Super Charging Fine-Grained Reactive Performance · milomg.dev

9 min. read · View original

What's a Reactive Library?

Reactivity is the future of JS frameworks! Reactivity allows you to write lazy variables that are efficiently cached and updated, making it easier to write clean and fast code.

I've been working on a new fine grained reactivity libary called Reactively inspired by my work on the SolidJS team. Reactively is currently the fastest reactive library in its category. You can use Reactively on its own right now, and in the future the ideas from Reactively will help SolidJS become even faster.

Let's explore some of the different algorithmic approaches to fine grained reactivity, I'll introduce you to my new reactivity library, and we'll benchmark three of the libraries to compare.

Introducing Reactively

Fine-grained reactivity libraries have been growing in popularity recently. Examples include new libraries like Preact Signals, µsignal, and now Reactively, as well as longer-standing libraries like Solid, S.js, and CellX. Using these libraries, programmers can make individual variables and functions *reactive*. *Reactive* functions run automatically, and re-run 'in reaction' to changes in their sources.

With a library like Reactively, you can easily add lazy variables, caching, and incremental recalculation to your typescript/javascript programs. Reactively is tiny (<1 kb) and has a simple API. Hopefully, Reactively makes it easy for you to explore the benefits of reactive programming.

Here's an example of using Reactively for a lazy variable:

ts

```
import { reactive }
from"@reactively/core";

constnthUser = reactive(10);

// fetch call is deferred until
needed

constlazyData = reactive(()
=>fetch(`https://data.mysite.io/users
?n=${nthUser.value}`));

if (needUsers) {

useBuffer(awaitlazyData.value);

}
```

```
ts

import { reactive }
from"@reactively/core";

constnthUser = reactive(10);

// fetch call is deferred until
needed

constlazyData = reactive(()
=>fetch(`https://data.mysite.io/users
?n=${nthUser.value}`));

if (needUsers) {

useBuffer(awaitlazyData.value);

}
```

Reactive libraries work by maintaining a graph of dependencies between reactive elements. Modern libraries find these dependencies automatically, so there's little work for the programmer beyond simply labeling reactive elements. The library's job is to efficiently figure out which reactive functions to run in responses to changes elsewhere in the graph. In this example, our dependency graph is quite simple:

Reactivity libraries are at the heart of modern web component frameworks like Solid, Qwik, Vue, and Svelte. And in some cases you can add fine-grained reactive state management to other libraries like Lit and React. Reactively comes with a decorator for adding reactive properties to any class, as well as prototype integration with Lit. Preact Signals comes with a prototype integration with React. Expect more integrations as these reactivity cores mature.

Goals of a reactive library

The goal of a reactive library is to run reactive functions when their sources have changed.

Additionally, a reactive library should also be:

- **Efficient**: Never overexecute reactive elements (if their sources haven't changed, don't rerun)
- **Glitch free**: Never allow user code to see intermediate state where only some reactive elements have updated (by the time you run a reactive element, every source should be updated)

Lazy vs. Eager Evaluation

Reactive libraries can be divided into two categories: lazy and eager.

In an eager reactive library, reactive elements are evaluated as soon as one of their sources changes. (In practice, most eager libraries defer and batch evaluations for performance reasons).

In a lazy reactive library, reactive elements are only evaluated when they are needed. (In practice, most lazy libraries also have an eager phase for performance reasons).

We can compare how a lazy vs eager library will evaluate a graph like this:

Lazy	Eager
A lazy library will recognize that the user is asking to update `D`, and will first ask `B` then `C` to update, then update `D` after the `B` and `C` updates have been completed	An eager library will see that `A` has changed, then tell `B` to update, then `C` to update and then `C` will tell `D` to update

Reactive Algorithms

The charts below consider how a change being made to `A` updates elements that depend on `A`.

