications and frontend frameworks. Previous work has primarily focused on the costs of the initial page load, where costs of resource loading may dominate [37], or in the performance of the browser's content rendering process [22]. We believe that with the rising use of web frontend frameworks, the potential impact of script execution on responsiveness beyond initial page load deserves to be better studied.

Before investigating frontend frameworks specifically, we will first need to understand how browsers render content in general. This will be explored in Chapter 2. In Chapter 3, we will review in detail some of the more popular open source web frontend frameworks and their approaches to rendering content. In Chapter 4 we will introduce benchmarks which we have implemented for measuring performance differences between the frameworks, the results of which will be discussed in Chapter 5. A conclusion with final thoughts will be presented in Chapter 6.

*As measured by the annual State of JS survey at https://stateofjs.com

2 Rendering in a browser

The browser of today, despite the growth in capabilities and complexity that it has experienced over the years [31], will happily a web page built three decades ago at the web's inception[†]. This is because the fundamental building blocks of a web page remain the same, as does the browser's fundamental mode of operation. A web page today, as then, consists of a document described in a markup language, and a set of resources associated with the document. When pointed to a URL containing a web page, the browser will fetch the page and proceed through a series of steps required to render it. In the rest of this chapter, we will examine this rendering process and some of the technologies that enable it as they are today.

2.1 HTML and DOM

HTML (HyperText Markup Language)[‡] is the markup language that was used to describe the earliest web pages [7], and in its updated form is used to describe the vast majority of them today. An HTML document describes the structure and content of a web page. It does this by describing the individual elements which the page contains, as well as metadata such as links to any external resources associated with the page such as scripts and stylesheets. Elements can be nested and may contain attributes such as styles, classes or hypertext references. Certain elements, such as input elements, may be directly visible to the user and can be interacted with, whereas others only specify structural or semantic information (figure 2.1).

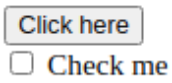
HTML is processed by browsers by parsing it into an object representation, called the Domain Object Model (DOM)[§]. Officially the term "DOM" encompasses both the tree data structure that represents the document, as well as the programming interfaces which scripts can use to access it. Commonly, however, "the DOM" is used to refer specifically to the data structure, with "DOM APIs" referring to the programming interfaces. We will follow this convention in this thesis.

[†]The first public website can still be visited at http://info.cern.ch/

[‡]https://html.spec.whatwg.org/

[§]https://dom.spec.whatwg.org/

```
<div>
  <div id="button-container">
    <button>Click here</button>
  </div>
  <div id="checkbox-container">
    <input type="checkbox" />
    <label>Check me</label>
  </div>
</div>
```



Check me

Figure 2.1: An HTML fragment and the resulting view. The *input* elements produce visible controls that can be interacted with, while the wrapping *div* elements produce no visible output.

In HTML, the structure of a document is represented through nested elements, expressed as tags. In the DOM, these are represented as nodes in a tree, each node and its attributes corresponding to an HTML element and its attributes (figure 2.2).

```
<div>
  <label id="label-name" for="name">Enter name</label>
  <input id="input-name" type="text" name="name" />
</div>
```

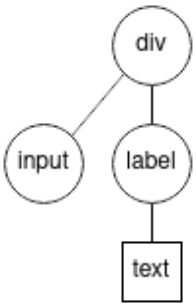


Figure 2.2: An HTML fragment and the corresponding DOM subtree.

The DOM node tree can be modified programmatically at runtime using DOM APIs [33]. These consist of imperative functions that allow querying and direct manipulation of the node tree, such as finding a particular node, modifying node attributes, removing nodes or adding new nodes (figure 2.3). Because DOM APIs allow for creating arbitrary types of nodes and node attributes, any DOM tree that can be built by parsing an HTML document can also be built using DOM APIs.

The DOM specification also includes an event system, where certain actions such as user interactions or network activity can cause events to be dispatched. Scripts can attach

```
<script>
  const label = document.querySelector("#label-name");
  const input = document.querySelector("#input-name");
  const newInput = document.createElement("input");
  newInput.setAttribute("type", "number");
  const parent = input.parentElement;
  parent.removeChild(input);
  parent.appendChild(newInput);
  label.textContent = "Enter age";
</script>
```

Figure 2.3: Using DOM APIs to modify the node tree.

listener functions which react to such events, which makes possible dynamic behaviour based on user interaction. Together, the event system and node tree modification enable arbitrarily complex applications to be built using DOM APIs alone.

2.2 CSS

Cascading Style Sheets (CSS) is a language for defining the styling and presentation of a document. CSS is based on rules which apply a set of styles to a set of HTML elements matching a particular selector. Selectors can be based on the tag name of an element or on its attributes, such as its *class* or *id* attributes.

As with HTML, CSS is parsed into an object model, called CSSOM (CSS Object Model). Whereas the DOM node tree is a structure of nodes representing elements, CSSOM is a tree of selectors that describe the styles that should be applied to a DOM node based on its type and attributes [10]. As with the DOM, CSSOM exposes various APIs to scripts which can be used to programmatically modify the CSSOM node tree. In practice, CSSOM APIs are not commonly used by web applications: any desired visual change can be achieved by describing the possible style variations in CSS with different rules, and then using DOM APIs to modify DOM node attributes so that the desired nodes match a different set of CSS selectors, with different styles being applied to them as a result (figure 2.4).

It is possible that there are performance differences between this commonly used approach and that of directly modifying the CSSOM node tree, but this is beyond the scope of our investigation. For our purposes, it is sufficient to be aware that web applications and web frontend frameworks will generally rely entirely on DOM APIs, with CSS being relegated to a more static role.

```
<p id="text">The script adds a class attribute to this element</p>
<style>
  .text-green {
    color: green;
  }
</style>
<script>
  const element = document.querySelector("#text");
  element.classList.add("text-green");
</script>
```

Figure 2.4: Using DOM APIs to change the styling of an element.

2.3 JavaScript

JavaScript* is a dynamically typed scripting language supported by all major browsers. Although having its origins in the browser, JavaScript is increasingly used in other contexts as well, with web application servers using the Node.js† runtime being the most prominent example. A full exploration of JavaScript's features is not possible or relevant here, but we will note some of the features that pertain to discussion in the following chapters.

Aside from primitive types such as numbers, everything in JavaScript is an object [39]. An object is a key-property container where both keys and properties can be of arbitrary types. Each property has associated accessors – getters and setters – which can be optionally defined as functions to be invoked when a property is read or written. Properties further have descriptors, which describe metadata related to the property, such as whether the property is enumerable or writable.

Objects in JavaScript utilize prototypic inheritance [39]. All objects are part of a prototype chain with a common object prototype sitting at the top, and objects can be created using another object as their prototype. If an accessed property is not found in an object, each object further up in the prototype chain will be checked in turn until either the property is found or the end of the chain is reached.

In browsers, JavaScript is executed by default in a single thread, that being the same thread that is used to render the user interface [20]. All JavaScript execution therefore has the potential to degrade user experience by blocking user interaction. Browsers allow the creation of separate worker threads that can communicate with the main thread using events, but worker threads cannot access most browser APIs, including DOM APIs. All web frontend frameworks presented here perform all their work in the main thread,

*https://262.ecma-international.org/
†https://nodejs.org/

JavaScript is a popular target for transcompilation for other languages that wish to target the browser as a platform. One prominent example is TypeScript*, a language developed by Microsoft. TypeScript is a superset of JavaScript which adds optional static typing to the language. It is of some interest to us in the context of web frontend frameworks, as its use in place of JavaScript is supported by many of them and required by some.

2.4 WebAssembly

WebAssembly† is a bytecode specification developed by the major browser vendors, with an explicit aim of enabling the creation of high-performance web applications using languages other than JavaScript [14]. It is a somewhat immature technology still undergoing fundamental development, and is not yet widely used. It is, however, at least partially supported by all major browsers and WebAssembly-based web frontend frameworks already exist. We will investigate one such framework in this thesis, for which reason we will introduce the basics concepts of WebAssembly here.

Unlike JavaScript, which is served as raw source code to be parsed and compiled by browsers, applications targeting WebAssembly are compiled ahead of time to WebAssembly bytecode. This compiled code is then executed in a WebAssembly virtual machine, with promises of "near-native" speeds of execution.

A central feature of WebAssembly is its embeddability: although having been born in the browser, it places only limited requirements on the host system and has consequently already been implemented on numerous other platforms‡. WebAssembly code is compiled into modules which provide an interface for importing and exporting functions from and to the host environment. In browsers, the host environment is the browser's JavaScript engine. This enables straightforward interaction between JavaScript and WebAssembly: WebAssembly modules can invoke imported JavaScript functions and export functions which can in turn be invoked using JavaScript.

This interoperation with JavaScript is important, because WebAssembly does not have direct access to DOM APIs [38]. Instead, all access must go through the JavaScript interoperability layer. In practice, a WebAssembly application that wishes to use DOM APIs can be bundled with a JavaScript counterpart that mediates API access. This raises

*https://www.typescriptlang.org/

†https://webassembly.org/

‡For examples, see https://github.com/appcypher/awesome-wasm-runtimes

questions over performance: on the one hand, code executed within the WebAssembly context itself is supposedly fast, but the requirement for interoperation with JavaScript is a potential source of overhead.

2.5 Critical Rendering Path

The process whereby the browser turns the various resources that describe a web page or web application into pixels on a user's screen is called the critical rendering path by browser vendors [11]. The various terms used by browser vendors for the different phases differ, but major browsers all perform this along the lines of the process shown in figure 2.5 [29].

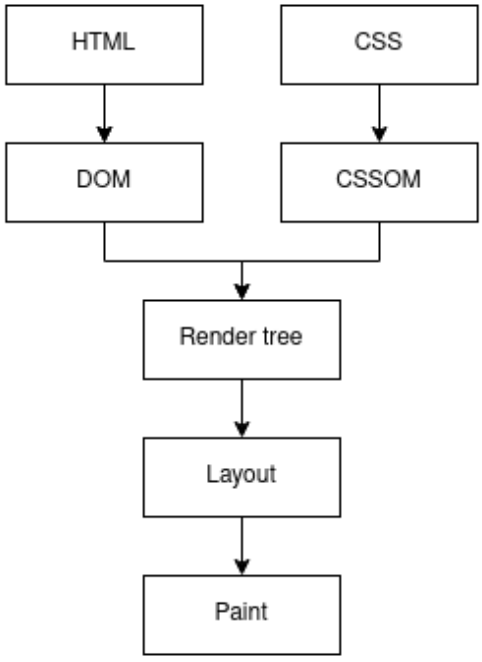


Figure 2.5: The various steps in the critical rendering path.

The process begins with parsing of source HTML to create the DOM tree. DOM creation is incremental: the node tree is constructed as the parser progresses through the HTML document. If, during parsing, the parser encounters links to other resources, it dispatches requests to fetch these and continues parsing and DOM construction. Once DOM construction is finished, any stylesheets are parsed to create the CSSOM.

While the DOM contains all the elements present in a document, it cannot be used to directly render the results, because some of the elements may not be visible. This includes

elements which according to rules present in the CSSOM are hidden. Therefore, once the DOM and CSSOM have been fully constructed, the browser creates a render tree using DOM and CSSOM. The render tree is a subset of the DOM tree containing only elements which will be rendered, with their styling details included from the CSSOM.

When the render tree is completed, the browser calculates the sizes and positions of each element in the tree. This process is variously called layout or reflow. Because the browser cannot know which elements are visible in the viewport until the entire render tree has been processed, no elements are actually painted until the entire tree has been processed. After the positions and sizes have been determined, the browser will finally paint the pixels on the screen, rendering the web page visible.

