MILLION.JS: A FAST, COMPILER-AUGMENTED VIRTUAL DOM FOR PERFORMANT JAVASCRIPT UI LIBRARIES

Aiden Bai
aiden.bai05@gmail.com

February 17, 2022

Abstract

The need for developing and delivering interactive web applications has grown rapidly. Thus, JavaScript User Interface (UI) libraries were introduced, at the detriment of performance and bundle size. To solve this problem, Million.js[1] was created to allow for the creation of compiler-augmented JavaScript UI libraries that are extensible, performant, and lightweight. This was accomplished by designing a computationally efficient diffing algorithm that relies on a compiler, and then measuring the performance with a series of relevant and exhaustive benchmarks. Additionally, built-in mechanisms are implemented to allow for imperative optimizations, allowing the compiler to directly skip runtime diffing. When compared to other methods of virtual DOM rendering, these findings showed that Million.js had superior performance, with 133% to 300% more operations per second than other Virtual DOM libraries.

Keywords: Web Programming, JavaScript, User Interface (UI), Document Object Model (DOM), Virtual DOM, Compiler, Static Analysis, Single-Page Application (SPA), Open Source, Mobile Websites, Functional Programming, Components, Optimization

[1] https://github.com/aidenybai/million

Introduction

The way that web developers think about building websites has changed. Initially, websites started off as static pages, which were very simple in interactivity and primarily relied on markup languages like Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS). As the usage of the web grew, JavaScript and the Document Object Model (DOM) was created to provide more complex interactivity. Static content was served by a monolithic server, and interactivity was supplemented with basic DOM operations. Today, the web has changed drastically, as websites have shifted focus from serving static content to full on single page applications (SPA). These web applications are rendered using JavaScript, and heavily use libraries like React.js, Vue.js, or Angular.js. These libraries provide greater functionality and flexibility. With the creation of these libraries, new declarative programming paradigms and innovations in internal design, such as virtual DOM and its variants [2]. However, the SPA architecture and virtual DOM libraries have come with the issues of detrimental performance, meaning that user experience is degraded as a result.

Historical Overview

JavaScript is a programming language that has historically been an invaluable part of web development. During the pioneer era of the web, web pages were statically served, meaning dynamic portions were limited to embedded images with the Graphics Interchange Format (GIF) or small CSS animations. Nowadays, with entire web pages being built with primarily JavaScript, it has changed from a toy animation language to a language with a wide ecosystem of code and libraries for numerous use cases [10].

JavaScript's first version was created by Brendan Eich of Sun Microsystems as an effort to add a scripting language to Netscape Navigator [7]. As time progressed, more and more web browsers like Google Chrome, Mozilla Firefox, and Internet Explorer have been created and have adopted JavaScript as their main scripting language, but with differing implementations. As an outcome of the stratification of JavaScript implementations, many of the companies that run these browsers banded together into the ECMA International consortium, allowing for the standardization of JavaScript as ECMAScript [7]. Throughout the years, ECMA International has pushed out new standardizations to accommodate for new features. However, most web development standards have settled on

ECMAScript 5 (ES5), even though most modern browsers support ECMAScript 6 (ES6). This is due to backward compatibility for users that still use older browsers [3].

Browser-side JavaScript has many uses. One of its main uses is to interact with HTML and XML documents [8]. It does this through the DOM, which is a tree-like Application Programming Interface (API). The DOM tree consists of nodes and objects, and has properties and methods which allow for manipulation of the nodes themselves by developers. The introduction of the DOM enabled the HTML and JavaScript to interface with each other, creating one of today's intrinsic parts of the web: interactivity, or the ability for the website's content to change during runtime [8].

After the creation and development of JavaScript and the DOM, web developers have created tools and libraries for simplifying development and to add functionality. One of these libraries is jQuery, a widely used DOM utility library. The goal of jQuery was to help developers solve issues and allow for cross-browser compatibility, while providing a clean and easy syntax to use [7]. jQuery was important because it helped web developers concisely manipulate the DOM, and segregate the HTML document and the JavaScript by removing the necessity of using event listener attributes to run JavaScript code [8]. These innovations allowed jQuery to flourish in the JavaScript community and ecosystem, which is why it is being used in a significant portion of production websites today [8].

