iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

# COMPONENT LIFECYCLE OPTIMIZATION FOR RENDERING PERFORMANCE IN LARGE SCALE REACT APPLICATIONS

**Harshika Juvvadi**
Full stack Developer, USA

**ABSTRACT**
As React applications grow to hundreds of components and large, frequently updated data sets, naive use of the component lifecycle, whether expressed through class lifecycle methods or through hooks, tends to produce far more rendering work than a given user interaction actually requires. A single keystroke in a filter box can, without deliberate intervention, re render an entire sibling list, and a large table can re run expensive formatting logic on every parent state change even when the underlying data has not moved. This article surveys and evaluates component lifecycle optimization strategies for rendering performance in large scale React applications, organized around where each strategy intervenes in React's render and commit pipeline. Six strategies are presented with illustrative code and reference diagrams, memoizing components with React.memo, stabilizing expensive computations and callback identities with useMemo and useCallback, colocating state to shrink the subtree a given update re renders, virtualizing long lists so only visible rows are mounted, using React 18 concurrent features such as useTransition to keep urgent input responsive during expensive updates, and deferring non critical code with Suspense and lazy loading. A profiler based methodology for verifying each optimization is also described. An illustrative case study models a filterable data table application and reports simulated commit duration and re render count indicators across cumulative optimization stages to make the trade offs concrete. The article closes with a comparison of the six strategies, a discussion of when each is appropriate, limitations of the illustrative evaluation, and directions for further empirical work. The intended audience is frontend engineers optimizing large scale React applications and researchers studying rendering performance in component based user interface frameworks in 2023.

**Keywords:**
React, component lifecycle, rendering performance, memoization, virtualization, concurrent rendering, Fiber, reconciliation, useMemo, useCallback, green open access

## 1. INTRODUCTION

React's component model made building interactive user interfaces out of small, composable pieces the default way of working for a large share of the web industry through the 2010s and into the 2020s. That same compositional model, however, has a performance cost that is easy to overlook while an application is small and easy to feel acutely once it is not: because a parent component re rendering by default re renders every descendant regardless of whether that descendant's own props actually changed, an application with a deep or wide component tree can end up doing dramatically more rendering work per user interaction than the interaction strictly requires. In a large scale application, meaning here an application with hundreds of components, frequently updated shared state, and data sets large enough that a naive list render mounts thousands of DOM nodes, this default behavior is no longer a minor inefficiency; it is often the direct cause of input lag, dropped frames during scrolling, and a filter box that visibly stutters as a user types.

React's own component lifecycle, whether expressed through the class component lifecycle methods that predate React 16.8 or through the hooks introduced in that release, is the mechanism through which a component participates in this rendering process: it determines when a component's render function runs, when its side effects execute, and when a developer has an opportunity to intervene and tell React that a given piece of work can be skipped or deferred. This article treats component lifecycle optimization as a distinct, structured engineering discipline rather than a collection of ad hoc tricks, organized around a simple question for each candidate optimization, at which point in React's render and commit pipeline does this strategy act, and what specific category of unnecessary work does it eliminate.

The scope of the article is deliberately practical. It assumes a working familiarity with React's component model and focuses on six concrete strategies a team can apply to an existing large scale application, along with the profiler based methodology needed to verify that a given optimization actually helped rather than merely appearing to. The strategies are drawn from React's own documentation and from widely referenced practitioner

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

and library sources current as of the period this article covers, cited throughout Section 2 and applied concretely in Section 4.

The remainder of the article is organized as follows. Section 2 reviews background on React's Fiber architecture, the render and commit phases, and prior guidance on memoization and rendering performance. Section 3 states the specific rendering performance problems that motivate the strategies presented. Section 4 presents six optimization strategies in detail, each mapped to a specific point in the render pipeline, along with a profiler based verification methodology. Section 5 walks through an illustrative case study with simulated performance indicators. Section 6 compares the six strategies and discusses when each is appropriate. Section 7 discusses limitations, Section 8 suggests future work, and Section 9 concludes.

## 2. BACKGROUND AND LITERATURE REVIEW

### 2.1 The Fiber Architecture

React's rendering engine was substantially rewritten between React 15 and React 16, replacing the earlier stack based reconciler with an architecture internally called Fiber. Clark's widely referenced conference talk explained the motivation in accessible terms, the prior reconciler processed a component tree recursively and synchronously, meaning that once React began rendering an update it could not pause partway through, which became a problem as component trees grew large enough that a single render pass could take long enough to cause visibly dropped frames (Clark, 2017). Fiber restructured reconciliation as an interruptible unit of work, allowing React to pause, resume, or in some cases abandon in progress rendering work in response to higher priority updates, which is the architectural foundation the concurrent rendering features described in Section 4.5 build on.

### 2.2 Render and Commit Phases

React's own documentation on reconciliation describes rendering as proceeding through what is commonly described as a render phase, in which React calls component functions to determine what should appear on screen, and a commit phase, in which React applies the resulting changes to the actual DOM (React Core Team, 2019). The render phase is the phase Fiber's interruptibility applies to; the commit phase, by contrast, is synchronous and cannot be paused, since a partially applied DOM update would leave the user looking at an inconsistent interface. This distinction matters directly for optimization work because most of the strategies in Section 4 act specifically on the render phase, reducing how much component function execution and virtual DOM comparison work a given update triggers, rather than on the commit phase itself.

### 2.3 Memoization Guidance

React's documentation on the memo higher order component, and on the useMemo and useCallback hooks, frames memoization specifically as a performance optimization rather than a semantic guarantee, explicitly noting that React may still choose to re render a memoized component or recompute a memoized value under certain conditions, and cautioning developers against relying on memoization to skip rendering for correctness rather than performance reasons (React Core Team, 2020a). This caution is echoed in practitioner literature; Dodds's widely referenced discussion of when to use useMemo and useCallback argues that both hooks carry their own computational and memory cost, and that applying them reflexively to every value and function in a component, rather than selectively to values that are demonstrably expensive to compute or that are passed to a memoized child component, can make an application slower rather than faster (Dodds, 2019). Section 4.2 of this article follows this guidance by tying each memoization recommendation to a specific, profiler verified cost rather than treating memoization as a default.