Scripts can affect and halt the critical rendering path. As described previously, scripts can use the DOM and CSSOM APIs to arbitrarily modify the DOM and CSSOM node trees. For this reason, the browser will by default stop DOM construction when it encounters a script tag, at which point it will fetch, parse and execute the script, before continuing with DOM construction. It is possible to declare a script resource as asynchronous, in which case the browser will not stop DOM construction, but will instead download the script resource in the background and execute it once DOM construction is complete.

The critical rendering path is triggered in full only when a page is initially loaded. Various actions can cause partial re-evaluation, however. Whenever DOM APIs are used to modify the DOM, the render tree must be at least partially reconstructed, which triggers the layout and paint phases as well. The layout and paint phases can also be triggered by user interaction without the need for render tree reconstruction: some HTML elements have built-in ways in which the user may interact with them which may cause the layout to change. Along with script execution, the costs of render tree construction and the layout and paint phases are the primary factors affecting an application's responsiveness when the DOM is modified using scripts.

3 Web frontend frameworks

A web application is an application distributed via a web server and accessed using a browser. Web applications can be rendered entirely on the server, which serves the result to the browser as static resources. When dynamic behaviour is introduced to the browser with scripts, the term ”web application frontend” is typically used for the portion of the application run in the browser.

In recent years, web application frontends are increasingly created using a new generation of JavaScript frameworks. Widespread use of JavaScript libraries and frameworks is nothing new per se, with jQuery in particular still having an overwhelming market share[†]. What sets the new generation of frameworks apart from jQuery and similar tools is that instead of just providing a more ergonomic interface to DOM APIs, they define a declarative programming model where the application developer does not need to use DOM APIs directly at all.

Although their overall market share is still modest, the popularity of such frameworks is growing. As an example, at the time of writing 658 out of 934 or approximately 70% of open positions advertised at Stackoverflow Jobs[‡] containing the keyword 'JavaScript' also contain the keyword 'React', the most widely used web frontend framework at the moment[§]. In the rest of this chapter, we will consider the reasons for their growing popularity and review their rendering strategies.

3.1 Motivations for using script-based rendering

Page navigation in a browser can be viewed as transitions in the state of the render tree. Using traditional page navigation, when the user clicks on a link, the browser discards the current page, downloads resources that make up the new page and builds a new render tree as described in the previous chapter [24]. Any render tree state transition which can be achieved using page navigation can also be achieved programmatically using DOM APIs, yet the performance models of the two approaches are very different.

[†]https://w3techs.com/technologies/history_overview/javascript_library

[‡]https://stackoverflow.com/jobs

[§]Based on https://2020.stateofjs.com and https://insights.stackoverflow.com/survey/2020

A render tree transition can be thought of as a solution to the tree edit distance problem, where steps to transform one tree into another are calculated and applied [6]. Assuming a constant cost for each individual edit action, the ideal solution is one where the minimum number of actions are performed. From this perspective, traditional page navigation equals giving up on even attempting to solve the problem and simply rebuilding the entire tree. The performance cost of a UI update is thus directly dependent on the complexity of the render tree, not on the changes made to it. Script-based rendering, on the other hand, can make incremental changes to the DOM, so that the cost of a UI update scales linearly with the number of changes to be made. Script-based rendering hence potentially enables far more responsive applications, particularly when small changes to the UI are made, as is typical in an application with a rich UI.

Another motivation for using script-based rendering is that UI transitions do not necessarily require resource loading, which can be the most important single factor impacting website performance [37]. Resource requests, when they are needed, can be completed asynchronously, allowing the user to continue to interact with the application in the meanwhile [24]. Script-based rendering, then, is superior for use cases requiring frequent small UI transitions. The downside is that this can be very complicated to manage in a complex application.

In a browser, a programmatic render tree transition consists of issuing a set of imperative DOM API calls. Typically, whenever the application state changes, appropriate changes should be done to update the UI to reflect the updated application state. Given an application with N possible valid application states, where for each possible state there exists, on average, valid transitions to a fraction k of other states, the number of possible valid state transitions is $kN(N-1)$. That is, unless from every state there exists only a single valid transition to another state ($k = 1/(N-1)$), the number of possible valid state transitions grows exponentially as the application's complexity grows linearly. Consequently, if every transition must be manually accounted for, as is the case when using the DOM APIs directly, UI management quickly becomes error-prone and unwieldy.

Web frontend frameworks solve this problem by providing an abstraction over DOM APIs. In the component-based approach used by modern frameworks, the application developer describes the application as a tree of components, and is concerned only with the state of the component tree. Whenever the state of the component tree changes, the framework takes on the task of generating the appropriate DOM API calls to update the UI to reflect the new application state. This transforms the programming model from imperative to

declarative, greatly simplifying the application developer's task. We believe this largely explains the popularity of frameworks implementing the component-based model.

Because every application must in the end perform DOM updates through DOM APIs, they all must decide on a particular strategy with which to generate the necessary DOM API calls. For any particular UI update, the required DOM API calls do not depend on choice of framework. However, as we will see, the amount of work required to determine the required changes can vary massively between different rendering strategies, and is therefore a primary factor determining performance differences between frameworks.

3.2 Design patterns in web frontend frameworks

When reviewing rendering strategies used in frontend frameworks, we found that they often share certain patterns, the most important of which we will describe below. Unsurprisingly, we also discovered differences, some of which have significant implications on performance. Significantly, some of these patterns can be used to estimate a rendering strategy's relative performance without having to measure it.

3.2.1 Model-View-ViewModel and data bindings

The various web frontend frameworks we reviewed vary in scope. Some are focused purely on rendering, whereas others provide a wider set of tools and in some cases enable targeting other platforms besides the browser. When used in the web application context, however, what they all have in common is that they implement a declarative programming model using some version of the Model-View-ViewModel (MVVM) pattern [13].

In the MVVM pattern, the *Model* is the application's data source, independent of the application's GUI. The *View* consists of the concrete GUI elements presented to the user. The *ViewModel* is an abstraction of the view: it contains a model of the data presented in the view, but is not itself dependent on a concrete implementation of the view. One central feature of the MVVM pattern is data binding: instead of requiring the application developer to imperatively update the GUI, changes to bound data are automatically propagated to the view. In the case of two-way data binding, changes to bound data in the view due to user input are also automatically propagated to the model or viewmodel.

In the context of web frontend frameworks, the term "component" is usually used instead of *ViewModel*, but they are conceptually equivalent. Each component defines a subset of

the DOM, and defines any data bindings that should exist between the component and the DOM. Data bindings do not only define values that should be displayed in the view as text nodes, but can have custom logic associated with them: for example, a data binding might be associated with an element which is conditionally rendered or hidden based on the truthiness of the binding's value (figure 3.1).

```
<template>
  <div v-if="showGreeting">
    Hello world
  </div>
</template>

<script>
export default {
  props: ["showGreeting"],
};
</script>
```

Figure 3.1: In this Vue component, the text "Hello world" is conditionally rendered based on the value of the "showGreeting" data binding.

3.2.2 Rendering patterns

Rendering strategies in web frontend frameworks fall into one of two categories depending on how they solve the tree edit distance problem. Some solve the problem explicitly by comparing two trees, one representing the current state of the DOM and the other a new desired state, and explicitly calculating the changes that must be applied to current tree to obtain the new one. This is usually called the "virtual DOM"-based rendering strategy. Solving the tree edit distance problem in the general case has a time complexity of $O(n^3)$ [25], but the authors of React, for example, claim to simplify this to $O(n)$ by making certain assumptions that generally hold in the web application context [28].

The other option is to solve the tree edit distance problem implicitly by applying incremental changes to the DOM based on changes to individual data bindings. Regardless of framework, the number of different possible types of data bindings is limited. Each data binding can then be associated with a specific type of work that can be performed to update the DOM whenever the value of the binding changes. For example, a binding related to conditional rendering might be associated with a function which alternatively deletes or adds the bound element to the DOM based on the binding's value. By walking through each data binding and applying the requisite work wherever necessary, the sum

total of the changes is a solution to the tree edit distance problem.

3.2.3 Performance considerations

The basic work that every framework must perform consists of creating and updating components and their associated elements and data bindings. All frameworks considered here do this in a loop which walks through the component tree and, with one exception, are able to guarantee a basic $O(n)$ time complexity for each loop. Significant performance differences exist, however, and they arise from differences in input sizes and fixed costs.

Differences in fixed costs between various frameworks are not straightforward to quantify. Although every framework must create and update components and data bindings, the specific actions they must perform per component and binding are heavily dependent on implementation details, and differences in costs are not necessarily obvious. One example of fixed costs which we encounter in numerous frameworks is dirty checking.

Dirty checking consists of storing a copy of a value, and later performing a check to see if the value has changed. As we will see, this is often associated with data bindings, where the framework stores a copy of value of a binding whenever it is pushed to the DOM, and later compares the copy to the current value to determine whether an update is needed. Dirty checking of objects is not straightforward: a reference comparison will detect if a variable has been reassigned, but not if an object's properties have been mutated. Determining mutations may require a comparison of every value of every property in an object, with the performance cost increasing in line with object complexity. Unless an object can be marked as dirty in advance, a dirty check must choose between accuracy and performance [3].

The other factor affecting performance is a render loop's input size. This is composed of two parts: the number of components, and the number of static elements and data bindings per component. When a component tree is initially created, all relevant components, elements and bindings must be created, and input sizes are therefore equivalent regardless of framework. When already existing components are updated, however, we witness significant differences in input sizes between different rendering strategies, both in the number of components and the amount of content per component.

Ideally, an update loop will check only exactly the components and data bindings which have changed. In practice, no framework we reviewed achieves this, and indeed one of our primary findings is that *the most significant differentiating factor in rendering strategy performance is the input size of an update loop.* Specifically, a rendering strategy's relative

performance can be estimated based on the following factors in the context of update loops:

- Which components are checked?
- Which elements are checked for each component?
- Does the framework utilize virtual DOM?

Virtual DOM incurs additional overhead due to the necessity to explicitly calculate the required DOM changes. In effect, a virtual DOM-based rendering strategy must perform two loops: one over the component tree to build a new virtual DOM tree, and another over the two virtual DOM trees to compare them and produce the required set of DOM API calls.

In the review of frameworks that is to follow, we will investigate the particulars of how frameworks differ across these factors. We will additionally bring up examples of differences in fixed costs and related optimizations.

3.3 Review of frameworks

We have reviewed the following frameworks:

- AngularJS
- Angular
- React
- Vue
- Svelte
- Blazor

The number of open source web frontend frameworks is vast and the selection of frameworks that we could review is necessarily limited. The selection has been based on each framework's popularity* and influence, as well as the progression shown in their rendering strategies. As some of the most popular open source frameworks, we believe they are a representative sample of rendering strategies in frontend frameworks in general.

*Based on https://stackoverflow.com/jobs and https://stateofjs.com/

3.3.1 AngularJS

AngularJS* is one of the earliest widely used web frontend frameworks with an explicit goal of enabling declarative programming model [16]. AngularJS is by now considered obsolete, having been superseded by a successor confusingly also called Angular. Even though obsolete, AngularJS makes a good starting point for our review due to its influence and unique rendering model with both strengths and flaws that have influenced later frameworks.

In AngularJS, a component typically consists of a controller written in JavaScript and an associated HTML template. Each controller manages a scope, which is a JavaScript object that contains application state. In AngularJS, data bindings are two-way, and are defined directly between a scope and a template: changes to data within the scope will be propagated to the view and vice versa irrespective of the controller. Controllers and their scopes can be nested, effectively forming a component tree. Scopes use prototype-based inheritance, which allows bindings to properties in an ancestor scope to be defined in any descendant component [3].