Let's consider two core challenges that each algorithm needs to address. The first challenge is what we call the diamond problem, which can be an issue for eager reactive algorithms. The challege is to not accidentally evaluate `A,B,D,C` and then evaluate `D` a second time because `C` has updated. Evaluating `D` twice is inefficient and may cause a user visible glitch.

The second challenge is the equality check problem, which can be an issue for lazy reactive algorithms. If some node `B` returns the same value as the last time we called it, then the node below it C doesn't need to update. (But a naive lazy algorithm might immediately try to update C instead of checking if B has updated first)

To make this more concrete, consider the following code: it is clear that C should only ever be run once because every time A changes, B will reevaluate and return the same value, so none of C's sources have changed.

```ts
constA = reactive(3);

constB = reactive(() =>A.value * 0);
// always 0

constC = reactive(() =>B.value + 1);
```

```ts
constA = reactive(3);

constB = reactive(() =>A.value * 0);
// always 0

constC = reactive(() =>B.value + 1);
```

MobX

In a blog post a few years ago, Michael Westrate described the core algorithm behind MobX. MobX is an eager reactive library, so let's look at how the MobX algorithm solves the diamond problem.

After a change to node `A`, we need to update nodes `B`, `C` and `D` to reflect that change. It's important that we update `D` only once, and only after `B` and `C` have updated.

MobX uses a two pass algorithm, with both passes proceeding from `A` down through its observers. MobX stores a count of the number of parents that need to be updated with each reactive element.

In the diamond example below, the first pass proceeds as follows: After an update to `A`, MobX marks a count in `B` and `C` that they now have one parent that needs to update, and MobX continues down from `B` and from `C` incrementing the count of `D` once for each parent that has a non `0` update count. So when the pass ends `D` has a count of 2 parents above it that need to be updated.

In a second pass, we update every node that has a zero count, and subtract one from each of its children, then tell them to reevaluate if they have a zero count and repeat.

Then, `B` and `C` are updated, and `D` notices that both of its parents have updated.

Now finally `D` is updated.

This quickly solves the diamond problem by separating the execution of the graph into two phases, one to increase a number of updates and another to update and decrease the number of updates left.

To solve the equality check problem MobX just stores an additional field that tells each node whether any of its parents have changed value when they updated

An implementation of this might look like the following (code by Fabio Spampinato):

js

```
set = (value) => {

this.value = valueNext;

// First recursively increment the counts

this.stale(1, true);

// Then recursively update the values and decrement the counts

this.stale(-1, true);

};

stale = (change: 1 | -1, fresh: boolean): void=> {
```

```
if (!this.waiting && change < 0)
return;

if (!this.waiting && change > 0) {

this.signal.stale(1, false);

  }

this.waiting += change;

this.fresh ||= fresh;

if (!this.waiting) {

this.waiting = 0;

if (this.fresh) {

this.update();

    }

this.signal.stale(-1, false);

  }

};
```

js

```
set = (value) => {

this.value = valueNext;

// First recursively increment the
counts
```

```
this.stale(1, true);

// Then recursively update the values
and decrement the counts

this.stale(-1, true);

};

stale = (change: 1 | -1, fresh:
boolean): void=> {

if (!this.waiting && change < 0)
return;

if (!this.waiting && change > 0) {

this.signal.stale(1, false);

  }

this.waiting += change;

this.fresh ||= fresh;

if (!this.waiting) {

this.waiting = 0;

if (this.fresh) {

this.update();

    }

this.signal.stale(-1, false);
```

```
  }

};
```

Preact Signals

Preact's solution is described on their blog. Preact started with the MobX algorithm, but they switched to a lazy algorithm.

Preact also has two phases, and the first phase "notifies" down from A (we will explain this in a minute), but the second phase recursively looks up the graph from `D`.

Preact checks whether the parents of any signal need to be updated before updating that signal. It does this by storing a version number on each node and on each edge of the reactive dependency graph.