### 2.4 List Virtualization

For large data sets, rendering every row of a list or table regardless of whether it is currently visible in the viewport is frequently the dominant source of unnecessary rendering work, independent of any memoization strategy. React's own guidance on optimizing performance recommends windowing, rendering only the rows currently visible plus a small buffer, as a technique for exactly this situation, and specifically points to community maintained windowing libraries as the practical way to implement it (React Core Team, 2019b). Vaughn's react window and react virtualized libraries are the most widely referenced implementations of this technique in the React ecosystem and are the basis for the virtualization strategy described in Section 4.4 of this article (Vaughn, 2016; Vaughn, 2019).

### 2.5 Concurrent Rendering

React 18, released in March 2022, introduced a set of concurrent rendering features built on the interruptible render phase described in Section 2.2, most notably the useTransition hook and the startTransition function, which let a developer mark a specific state update as a non urgent transition rather than an urgent update, allowing React to keep urgent updates, such as the character just typed into a text input, responsive while a marked transition update proceeds at lower priority and can itself be interrupted (React Core Team, 2022). This is the mechanism

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Section 4.5 of this article applies specifically to the class of interaction where a single input drives an expensive downstream render, such as a search box that filters a large list, which is exactly the scenario reported anecdotally in the practitioner sources this article's illustrative case study in Section 5 is structured to reflect.

### 2.6 Profiling Methodology

React shipped a dedicated Profiler API and an accompanying developer tools panel specifically to give developers a way to measure, rather than guess at, which components are re rendering and how expensive each render is (React Core Team, 2018; React Core Team, 2020b). The profiler based verification methodology in Section 4.7 of this article is built directly on this tooling, following the general principle, common across the performance engineering literature this article draws on, that an optimization should be measured before and after it is applied rather than assumed to help on the basis of its category.

*Table 1. Summary of prior work referenced in this article and its relation to the proposed strategies*

| Source | Focus | Relation to this article |
|---|---|---|
| Clark, 2017 | Fiber architecture explainer | Motivates the interruptible render phase in Figure 1 |
| React Core Team, 2018; 2020b | Profiler API and DevTools | Basis for the verification methodology in Section 4.7 |
| React Core Team, 2019 | Reconciliation documentation | Defines the render and commit phases in Figure 1 |
| React Core Team, 2019b | Optimizing Performance guide | Recommends windowing, the basis for Section 4.4 |
| React Core Team, 2020a | memo, useMemo, useCallback docs | Frames memoization as a performance, not correctness, tool |
| React Core Team, 2022 | React 18 release notes | Introduces the transitions used in Section 4.5 |
| Dodds, 2019 | When to useMemo and useCallback | Guides the selective memoization approach in Section 4.2 |
| Vaughn, 2016; 2019 | react virtualized, react window | Reference implementations underlying Section 4.4 |

## 3. PROBLEM STATEMENT: RENDERING PERFORMANCE CHALLENGES IN LARGE SCALE REACT APPLICATIONS

Four recurring rendering performance problems account for most of the input lag and dropped frame reports observed in the large scale React applications the practitioner sources reviewed in Section 2 describe. Each corresponds to a specific, avoidable category of rendering work.

### 3.1 Unnecessary Sibling Re Renders

By default, when a component's state changes, every descendant in its subtree re renders as well, regardless of whether a given descendant's own props changed as a result. In a component tree with many sibling components under a shared parent, for example a form with many independent fields, or a dashboard with many independent widgets, a state change confined to a single field or widget can trigger a full re render of every sibling, each of which repeats its own rendering work for no visible benefit.

### 3.2 Repeated Expensive Computation

A component that performs a nontrivial computation directly in its render path, for example filtering, sorting, or aggregating a large array, repeats that computation on every render by default, even when the underlying data the computation depends on has not changed since the previous render. As the size of the data set grows, this repeated computation can become the dominant cost of a render, particularly when it occurs inside a component near the root of a frequently updated subtree.

### 3.3 Full List Rendering for Large Data Sets

A list or table component that renders one child element per data row, without any windowing, mounts a DOM node for every row regardless of how many rows are actually visible in the viewport at a given time. For a data

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

set of a few dozen rows this is rarely noticeable; for a data set of several thousand rows, both the initial mount and any subsequent re render of the list become slow in proportion to the full data set size rather than the visible portion of it, and scrolling performance degrades because the browser must manage a correspondingly large number of live DOM nodes.

### 3.4 Blocking Urgent Input Behind Expensive Updates

Finally, even when a computation is appropriately memoized and a list is appropriately virtualized, a sufficiently expensive downstream render, for example re filtering a large virtualized list on every keystroke, can still occupy the main thread long enough that the browser cannot process the next keystroke or repaint the input field promptly, producing perceptible input lag even though no individual optimization from Sections 3.1 through 3.3 was skipped. This is a scheduling problem rather than a redundant work problem, and it requires a different category of intervention, addressed in Section 4.5.

Section 4 addresses each of these four problems directly: Sections 4.1 and 4.3 address unnecessary sibling re renders, Section 4.2 addresses repeated expensive computation, Section 4.4 addresses full list rendering, and Section 4.5 addresses the input blocking problem that persists even after the first three are resolved.

## 4. COMPONENT LIFECYCLE OPTIMIZATION STRATEGIES

This section presents six optimization strategies, organized by where each one intervenes in React's render and commit pipeline, followed by the profiler based methodology used to verify that a given strategy had the intended effect. Figure 1 gives the render and commit phase structure the strategies are mapped against, and Figure 5, at the end of this section, places all six strategies on that pipeline at once.