AngularJS's rendering strategy is binding-based. Templates in AngularJS can contain custom syntax – custom elements representing components, and custom element attributes representing data bindings – called directives (figure 3.2). Each directive is associated with some type of custom behaviour, such as rendering a named component or rendering an element conditionally. The browser, when it parses the initial HTML template to build the DOM, will store the directives in the DOM as node types and node attributes which have no intrinsic meaning. AngularJS will then walk through the DOM, find any directives and perform the work associated with each of them, which builds out the component tree and registers any data bindings. Any further updates beyond the initial render are managed through dirty checking.

AngularJS does not detect changes to application state directly, and instead expects the application developer to use library functions that trigger an update loop either explicitly or as part of modifying state within a scope. During an update loop, AngularJS performs a dirty check on every data binding within the current scope as well as any ancestor and descendant scopes, and for each dirty binding performs the required work to update the DOM. Because AngularJS applies updates based on bindings, it does not need to update static content after a component's initial render. In an optimal case, an update loop

*https://angularjs.org/

```
<ul>
  <li ng-repeat="alert in alerts">
    <alert-icon></alert-icon>
    <span>{{ alert }}</span>
  </li>
</ul>
```

Figure 3.2: Directives in AngularJS: custom HTML syntax for dynamic behaviour based on data bindings. Here, members of the ”alerts” array are rendered in a list, and a custom ”alert-icon” component is rendered for each.

in AngularJS checks a subtree of the component tree consisting of the component that initiates the update loop, its descendants and its direct ancestors.

The optimal case, however, is not guaranteed, due to AngularJS's two-way data binding and prototypic inheritance in scopes. Updates to the DOM resulting from dirty bindings may potentially lead to changes to binding values further up in the component tree, which can lead to further cascading changes in an unpredictable manner. In effect, AngularJS permits update loops to have cycles, which can both lead to difficult to understand bugs [17] and an unbounded time complexity [36].

AngularJS's basic rendering strategy – incremental updates based on changes to bindings – is still used by other frameworks today, and the directive-based syntax remains popular. Two-way data binding, however, has not survived, and neither has prototype-based inheritance in component state.

3.3.2 Angular

AngularJS's successor framework Angular* borrows many concepts from its predecessor, but is an entirely reimagined framework. Angular is a compiler-based framework: applications are written using TypeScript and compiled into JavaScript, which can then be executed in the browser with the help of Angular's JavaScript runtime.

Angular solves the problem of cyclical dependencies in bindings in AngularJS by using one-way data binding and treating component state as local by default unless explicitly shared with children. Instead of a graph with cycles, a render loop becomes a tree: each component and binding needs to be checked at most exactly once per loop by walking down the component tree [36]. This guarantees $O(n)$ time complexity for each update loop. Unsurprisingly, one way data binding and component-local state have become standard

*https://angular.io/

patterns adopted by all the other frameworks reviewed here.

A component in Angular consists of a TypeScript class definition and an HTML template extended with AngularJS-style directives, which are then compiled into a runtime component definition. The core of a runtime component is its template function, which handles rendering for the component. The template function is generated by the Angular compiler by walking through the component's template and class definition, which it uses to produce code for generating each element and for updating any data bindings. The resulting template function contains two branches: one for the initial render of the component and another for any subsequent updates [1].

A render loop in Angular consists of walking through the component tree and calling the template function of each encountered component to either create or update the component. Component creation is straightforward: the template function creates any necessary elements and sets up any value bindings. The update branch, on the other hand, will walk through each binding, perform a dirty check, and update the DOM if necessary. The update branch does not need to consider static elements which will never change: only elements with data bindings need to be checked. In AngularJS, bindings are distinguished from static content at runtime by walking through the DOM. Angular, in contrast, processes directives at compile time, using them to build the template function in such a way that the update branch only handles bindings, not static content. [19].

Unlike AngularJS, Angular does not need the application developer to manually trigger a render loop. Angular borrows a concept from the Dart language* called zones [8]. A zone is an execution context within which code – including asynchronous function calls – can be executed and which exposes hooks that allows callback functions to be executed before and after the code wrapped in the zone finishes executing. Angular patches any browser events that can cause application state changes, such as user input events and HTTP data requests, and wraps them in zones which trigger a render loop at the end of the zone's execution [2]. Zones do not, however, enable Angular to know which components require an update, only that an update is needed, and an update loop will consequently walk through the entire component tree.

Angular's deterministic render loop makes it a straightforward improvement to its predecessor in terms of performance. The compiler-based approch also ensures that updating each component is cheap as static content does not need to be checked. Update loops as a whole are still inefficient, however, due to the need to check the entire component tree.

*https://dart.dev

3.3.3 React

React*, developed by Facebook and released as open source in 2013, introduced a rendering strategy based on explicitly solving the tree edit distance problem [6]. In this approach, the application manages a data structure called a virtual DOM (vDOM), which represents a particular state of the real DOM. The steps required to update the DOM can be produced at any given time by comparing two vDOM trees, one representing the current state of the DOM and the other a new, desired state. In React jargon, this is called reconciliation.

As with Angular and the other frameworks reviewed here, a React application consists of a tree of components. Each component contains a render function, which as its output produces a single vDOM node, which is an object that describes a set of elements and their attributes (figure 3.3). A render loop consists of walking through the component tree and calling the render functions of each component encountered. The newly produced vDOM is then reconciled with the previous one, and the DOM updated accordingly. Like AngularJS, React does not detect state changes directly, requiring application developers to schedule render loops manually by using specific library functions to update application state. Unlike Angular, a render loop in React is performed only for the subtree of a component which initiates the render loop, not the entire component tree [32].

```
export default class Child extends React.Component {
  render() {
    return <div>-</div>;
  }
}
```

Figure 3.3: A render function in a React component which procuces a very simple VDOM node.

React's rendering strategy is remarkably simple: there is no fundamental difference between an initial render loop and subsequent update loops, or between static elements and data bindings. Components simply describe the desired state of the DOM as a function of application state, and reconciliation is used to determine how to update the DOM to the desired state. This simplicity comes with costs: React does not differentiate between static and dynamic content, and must process both on each render loop. Render functions may also perform arbitrarily complex computations, and an update to a component will by default require its entire subtree to be reconstructed and reconciled, regardless of whether the output of any child components has actually changed. A more performant render loop can be achieved by skipping the rendering of components whose output has not changed,

*https://reactjs.org

or by optimizing render function performance, both of which are possible to do in React, but must usually be done manually.

React allows application developers to instruct the framework to skip rendering a component and its children based on custom logic. Typically, dirty checking of a component's props and state is used: if they have not changed, the component's output has likely not changed either. Aside from the usual pitfalls with dirty checking of objects, this model can be error-prone: if a parent component decides to skip a render loop, the entire subtree will not be updated, even if some descendants would require a re-render [26].

Render functions in React are plain JavaScript functions which can perform any type computation. An expensive render function might perform computations on a large array of objects, for example, and will by default do so every time the render function is invoked, even if the output of the computation has not changed. The results of expensive computations can be memoized, but this must be done manually and will require a dirty check on the inputs of the memoized computation on all subsequent renders.

Render loops are the most performance-heavy operation in every framework, and thus the most likely cause of blocking user interaction. React has in recent versions introduced an alternative approach to address this by performing a render loop incrementally [9]. In this implementation, React can yield execution back to the browser in the middle of a render loop, allowing user interaction and animations to be interleaved with render loop processing. This potentially improves user experience even if the total cost of a render loop is not improved. This feature is currently only in experimental use, however, and by default the entire render loop is processed in one go.

Perhaps partly due to its conceptual clarity, React has been hugely influential and sees the most wide use of all open source web frontend frameworks. Its performance, however, remains unoptimal, which has inspired challengers to come up with solutions to its perceived shortcomings.

3.3.4 Vue

Vue* utilizes a rendering strategy based on a virtual DOM and reconciliation, and is conceptually almost identical to React. Vue components are essentially equivalent to React components: a component is an object containing a render function, which as its output produces a vDOM node built based on application state. There are differences in syntax:

*https://vuejs.org

for instance, Vue allows the render function to be defined as a HTML template with directive-based syntax similar to Angular, but this is only syntactic sugar: the template is parsed into a render function.

The primary innovation that Vue implements over React is a limited reactivity system. In reactive programming, the application developer can define values as being produced through computations dependent on other values. The runtime environment which implements the reactive programming model ensures that changes to values are automatically propagated across dependency graphs [5]. Spreadsheets are the canonical example.

React, despite the name, does not implement a reactive programming model. React does not keep track of dependencies between values: dependencies between computations exist only in the sense of the order in which they are evaluated within a render function. Unless memoized, each computation will always be re-evaluated regardless of whether it is actually necessary. Vue, on the other hand, allows application developers to explicitly declare reactive functions which are re-computed only when values they depend on change (figure 3.4). This is achieved using proxies.

```
<template>
  <div>{{ greeting }}</div>
</template>

<script>
export default {
  props: ["name"],
  computed: {
    greeting() {
      return `Hello ${this.name}`;
    },
  },
};
</script>
```

Figure 3.4: The "greeting" function reactively produces an output which depends on the value of the "name" input received from a parent component. Whenever this value changes, the function is automatically re-evaluated.

A proxy is a special type of JavaScript object which enables intercepting calls to properties in a target object [35]. At runtime, Vue converts all referenced data into proxies, which hook to the proxied object's property accessors. When a value mediated by a proxy is accessed, the proxy registers the accessing context, such as a reactive function, as being dependant on the value. Whenever the value later changes, any dependant functions are re-evaluated. Reactive values achieve the same render function performance optimization

as value memoization in React, but do so automatically and without the need for dirty checking.

More significantly, Vue's reactivity system automatically solves the problem of determining which components need to be checked in an update loop. A component's render function is treated the same way as reactive functions: it is registered as being dependent on every value referenced in it. Whenever any of the values changes, the component is scheduled for a re-render [27]. In effect, components are marked dirty automatically, and Vue only needs to update dirty components during a render loop.

Vue, although not requiring one, provides an optional compiler which can be used to compile the application in advance. In addition to introducing some additional syntax intended as quality of life improvements for application development, the compiler will perform static analysis on application code and optimize it for runtime performance [12].

Without compilation, the Vue runtime must perform a number of checks while processing the output of a render function. For example, when rendering an element, the runtime must check whether the element is a Vue component or a native HTML element. The compiler adds hints to the compiled code that the runtime can use to eliminate branching of this kind. Possibly the most significant optimization, however, is that the compiler determines which portions of the render function are static – ie. will never change after the initial render, and hoist them out of the render function, to be processed separately only on the component's initial render. This achieves the same end result as Angular's branching template function: an update to a component can ignore static content and process data bindings only.

While borrowing its core concepts from React, Vue's reactivity system and compile-time optimizations allow it to reduce the input sizes of update loops compared to React's more naive virtual DOM implementation. Input sizes are close to optimal: only dirty components are checked, and for each component only its data bindings. The use of virtual DOM, however, still causes additional overhead.

3.3.5 Svelte

Similar to Angular, Svelte* is a compiler-based framework. A Svelte component consists of an HTML template and an associated script (figure 3.5), which are compiled into a runtime component definition. As with Angular, Svelte does not utilize a virtual DOM, but each

*https://svelte.dev

component instead incrementally modifies the DOM based on changes to bindings. Like Angular and Vue, updating a component only checks bindings, not static content (figure 3.6). Change detection in Svelte is based on a reactivity system functionally equivalent to that of Vue, but based on compile-time code generation.

