ChatGPT

React Fiber Architecture – Advanced Overview

Introduction

React Fiber is a complete reimplementation of React's core reconciliation algorithm, developed after over two years of React team research [1]. Its primary mission is to make React *more suitable for advanced UI scenarios* (e.g. animation, layout, gestures) by breaking rendering work into incremental chunks across multiple frames [2]. In practice, Fiber enables React to *pause, resume, or reuse* work when new updates arrive, assign *priorities* to different updates, and introduce new concurrency primitives [3]. These changes allow React apps to remain highly responsive and smooth even under complex or time-consuming updates.

Fiber still follows React's core model where every update conceptually "re-renders" the entire app, but it **diffs** the new virtual tree against the previous one to compute minimal changes [4]. By decoupling *reconciliation* (the diff algorithm) from the *renderer* (DOM, native, etc.), Fiber shares the reconciler across platforms and focuses on new scheduling capabilities [5].

Goals

- **Incremental Rendering:** Split rendering work into small units spread across frames, improving animation smoothness and avoiding jank [2].
- **Scheduling and Prioritization:** Allow React to *pause* or *abort* work in progress, *reuse* work that hasn't changed, and handle updates with different priorities (e.g. user interactions vs background data) [3] [6].
- **Concurrency Support:** Introduce new primitives (such as *time-slicing* and *concurrent modes*) so multiple tasks can co-exist without blocking the UI [3] [7].

These goals address real-world performance concerns: animations can run smoothly, expensive updates can be deferred, and high-priority events (like user input) can interrupt long-running renders [8] [9]. In essence, Fiber is designed to keep interfaces responsive even under heavy workloads.

Core Concepts

- **Reconciliation:** React's diffing algorithm computes the minimal changes between two virtual trees. Even in Fiber, the *high-level reconciliation* strategy remains the same: entire trees are generated on each update, and React diffs the new tree against the old one [4]. The reconciler follows two main heuristics: different component types result in unrelated trees (so React will replace rather than diff them) and lists are diffed using stable keys [10]. Thus, Fiber retains React's fundamental model of tree diffing.

- **Reconciler vs Renderer:** React separates *calculating* what changed (reconciler) from *applying* those changes (renderer). Fiber reimplements the reconciler (the "virtual DOM" diff engine) but leaves actual rendering to platform-specific renderers (e.g. `react-dom` or React Native) [5]. This separation allows Fiber to change how updates are scheduled without altering how changes are painted to the screen.

- **Scheduling (Pull vs Push):** Traditional React (the "stack" reconciler) walks the tree in one go (push model), executing all updates in a single tick. In contrast, Fiber adopts a *pull-based* scheduling approach [11] [9]. This means React can decide *when* and *how much* of the update to compute. For example, if updates are too frequent (faster than the frame rate), Fiber can *batch* or *defer* them. If a user interaction (high priority) comes in, Fiber can interrupt lower-priority work, process the interaction first, then resume the rest [8] [9]. In short, scheduling enables *time-slicing*: React "pulls" work in increments, yielding control back to the browser when needed, instead of blindly processing the entire tree immediately.

- **Unit of Work – The Fiber:** At its heart, Fiber introduces the concept of a "unit of work." Instead of relying on the call stack alone, Fiber uses *Fiber objects* (in-memory stack frames) to represent pieces of the work to be done [12]. You can think of each Fiber as analogous to a function call frame: it captures the information needed to process a component (type, props, state) and produce output. By keeping these Fiber objects around, React can pause execution at any point, switch to other work, and resume later exactly where it left off [13] [14].

- **Call Stack Analogy:** Normally, a function call pushes a frame on the browser's call stack and runs to completion. For UIs, blocking the stack is problematic (e.g. long renders drop animation frames). Fiber effectively *reimplements* this call stack as a heap-based linked structure [13]. This lets React pause "stack frames" (Fibers) and resume them later. In other words, React's components become functions whose execution can be scheduled and interrupted.

Fiber Node Structure

A **Fiber node** is a JavaScript object holding all information about a component instance during reconciliation. Key fields include:

- `type` **and** `key`: The component type (e.g. function, class, or string for host components like `'div'`) and the React key prop [15]. The `type` identifies what component to render, and together with `key` it is used to determine if a Fiber corresponds to the same component as before (for reuse) [15] [16].

- `child` **and** `sibling`: References to other Fibers that form the tree structure [17]. The `child` pointer links to the first rendered child component; the `sibling` pointer links to the next child returned by the same parent. For example:

```
function Parent() {
  return [<Child1 />, <Child2 />];
}
```

  Here, `Parent.child` would point to the Fiber for `Child1`, and `Child1.sibling` would point to the Fiber for `Child2` [18]. Together, `child` and `sibling` form a singly-linked list of children per parent.