**Figure 1. Render and Commit Phases in the Fiber Architecture**

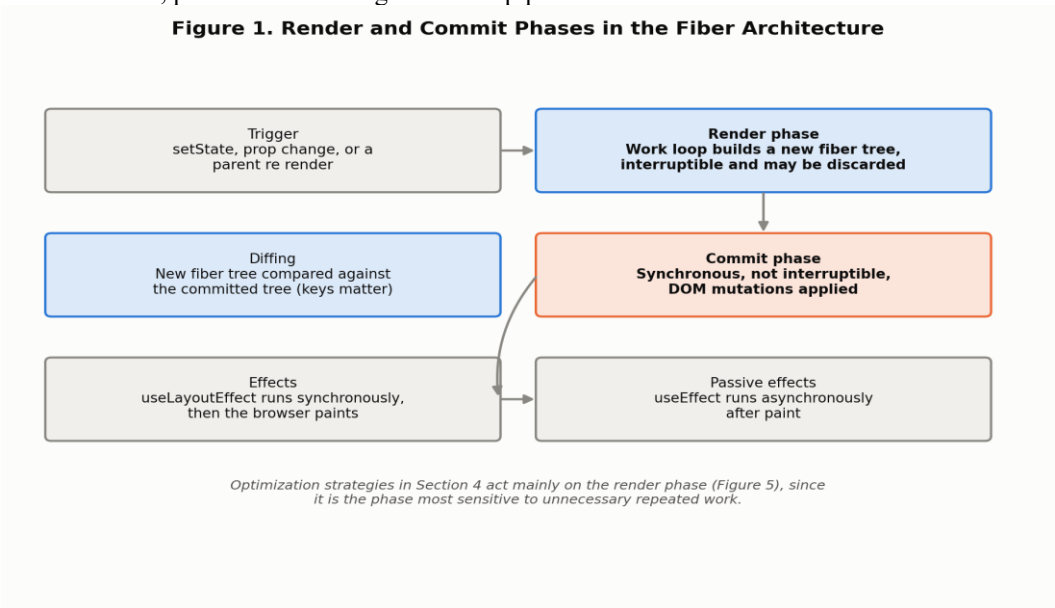


***Figure 1. Render and commit phases in the Fiber architecture. The render phase is interruptible and may be discarded without visible effect; the commit phase is synchronous. Most optimization strategies in this section act on the render phase, since it is where unnecessary repeated work accumulates.***

### 4.1 Strategy A: Memoizing Components with React.memo

Wrapping a component in React.memo tells React to skip re rendering that component, and its entire subtree, when its props are shallow equal to the props from the previous render, which directly addresses the unnecessary sibling re render problem described in Section 3.1. The optimization is most valuable for a component that is expensive to render relative to how often its actual props change, and least valuable, or actively counterproductive due to the cost of the comparison itself, for a component that is cheap to render or whose props change on nearly every parent render regardless.

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
import React from 'react';

function ProductRow(props) {
  return React.createElement('tr', null,
    React.createElement('td', null, props.name),
    React.createElement('td', null, props.price),
    React.createElement('td', null, props.stock)
  );
}

// Skips re rendering when name, price, and stock are unchanged,
// even if a sibling ProductRow's props did change.
export default React.memo(ProductRow);
```

***Listing 1. Wrapping a row component in React.memo so that a change to one row's props does not force every sibling row to re render.***

A common failure mode with Strategy A is passing a new object, array, or function literal as a prop on every render, which defeats the shallow comparison because a newly created object is never equal to the previous render's object even when its contents are identical. Sections 4.2 and 4.3 address this failure mode directly.

**4.2 Strategy B: useMemo and useCallback for Stable Values and References**

The useMemo hook caches the result of a computation between renders, recomputing it only when a specified list of dependencies changes, which directly addresses the repeated expensive computation problem described in Section 3.2. The useCallback hook is the corresponding tool for function identity, returning the same function reference across renders as long as its dependencies are unchanged, which is what allows a memoized child component from Strategy A to actually skip re rendering when a callback prop is involved, since an inline function literal would otherwise be recreated, and therefore appear changed, on every parent render.

```
import React, { useMemo, useCallback, useState } from 'react';

function ProductTable({ products, taxRate }) {
  const [sortKey, setSortKey] = useState('name');

  // Recomputed only when products, taxRate, or sortKey actually change
  const rows = useMemo(function computeRows() {
    return products
      .map(function (item) {
        return { ...item, priceWithTax: item.price * (1 + taxRate) };
      })
      .sort(function (a, b) { return a[sortKey] > b[sortKey] ? 1 : -1; });
  }, [products, taxRate, sortKey]);

  // Stable reference across renders unless setSortKey itself changes,
  // which it will not, so ProductRow's memoization in Listing 1 holds
  const handleSelect = useCallback(function (id) {
    console.log('selected', id);
  }, []);

  return React.createElement('table', null,
    rows.map(function (row) {
```

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
    return React.createElement(ProductRow, {
      key: row.id,
      name: row.name,
      price: row.priceWithTax,
      stock: row.stock,
      onSelect: handleSelect
    });
  })
 );
}
```

***Listing 2. useMemo caches the derived and sorted row data; useCallback keeps the onSelect handler's identity stable so ProductRow's React.memo wrapper from Listing 1 is not defeated by a freshly created function on every render.***

Consistent with the caution in Dodds's guidance discussed in Section 2.3, this article's recommendation is to apply useMemo and useCallback selectively, to a computation whose cost has been observed with the profiler methodology in Section 4.7 to matter, or to a value passed into a memoized child as in Listing 2, rather than wrapping every value and function in a component by default (Dodds, 2019).