Svelte, like Vue, allows declaring reactive values. At compile time, Svelte builds a dependency graph for every value which is referred to either in the component's template or by any reactive value. For every value within a dependency graph, the Svelte compiler finds every code location where the value is updated or reassigned, and inserts a call to the Svelte runtime which marks the value and its transitive dependants as dirty and schedules an update for the component [15]. An update loop consists of walking through every dirty component, recomputing any dirty values in the component and then updating the DOM wherever the recomputed values are used in data bindings.

Svelte does not explicitly track dependencies at runtime in any way. The correctness of an update loop depends entirely on correctness of code generation: reactive values must be updated in the correct order according to their positions in each dependency graph, and there may not be cycles in any dependency graph. At runtime, reactivity only exists – similarly to React – in the sense of the order within which computations are evaluated in a component's update function. Unlike React, however, the evaluation order is explicitly generated from a compile-time dependency graph, which guarantees its correctness at runtime. The result is functionally equivalent to Vue's runtime reactivity system, but has no runtime costs associated with proxies.

The input size of an update loop in Svelte is exactly equivalent to that of Vue: only dirty components are checked, and for each component only bindings are checked. Svelte, however, does not use a virtual DOM, and therefore has smaller costs overall. It is theoretically possible to further reduce input sizes by only checking dirty bindings – rather than all bindings – for each component. However, because Svelte marks bindings as dirty when they become dirty, dirty checks always consist of a simple boolean comparison regardless of the complexity of the binding's value, and therefore reducing the number of bindings checked might not make much of a difference. Overall, of the frameworks we have reviewed, Svelte's rendering strategy appears to require the least amount of work. Fixed costs, however, are likely to play a part in determining real world performance.

```
<script>
  export let name;
  $: greeting = `Hello ${name};`;
</script>

<div>This content is static</div>
<div>{greeting}</div>
```

Figure 3.5: A Svelte implementation of the the Vue component in figure 3.4. The "$:" syntax denotes the "greeting" variable as being reactive.

```
c() {
  div0 = element("div");
  div0.textContent = "This content is static";
  t1 = space();
  div1 = element("div");
  t2 = text(/*greeting*/ ctx[0]);
},
m(target, anchor) {
  insert(target, div0, anchor);
  insert(target, t1, anchor);
  insert(target, div1, anchor);
  append(div1, t2);
},
p(ctx, [dirty]) {
  if (dirty & /*greeting*/ 1) set_data(t2, /*greeting*/ ctx[0]);
}
```

Figure 3.6: A part of the Svelte compiler output for the component in figure 3.5. Pictured are the lifecycle hooks used to create and update the component. Note how the update function ("p") ignores static content processed in the create ("c") and mount ("m") functions.

3.3.6 Blazor

Blazor* is the final framework to be introduced here. In Blazor's case, it is not its rendering strategy which is novel, but its implementation which is based on WebAssembly. Blazor applications are written in C# and executed within a .Net runtime that has been compiled ahead of time to WebAssembly bytecode. Blazor also includes a JavaScript runtime which manages interoperation between the browser and the .Net components, particularly access to DOM APIs.

Aside from using a different programming language, a Blazor application is not conceptually any different from that of any of the other frameworks introduced here. A Blazor application consists of a tree of components, with each component describing a portion of the DOM using one-way data binding and local component state optionally shared with

*https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

child components.

Details of Blazor's rendering strategy are not available as readily as those of the other frameworks considered here, but it seems conceptually identical to React's, being based on virtual DOM and reconciliation [30]. Render loops are triggered either manually or on component state changes and, as with React, will by default render the entire subtree of the component that initiated the loop. As with React, application developers can designate manually which components can skip a render loop. Blazor contains a default optimization, an equivalent of which is optional in React, which skips re-rendering a component if its inputs have not changed. Blazor will not attempt to dirty check reference type inputs, however, opting by default to always re-render any component with complex inputs [4].

Because Blazor components are executed within the WebAssembly context, all interaction with the DOM – both handling user input and updates pushed to the DOM – must go through the JavaScript interoperability layer. The potential overhead caused by this is of primary interest in the viability of any WebAssembly-based framework, beyond the details of its rendering strategy. Blazor is the most mature WebAssembly-based frontend framework as of yet, which makes it a suitable choice for a performance comparison with its JavaScript-based competitors.

3.4 Summary

The frameworks we have introduced display an increasing sophistication in their rendering strategies, with performance improvements often an explicit goal. The selection does not cover the entire gamut of rendering strategies found in open source frontend frameworks, but is a representative sample nevertheless. Interesting projects that have not been introduced here include Ember*, a long-established framework with a compiler-based approach which utilizes a virtual machine-inspired runtime and a binding-based rendering strategy, as well as Inferno†, a virtual DOM framework built explicitly with performance in mind and brimming with micro-optimizations.

In table 3.1 below, we have summarized the frameworks discussed in this chapter according to factors affecting update loop input sizes as outlined in section 3.2.3.

Table 3.1: Summary of the reviewed frameworks.

Framework	Components checked	Elements checked	Virtual DOM
AngularJS	All	Bindings only	No
Angular	All	Bindings only	No
React	Subtree of updated component	All	Yes
Vue	Dirty components only	Bindings only	Yes
Svelte	Dirty components only	Bindings only	No
Blazor	Subtree of updated component	All	Yes

*https://emberjs.com/
†https://infernojs.org/

4 Measuring rendering performance

In this chapter, we describe a set of benchmarks we have implemented to measure the rendering performance of the frameworks introduced previously. With these benchmarks, we have attempted to verify and measure the expected relative performance differences based on update loop input sizes. We also measure relative differences in fixed costs associated with creating components.

4.1 Aims and methodology

As outlined in the previous chapter, the script execution costs of each rendering strategy can be expected to vary depending on whether a render loop is used to primarily create new components or to update existing ones. The largest differences are expected to appear in update loops, particularly under circumstances where components with large subtrees are updated or when updated components primarily contain static content. The aim of the benchmarks described in this chapter is to verify the existence of these expected differences, as well as to measure their relative magnitudes. Due to the simplicity of the components we use, our benchmarks do not quantify absolute differences in rendering performance except in an approximate way.

Web frontend framework performance comparisons typically measure the entire duration of a render cycle[†]. A problem with this approach is that a full render cycle includes work done by the browser such as performing the layout and paint phases of the critical rendering path. These costs, while important, are separate from and additional to script execution. Script execution represents the true amount of work performed by a framework and is therefore an accurate measure of its performance costs. We have therefore measured, as accurately as possible, the duration of script execution in addition to the duration of the entire render cycle.

There is no standard API for measuring script execution times in browsers. Chromium-based browsers, however, implement the Chrome Devtools Protocol (CDP)[‡], which enables programmatic control over the browser, including the ability to trace script execution. We

[†]See for example https://github.com/krausest/js-framework-benchmark

[‡]https://chromedevtools.github.io/devtools-protocol/

have therefore implemented all tests using CDP. Tracing is based on CPU polling, which in our tests we perform at intervals of 250 microseconds, giving millisecond-level accuracy in our results. We repeat every individual test 10 times with each framework, and each result is the mean value of these 10 samples,

In each benchmark described in this chapter, we have implemented an application identically with each framework. We then perform a particular action and measure the time taken for script execution in the render cycle that follows the action. Each scenario measures the costs of a particular aspect of rendering by varying the shape of the component tree and the actions performed.

Benchmarks were implemented for all frameworks described in the previous chapter except for AngularJS, due to its having been superseded by Angular. For each benchmark, we present a subset of results across a range of input sizes. Full results, including full render cycle durations for each benchmark, can be found in appendix B.

4.2 Benchmarks

The test scenarios we implemented are designed to measure different aspects of rendering performance. We have divided these into three groups based on the purpose of each benchmark.

4.2.1 Group 1: Component and element creation

This set of benchmarks was implemented with the aim of measuring performance differences in a render loop that consists primarily of creating new components and elements, such as on initial page load. The group includes three test scenarios.

In the first scenario, we measure time taken to render static elements. The implementation consists of a single component which renders N static elements. The results are presented in table 4.1

In the second scenario, we measure the cost of creating components. A single parent component is created with N child components, each of which outputs a single static element. The resulting DOM tree is exactly identical to that in the previous scenario, but there is additional overhead due to component construction. Table 4.2 has the results.

The final scenario in this group also measures the cost of creating components, but the

shape of the component tree is different. The motivation for using a different shape of component tree is to determine whether there are costs associated with having a large number of child components in a single component. In this case, we build a binary tree-shaped component tree, where each non-leaf component has exactly 2 children, for a total of N components. As in the previous scenario, each component outputs a single static element. Additionally, each component contains a single data binding used to determine whether it should render children, which makes each individual component slightly more expensive than in the previous scenario. The results are found in table 4.3.

Table 4.1: Script execution time (ms) when creating N static elements.

Framework	N=100	N=500	N=1000	N=5000	N=10000	N=25000	N=50000
Angular	3	9	16	85	177	844	2520
React	2	9	11	77	200	956	3559
Vue	1	3	6	28	47	95	173
Svelte	1	2	3	14	24	63	98
Blazor	3	8	13	61	123	371	964

Table 4.2: Script execution time (ms) when creating N components with a single parent.

Framework	N=100	N=500	N=1000	N=5000	N=10000	N=25000	N=50000
Angular	5	16	35	102	140	208	364
React	3	16	28	122	278	1128	3840
Vue	4	20	34	105	169	282	473
Svelte	1	5	10	40	72	152	252
Blazor	9	28	52	268	557	1501	3451

Table 4.3: Script execution time (ms) when creating N components as a binary tree.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	20	75	120	216	297	469	774
React	7	32	55	137	233	394	733
Vue	16	53	84	223	313	485	897
Svelte	3	10	22	83	142	233	482
Blazor	17	59	128	485	966	1870	3644

4.2.2 Group 2: Updating components

As described previously, we expect there to be significant differences in the number of components that each framework must touch during an update loop, independently of each component's contents. This second set of tests was implemented for verifying and measuring these differences. The group contains two benchmarks.

In both benchmarks, we utilize the same test scenario. The scenario contains a component tree in the shape of a binary tree with a total of N components, where each non-leaf node has two child components. The root component and all leaf components contain a single input which can be used to perform an update that targets only the component itself. Intermediate components produce only a single static element with no user-visible output. The two benchmarks are identical in their setup, differing only in the target component to be updated. In the first case, we update the root component, the results of which can be found in table 4.4. Table 4.5 shows the results of the second benchmark, where we update a single leaf component.

Table 4.4: Script execution time (ms) when updating the root component of a component tree of N components.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	3	12	14	32	32	43	103
React	7	23	42	92	148	211	379
Vue	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Svelte	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Blazor	3	3	2	3	3	2	3

Table 4.5: Script execution time (ms) when updating a leaf component of a component tree of N components.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	3	13	14	33	33	44	104
React	0	0	1	4	3	5	4
Vue	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Svelte	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Blazor	1	1	1	3	5	4	8

4.2.3 Group 3: Updating static content

The final set of benchmarks measures the cost of updating components containing a high proportion of static content compared to data bindings, which is likely to be typical in real applications. The purpose of these tests is to verify the expected differences in input sizes that must be processed for each component. The group contains two benchmarks.

As in the previous group, both benchmarks utilize the same scenario. Again, a component tree in the shape of a binary tree is used. Each component outputs 50 static elements that do not change after the initial render, as well as an input which can be used to update the component locally, as in the previous scenario. Additionally, the root component contains an input which can be used to force an update of all components in the component tree. Table 4.6 contains the results of updating the root component but no other components, and table 4.7 the results of updating all components.

In this group of tests, the component tree is smaller than in the previous tests. Beyond 8192 components and approximately 2 million DOM nodes, certain frameworks would allocate more memory than available and cause the browser to crash.

Table 4.6: Script execution time (ms) when updating the root component in a component tree of N components where each component contains primarily static content.