Although jQuery had a monumental impact on web development, many web developers were still unsatisfied with the ergonomics of jQuery and the DOM, due to imperative APIs. In 2013. Software engineers at Facebook started to work on a JavaScript UI library called React.js. React.js uses a virtual DOM engine, which allows the DOM to be modelled as a superset, utilizing a diff algorithm to determine pinpoint DOM manipulations to reduce reflow and repaints [5]. React.js' invention of the virtual DOM allowed for a new way of declaratively writing efficient web applications. React.js sparked a revolution in JavaScript UI libraries, leading to the incorporation of the virtual DOM in newer JavaScript UI libraries like Ember 2.0 and Vue.js [4].

Current Trends and Practices

The usage of JavaScript in the web has expanded over the years. JavaScript now handles an immense amount of interactivity and powers SPAs, continuing to

be used and integrated into the web. Many developers have created JavaScript UI libraries to enhance the development experience of the aforementioned web applications, such as jQuery, React.js, and Vue.js. These JavaScript UI libraries use several methodologies of development. For example, React.js focuses on providing declarative components at a primitive level, meaning that abstractions in web applications are built up using primitive APIs. Another example is Vue.js, which focuses on being a progressive framework, allowing a wide range of developers to write declarative JavaScript incrementally regardless of experience. "Vue.js can be easily introduced into an app by simply including a script tag, and more can be added according to one's needs" [7].

Most mainstream and modern libraries construct their APIs to be easy, declarative, scalable, and reactive. Often these libraries utilize techniques like virtual DOM architectures to implement such functionalities or paradigms at the DOM level [6]. The virtual DOM technique is utilized by both Vue.js and React.js, with varying differences in application of templating, such as the use of JSX, Functional Components, and Hooks in React.js or Single-File Components and directives in Vue.js. "React.js has its own JavaScript language called JSX. This provides an extension to JavaScript by adding XML-like syntax that is similar to HTML" [7]. Other libraries such as jQuery take advantage of direct DOM manipulation, opting for a more imperative API but simultaneously simpler to use than native code. The variety of choice in libraries allows developers to choose a paradigm based on the conditions of their project [5].

As of late, new techniques like Server Side Rendering (SSR) and Server Side Generation (SSG) have become more prevalent in meta-frameworks, such as Next.js and Nuxt.js. These techniques attempt to improve the runtime performance of JavaScript UI libraries like React.js and Vue.js, respectively, by utilizing prerendering, code-splitting, static-hoisting, and bundling techniques [7]. These techniques, when combined together with caching, can significantly improve the user experience.

Controversies and Debates

The lineup of popular open-source [9] JavaScript libraries, like React.js, Vue.js, and Angular.js contain codebases that have received criticism for unnecessary abstractions and code bloat. "[Many JavaScript UI libraries are] over-engineered and too complicated ... However, that has not stopped its adoption and legions

of adoring fans" [4]. This often results in an indirect effect of degradation of user experience and web performance.

Another controversy in the JavaScript library community is how markup and JavaScript logic are handled. "Ember will serve as the diametric comparison for React.js in terms of separating logic from markup" [4]. React.js encourages developers to blend and mix markup and logic together through JSX, however JavaScript UI libraries like Ember strictly segregate markup and logic into different partitions. These two different philosophies have been criticized either for being too loose or opinionated, prompting different web developers on different sides to conflict. One could argue that this type of stratification of the community results in harming all parties and these issues can even propagate up to development at the enterprise level.

A rising issue that web developers face is the amount of abstraction in the architecture of modern JavaScript libraries. The amount of abstractions ranges widely, from libraries like React.js focusing on a primitive developer experience with a low learning curve, and Angular.js being the direct opposite. Both solutions have differing use cases, but from a holistic perspective, among the general developer community, primitive construction is much more appealing, bridging the way for widespread adoption by developers.

Finally, some web developers question the relevance of extreme performance in JavaScript UI libraries. Often, they will argue that the performance of mainstream JavaScript libraries are adequate for most use cases. While this is true, accessing websites via a mobile device or for applications with heavy data sets requires optimal performance, which invalidates the claim of subpar performance being a heuristic solution.

Methodology

I started off by creating the API, performing tests through an automated pipeline, and publishing the package on NPM. At each point in the process, I enforced code styling and conventions with tools like Prettier and ESLint, ensuring that maintainability was nurtured, while minimizing barriers of entry for other contributors.