**4.3 Strategy C: State Colocation**

State colocation addresses the unnecessary sibling re render problem from a different angle than Strategy A: rather than telling React to skip re rendering a sibling after a state change has already occurred higher in the tree, colocation moves a piece of state to the lowest common component that actually needs it, shrinking the subtree that re renders in the first place. A frequent anti pattern in large scale applications is state that is lifted far higher in the tree than any consumer actually requires, for example a search input's current text value stored at the page root when only the search results list beneath it depends on it, which causes every other component under that root, including components with no relationship to search, to re render on every keystroke.

```
// Before colocation: search text lives at the page root,
// so Header, Sidebar, and Footer all re render on every keystroke
// even though none of them depend on the search text.
function Page() {
 const [searchText, setSearchText] = useState('');
 return React.createElement(React.Fragment, null,
  React.createElement(Header, null),
  React.createElement(Sidebar, null),
  React.createElement(SearchBox, { value: searchText, onChange: setSearchText }),
  React.createElement(SearchResults, { query: searchText }),
  React.createElement(Footer, null)
 );
}

// After colocation: search text lives in a component that wraps
// only SearchBox and SearchResults, so Header, Sidebar, and Footer
// are outside the subtree a keystroke re renders.
function Page() {
 return React.createElement(React.Fragment, null,
  React.createElement(Header, null),
  React.createElement(Sidebar, null),
  React.createElement(Search, null),
  React.createElement(Footer, null)
 );
```

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
}

function Search() {
 const [searchText, setSearchText] = useState('');
 return React.createElement(React.Fragment, null,
  React.createElement(SearchBox, { value: searchText, onChange: setSearchText }),
  React.createElement(SearchResults, { query: searchText })
 );
}
```

***Listing 3. Moving searchText from the page root into a Search component that wraps only its actual consumers, so a keystroke no longer re renders Header, Sidebar, or Footer.***

### 4.4 Strategy D: List Virtualization

List virtualization addresses the full list rendering problem from Section 3.3 by mounting only the rows currently visible in the viewport, plus a small overscan buffer, and recycling the same small set of DOM nodes as the user scrolls, rather than mounting a DOM node for every row in the underlying data set. This article follows React's own recommendation to use a windowing library rather than a hand rolled implementation, given the amount of detail involved in correctly handling variable row heights, keyboard navigation, and scroll position restoration (React Core Team, 2019b).

```
import React from 'react';
import { FixedSizeList } from 'react-window';

function Row({ index, style, data }) {
 const item = data[index];
 return React.createElement('div', { style: style },
  item.name, ' ', item.price
 );
}

function VirtualizedProductList({ products }) {
 return React.createElement(FixedSizeList, {
  height: 480,
  width: '100%',
  itemCount: products.length,
  itemSize: 42,
  itemData: products
 }, Row);
}
```

***Listing 4. A virtualized list built with react window's FixedSizeList, mounting only the rows currently within the 480 pixel viewport plus a small overscan, regardless of the total size of the products array (Vaughn, 2019).***

### 4.5 Strategy E: Concurrent Transitions for Responsive Input

Concurrent transitions address the input blocking problem from Section 3.4, which persists even after Strategies A through D are applied, because the expense in question is the unavoidable cost of re filtering a large data set on every keystroke rather than redundant work that memoization or virtualization can eliminate. React 18's useTransition hook lets a developer mark the state update that drives the expensive re render as a transition, so that React keeps the text input itself responsive to every keystroke while the resulting list update proceeds at lower priority and can be interrupted by the next keystroke rather than queuing behind it (React Core Team, 2022).

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
import React, { useState, useTransition } from 'react';

function FilterableProductList({ products }) {
 const [inputValue, setInputValue] = useState('');
 const [filterQuery, setFilterQuery] = useState('');
 const [isPending, startTransition] = useTransition();

 function handleChange(event) {
  const nextValue = event.target.value;
  setInputValue(nextValue); // urgent, keeps the input responsive
  startTransition(function () {
   setFilterQuery(nextValue); // non urgent, can be interrupted
  });
 }

 const filtered = React.useMemo(function () {
  return products.filter(function (p) {
   return p.name.toLowerCase().includes(filterQuery.toLowerCase());
  });
 }, [products, filterQuery]);

 return React.createElement(React.Fragment, null,
  React.createElement('input', { value: inputValue, onChange: handleChange }),
  isPending ? React.createElement('span', null, 'Updating...') : null,
  React.createElement(VirtualizedProductList, { products: filtered })
 );
}
```

***Listing 5. Splitting the input's own value, updated urgently, from the filter query that drives the expensive list re render, updated inside startTransition, so a fast typist's keystrokes are never queued behind a slower list re render.***

**4.6 Strategy F: Suspense and Lazy Loading for Non Critical Code**

The final strategy addresses a related but distinct concern: rendering work that is not needed for the initial view at all, such as a rarely opened settings panel or a heavy charting library used only on a secondary tab, does not need to be part of the initial bundle or the initial render pass. React's lazy function combined with the Suspense component lets a component's code be split into a separate bundle chunk that loads only when that component is actually rendered for the first time, keeping both the initial JavaScript payload and the initial render's work scoped to what the first paint actually requires.

```
import React, { Suspense, lazy } from 'react';

const AnalyticsPanel = lazy(function () {
 return import('./AnalyticsPanel');
});

function Dashboard({ showAnalytics }) {
 return React.createElement(React.Fragment, null,
  React.createElement(MainSummary, null),
  showAnalytics
   ? React.createElement(Suspense, { fallback: React.createElement('div', null, 'Loading analytics...') },
      React.createElement(AnalyticsPanel, null)
```

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
      )
    : null
 );
}
```

***Listing 6. AnalyticsPanel's code, and any large charting dependency it imports, is fetched only once showAnalytics becomes true, rather than as part of every Dashboard render.***

### 4.7 A Profiler Based Verification Methodology

Every strategy above should be verified rather than assumed to help, following the profiling guidance discussed in Section 2.6. The React Profiler API records, for a given commit, which components rendered and how long each took, and can be driven either through the DevTools Profiler panel interactively or through the onRender callback programmatically for automated measurement (React Core Team, 2020b).