Framework	N=128	N=256	N=512	N=1024	N=2048	N=4096	N=8192
Angular	3	5	14	23	20	24	148
React	34	43	64	98	225	278	806
Vue	< 1	< 1	< 1	< 1	< 1	1	8
Svelte	< 1	< 1	< 1	< 1	< 1	< 1	< 1
Blazor	4	3	3	8	10	7	8

Table 4.7: Script execution time (ms) when updating the entire component tree of N components where each component contains primarily static content.

Framework	N=128	N=256	N=512	N=1024	N=2048	N=4096	N=8192
Angular	4	8	17	27	29	44	238
React	34	44	66	101	235	289	841
Vue	20	32	42	72	91	149	311
Svelte	2	3	5	10	20	54	80
Blazor	28	60	101	250	502	1020	2013

5 Discussion

This chapter contains our analysis and interpretation of the results of the benchmarks. We will firstly discuss and analyse the results of each benchmark. Based on our findings, we will then review the research questions introduced in chapter 1. Finally, we will consider the validity and applicability of our results.

5.1 Analysis of benchmarks

An overall impression of the results is that they are in line with what would be expected given the characteristics of each rendering strategy as outlined in table 3.1. There are, however, unexpected results as well.

5.1.1 Group 1: Component and element creation

Based on the rendering strategies used by each framework, we would not expect fundamental differences in their performance when initially rendering content. Two things stand out from the results, however.

Firstly, Angular and React exhibit nonlinear performance costs when rendering a single component containing a large number of child components or elements. This becomes most obvious when more than 10000 elements are rendered in a single component, as seen in figure 5.1. This effect disappears when components are rendered as a binary tree, where no single component has a large number of children, as shown in figure 5.2.

Secondly, it appears that there are significant differences in fixed costs. Blazor is a clear outlier with significantly worse performance than its JavaScript-based competitors. Angular, React and Vue post comparable results, but the most performant framework, Svelte, has an edge of at least 50% over its competitors, as seen in table 5.1. While we expected Svelte to perform well when updating components, the reasons for its superior performance here must be explained by fixed costs rather than input sizes.

From tables 4.1 and 4.2, we note that Vue, Svelte and Blazor exhibit linear performance. From their results, it appears that rendering a single element within a separate component

Table 5.1: Relative costs of script execution when rendering N components as a binary tree.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	6.7	7.5	5.5	2.6	2.1	2.0	1.6
React	2.3	3.2	2.5	1.7	1.6	1.7	1.5
Vue	5.3	5.3	3.8	2.7	2.2	2.1	1.9
Svelte	1	1	1	1	1	1	1
Blazor	5.7	5.9	5.8	5.8	6.8	8.0	7.5

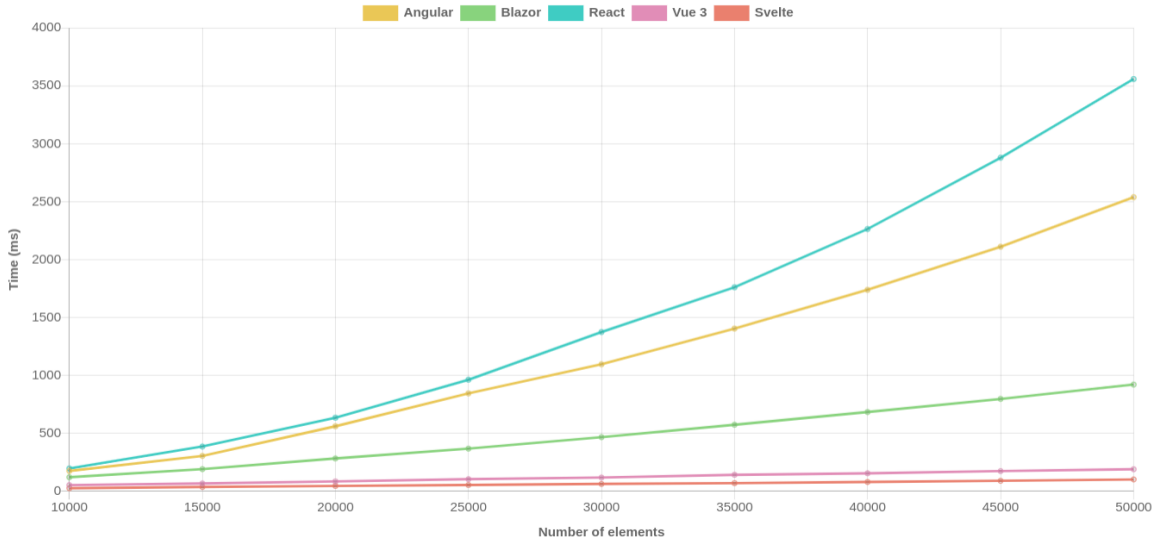


Figure 5.1: Script execution time (ms) when rendering N elements in a single component.

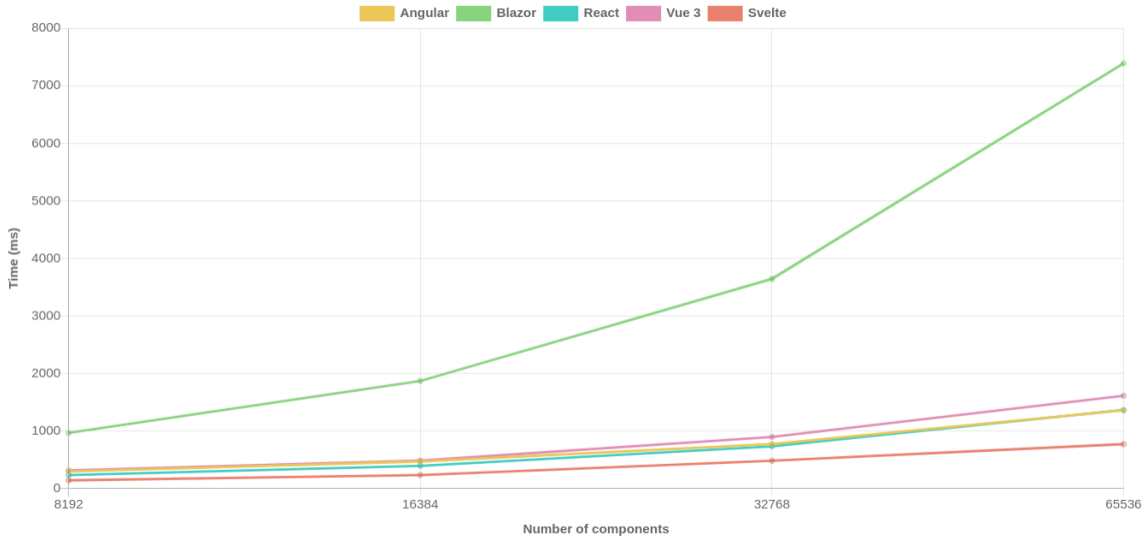


Figure 5.2: Script execution time (ms) when rendering N components as a binary tree.

Figure 5.3: Comment nodes inserted by Angular in the DOM.

roughly triples the performance cost compared to rendering it directly inside the parent component. The costs of a single component seem hence to be reasonably low, considering that in real-world use most components should render a far larger number of elements.

To put the results in context, we can compare script execution times to the duration of the full render cycle. Table 5.2 contains for each framework the duration of the full render cycle in the binary tree component creation scenario, including script execution times. This is the true measure of rendering costs as it appears to a user. The relative differences here are significantly smaller than if we only consider script execution times. Given that the components we render are very simple and thus cheap to build, and that the browser must perform a full render tree construction cycle, this is perhaps to be expected.

The outlier here is Angular, where the difference in the cost of a full render cycle compared to other frameworks is in fact significantly greater than the difference in script execution times alone. This appears to originate from the fact that Angular generates comment nodes in the DOM when a conditional rendering directive is used (figure 5.3). While comment nodes do not contain any meaning for the document's structure or content, they increase the size of the DOM tree and consequently increase the cost of render tree construction. Such costs are real and inherent to the framework's rendering implementation, even if not part of script execution costs. Blazor exhibits similar behaviour under other circumstances.

5.1.2 Group 2: Updating components

When components are updated, we expect to see the best performance from the frameworks that have to touch the least amount of components in the update loop. This is

Table 5.2: Duration of a full render cycle (ms) when creating a binary tree of N components.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	34	117	204	556	992	1878	3654
React	14	45	77	249	464	858	1669
Vue	24	72	115	340	549	953	1836
Svelte	7	22	53	198	375	696	1407
Blazor	23	74	153	585	1172	2272	4446

exactly what we see in the results as well. When updating a leaf component, Angular stands out as the only framework with a less-than-trivial rendering cost, having identical performance regardless of whether a leaf or a root component is updated, as seen in tables 4.4 and 4.5. Measured in absolute terms, the cost is still relatively low, but this might not be the case in a real-world application with more complex components to render. In effect, Angular's rendering strategy places a minimum cost on every update that scales linearly with the complexity of the view, which may make it unsuitable for applications with complex views and real-time responsiveness requirements.

As expected, updating the root component also confirms the costs of React's rendering strategy which must re-render the entire subtree of the updated component, as seen in figure 5.4. The relative performance difference to the best performing strategies is dramatic, and measured in several orders of magnitude. Blazor's default optimization of not re-rendering components with unchanged inputs ensures that it performs well here, but if the child components were to contain reference type inputs, we would witness a curve similar to that of React. Other frameworks perform similarly regardless of where the component is located, with Svelte and Vue in particular standing out with a sub-millisecond update loop.

Again, it is useful to put these results in context by comparing script execution times to the duration of a full render cycle. The most striking differences are found when the root component is updated, with the full render cycle costs shown in table 5.3. Even though the differences are smaller than if only script execution costs are considered, they still remain at an order of magnitude of difference in the full render cycle duration between React and Svelte. Because the changes made to the DOM are very minor, the browser's layout process is quick, and script execution costs become the dominant factor in the duration of the render cycle.

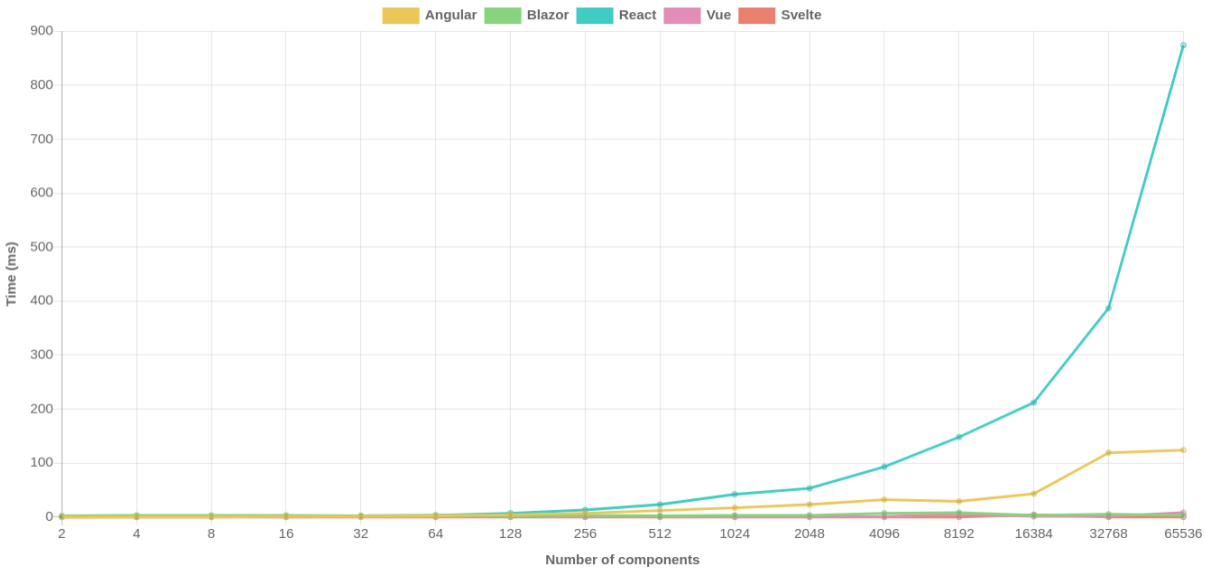


Figure 5.4: Script execution time (ms) when updating the root component of a component tree with N components.