On a graph like the following where A has just changed but B and C have not yet seen that update we could have something like this:

Then when we fetch D and the reactive elements update we might have a graph that looks like this

Additionally, Preact stores an extra field that stores whether any of its sources have possibly updated since it has last updated. Then, it avoids walking up the entire graph from `D` to check if any node has a different version when nothing has changed.

We can also see how Preact solves the equality check problem:

When `A` updates, `B` will rerun, but not change its version because it will still return `0`, so `C` will not update.

Reactively

Like Preact, Reactively uses one down phase and one up phase. Instead of version numbers, Reactively uses only graph coloring.

When a node changes, we color it red (dirty) and all of its children green (check). This is the first down phase.

In the second phase (up) we ask for the value of `F` and follow a procedure internally called `updateIfNecessary()` before returning the value of `F`. If `F` is uncolored, it's value does not need to be recomputed and we're done. If we ask for the value of `F` and it's node is red, we know that it must be re-executed. If `F`'s node is green, we then walk up the graph to find the first red node that we depend on. If we don't find a red node, then nothing has changed and the visited nodes are set uncolored. If we find a red node, we update the red node, and mark its direct children red.

In this example, we walk up from F to E and discover that C is red. So we update C, and mark E as red.

Then, we can update E, and mark its children red:

Now we know we must update F. F asks for the value of D and so we `updateIfNecessary` on D, and repeat a similar traversal this time with D and B.

And finally we get back to a fully evaluated state:

In code, the `updateIfNecessary` process looks like this:

ts

```
/** update() if dirty, or a parent turns out to be dirty. */
updateIfNecessary() {
// If we are potentially dirty, see if we have a parent who has actually changed value
if (this.state === CacheCheck) {
for (constsourceofthis.sources) {
source.updateIfNecessary(); // Will change this.state if source calls update()
if (this.state === CacheDirty) {
// Stop the loop here so we won't
trigger updates on other parents
unnecessarily
// If our computation changes to no
longer use some sources, we don't
// want to update() a source we used
last time, but now don't use.
break;
      }
    }
  }
// If we were already dirty or marked
dirty by the step above, update.
if (this.state === CacheDirty) {
this.update();
  }
// By now, we're clean
this.state = CacheClean;
}
```

ts

```
/** update() if dirty, or a parent turns out to be dirty. */
updateIfNecessary() {
// If we are potentially dirty, see if we have a parent who has actually changed value
if (this.state === CacheCheck) {
for (constsourceofthis.sources) {
source.updateIfNecessary(); // Will change this.state if source calls update()
if (this.state === CacheDirty) {
// Stop the loop here so we won't trigger updates on other parents unnecessarily
// If our computation changes to no longer use some sources, we don't
// want to update() a source we used last time, but now don't use.
break;
      }
    }
```

```
  }
// If we were already dirty or marked dirty by the step above, update.
if (this.state === CacheDirty) {
this.update();
  }
// By now, we're clean
this.state = CacheClean;
}
```

Ryan describes a related algorithm that powers Solid in his video announcing Solid 1.5.

Benchmarks

Current reactivity benchmarks (Solid, CellX, Maverick) are focused on creation time, and update time for a static graph. The existing benchmarks aren't very configurable, and don't test for dynamic dependencies.

We've created a new and more flexible benchmark that allows library authors to create a graph with a given number of layers of nodes and connections between each node, with a certain fraction of the graph dynamically changing sources, and record both execution time and GC time.

In early experiements with the benchmarking tool, what we've discovered so far is that Reactively is the fastest (who would've guessed 😉).

The frameworks are all plenty fast for typical applications. The charts report the number of updated reactive elements per millisecond on an M1 laptop. Typical applications will do much more work than a framework benchmark, and at these speeds the frameworks are unlikely to bottleneck overall performance. Most important is that the framework not run any user code unnecessarily.