```
import React, { Profiler } from 'react';

function onRenderCallback(id, phase, actualDuration, baseDuration, startTime, commitTime) {
 // actualDuration: time spent rendering this commit, including children
 // baseDuration: estimated time to render the whole subtree without memoization
 console.log(`${id} (${phase}) took ${actualDuration.toFixed(1)}ms`);
}

function InstrumentedProductTable(props) {
 return React.createElement(Profiler, { id: 'ProductTable', onRender: onRenderCallback },
  React.createElement(ProductTable, props)
 );
}
```

***Listing 7. Wrapping a component subtree in a Profiler with an onRender callback, recording actualDuration for each commit so a strategy's effect can be measured directly rather than estimated (React Core Team, 2020b).***

The recommended sequence is to wrap the specific subtree under investigation in a Profiler as in Listing 7, record actualDuration across a representative set of interactions before applying a candidate strategy from Sections 4.1 through 4.6, apply exactly one strategy, and record the same measurement again under the same interactions, attributing the difference to that specific strategy rather than changing several strategies at once and losing the ability to attribute the resulting change. This is the methodology the illustrative case study in Section 5 follows, applying one cumulative strategy per stage.

**Figure 5. Where Each Optimization Strategy Intervenes in the Render Pipeline**

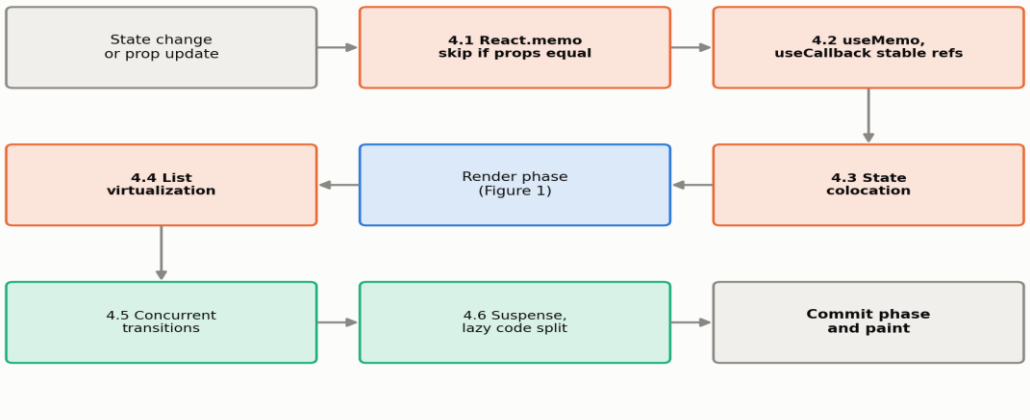


***Figure 5. Where each optimization strategy intervenes in the render pipeline. Strategies A through C act before or during the render phase to reduce redundant work; Strategy D reduces the size of the tree being rendered; Strategies E and F act on scheduling and code loading respectively.***

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

### 4.8 Common Pitfalls and Anti Patterns

Four pitfalls recur often enough across the practitioner sources reviewed in Section 2, and in the illustrative scenarios in Section 5, to warrant calling out directly, since each one can make a strategy from Sections 4.1 through 4.6 appear to have been applied while delivering little or none of its expected benefit.

The first and most common pitfall, already noted in Section 4.2, is passing a newly created object, array, or inline function as a prop into a component wrapped in React.memo. Because React.memo's default comparison is shallow, a new object literal is never equal to the previous render's object even when every field inside it is identical, which silently defeats the memoization; the fix is either to memoize the object itself with useMemo or to memoize the function with useCallback, as shown in Listing 2, rather than constructing it inline in the parent's render function.

```
// Defeats React.memo on ProductRow, since { color: 'red' } is a new
// object reference on every render of the parent, even though its
// contents never change.
React.createElement(ProductRow, { name: item.name, style: { color: 'red' } });