- `return`: Points to the parent Fiber, similar to a stack frame's return address [19]. In the example above, both `Child1.return` and `Child2.return` would point back to the `Parent` Fiber [19].

- `pendingProps` **and** `memoizedProps`: The incoming props (arguments) for the current render and the props used in the last completed render [20]. When `pendingProps` equals `memoizedProps`, it means the inputs have not changed and React can reuse the previous output, avoiding unnecessary work [20]. Essentially, `memoizedProps` caches the last rendered props.

- `pendingWorkPriority`: A numeric priority level for this fiber's update [21]. Lower numbers mean higher priority (except 0 means no work). The scheduler uses this field to decide which fiber to work on next. For example, `requestAnimationFrame` updates might get higher priority than data-fetch updates [21].

- `alternate`: Points to the *other* Fiber representing the same component between renders [22]. React keeps two Fibers per component instance: the *current* (rendered) Fiber, and the *work-in-progress* Fiber. The `alternate` of each links these two versions. This lets React "double-buffer" the tree: build the new tree (work-in-progress) while keeping the old one intact [22]. The `cloneFiber` function lazily creates or reuses the alternate to minimize allocations [23].

- `output` (and **host fields**): For host components (DOM or native elements), the Fiber contains the rendered output (e.g. a DOM node or native view) [24]. This `output` is what ultimately gets passed to the renderer to update the UI [24]. In function/class components, `output` is conceptually the return value of the render function, which becomes the child components.

These fields (and others like state and effect lists) together define the Fiber tree. Fibers form a linked tree of nodes that React traverses and updates incrementally.

Reconciliation Process

React Fiber's reconciliation process follows the familiar virtual-DOM diffing pattern, with some enhancements:

- **Tree Generation and Diffing:** On each update, React "renders" the component tree to produce a new virtual tree. Fiber then compares this new tree to the previous one to compute minimal changes [4]. If a component's type has changed (e.g. `<span>` to `<div>` or one component class to another), Fiber discards the old subtree and mounts a new one (since diffing by type is assumed futile) [25]. Otherwise, it reconciles children, matching keys of list items to align old and new items [26].

- **Key-Based List Diffing:** When rendering arrays or iterables of children, Fiber uses `key` props to match up items [26]. Stable, unique keys allow it to preserve items across renders. This is identical to the older reconciler's behavior.

- **Reconciler/Renderer Separation:** The reconciliation phase only computes *what* should change. After reconciliation, Fiber hands off to the renderer (e.g. `react-dom`) to *apply* those changes to the actual UI [5]. Because the reconciler is shared, React DOM and React Native can both benefit from Fiber's algorithm.

Internally, Fiber traverses and updates the tree *incrementally*: it processes a unit of work (a Fiber), performs any necessary diff for that component, then schedules the next unit. This approach contrasts with the legacy stack reconciler that would synchronously traverse everything at once.

Scheduling

Fiber's scheduling system is what distinguishes it from the older reconciler. Key points:

- **Pull-Based Approach:** Rather than recursing deep immediately, Fiber pulls work in small increments and can yield between them. This is akin to using `requestIdleCallback` and `requestAnimationFrame` to schedule low/high priority tasks [27]. Fiber's architecture explicitly allows interrupting the work loop and continuing it later [27] [7].

- **Priorities and Time-Slicing:** Each update is assigned a priority. React can then choose to do higher-priority work (e.g. responding to user input) before lower-priority tasks (e.g. rendering a newly loaded data list) [8] [9]. If the browser is busy (e.g. an animation frame is about to render), Fiber can *pause* its work loop and let the browser paint, then resume on the next idle period [8] [27].

- **Pausing and Resuming:** A Fiber tree may not complete in one go. React walks the tree until it chooses to yield (perhaps because it must keep 60fps), then it will pause. Later it "re-enters" the tree at the point it left off, resuming from the last unfinished Fiber [13] [28]. This mechanism is invisible to developers but ensures UI updates never block the main thread for too long.

- **Aborting and Reusing Work:** If a new update arrives while processing an older one, Fiber can abort the in-progress work and start a new render, reusing any already-completed child Fibers as much as possible. This avoids wasted work. The presence of `memoizedProps` and the alternate Fiber tree allows React to pick up from the current visible state or reuse parts of it [20] [22].

