# Virtual DOM is pure overhead

Rich Harris

If you've used JavaScript frameworks in the last few years, you've probably heard the phrase 'the virtual DOM is fast', often said to mean that it's faster than the *real* DOM. It's a surprisingly resilient meme — for example people have asked how Svelte can be fast when it doesn't use a virtual DOM.

It's time to take a closer look.

## What is the virtual DOM?

In many frameworks, you build an app by creating `render()` functions, like this simple React component:

```
function HelloMessage(props) {
    return (
        <div className="greeting">
            Hello {props.name}
        </div>
    );
}
```

You can do the same thing without JSX. . .

```
function HelloMessage(props) {
    return React.createElement(
        'div',
        { className: 'greeting' },
        'Hello ',
        props.name
    );
}
```

. . . but the result is the same — an object representing how the page should now look. That object is the virtual DOM. Every time your app's state updates (for example when the `name` prop changes), you create a new one. The framework's job is to *reconcile* the new one against the old one, to figure out what changes are necessary and apply them to the real DOM.

## How did the meme start?

Misunderstood claims about virtual DOM performance date back to the launch of React. In Rethinking Best Practices, a seminal 2013 talk by former React core team member Pete Hunt, we learned the following:

> This is actually extremely fast, primarily because most DOM operations tend to be slow. There's been a lot of performance work on the DOM, but most DOM operations tend to drop frames.

Screenshot from Rethinking Best Practices at JSConfEU 2013

But hang on a minute! The virtual DOM operations are *in addition to* the eventual operations on the real DOM. The only way it could be faster is if we were comparing it to a less efficient framework (there were plenty to go around back in 2013!), or arguing against a straw man — that the alternative is to do something no-one actually does:

```
onEveryStateChange(() => {
    document.body.innerHTML = renderMyApp();
});
```

Pete clarifies soon after…

> React is not magic. Just like you can drop into assembler with C and beat the C compiler, you can drop into raw DOM operations and DOM API calls and beat React if you wanted to. However, using C or Java or JavaScript is an order of magnitude performance improvement because you don't have to worry…about the specifics of the platform. With React you can build applications without even thinking about performance and the default state is fast.

…but that's not the part that stuck.

## So… is the virtual DOM *slow*?

Not exactly. It's more like 'the virtual DOM is usually fast enough', but with certain caveats.

The original promise of React was that you could re-render your entire app on every single state change without worrying about performance. In practice, I don't think that's turned out to be accurate. If it was, there'd be no need for optimisations like `shouldComponentUpdate` (which is a way of telling React when it can safely skip a component).

Even with `shouldComponentUpdate`, updating your entire app's virtual DOM in one go is a lot of work. A while back, the React team introduced something called React Fiber which allows the update to be broken into smaller chunks. This means (among other things) that updates don't block the main thread for long periods of time, though it doesn't reduce the total amount of work or the time an update takes.

## Where does the overhead come from?

Most obviously, diffing isn't free. You can't apply changes to the real DOM without first comparing the new virtual DOM with the previous snapshot. To take the earlier `HelloMessage` example, suppose the `name` prop changed from 'world' to 'everybody'.

1. Both snapshots contain a single element. In both cases it's a `<div>`, which means we can keep the same DOM node
2. We enumerate all the attributes on the old `<div>` and the new one to see if any need to be changed, added or removed. In both cases we have a single attribute — a `className` with a value of `"greeting"`
3. Descending into the element, we see that the text has changed, so we'll need to update the real DOM

Of these three steps, only the third has value in this case, since — as is the case in the vast majority of updates — the basic structure of the app is unchanged. It would be much more efficient if we could skip straight to step 3:

```
if (changed.name) {
    text.data = name;
}
```

(This is almost exactly the update code that Svelte generates. Unlike traditional UI frameworks, Svelte is a compiler that knows at *build time* how things could change in your app, rather than waiting to do the work at *run time*.)

## It's not just the diffing though

The diffing algorithms used by React and other virtual DOM frameworks are fast. Arguably, the greater overhead is in the components themselves. You wouldn't write code like this…

```
function StrawManComponent(props) {
    const value = expensivelyCalculateValue(props.foo);

    return (
        <p>the value is {value}</p>
    );
}
```

…because you'd be carelessly recalculating `value` on every update, regardless of whether `props.foo` had changed. But it's extremely common to do unnecessary computation and allocation in ways that seem much more benign:

```
function MoreRealisticComponent(props) {
    const [selected, setSelected] = useState(null);

    return (
        <div>
```

```
            <p>Selected {selected ? selected.name : 'nothing'}</p>

            <ul>
                {props.items.map(item =>
                    <li>
                        <button onClick={() => setSelected(item)}>
                            {item.name}
                        </button>
                    </li>
                )}
            </ul>
        </div>
    );
}
```

Here, we're generating a new array of virtual `<li>` elements — each with their own inline event handler — on every state change, regardless of whether `props.items` has changed. Unless you're unhealthily obsessed with performance, you're not going to optimise that. There's no point. It's plenty fast enough. But you know what would be even faster? *Not doing that.*

React Hooks doubles down on defaulting to doing unnecessary work, with predictable results.

The danger of defaulting to doing unnecessary work, even if that work is trivial, is that your app will eventually succumb to 'death by a thousand cuts' with no clear bottleneck to aim at once it's time to optimise.

Svelte is explicitly designed to prevent you from ending up in that situation.

## Why do frameworks use the virtual DOM then?

It's important to understand that virtual DOM *isn't a feature*. It's a means to an end, the end being declarative, state-driven UI development. Virtual DOM is valuable because it allows you to build apps without thinking about state transitions, with performance that is *generally good enough*. That means less buggy code, and more time spent on creative tasks instead of tedious ones.

But it turns out that we can achieve a similar programming model without using virtual DOM — and that's where Svelte comes in.