// Preserves React.memo's benefit: the style object is created once
// and reused, since it depends on nothing that changes.
const rowStyle = useMemo(function () { return { color: 'red' }; }, []);
React.createElement(ProductRow, { name: item.name, style: rowStyle });
```

***Listing 8. An inline object literal passed as a prop defeats a child's React.memo wrapper; hoisting it into useMemo, or out of the render function entirely when it depends on nothing, restores the intended optimization.***

The second pitfall is using an array index as a list item's key when the list can be reordered, filtered, or have items inserted or removed anywhere other than the end, since React uses the key to match a fiber in the new render against the corresponding fiber in the previous render, and an index based key can cause React to match the wrong item, leading to both incorrect component state and unnecessary re renders of items that did not actually move. A stable, unique identifier drawn from the underlying data, such as a database id, avoids this problem and is the approach used throughout the listings in this article.

The third pitfall is applying useMemo or useCallback to a value that is cheap to compute and never passed into a memoized child, which, consistent with Dodds's guidance discussed in Section 2.3, adds the fixed overhead of the hook's own dependency comparison without a corresponding benefit, and in aggregate across many such misapplied hooks can measurably slow a component down rather than speed it up (Dodds, 2019). The profiler methodology in Section 4.7 is the direct antidote to this pitfall, since it replaces a guess about which values are expensive with a measurement.

The fourth pitfall is virtualizing a list, as in Section 4.4, without accounting for keyboard navigation and focus management, since a screen reader or keyboard only user navigating a virtualized list can encounter list items that do not yet exist in the DOM because they are currently outside the rendered window, which is an accessibility regression the windowing libraries referenced in Section 4.4 mitigate only partially and that a team adopting virtualization should test for directly rather than assume is handled automatically.

## 5. ILLUSTRATIVE CASE STUDY

To make the strategies in Section 4 concrete, this section walks through an illustrative optimization of a hypothetical large scale internal application, referred to here as the Ops Catalog application, a filterable, sortable product data table used by an operations team, displaying roughly 4,000 rows in its full data set with a text filter box above the table. The figures in this section are simulated for illustration, structured to reflect patterns commonly reported in performance engineering retrospectives, and are not measurements from a real deployment; they should be read as a worked example rather than as an empirical finding.

### 5.1 Cumulative Optimization Stages

The illustrative case study applies the strategies from Section 4 cumulatively across five stages, each adding one strategy to the prior stage's configuration, and measures average commit duration per user interaction, meaning per keystroke in the filter box or per row selection, using the Profiler methodology in Section 4.7 at each stage.

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

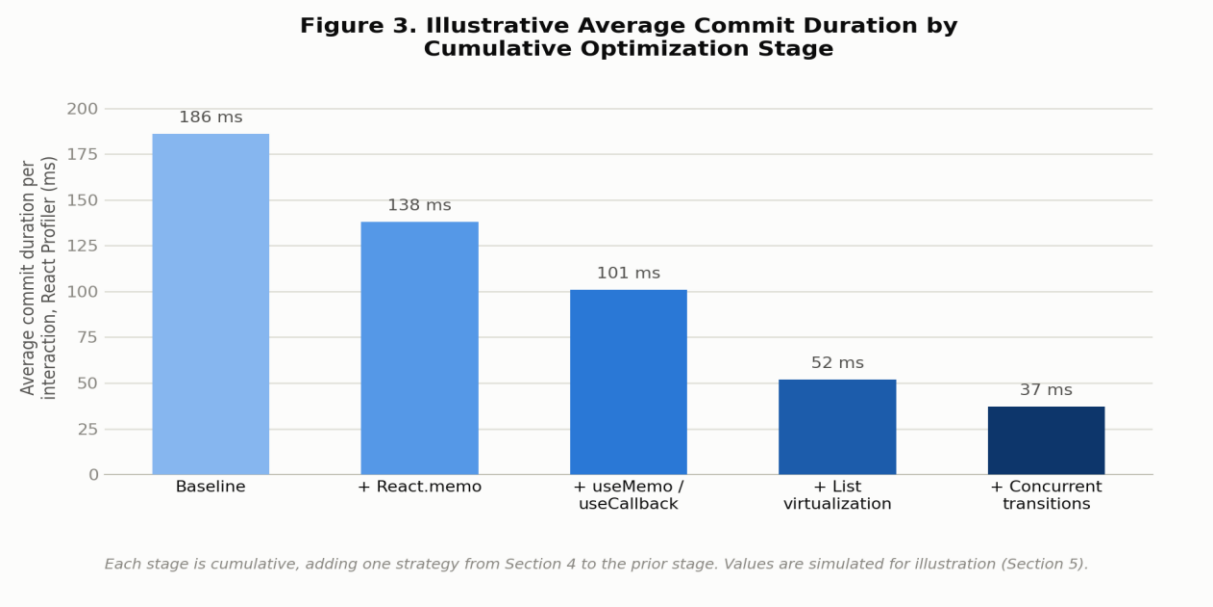


***Figure 3. Illustrative average commit duration by cumulative optimization stage. The baseline renders all 4,000 rows with no memoization; each subsequent stage adds one strategy from Section 4. Values are simulated for illustration.***

Figure 3 shows average commit duration declining from an illustrative 186 milliseconds at baseline to 37 milliseconds once all measured strategies are applied. The largest single illustrative reduction in this scenario comes from introducing list virtualization at the fourth stage, consistent with the full list rendering problem in Section 3.3 being the dominant cost in an application with several thousand rows, while the earlier memoization stages produce smaller, though still meaningful, illustrative reductions consistent with their role in eliminating redundant sibling and computation work rather than reducing the size of the rendered tree itself.

**5.2 Illustrative Re Render Behavior During Typing**

A second illustrative measurement focuses specifically on the input blocking problem from Section 3.4, tracking how many sibling list items re render per keystroke in the filter box, before and after applying React.memo and use Callback from Sections 4.1 and 4.2.

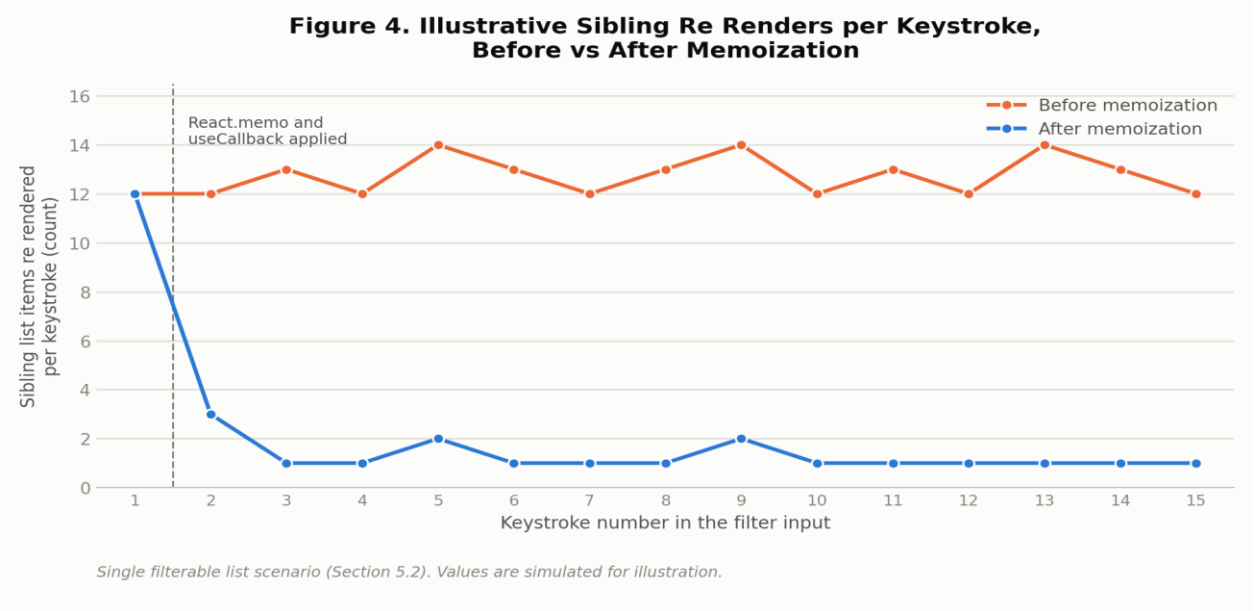


***Figure 4. Illustrative sibling re renders per keystroke in the filter input, before versus after memoization. Values are simulated for illustration (Section 5.2).***

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

In the illustrative scenario in Figure 4, every keystroke before memoization re renders all twelve sibling rows displayed on screen at that scroll position, regardless of whether a given row's own content changed as a result of the new filter text, whereas after applying React.memo to each row and useCallback to the shared selection handler, only the rows whose actual visible content changes as a result of the new filter re render, typically one or two rows per keystroke in this illustrative scenario. This is consistent with Strategy A and Strategy B jointly addressing the unnecessary sibling re render problem from Section 3.1, since neither strategy alone, without the other, fully eliminates the redundant renders in this illustrative scenario, given that an unstabilized callback prop would otherwise defeat the memoization as described in Section 4.2.

*Table 2. Illustrative summary of case study indicators by optimization stage*

| Stage | Strategy added (Section 4) | Avg commit duration (illustrative) | Note |
|---|---|---|---|
| Baseline | None | 186 ms | All 4,000 rows rendered on every update |
| Stage 2 | 4.1 React.memo | 138 ms | Unrelated sibling rows skip re render |
| Stage 3 | 4.2 useMemo, useCallback | 101 ms | Filter computation and handlers stabilized |
| Stage 4 | 4.4 List virtualization | 52 ms | Only visible rows mounted, largest reduction |