Key Scheduling Insights: React's design principles emphasize that not all updates must be immediate. By deferring non-critical updates, the app avoids frame drops [9]. For example, if something is offscreen, its rendering can be delayed; if too many data updates are arriving, Fiber can batch them; and user-initiated animations get top priority over background tasks [8]. Altogether, Fiber's scheduler makes React *more adaptive* to the environment.

Comparison with Stack Reconciler

The legacy (pre-Fiber) reconciler is often called the "stack reconciler" because it uses the synchronous call stack to process updates. Its behavior can be summarized as follows:

- **Synchronous Depth-First Traversal:** The stack reconciler simply performs a recursive depth-first traversal of the tree on each update, completing it in one go. It cannot easily pause or yield mid-tree, nor can it schedule sub-tasks by priority.

- **Call Stack Dependent:** It leverages the JavaScript call stack for recursion. A deep tree or slow computation means a long, blocking call stack.

Fiber reimagines this model:

- **Heap-Based Stack Frames:** Instead of relying solely on the call stack, Fiber uses Heap objects (Fiber nodes) as virtual stack frames [13]. Each Fiber acts like a frame that can be stored and manipulated.

- **Interruptible Work Loop:** Because Fibers live on the heap, React can interrupt the "call stack" at any point and resume later [13] [28]. This means the work loop is fully interruptible and resumable—a capability the stack reconciler lacked.

In short, whereas the old reconciler "just blindly processes from start to end," Fiber "can actually pause on a task, deal with other tasks, and resume it" [13]. This difference enables all the scheduling flexibility in Fiber.

Benefits

- **Smoother UX and Animation:** By splitting rendering into chunks, Fiber prevents main-thread monopolization. This means animations and interactions can stay at 60fps because React avoids long blocking updates [9]. Frame-drop scenarios (like when loading or updating large data) are mitigated by yielding during idle time or giving precedence to user input [9] [8].

- **Improved Responsiveness:** Time-slicing and priority scheduling let high-priority updates (e.g. keystrokes, clicks) interrupt lower-priority renders. Users perceive the app as faster because their actions take effect immediately even if a big render is still pending [8] [9].

- **Concurrency and New Features:** The dual-tree (current vs work-in-progress) model and interruptible rendering pave the way for React's concurrent features (like Suspense and error boundaries). For example, if a subtree throws an error or is awaiting data, Fiber can pause it and handle the error without crashing the whole app. These capabilities were difficult with the old reconciler.

- **Efficient Diffing:** Memoizing props and reusing Fiber nodes minimizes unnecessary work [20] [22]. If props haven't changed, Fiber can bail out on a subtree quickly. Similarly, the ability to reuse an alternate tree avoids remounting when not needed.

Overall, Fiber makes React more *adaptive*: it adapts to device performance and user needs by scheduling work intelligently. The result is often faster, smoother applications, especially under complex update scenarios.

Real-World Implications

React Fiber's architecture was motivated by real-world app complexity. In practice:

- **Complex Animations and Gestures:** UIs with heavy animations (e.g. interactive charts, drag-and-drop, gesture-driven interfaces) benefit because Fiber can keep animations running smoothly by postponing non-urgent rendering tasks [2].

- **Large Component Trees:** Applications with deep or large component hierarchies (like dashboards or pages with many widgets) no longer incur jank from one big synchronous reconciliation. Instead, rendering can be spread out or chunked across frames.

- **Mobile and React Native:** On mobile devices, keeping the UI thread free is even more critical. Fiber's scheduling model fits well with React Native's need for smooth gestures. The same reconciler is shared across platforms, so improvements carry over.

- **Concurrent Mode and Suspense:** While outside the scope of this doc, it's worth noting Fiber underpins React's newer concurrency features (introduced after this doc). Concepts like pausing, yielding, and priorities directly enable things like `React.Suspense` and `Concurrent Mode`.

- **Performance Tuning:** Understanding Fiber helps developers optimize React apps. For instance, knowing that offscreen components get deferred [8] can influence when and how you trigger updates. Prioritizing user inputs or using `unstable_scheduleCallback` becomes feasible with Fiber's priority system.

In summary, React Fiber aligns the library with the demands of modern web and native apps: dynamic interfaces, varied input types, and unpredictable data. It trades the simplicity of an all-at-once update for the flexibility and performance of incremental, prioritized rendering.

Sources: Insights are drawn directly from the official React Fiber documentation in the [react-fiber-architecture GitHub repo](#) [1] [4] [15] [20] (as well as linked React docs in those sources). Each section above quotes or paraphrases lines from that document.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28]
GitHub - acdlite/react-fiber-architecture: A description of React's new core algorithm, React Fiber
https://github.com/acdlite/react-fiber-architecture