Development

I constructed the codebase using TypeScript, a superset of the JavaScript language. The project did not use any NPM dependencies to reduce bundle size, and the build pipeline was managed by Rollup, with adapters for TypeScript. The project holistically can be represented as a variant of the virtual DOM pattern. This means it still includes a reconciliation algorithm, but also implements compiler flags, deltas, and keyed node patterns.

In *Figure 1*, I created a three step process of the initial render of content to a page.

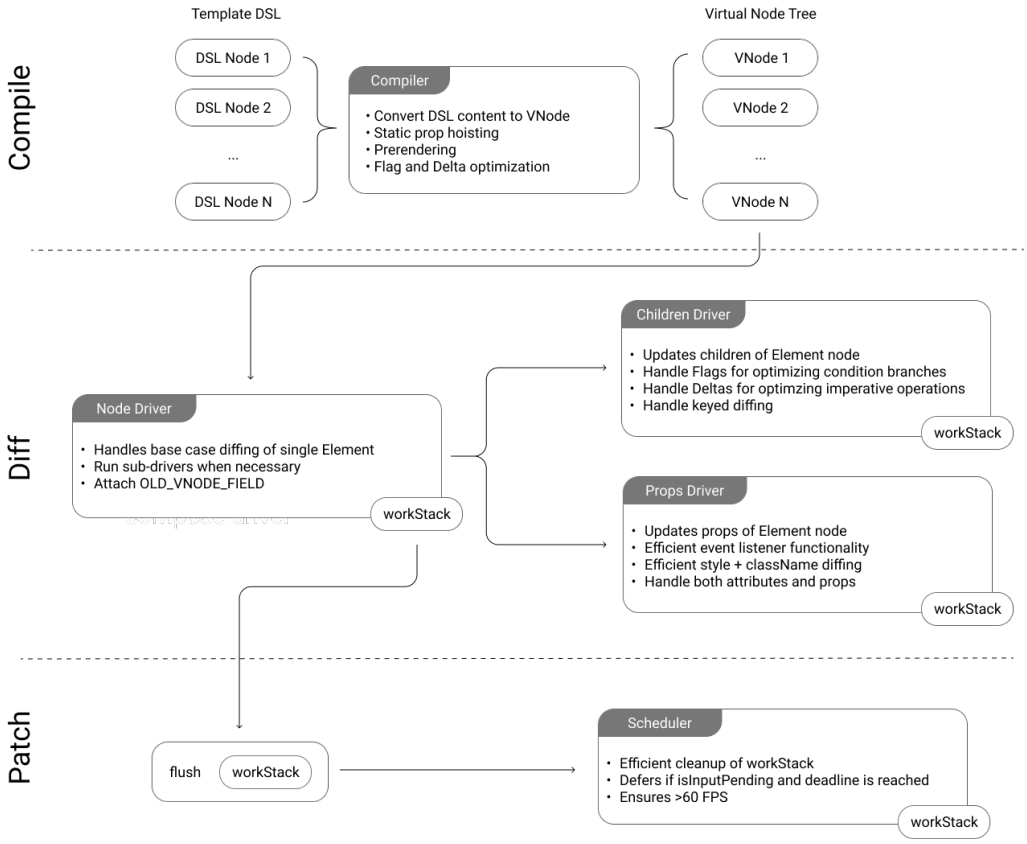

Figure 1. Flowchart of Million.js render process

Compile The compile step occurs only once in either the server through SSR or with the developer through SSG. This process takes a payload in a template DSL, such as JSX, or framework-specific SFCs. I didn't need to write a full compiler for this step, as most JavaScript UI libraries already can produce their own optimizations, such as prerendering when generating a virtual node tree. However, I still wrote compatibility layers for JSX for further optimizations to the runtime as a Babel plugin as a proof-of-concept.

Diff & Patch The diff and patch steps are part of the runtime process and are run on initial load and every time there is a render call. Virtual node trees, generated from the compile step, are passed into special blocks of logic called drivers. These drivers can be composed together for extensibility. A composed driver produces a work stack, which is flushed during the patch phase and optionally scheduled to ensure 60 frames per second for the user. A simplified diagram of diffing and patching trees is demonstrated in *Figure 2*.