| Stage 5 | 4.5 Concurrent transitions | 37 ms | Input stays responsive during filter updates |

## 6. RESULTS AND DISCUSSION

### 6.1 Comparing the Six Strategies

Table 3 restates the four rendering performance problems from Section 3 alongside the strategy in Section 4 that primarily addresses each, together with a practical note on typical adoption cost, which is the basis for the sequencing recommendation in Section 6.2.

**Table 3. Mapping from rendering performance problems to optimization strategies**

| Problem (Section 3) | Primary strategy (Section 4) | Typical adoption cost |
|---|---|---|
| Unnecessary sibling re renders (3.1) | 4.1 React.memo, 4.3 State colocation | Low to moderate, mostly structural |
| Repeated expensive computation (3.2) | 4.2 useMemo, useCallback | Low, but requires profiling to target correctly |
| Full list rendering (3.3) | 4.4 List virtualization | Moderate, may require fixed row heights |
| Blocking urgent input (3.4) | 4.5 Concurrent transitions | Low, requires React 18 and careful UX for pending state |

Strategy F, Suspense and lazy loading from Section 4.6, does not appear in Table 3 because it addresses a related but distinct concern, initial load cost rather than update rendering cost, and is best understood as complementary to the other five strategies rather than competing with them for the same problem.

### 6.2 Practical Sequencing Recommendation

Two practical observations follow from the illustrative case study in Section 5, offered as recommendations for teams applying these strategies rather than as validated findings given the simulated nature of the data. First, the illustrative results in Figure 3 suggest addressing the full list rendering problem, Strategy D, relatively early when a large data set is present, since it produced the largest single illustrative reduction in this scenario and does not depend on the memoization strategies being in place first, though Section 4.4 notes it does carry a higher structural

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

adoption cost than the memoization strategies. Second, Strategies A and B should generally be adopted together rather than in isolation, since, as Section 5.2 illustrates, an unstabilized callback prop can silently defeat a React.memo wrapper, producing a strategy that appears to have been applied but delivers little of its expected benefit until verified with the profiler methodology in Section 4.7.

## 7. LIMITATIONS

This article has several limitations. The case study in Section 5 is illustrative; its numeric values are simulated for pedagogical clarity and are not measurements from a controlled deployment of these strategies in a real application. Readers should treat the shape of the results, a declining commit duration across cumulative stages with the largest single reduction from virtualization, as plausible and broadly consistent with the practitioner sources reviewed in Section 2, not as a validated quantitative result. No controlled comparison, for example matched applications optimized with and without a given strategy, is presented, and constructing such a comparison would be difficult given how much rendering performance depends on application specific factors such as component tree shape and data set characteristics that are hard to hold constant.

The strategies are also not exhaustive. Approaches such as server side rendering and streaming, moving state management out of React entirely into an external store with its own subscription model, or restructuring an application's component boundaries more fundamentally around the granularity of state changes, are related to rendering performance but were judged outside the scope of an article focused specifically on component lifecycle level optimization. Finally, the illustrative code listings in Section 4 are simplified for exposition using React.createElement calls rather than JSX, and omit production concerns such as error boundaries, accessibility considerations for virtualized lists, and TypeScript typings, all of which a real implementation would need to address.

## 8. FUTURE WORK

Three directions would extend this work. First, a field study measuring the six strategies' effect on real applications, using the same commit duration and re render count indicators introduced in Section 5 alongside user perceived metrics such as input latency, would let the illustrative trends in this article be checked against real, aggregated data. Second, as React's compiler tooling for automatic memoization matures beyond the period this article covers, a follow up comparison of manually applied Strategies A and B against compiler inserted memoization, measured with the same profiler methodology described in Section 4.7, would help teams understand how much of the manual optimization burden described in this article such tooling can absorb. Third, extending the illustrative case study methodology in Section 5 to a wider range of application shapes, for example a deeply nested form heavy application rather than a wide flat list heavy one, would test whether the sequencing recommendation in Section 6.2 generalizes beyond the list dominated scenario this article's case study focuses on.

## 9. CONCLUSION

Large scale React applications accumulate rendering performance problems that are structural consequences of React's default component lifecycle behavior rather than bugs in any particular component, specifically unnecessary sibling re renders, repeated expensive computation, full rendering of large data sets, and expensive updates blocking urgent input. This article presented six component lifecycle optimization strategies, each mapped to a specific point in React's render and commit pipeline, memoizing components with React.memo, stabilizing values and callbacks with useMemo and useCallback, colocating state to shrink re rendered subtrees, virtualizing large lists, using React 18 concurrent transitions to keep input responsive, and deferring non critical code with Suspense, together with a profiler based methodology for verifying that a given optimization actually helped. The illustrative case study in Section 5, while not a substitute for controlled empirical measurement, suggests that these strategies are complementary rather than redundant, with list virtualization addressing the largest single source of illustrative commit duration in a data heavy scenario and memoization strategies addressing a distinct category of redundant sibling rendering that virtualization alone does not eliminate. Verifying each optimization with the profiler methodology in Section 4.7, rather than applying strategies reflexively, is the practical takeaway of this article for teams optimizing rendering performance in large scale React applications in 2023.