Table 5.3: Duration of a full render cycle (ms) when updating the root component in a binary tree of N components.

Framework	N=128	N=512	N=1024	N=4096	N=8192	N=16384	N=32768
Angular	8	17	26	53	60	98	224
React	12	32	47	105	166	237	439
Vue	4	9	5	13	20	26	48
Svelte	7	8	4	10	12	22	24
Blazor	5	6	9	20	28	41	51

5.1.3 Group 3: Updating static content

When updating static content, we would expect to see a significant advantage for frameworks which only update bindings, not static content. This is exactly what we see, with React and Blazor in particular again having a significant disadvantage due to their inability to differentiate between static and dynamic content. This difference is very significant, reaching an order of magnitude as shown in figure 5.5 and table 5.4. Svelte is again the best performing framework overall, although the absolute difference to Angular and Vue is small enough when the number of components is low that Angular is shown to outperform Svelte when N=4096.

We can again compare script execution times to the duration of the full render cycle, which

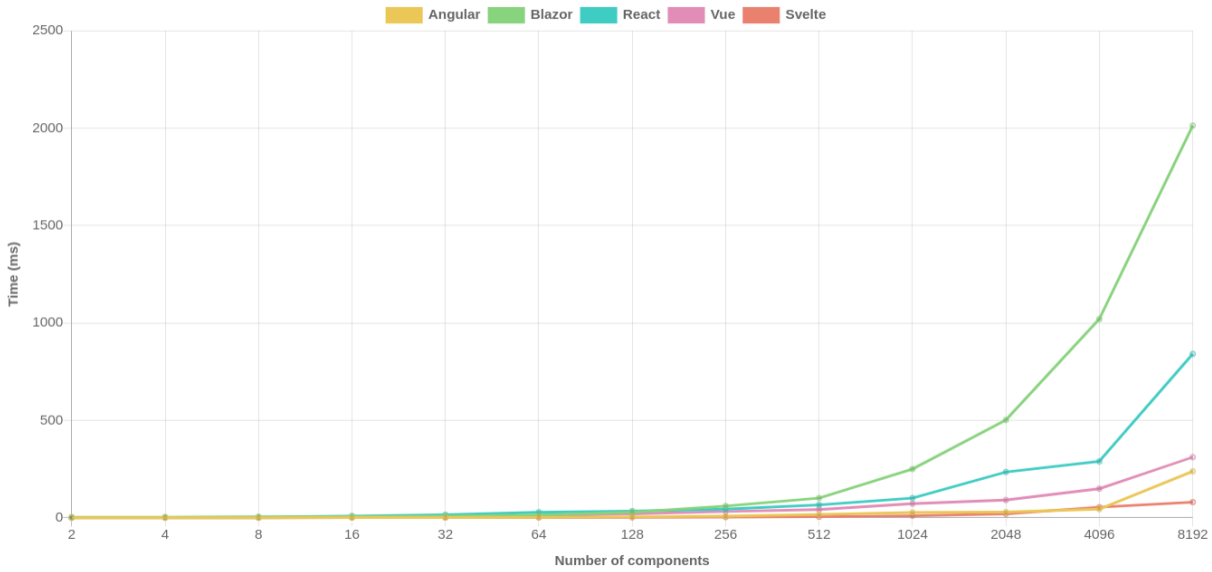


Figure 5.5: Script execution times (ms) when updating all components in a component tree of N components where each component contains primarily static content.

Table 5.4: Relative costs of script execution when updating all components in a tree of N components.

Framework	N=128	N=256	N=512	N=1024	N=2048	N=4096	N=8192
Angular	2.0	2.7	3.4	2.7	1.5	0.8	3.0
React	17.0	14.7	13.2	10.1	11.8	5.4	10.5
Vue	10.0	10.7	8.4	7.2	4.55	2.8	3.9
Svelte	1	1	1	1	1	1	1
Blazor	14.0	20.0	20.2	25.0	25.1	18.9	25.1

is shown in table 5.5. Here, layout costs are somewhat significant due to a large number of changes in the DOM, but costs are still dominated by script execution. Although the relative costs differences are smaller, they are still very significant when comparing Blazor and React to the three other frameworks which update only dynamic content.

Table 5.5: Duration of a full render cycle when updating all components in a tree of N components.

Framework	N=128	N=256	N=512	N=1024	N=2048	N=4096	N=8192
Angular	11	31	47	72	115	211	577
React	43	67	91	142	295	419	1068
Vue	29	56	67	106	152	270	542
Svelte	7	13	29	56	82	174	387
Blazor	41	72	126	303	605	1213	2396

5.2 Revisiting research questions

In chapter 3, we found differences with potential performance implications in the rendering strategies of frontend frameworks. The benchmarks we describe in chapter 4 confirmed that these differences are both real and significant. We will summarize our findings by answering the research questions listed in chapter 1.

5.2.1 RQ1: What rendering strategies are used in web frontend frameworks?

The rendering strategies we reviewed can be divided into two categories. In the first group are those of React, Vue and Blazor, which utilize a virtual DOM-based approach. This strategy consists of building a representation the desired state of the DOM, and explicitly comparing it with a representation of the DOM's current state and thereby deducing the changes that must be applied to the DOM to obtain the desired state.

The other group consists of AngularJS, Angular and Svelte, whose rendering strategies are based on data bindings. Each data binding is associated with some type of work which can be performed on the DOM when the value of the binding changes. Keeping the DOM up to date consists of tracking changes to data binding values and applying the required work whenever changes are detected.

Rendering strategies can also further be characterized by how they determine when the UI should be updated. AngularJS, React and Blazor depend on the application developer to initiate an update through the use of relevant library functions. Angular uses a unique solution which patches browser functionality and automatically schedules an update whenever certain activities such as user interaction take place. Vue and Svelte rely on a reactivity system, which automatically schedules an update whenever application state is updated.

5.2.2 RQ2: Which factors affect the performance of each strategy?

The performance of a rendering strategy is decided by fixed costs of creating and updating individual elements, and the number of elements and components which must be created or updated. Fixed costs are the primary determining factor when components and elements

are initially created, as the input sizes are similar regardless of rendering strategy.

For update loops, we find that input sizes are the dominant factor in performance. During an update loop, React and Blazor opt by default to re-render every component in the subtree of the component which initiates the update loop. Alone of all frameworks, AngularJS may have cycles in an update loop and walk through the same component multiple times. Angular, its successor, walks through the entire component tree every time. Vue and Svelte utilize their reactivity systems to determine exactly which components require an update, and update those only.

Input sizes for updating individual components also differ between frameworks. AngularJS, Angular, Vue and Svelte only check data bindings for each component to be updated, while React and Blazor opt to reproduce all the contents of every component each time.

React, Vue and Blazor must perform a separate loop for virtual DOM reconciliation, which adds a cost specific to that strategy. The use of WebAssembly in the case of Blazor adds additional costs by requiring that all DOM API access is mediated by a JavaScript interoperability layer.

Finally, Angular and Blazor's rendering implementation includes the insertion of comment nodes in the DOM under certain circumstances. These increase the size of the DOM node tree – potentially significantly – and thereby incur costs to render tree construction.

5.2.3 RQ3: What are the measured performance differences between strategies?

We measure significant differences of at least 50% in fixed costs of creating elements and components between Svelte with the most performant rendering strategy and its closest challengers. We also find exponentially scaling performance costs in the case of Angular and React when a single component contains a very large number of child components or elements.

The most significant differences are found when updating existing components. Here, differences in rendering strategies can rise to several orders of magnitude in certain cases when components with large subtrees are updated. When components contain a large proportion of static content, we again find differences of up to an order of magnitude in performance even when an equal number of components is updated.

The measured performance differences are found to be exactly in line with what would be expected given the input sizes of each rendering strategy as outlined in table 3.1. The overall strongest update performance is found in Vue and Svelte, which are the only frameworks we reviewed that are able to guarantee optimal input sizes measured in the number of components. Angular's per-component update performance is comparable to Svelte and Vue, but update loop performance can be orders of magnitude worse when a small number of components is updated. The worst update performance of the JavaScript-based frameworks is found with React, with performance ranging from several times to several orders of magnitude slower than competitors in most update tasks. Its performance in component and element creation, however, is competitive with other JavaScript-based frameworks.

The sole WebAssembly-based framework, Blazor, is an outlier with performance several times worse even than React which has an essentially identical rendering strategy. Our results do not allow us to conclude whether this is an inherent weakness of WebAssembly, or one particular to Blazor.

5.2.4 Final thoughts

The purpose of using script-based rendering in browser applications is to enable rich, responsive user interfaces. Responsiveness is trivial when the UI is simple and contains few components. Therefore, the differentiating factor between rendering strategies is how script execution costs scale as the size and complexity of the component tree grows.

As we have seen, the script execution costs of a given UI update will with certain rendering strategies depend only on the complexity of the update itself, while others scale with the complexity of the entire component tree. Rendering strategies of the latter type are characterized by an inability to automatically determine dirty components, and as a result suffer from a performance cost scaling profile similar to traditional page navigation. Frameworks using such rendering strategies may be increasingly incapable of ensuring responsive performance as an application's complexity grows. At a minimum, they are likely to require manual optimization of a kind which in other frameworks can be achieved automatically.

Our results do not allow us to determine what, in practice, constitutes a view complex enough that script execution costs may become an issue. We found update loops exceeding 100ms in some cases with component trees of approximately 1000 components, but our

benchmarks use far simpler components than would be found in real world applications. Performance issues may therefore in practice arise even with smaller component trees, particularly for users with resource constrained devices. We have not surveyed component trees in real-world applications and therefore cannot speculate on what might be a typical size or complexity of a component tree.

A particular case where script execution costs are likely to be of practical concern are applications which render large lists of items, especially when each item is complex and composed of numerous components. When the component containing the list is updated, such as when adding or removing items, differences in the scaling of script execution costs similar to our benchmark scenarios can be expected. Anecdotally, we have implemented business applications containing such lists which render in excess of 10000 components in a single view, and the associated performance issues were one of the motivating factors for this thesis.

In the frameworks we have reviewed, we find that the following patterns result in significant performance gains:

- Use of a reactivity system to automatically detect dirty components
- Use of an optimizing compiler to generate component update code which ignores static content
- Use of binding-based rendering, rather than virtual DOM

The relative performance of the rendering strategies we have reviewed can be approximately estimated based solely on update loop input sizes. We believe this to be a valid and practically useful approach to characterizing and comparing script-based rendering strategies in the browser application context in general.

5.3 Validity of results

All tests were performed on a desktop computer with a Ryzen 5 3600 CPU and 16GB of RAM. Our results are not directly applicable to devices with different specifications. The relative amount of work that each framework must perform remains the same regardless of device, however, and thus their relative results should be similar as well.

We did not measure memory usage, which may be an even greater concern to low-end devices than processing power requirements. During testing, we ran into browser crashes due to excessive memory allocation. This was most commonly encountered in the case of Angular and Blazor, which insert significant amounts of comment nodes in the DOM and thereby increase its size and memory consumption.