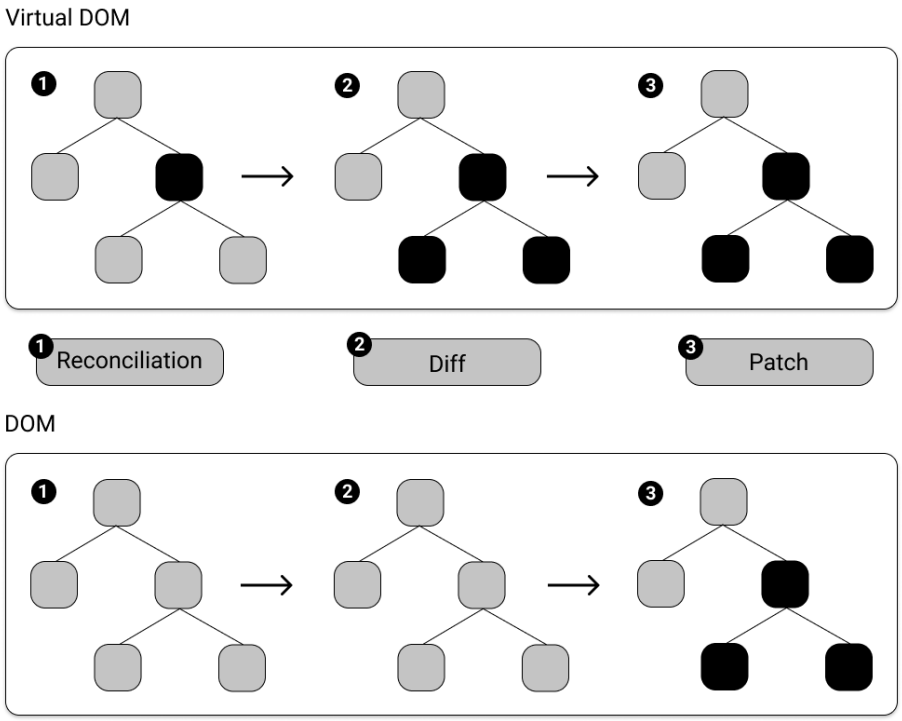

Figure 2. Simplified flowchart of virtual DOM and DOM interaction.

Optimizations

Million.js is a virtual DOM. The virtual DOM exists because swapping of the innerHTML property for DOM manipulation is generally not recommended, as it is computationally expensive to use an in-browser HTML parser to modify a DOM tree. A virtual DOM attempts to remedy this by creating a virtual representation of the DOM inside JavaScript objects, which are computationally inexpensive. Every JavaScript object, or virtual node, can be diffed to determine modifications and patched in the DOM. This allows for pinpoint DOM modifications, thereby reducing overhead of interacting with the DOM.

However, unlike other virtual DOM libraries, Million.js integrates ways for a compiler to optimize condition branches and reduce extraneous diffing. This allows for significantly better time and space complexity. In fact, Million.js can achieve $O(1)$ best case time complexity for diffing of a virtual node tree when optimized correctly, compared to other virtual DOM libraries which have $O(n)$ best case time complexity.

Keyed Nodes Most efficient virtual DOM engines use a keying mechanism to identify virtual nodes during diffing. This allows for more precise DOM manipulations, reducing overhead. However, in most cases, keyed children requires substantial diffing, resulting in an extremely inefficient time complexity. Million.js instead uses keys to determine whether a virtual node is static, reducing time complexity on both fronts. The steps to the keyed algorithm can be seen in *Figure 3*.