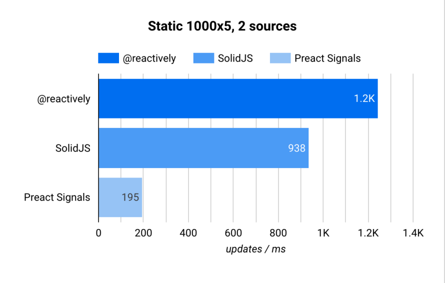
That said, there's learning here to improve performance of all the frameworks.

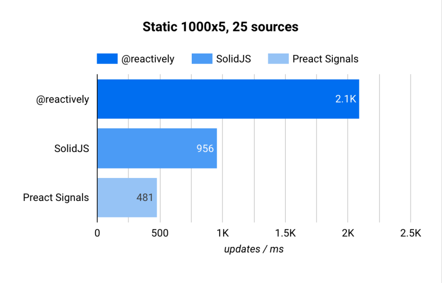
- The Solid algorithm performs best on wider graphs. Solid is consistent and stable, but generates a little garbage which limits speed under these extreme benchmark conditions.
- The Preact Signal implementation is fast and very memory efficient. Signal works especially well on deep dependency graphs, and not as well on wide dependency graphs. The benchmarking also caught a performance bug with dynamic graphs that will undoubtedly be fixed soon.

> It should be noted that there is a second part of the implementation of each framework that makes a big difference in performance: memory management & data structures. Different data structures have different characteristics for insert and delete, as well as

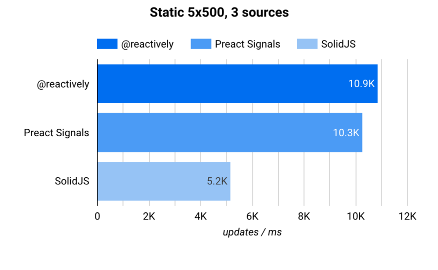
> very different cache locality in JavaScript (which can drastically change iteration times). In a future blog post we'll look at the data structures and optimizations used in each framework (such as S.js's slots optimization, or Preact's hybrid linked list nodes).

Wide graphs

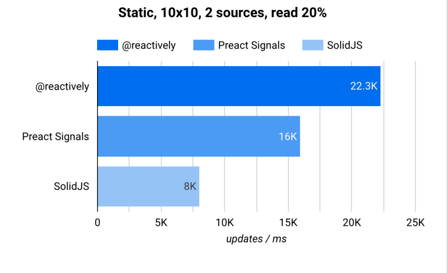




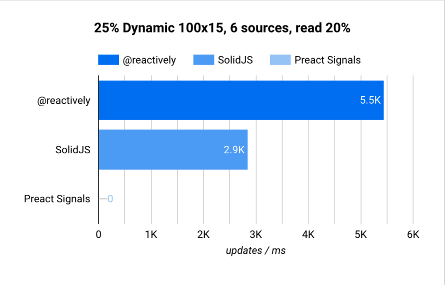
Deep Graph

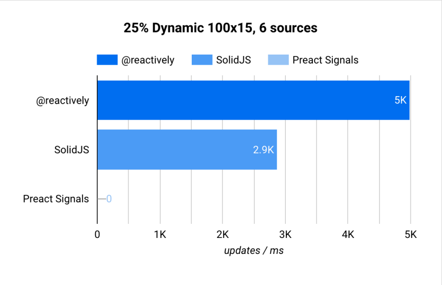


Square Graph



Dynamic Graphs


25% Dynamic 100x15, 6 sources, read 20%
@reactively
SolidJS
Preact Signals
@reactively
SolidJS
Preact Signals
5.5K
2.9K
1K
2K
3K
4K
5K
6K
updates / ms


25% Dynamic 100x15, 6 sources
@reactively
SolidJS
Preact Signals
@reactively
SolidJS
Preact Signals
5K
2.9K
1K
2K
3K
4K
5K
updates / ms