All tests were performed using Chrome, and the results are likely not identical in other browsers. Again, however, the fundamental differences in the frameworks' rendering strategies can be expected to produce similar relative results regardless of browser. The one possible exception here is Blazor, where the browser's WebAssembly implementation could potentially significantly affect its performance.

The components used in our scenarios are necessarily very simple. In a real-world application, the number of components would likely seldom reach the higher ends of the ranges we have tested, while any individual component's complexity should in almost all cases be higher. These and other factors which the tested components do not account for will likely increase the costs of rendering individual components. This is likely to increase costs particularly for rendering strategies which do not optimize static content or which perform expensive dirty checks.

The components used in our benchmarks use no custom styling at all and use only the simplest of elements with minimal attributes. Although we have brought up the proportion of script execution costs vs. full render cycle costs where it seems relevant, it is likely that the aspects of rendering not related to script execution would be much more expensive in a real-world application due to more complex layouts. Our benchmarks hence do not represent the relative importance of optimizing script execution times compared to other factors.

Finally, the frameworks we have reviewed are continuously being developed, and their respective results are only valid for the versions we have tested. A list of software versions used in this thesis can be found in appendix A.

5.4 Future work

With our benchmarks, we have attempted to build an objective set of tests that accurately represents some of the fundamental differences between different rendering strategies. While we believe that they fulfill this purpose, we have not attempted to ensure

that the benchmarks are comprehensive, nor can we ensure that they represent real world workloads. A survey of component tree shapes and component complexity in real world applications could help in designing a standardized set of tests that more accurately represents typical applications.

Our benchmarks only compare the performance differences between different frameworks, but not with different rendering strategies entirely. Most obviously useful would be a comparison with server-side rendering and traditional page navigation. There also exists a growing number of open source projects which implement a hybrid approach, where fragments of HTML are rendered server-side and pushed over WebSockets to the browser, where a minimal amount of scripting is used to mount and unmount fragments to and from the DOM. Examples of such projects include Phoenix LiveView* and StimulusReflex for Ruby on Rails†. They exist as a direct alternative to client-side rendering, and are an interesting target for a performance comparison.

This review is not a comprehensive overview of the open source web frontend framework ecosystem. A more wide review of the entire landscape would be useful, and might turn up rendering strategies that differ in important ways from the ones we have covered. In particular, our focus has been primarily on the input sizes of different rendering strategies, but sources of fixed costs could prove to be an equally interesting subject of study.

*https://phoenixframework.org/
†https://stimulusreflex.com/

6 Summary

Traditional page navigation requires the browser to fully re-render a web page whenever any dynamic interaction occurs. This is often an inadequate approach for applications with rich user interfaces and real-time interaction requirements. Using browser APIs to dynamically modify the document overcomes these shortcomings, but is itself fraught with complexity due to the difficulty of managing state transitions in a complex application. Web frontend frameworks solve this problem by providing a declarative abstraction over rendering, and have as a consequence surged in popularity in the past decade.

The rendering strategies used in frontend frameworks vary not only in their technical implementation, but in their performance characteristics. These characteristics may not be obvious to the application developer using such a framework, yet can have significant consequences for the runtime performance of any application implemented using it. When choosing a framework for building a web application, it is therefore crucial to understand the fundamental characteristics of its rendering strategy.

In our review of some of the most popular and influential frontend frameworks, we find that there are major differences particularly in the ways they update existing content. While some frameworks are able to limit an update loop to concern only those parts of the application which actually need to be updated, others may need to walk through every component, regardless of whether they need to be updated or not. Similarly, some frameworks are able to significantly optimize rendering of static content by only rendering it once, whereas others will re-render it on every update.

By benchmarking the frameworks in various situations, we find that these theoretical differences translate directly into differences in practical performance, often with an order of magnitude or more of difference between different strategies. Moreover, we find that there are significant differences even in fixed costs of creating components and elements, something that every framework must perform an equal amount of when initially rendering a view. Overall, we find that significant performance gains are obtained through using a compiler to optimize rendering of static content, implementing a reactivity system to accurately track which components need to be updated, and updating the UI based on individual changes to data bindings rather than explicitly computing the steps required to update the UI to the desired state.

The modern browser is the most ubiquitous application platform in existence today. Given the ease of sharing content on the Web and the browser's availability on a huge variety of devices, this is unlikely to change soon. As the amount of script-driven content increases, web application performance becomes an increasingly pertinent question particularly on resource constrained devices. A better understanding of the tools used to produce content, including their performance characteristics, is therefore a necessity for ensuring that the Web remains a platform accessible to all.

Bibliography

[1] Alex Rickabaugh. *Deep Dive into the Angular Compiler — Alex Rickabaugh — #AngularConnect*. Sept. 2019. URL: https://www.youtube.com/watch?v=anphffaCZrQ (visited on 02/17/2021).

[2] *Angular - NgZone*. URL: https://angular.io/guide/zone (visited on 02/22/2021).

[3] *AngularJS: Scopes*. URL: https://docs.angularjs.org/guide/scope (visited on 03/21/2021).

[4] *ASP.NET Core Blazor WebAssembly performance best practices*. en-us. URL: https://docs.microsoft.com/en-us/aspnet/core/blazor/webassembly-performance-best-practices (visited on 03/21/2021).

[5] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. "A survey on reactive programming". en. In: *ACM Computing Surveys* 45.4 (Aug. 2013), pp. 1–34. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2501654.2501666. URL: https://dl.acm.org/doi/10.1145/2501654.2501666 (visited on 02/22/2021).

[6] P. Bille. "A survey on tree edit distance and related problems". en. In: *Theoretical Computer Science* 337.1-3 (June 2005), pp. 217–239. ISSN: 03043975. DOI: 10.1016/j.tcs.2004.12.030. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397505000174 (visited on 02/19/2021).

[7] D. J. Bouvier. "Versions and standards of HTML". In: *ACM SIGAPP Applied Computing Review* 3.2 (Oct. 1995), pp. 9–15. ISSN: 1559-6915. DOI: 10.1145/228228.228232. URL: http://doi.org/10.1145/228228.228232 (visited on 03/28/2021).

[8] Brian Ford. *Brian Ford - Zones - NG-Conf 2014*. Jan. 2014. URL: https://www.youtube.com/watch?v=3IqtmUscE_U&t=150 (visited on 02/22/2021).

[9] A. Clark. *acdlite/react-fiber-architecture*. original-date: 2016-07-19T01:15:34Z. Feb. 2021. URL: https://github.com/acdlite/react-fiber-architecture (visited on 02/17/2021).

[10] *Constructing the Object Model — Web Fundamentals*. en. URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model (visited on 02/17/2021).

[11] *Critical rendering path - MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path (visited on 02/17/2021).

[12] Evan You. *Evan You on what's coming in Vue 3.0*. VueConf Toronto, Jan. 2019. URL: https://www.youtube.com/watch?v=8Hgt9HYaCDA (visited on 02/22/2021).

[13] J. Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. en-us. Oct. 2005. URL: https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps (visited on 02/25/2021).

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 185–200. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363 (visited on 04/05/2021).

[15] L. T. Hau. *Compile Svelte in your head (Part 1) — Tan Li Hau*. URL: https://lihautan.com/compile-svelte-in-your-head-part-1/#compile-svelte-in-your-head (visited on 03/21/2021).

[16] M. Hevery and A. Abrons. "Declarative web-applications without server: demonstration of how a fully functional web-application can be built in an hour with only HTML, CSS & Javascript Library". en. In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*. Orlando, Florida, USA: ACM Press, 2009, p. 801. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640022. URL: http://dl.acm.org/citation.cfm?doid=1639950.1640022 (visited on 02/18/2021).

[17] P. Hunt, P. O'Shannessy, D. Smith, and T. Coatta. "React: Facebook's Functional Turn on Writing JavaScript: A discussion with Pete Hunt, Paul O'Shannessy, Dave Smith and Terry Coatta". In: *Queue* 14.4 (Aug. 2016), pp. 96–112. ISSN: 1542-7730. DOI: 10.1145/2984629.2994373. URL: https://doi.org/10.1145/2984629.2994373 (visited on 02/19/2021).

[18] *JavaScript Reaches the Final Frontier: Space*. en. URL: https://www.infoq.com/news/2020/06/javascript-spacex-dragon/ (visited on 03/27/2021).

[19] Kara Erickson. *How Angular works — Kara Erickson — #AngularConnect*. Sept. 2019. URL: https://www.youtube.com/watch?v=S0o-4yc2n-8 (visited on 02/17/2021).

[20] *Main thread - MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Main_thread (visited on 03/28/2021).

[21] A. Mesbah and A. van Deursen. "Migrating Multi-page Web Applications to Single-page AJAX Interfaces". In: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. CSMR '07. USA: IEEE Computer Society, Mar. 2007, pp. 181–190. ISBN: 978-0-7695-2802-1. DOI: 10.1109/CSMR.2007.33. URL: https://doi.org/10.1109/CSMR.2007.33 (visited on 03/27/2021).

[22] L. A. Meyerovich and R. Bodik. "Fast and parallel webpage layout". In: *Proceedings of the 19th international conference on World wide web*. WWW '10. New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 711–720. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772763. URL: http://doi.org/10.1145/1772690.1772763 (visited on 04/03/2021).

[23] R. B. Miller. "Response time in man-computer conversational transactions". en. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*. San Francisco, California: ACM Press, 1968, p. 267. DOI: 10.1145/1476589.1476628. URL: http://portal.acm.org/citation.cfm?doid=1476589.1476628 (visited on 04/03/2021).

[24] L. D. Paulson. "Building rich web applications with Ajax". In: *Computer* 38.10 (Oct. 2005). Conference Name: Computer, pp. 14–17. ISSN: 1558-0814. DOI: 10.1109/MC.2005.330.

[25] M. Pawlik and N. Augsten. "Efficient Computation of the Tree Edit Distance". en. In: *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–40. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/2699485. URL: https://dl.acm.org/doi/10.1145/2699485 (visited on 02/19/2021).

[26] *React Top-Level API – React*. en. URL: https://reactjs.org/docs/react-api.html (visited on 03/21/2021).

[27] *Reactivity in Depth — Vue.js*. URL: https://v3.vuejs.org/guide/reactivity.html#what-is-reactivity (visited on 03/21/2021).

[28] *Reconciliation – React*. en. URL: https://reactjs.org/docs/reconciliation.html (visited on 02/19/2021).

[29] *Render-tree Construction — Web Fundamentals*. en. URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction (visited on 03/28/2021).

[30] S. Sanderson. *Shadow/virtual DOM in blazor · Issue #23428 · dotnet/aspnetcore*. en. URL: https://github.com/dotnet/aspnetcore/issues/23428 (visited on 03/21/2021).

[31] P. Snyder, L. Ansari, C. Taylor, and C. Kanich. "Browser Feature Usage on the Modern Web". In: *Proceedings of the 2016 Internet Measurement Conference*. IMC '16. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 97–110. ISBN: 978-1-4503-4526-2. DOI: 10.1145/2987443.2987466. URL: https://doi.org/10.1145/2987443.2987466 (visited on 03/27/2021).

[32] *State and Lifecycle – React*. en. URL: https://reactjs.org/docs/state-and-lifecycle.html (visited on 03/21/2021).

[33] A. Taivalsaari, T. Mikkonen, C. Pautasso, and K. Systä. "Comparing the Built-In Application Architecture Models in the Web Browser". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. Apr. 2017, pp. 51–54. DOI: 10.1109/ICSA.2017.23.

[34] A. Taivalsaari, T. Mikkonen, K. Systä, and C. Pautasso. "Web User Interface Implementation Technologies: An Underview:" en. In: *Proceedings of the 14th International Conference on Web Information Systems and Technologies*. Seville, Spain: SCITEPRESS - Science and Technology Publications, 2018, pp. 127–136. ISBN: 978-989-758-324-7. DOI: 10.5220/0006885401270136. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006885401270136 (visited on 09/24/2020).