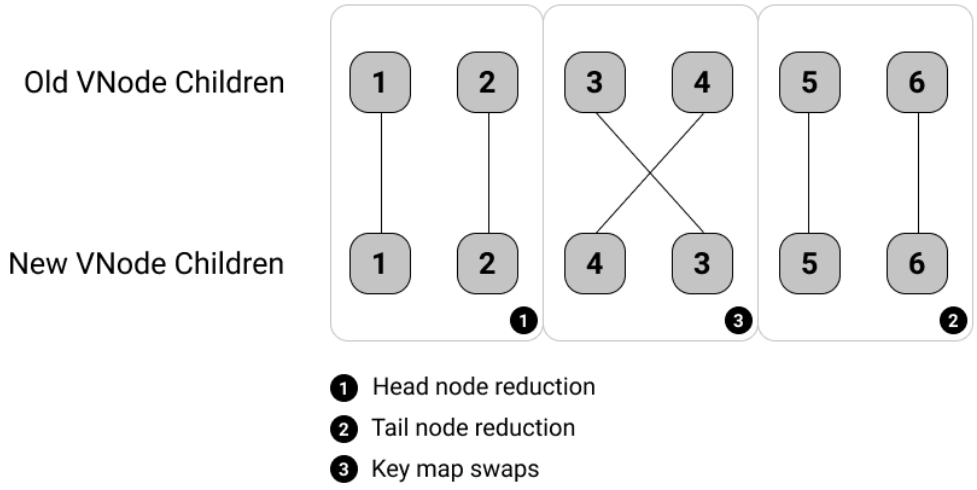

Figure 3. Keyed children diffing algorithm diagram

Compiler Flags Since virtual DOM engines generally encourage usage of a preprocessor to compile a template DSL, such as JSX or SFCs, custom compilation hints can be leveraged. Compiler flags allow for deterministic patching of the current node and children nodes, optimizing condition branches. This, Compiler flags reduce the time needed to diff. The JSX compatibility layer for Million.js detects the content of children and applies such flags during build time.

Deltas For some cases, particularly with benchmarks, manipulation of the DOM is simple and predictable. Diffing for such cases is unnecessary work, which is why Million.js supports a concept called deltas, providing a programmatic API for applying deterministic, imperative modifications to the DOM.

Scheduling While optimizing for raw performance can produce better numerical benchmark scores, this optimization does not ensure a consistent 60 frames per second experience during webpage navigation. With the ability to upgrade the patching process with scheduling, patches to the DOM can be deferred to a later time if the main thread is blocked, mitigating "page-freeze" scenarios.

Quality Assurance

Unit Testing "The TDD [Test-Driven Development] practice will yield code with superior external code quality when compared with code developed with a waterfall-like practice" [1]. For unit tests, I used Vitest to run code assertion to automate generic tests that run on lifecycle hooks to ensure that established code is working properly. During the final stages of development, I utilized the native coverage report system Jest provides to determine which parts of the codebase need more testing, and integrated the test runner into a GitHub Action continuous integration script.

Ad-hoc Debugging Although TDD can reduce the density of defects [10], tightening the development feedback loop allows for rapid, ad-hoc debugging. I used Vite.js' development server to serve content over ES modules and Chrome DevTools to quickly find issues and fix them.

Package Publishing First, I created a production build process with Tsup. Then, I published the distribution bundles on NPM, Unpkg, Skypack, and JSDelivr. After that, I ensured that Dependabot does not display any security concerns, and any subsequent publishes should follow the semantic versioning policy through GitHub releases. This is to allow for users to consistently use specific versions and update dependencies only if necessary.

Benchmarking To gauge the performance of Million.js and similar methods of rendering, I used the Benchmark.js library, coupled with Vite for the implementation. I also followed the official JS Framework Benchmark guidelines and ensured that the benchmarks suites were standardized. The results of the benchmark scores and deviations are listed in *Table 1* and *Table 2*.

Results

Ten different measurement benchmark suites were tested on seven different methods of webpage rendering, as seen in *Table 1*. The first type of method is the raw rendering method, specifically DOM and innerHTML. The second type of method is the virtual DOM library, specifically Million.js[2], Snabbdom[3], virtual-dom[4], tiny-vdom[5], and simple-virtual-dom[6]. The virtual DOM libraries, with the exception of Million.js and Snabbdom, had less operations per second when compared with raw DOM rendering methods, with the worst performing methods being tiny-vdom and simple-virtual-dom.

Raw rendering methods, like DOM and innerHTML had relatively higher operations per second in aggregate insertions or deletions, but relatively lower operations per second in partial or complex updates when compared to virtual DOM methods. Additionally, methods with keyed diffing, specifically Million.js and Snabbdom, had higher operations per second when compared to the other virtual DOM libraries.

Additionally, Million.js has a much smaller bundle size at 0.75 kilobytes compared to Snabbdom and virtual-dom, but slightly larger than tiny-vdom at 0.32 kilobytes (see *Table 2.*). Since Million.js is composable, the bundle size can become even lower through tree shaking and dead code elimination.