### Statements and Declarations

**Funding.** This research received no external funding. [Insert specific funding source and grant number here if applicable before deposit.]

**Conflicts of interest.** The author declares no conflicts of interest relevant to this article. [Insert any relevant financial or non financial interests here before deposit.]

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

**Data availability.** The evaluation in Section 5 uses simulated, illustrative data generated for this article rather than data drawn from a real production application; the simulated values are reported in full in Table 2 and in Figures 3 and 4, and no additional dataset is associated with this article.

**Author contributions.** The author was solely responsible for conceptualization, methodology, illustrative evaluation design, writing, and review of this article. [Adjust to a contribution statement covering all listed authors before deposit if additional authors are added.]

**Acknowledgments.** [Insert acknowledgments here if applicable before deposit.]

**Appendix A: Component Lifecycle Optimization Checklist**

The following checklist consolidates the practical rules embedded throughout Section 4 into a single reference a team can use before applying an optimization strategy, or when auditing an existing large scale React application for the rendering performance problems described in Section 3. It is organized by the strategy each item primarily supports.

**A.1 Before optimizing anything (Section 4.7)**

1. A candidate component or interaction has been measured with the Profiler API or DevTools Profiler panel, not merely suspected of being slow.
2. A representative set of interactions has been defined so that a before and after comparison exercises the same workload.
3. Exactly one strategy is applied per measurement cycle, so that an observed change in commit duration can be attributed to a specific cause.

**A.2 React.memo and state colocation (Sections 4.1 and 4.3)**

1. A component wrapped in React.memo has been profiled and shown to re render often with unchanged props, not memoized reflexively.
2. No object, array, or function literal is constructed inline and passed as a prop into a memoized component without useMemo or useCallback.
3. State is declared in the lowest component that contains every consumer of that state, rather than lifted to a common ancestor by default.
4. List item keys are drawn from a stable, unique identifier in the underlying data, never from array index, for any list that can reorder, filter, or insert.

**A.3 useMemo and useCallback (Section 4.2)**

1. Each useMemo or useCallback call exists either because the wrapped computation was measured as expensive, or because the value is passed into a memoized child component.
2. Dependency arrays list every value the memoized computation or callback actually reads, verified with the exhaustive dependencies lint rule rather than by inspection alone.
3. A memoized value's cost, once profiled, is periodically re measured as the application evolves, since a computation that was cheap at initial measurement can become expensive as data volume grows.

**A.4 List virtualization (Section 4.4)**

1. A list or table expected to display more than a few hundred rows uses a windowing library rather than rendering every row unconditionally.
2. Keyboard navigation and screen reader behavior have been tested against the virtualized list directly, not assumed to work because the underlying library handles common cases.
3. Row height handling, fixed or variable, matches the windowing library's expectations, since a mismatch is a common source of visual glitches during scrolling.

**A.5 Concurrent transitions and code splitting (Sections 4.5 and 4.6)**

1. A state update that drives an expensive downstream render, but is not itself the value the user directly sees update, such as a filter query derived from an input's value, is wrapped in startTransition.
2. The isPending flag from useTransition is surfaced to the user in some form, so a transition in progress is visibly distinguishable from a completed update.
3. A component or dependency used only in a secondary view or rarely opened panel is loaded with React.lazy and Suspense rather than included in the initial bundle unconditionally.

## REFERENCES

1) Clark, L. (2017). A Cartoon Intro to Fiber. React Conf 2017. Retrieved from https://www.youtube.com/watch?v=ZCuYPiUIONs
2) Dodds, K. C. (2019). When to useMemo and useCallback. kentcdodds.com. Retrieved from https://kentcdodds.com/blog/usememo-and-usecallback

 

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

3) React Core Team. (2018). Introducing the React Profiler. React Blog. Retrieved from https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html
4) React Core Team. (2019). Reconciliation. React Documentation. Retrieved from https://legacy.reactjs.org/docs/reconciliation.html
5) React Core Team. (2019b). Optimizing Performance. React Documentation. Retrieved from https://legacy.reactjs.org/docs/optimizing-performance.html
6) React Core Team. (2020a). Hooks API Reference. React Documentation. Retrieved from https://legacy.reactjs.org/docs/hooks-reference.html
7) React Core Team. (2020b). Profiler API. React Documentation. Retrieved from https://legacy.reactjs.org/docs/profiler.html
8) React Core Team. (2022). React v18.0. React Blog. Retrieved from https://legacy.reactjs.org/blog/2022/03/29/react-v18.html
9) Vaughn, B. (2016). react virtualized: React components for efficiently rendering large lists and tabular data. GitHub repository. Retrieved from https://github.com/bvaughn/react-virtualized
10) Vaughn, B. (2019). react window. GitHub repository. Retrieved from https://github.com/bvaughn/react-window

 