[35] T. Van Cutsem and M. S. Miller. "Proxies: design principles for robust object-oriented intercession APIs". In: *ACM SIGPLAN Notices* 45.12 (Oct. 2010), pp. 59–72. ISSN: 0362-1340. DOI: 10.1145/1899661.1869638. URL: https://doi.org/10.1145/1899661.1869638 (visited on 04/05/2021).

[36] Victor Savkin. *Change Detection Reinvented Victor Savkin*. Mar. 2015. URL: https://www.youtube.com/watch?v=jvKGQSFQf10 (visited on 02/25/2021).

[37] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. "Why are web browsers slow on smartphones?" In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile '11. New York, NY, USA: Association for Com-

puting Machinery, Mar. 2011, pp. 91–96. ISBN: 978-1-4503-0649-2. DOI: 10.1145/2184489.2184508. URL: http://doi.org/10.1145/2184489.2184508 (visited on 03/28/2021).

[38] *WebAssembly Concepts — MDN*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts (visited on 03/28/2021).

[39] A. Wirfs-Brock and B. Eich. "JavaScript: the first 20 years". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (June 2020), 77:1–77:189. DOI: 10.1145/3386327. URL: https://doi.org/10.1145/3386327 (visited on 03/25/2021).

Appendix A Software versions

The following software versions were used in this thesis:

Name	Version
Angular	11.2.3
React	17.0.1
Vue	3.0.7
Svelte	3.35.0
Blazor	5.0.3
Chrome	89.0.4389.90

Appendix B Full benchmark results

A full list of results from our benchmarks is included below. For each benchmark, we measured script execution times and total render cycle times separately, and have included their respective results accordingly. All results are in milliseconds, rounded to the closest millisecond.

Table B.1: Group 1: Creating static elements - script execution time

N	Angular	React	Vue	Svelte	Blazor
1	0	0	0	0	1
100	2	2	2	1	3
200	4	3	3	1	4
300	7	5	3	1	5
400	8	6	5	1	6
500	9	9	5	2	7
600	11	7	7	2	8
700	12	8	7	2	9
800	13	9	10	2	10
900	15	10	11	2	11
1000	17	11	11	3	13
2000	36	25	24	7	23
3000	55	40	29	9	36
4000	72	56	33	13	47
5000	84	76	36	16	60
6000	100	95	39	17	71
7000	114	115	43	19	81
8000	132	138	47	20	92
9000	154	166	50	23	103
10000	175	197	53	26	121
15000	305	386	67	37	191
20000	561	635	85	46	283
25000	845	963	104	54	368
30000	1096	1375	118	63	467
35000	1405	1761	142	70	574
40000	1739	2264	155	80	684
45000	2111	2880	174	90	797
50000	2539	3559	190	102	921

Table B.2: Group 1: Creating static elements - total render time

N	Angular	React	Vue	Svelte	Blazor
1	5	5	5	4	5
100	7	7	7	4	6
200	8	10	7	5	8
300	12	11	7	6	10
400	13	13	9	5	10
500	16	18	11	7	12
600	19	16	11	9	14
700	23	18	15	7	14
800	24	20	18	9	19
900	27	19	18	10	21
1000	32	21	20	8	20
2000	62	51	50	14	44
3000	92	77	65	32	66
4000	118	103	77	57	87
5000	142	134	89	77	109
6000	170	164	103	89	128
7000	196	196	125	102	148
8000	225	231	141	115	169
9000	259	270	156	129	189
10000	294	313	171	143	224
15000	484	562	244	215	349
20000	802	873	329	286	502
25000	1150	1264	410	358	644
30000	1463	1738	488	431	792
35000	1839	2185	572	498	959
40000	2233	2750	644	567	1124
45000	2667	3429	728	637	1297
50000	3160	4171	802	719	1476

Table B.3: Group 1: Creating components with a single parent - script execution time

N	Angular	React	Vue	Svelte	Blazor
1	0	0	0	0	1
100	5	3	4	1	9
200	7	7	9	2	13
300	10	11	11	2	18
400	13	14	15	4	23
500	16	16	20	5	28
600	20	18	23	7	32
700	24	21	27	8	37
800	29	25	30	9	42
900	33	28	34	9	47
1000	35	28	34	10	52
2000	59	55	67	21	106
3000	76	77	80	30	161
4000	88	98	93	34	214
5000	102	122	105	40	268
6000	108	151	122	47	318
7000	120	177	129	50	371
8000	126	207	139	55	421
9000	132	243	147	60	493
10000	140	278	169	72	557
15000	164	507	193	95	858
20000	186	782	242	131	1181
25000	208	1128	282	152	1501
30000	231	1532	309	168	1826
35000	256	2019	355	190	2222
40000	280	2553	390	213	2615
45000	339	3169	428	230	3016
50000	364	3840	473	252	3451

Table B.4: Group 1: Creating components with a single parent - total render time

N	Angular	React	Vue	Svelte	Blazor
1	4	4	4	4	5
100	11	8	9	5	13
200	16	13	15	8	17
300	22	18	18	10	24
400	28	23	24	10	30
500	34	27	31	13	36
600	41	29	34	15	42
700	48	31	39	18	47
800	56	35	44	18	53
900	62	42	49	23	59
1000	66	42	50	21	65
2000	118	83	92	42	127
3000	163	114	116	66	192
4000	206	146	139	78	255
5000	252	179	162	99	319
6000	293	220	192	121	378
7000	334	257	211	135	442
8000	375	300	232	151	504
9000	417	347	250	168	587
10000	455	394	286	194	663
15000	646	684	372	279	1019
20000	837	1023	482	379	1401
25000	1023	1432	585	469	1775
30000	1211	1898	671	548	2156
35000	1413	2450	779	635	2607
40000	1600	3043	875	723	3059
45000	1827	3719	980	813	3527
50000	2019	4453	1090	900	4007

Table B.5: Group 1: Creating components as a binary tree - script execution time

N	Angular	React	Vue	Svelte	Blazor
2	2	1	1	0	3
4	2	1	2	1	3
8	3	1	2	1	3
16	4	2	3	1	4
32	6	2	5	1	6
64	11	4	8	2	9
128	20	7	16	3	17
256	41	15	31	6	30
512	75	32	53	10	59
1024	120	55	84	22	128
2048	162	86	132	43	241
4096	216	137	223	83	485
8192	297	233	313	142	966
16384	469	394	485	233	1870
32768	774	733	897	482	3644
65536	1358	1365	1611	771	7390

Table B.6: Group 1: Creating components as a binary tree - total render time

N	Angular	React	Vue	Svelte	Blazor
2	6	5	6	5	6
4	6	5	6	5	6
8	6	6	6	5	6
16	8	6	6	4	7
32	12	6	8	5	10
64	18	10	12	7	13
128	34	14	24	7	23
256	65	26	39	10	39
512	117	45	72	22	74
1024	204	77	115	53	153
2048	332	142	190	101	290
4096	556	249	340	198	585
8192	992	464	549	375	1172
16384	1878	858	953	696	2272
32768	3654	1669	1836	1407	4446
65536	7210	3253	3483	2625	8991

Table B.7: Group 2: Updating the root component - script execution time

N	Angular	React	Vue	Svelte	Blazor
2	0	1	0	0	2
4	0	1	1	0	3
8	0	1	1	0	3
16	1	1	1	0	3
32	1	2	1	0	2
64	2	3	1	0	3
128	4	7	1	0	2
256	7	13	1	0	3
512	12	23	1	0	2
1024	17	42	1	0	3
2048	23	53	1	0	3
4096	32	93	1	0	7
8192	29	148	5	0	8
16384	43	212	1	0	3
32768	119	387	1	0	5
65536	124	874	8	0	3

Table B.8: Group 2: Updating the root component - total render time

N	Angular	React	Vue	Svelte	Blazor
2	5	4	5	5	6
4	5	5	5	4	6
8	5	5	4	3	6
16	4	4	4	5	6
32	4	6	4	5	6
64	5	8	5	6	6
128	8	12	4	7	5
256	12	16	5	7	6
512	17	32	9	8	6
1024	26	47	5	4	9
2048	37	59	6	5	10
4096	53	105	13	10	20
8192	60	166	20	12	28
16384	98	237	26	22	41
32768	224	439	48	24	51
65536	330	975	101	42	86

Table B.9: Group 2: Updating a leaf component - script execution time

N	Angular	React	Vue	Svelte	Blazor
2	0	0	0	0	1
4	0	0	0	0	1
8	0	0	0	0	1
16	1	0	0	0	1
32	1	0	0	0	1
64	2	0	0	0	1
128	4	0	0	0	1
256	8	0	0	0	1
512	11	0	0	0	1
1024	16	1	0	0	1
2048	23	3	0	0	1
4096	31	3	0	0	5
8192	32	5	0	0	6
16384	44	3	0	0	2
32768	115	3	0	0	8
65536	125	7	6	0	2

Table B.10: Group 2: Updating a leaf component - total render time

N	Angular	React	Vue	Svelte	Blazor
2	5	4	4	5	4
4	5	4	5	5	4
8	4	4	4	4	4
16	5	5	5	5	5
32	5	5	5	6	5
64	5	7	4	7	6
128	8	3	4	7	5
256	13	4	4	6	4
512	18	4	5	4	4
1024	25	5	5	3	6
2048	37	8	6	3	7
4096	52	14	13	6	18
8192	63	26	24	5	26
16384	100	27	25	11	40
32768	221	51	45	5	60
65536	330	87	91	7	86

Table B.11: Group 3: Updating the root component - script execution time

N	Angular	React	Vue	Svelte	Blazor
2	0	1	1	0	3
4	0	2	0	0	2
8	0	5	1	0	3
16	0	10	0	0	2
32	1	15	0	0	2
64	2	28	1	0	3
128	3	34	1	0	4
256	5	43	1	0	3
512	14	64	1	0	3
1024	23	98	4	0	13
2048	20	225	1	0	10
4096	24	278	1	0	8
8192	148	806	8	0	7

Table B.12: Group 3: Updating the root component - total render time

N	Angular	React	Vue	Svelte	Blazor
2	5	5	5	5	7
4	6	6	6	5	7
8	7	11	8	8	9
16	4	13	8	12	6
32	5	19	5	26	6
64	6	32	15	4	7
128	7	40	6	4	13
256	24	61	7	5	8
512	29	77	12	14	16
1024	34	114	14	25	48
2048	40	240	16	17	62
4096	59	307	24	30	100
8192	219	850	58	101	83

Table B.13: Group 3: Updating all components - script execution time

N	Angular	React	Vue	Svelte	Blazor
2	0	1	1	0	3
4	0	2	1	0	3
8	0	4	2	0	4
16	1	9	3	1	5
32	1	15	6	1	8
64	2	28	10	1	14
128	4	34	20	2	28
256	8	44	32	3	60
512	17	66	42	5	101
1024	27	101	72	10	250
2048	29	235	91	20	502
4096	44	289	149	54	1020
8192	238	841	311	80	2013

Table B.14: Group 3: Updating all components - total render time

N	Angular	React	Vue	Svelte	Blazor
2	6	5	5	6	7
4	7	6	6	5	8
8	5	8	8	9	9
16	5	15	10	7	9
32	7	19	12	10	14
64	9	35	21	5	20
128	11	43	29	7	41
256	31	67	56	13	72
512	47	91	67	29	126
1024	72	142	106	56	303
2048	115	295	152	82	605
4096	211	419	270	174	1213
8192	577	1068	542	387	2396