[2]https://github.com/aidenybai/million
[3]https://github.com/snabbdom/snabbdom
[4]https://github.com/Matt-Esch/virtual-dom
[5]https://github.com/aidenybai/tiny-vdom
[6]https://github.com/livoras/simple-virtual-dom

Method	DOM	million	innerHTML	snabbdom	virtual-dom	tiny-vdom	simple-virtual-dom
Append 1,000 rows to a table of 10,000 rows (ops/s)	21.7	21.8	9.4	6.6	5.9	5.1	4.6
Clear a table with 1,000 rows (ops/s)	207.0	190.0	172.0	128.0	97.7	93.7	64.1
Create 10,000 rows (ops/s)	36.8	30.5	36.1	22.2	22.3	12.9	9.3
Create 1,000 rows (ops/s)	36.0	30.3	35.8	22.2	22.4	12.7	9.0
Update every 10th row for 1,000 rows (ops/s)	220.0	221.0	135.0	76.4	69.8	58.7	51.2
Remove a random row from 1,000 rows (ops/s)	225.0	202.0	138.0	74.2	57.6	41.7	31.4
Update 1000 rows (ops/s)	119.0	127.0	123.0	50.7	55.5	42.3	33.2
Select a random row from 1,000 rows (ops/s)	210.0	208.0	123.0	72.5	66.6	46.8	45.4
Swap 2 rows for table with 1,000 rows (ops/s)	224.0	198.0	118.0	73.5	57.9	46.8	31.6
Geometric mean (ops/s)	105.7	98.8	74.3	44.1	39.8	29.7	23.1

Table 1. JS Framework Benchmark compatible benchmarks tested on Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.71 Safari/537.36. Aggregated from https://million.aidenybai.com

Method	DOM		million		innerHTML		snabbdom		virtual-dom		tiny-vdom		simple-virtual-dom	
Benchmarks	**Value**	**Deviation**	**Value**	**Deviation**	**Value**	**Deviation**	**Value**	**Deviation**	**Value**	**Deviation**	**Value**	**Deviation**	**Value**	**Deviation**
Append 1,000 rows to a table of 10,000 rows (ops/s)	**21.72**	**0.90%**	21.78	3.33%	9.38	2.91%	6.59	5.88%	5.91	3.53%	5.13	3.50%	4.58	2.79%
Clear a table with 1,000 rows (ops/s)	**207**	**2.35%**	190	2.42%	172	2.87%	128	4.31%	97.68	3.77%	93.67	3.58%	64.13	0.75%
Create 10,000 rows (ops/s)	**36.83**	**4.56%**	30.49	4.99%	36.05	4.84%	22.2	1.78%	22.34	1.16%	12.86	3.39%	9.25	1.25%
Create 1,000 rows (ops/s)	**35.99**	**5.21%**	30.34	4.67%	35.82	4.57%	22.22	2.10%	22.36	1.57%	12.72	3.60%	8.99	1.37%
Update every 10th row for 1,000 rows (ops/s)	220	3.55%	**221**	**3.31%**	135	3.55%	76.41	5.40%	69.76	3.43%	58.74	4.03%	51.19	3.75%
Remove a random row from 1,000 rows (ops/s)	**225**	**5.22%**	202	3.82%	138	3.67%	74.16	8.85%	57.59	6.67%	41.74	7.81%	31.42	3.88%
Update 1000 rows (ops/s)	119	1.91%	**127**	**3.33%**	123	2.78%	50.69	0.43%	55.54	6.66%	42.32	5.18%	33.2	0.80%
Select a random row from 1,000 rows (ops/s)	**210**	**5.88%**	208	5.73%	123	5.30%	72.49	7.47%	66.64	7.19%	46.79	7.24%	45.43	7.29%
Swap 2 rows for table with 1,000 rows (ops/s)	**224**	**6.77%**	198	4.41%	118	5.60%	73.51	8.17%	57.85	8.93%	46.79	6.26%	31.6	5.70%
Geometric mean (ops/s)	105.73	3.46%	98.76	3.88%	74.33	3.88%	44.15	3.66%	39.76	3.96%	29.68	4.71%	23.10	2.29%
Minified+brotli size (kb)	N/A		0.75		N/A		6.44		9.85		0.32		22.2	
Architecture	N/A		Composable		N/A		Composable		Abstracted		Abstracted		Abstracted	
GitHub Stars	N/A		1,226		N/A		9,599		11,217		71		1,652	
NPM installs (2021)	N/A		6,154		N/A		42,877		57,884		43		820	
Age (days)	N/A		166		N/A		2,378		2,849		56		2,184	

Table 2. Raw qualatative observations and quantative JS Framework Benchmark compatable benchmarks with deviations.

The geometric mean of benchmark results is visualized in *Figure 4*. Additionally, due to uncertainties with benchmark measurements, an average deviation of 1.856 operations per second is also visualized in *Figure 3*. The DOM method has the highest operations per second of the methods tested, with an geometric mean of 105.7 operations per second. Million.js is relatively comparable to the DOM method, and significantly outperforms the latter methods by 133% to 300%.

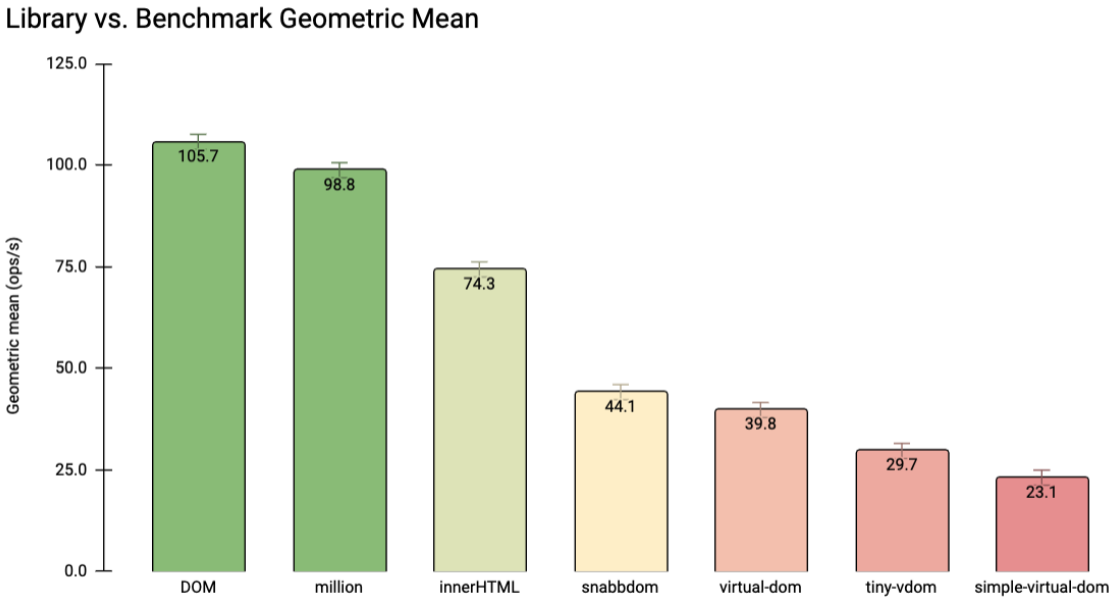

Figure 4. Column graph of the geometric mean of render speeds for benchmarks measured in operations per second.

Discussion

The goal of this project was to create a virtual DOM library and benchmark it against other popular methods to compare performance. These results clearly display that Million.js has a higher benchmark score when compared to other libraries. With deviation taken to account in *Figure 3*, it shows that error can be disregarded when taking a holistic analysis of the data.

Where nuance comes in is the consideration of applying libraries into practical applications and thereby the significance of benchmark performance. For instance, inserting, updating, and deleting one thousand rows is generally not prevalent among commonly used web applications. In reality, it may be something simple, such as opening and closing a modal. Some developers may argue that benchmarks do not accurately represent what is reality and should be taken with a grain of salt.

Although this is true to an extent, it does not take into consideration what benchmarks actually test. At face value, it seems that benchmarks only test extremely esoteric tasks, such as insertions and partial updates, but in reality it is also testing something much more fundamental—the rendering engines of the libraries. It is possible it may vary depending on tasks, but the rendering engines remain constant and therefore can be a valid measure of determining performance. Not only this, to ensure consistency, the well-known JS Framework Benchmark

guidelines, which are well accepted amongst the JavaScript community as a metric for performance, were used in the benchmarking process.

Additionally, a general inconsistency among all codebases alike are bugs, which could have a detrimental impact on benchmarks and performance in general. This is to be expected, as all systems are prone to this issue if programmed by a human, and this is no exception. These bugs are generally mitigated by offering issue-tracking on the Million.js Github repository [9], where users are able to submit issue posts that the maintainer can see and take action on. Several preventative measures are also taken, notably unit testing, continuous integration testing, and ad-hoc testing.

Despite the fact these benchmarks provide computational measurements of each method, it doesn't account for page interactivity and main thread blocking. When presented with a large number of entries in a list rendering example, the computations will block the main thread of JavaScript, creating the "page-freeze" situation, as tasks need to wait for the main thread to be unblocked. Benchmarks inherently assume that the user doesn't experience CPU performance limitations, which is untrue, especially with the advent of mobile devices. This is why Million.js implements scheduling, or the action of intentionally debouncing render calls if the main thread becomes blocked.

Conclusion

In this experiment, I constructed a virtual DOM that was capable of being a compilation target for JavaScript compilers, utilizing techniques to reduce unnecessary runtime complexity. Simultaneously, I designed the API to be composable, allowing for greater extensibility. My results showed that Million.js had significantly better performance when compared to other virtual DOM libraries, ranging from 133% to 300% higher operations per second in the geometric mean of benchmark scores. Additionally, because of the framework-agnostic architecture of Million, it can be applied in both runtime and compiled environments, making it extremely flexible. Overall, with promising results in benchmarks, Million.js has the potential to significantly improve the performance and internal ergonomics of virtual DOM-based JavaScript UI libraries.

Due to this, Million.js has received notable media feedback from the web development community. Million.js has been featured in major developer publications like YC Hacker News, JavaScript Weekly, and DEV. On GitHub, Million.js has

received over 1.3k stars and over 100k unique visitors, signifying overwhelming developer interest. Not only this, Million.js is also being used in real-world production applications at consulting agencies.

In the future, I plan to create a JSX-based compiler and JavaScript UI library on top of Million.js as a proof-of-concept and discover additional optimizations that could be added to Million.js. Additionally, I will further promote and negotiate with JavaScript library developers to integrate Million.js into their own projects, allowing for the expansion of the Million.js ecosystem. I hope to see the advent of Million.js spark greater discussion and experimentation with compilers for the optimization of the virtual DOM.

References

[1] George, B., & Williams, L. (2004). A structured experiment of test-driven development. Penn State University. Retrieved June 17, 2021, from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.147&rep=rep1&type=pdf

[2] Grov, M. (2015, May 20). Building User Interfaces Using virtual DOM. University of Olso. Retrieved June 17, 2021, from https://www.duo.uio.no/bitstream/handle/10852/45209/mymaster.pdf?sequence=7&isAllowed=y

[3] Guha, A., Saftoiu, C., & Krishnamurthi, S. (2015, October 3). The Essence of JavaScript. Brown University. Retrieved June 17, 2021, from https://arxiv.org/pdf/1510.00925.pdf

[4] Holland, B. (2016, January 11). What To Expect From JavaScript In 2016 - Frameworks [What To Expect From JavaScript In 2016 - Frameworks]. Telerik. Retrieved October 18, 2020, from https://www.telerik.com/blogs/what-to-expect-from-javascript-in-2016-frameworks

[5] Levlin, M. (2020). DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte. Åbo Akademi University. Retrieved June 17, 2021, from https://www.doria.fi/bitstream/handle/10024/177433/levlin_mattias.pdf?sequence=2

[6] Mirshokraie, S., Mesbah, A., & Pattabiraman, K. (2013, January). [Efficient JavaScript mutation testing]. University of British Columbia. https://blogs.ubc.ca/karthik/files/2013/01/mirshokraie-icst13.pdf

[7] Muyldermans, D. (2019, May 10). How does the virtual DOM compare to other DOM update mechanisms in JavaScript frameworks? University of Dublin. http://www.daisyms.com/THESIS.pdf

[8] Peterson, M. (2020, May 28). JavaScript DOM Manipulation Performance. Blekinge Institute of Technology. Retrieved June 17, 2021, from https://www.diva-portal.org/smash/get/diva2:1436661/FULLTEXT01.pdf

[9] Sen, R. (n.d.). A Strategic Analysis of Competition Between Open Source and Proprietary Software. Texas A&M University. https://econwpa.ub.uni-muenchen.de/econ-wp/io/removed/0510004.pdf

[10] Zou, Y. (n.d.). Coverage for Effective Testing of Dynamic Web Applications. Purdue University. https://

www.researchgate.net/profile/Zebao-Gao/publication/
266659080_Virtual_DOM_coverage_for_effective_testing_of_
dynamic_web_applications/links/55a9696a08aea9946721dc60/
Virtual-DOM-coverage-for-effective-testing-of-dynamic-web-applications.
pdfa