Technical Report: An Asynchronous Call Graph for JavaScript

Dominik Seifert
d01922031@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Michael Wan
b07201003@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Jane Hsu
yjhsu@csie.ntu.edu.tw
National Taiwan University
Taiwan, Taipei

Benson Yeh
pcyeh@ntu.edu.tw
National Taiwan University
Taiwan, Taipei

ABSTRACT

This Technical Report serves as a supplementary document to the original article "An Asynchronous Call Graph for JavaScript". It provides background information and showcases results.

1 TERMINOLOGY

In this section, we aim to provide more definitions and details related to terminology used in the paper.

1.1 Contexts and Call Graph Roots

In the following, we define **context** and **CGR** more concretely. To that end, we discuss how contexts are recorded in an actual implementation. We employ a common shadow stack approach, which is used to determine which context any traced event belongs to.

(1) We refer to a function, file, script tag as a "static execution context" (short: **static context**).
(2) Given a static execution context f, we denote its i'th execution f_i. f_i is an "execution context" (short: **context**). Each context starts when it is pushed onto and ends when popped from the stack.
(3) If in context f_i, some function g is called, then, for some j, g_j is a child —or **callee**— of f_i, and f_i is a parent —or **caller**— of g_j.
(4) Any context f_i is considered a Call Graph Root (CGR), if it has no parent caller. This implies that f_i was **directly invoked by the JS engine's event loop**, or in other words, when the synchronous call stack is empty.
(5) Given some CGR r, `cgr(x) = r = cgr(r)` holds for all contexts and events x executed after r is pushed onto and before it is popped from the stack.
(6) ES2017[1] introduced `async` functions, which need special attention due to their property of interruptibility: we refer to the i^{th} execution of some `async` function h as h_i. The context h_i is considered a **real context**. When executed, we add a **virtual context** h_i^1 as a child to h_i and push it onto the shadow stack. Furthermore, any `await p;` expression tells the scheduler to **interrupt** the current control flow for one tick of the asynchronous queue, or, if `p` is a promise, until that promise has been settled. Upon interruption, the current virtual context is popped from the shadow stack. Once the `await` expression concludes, that is, after the promise has settled, h_i is re-queued and its execution continues upon the next tick. When continuing for the k'th time, a new virtual context h_i^k is added as child of h_i.
(7) The context h_i^k also represents the **asynchronous continuation** of the interrupted real context h_i.
(8) Since a real context of an async function h_i has the exceptional property of spanning multiple CGRs, the ACG reasons about its children h_i^k instead. We thus deduce two more observations:
(9) The virtual context h_i^1 is considered a CGR, iff h_i has no parent.
(10) Each virtual context $h_i^k, k > 1$ is always a CGR.

It is worth noting that since in an actual implementation, not all contexts might be recorded, rule 4 might observe slightly different semantics, that is: f_i is considered a CGR, if it is the first recorded context without a recorded parent. In this case, the shadow stack is empty, while the JS engine's actual stack might not be.

1.2 AE Comparison

To aid understanding, Fig. 1 illustrates the three types of Asynchronous Events (AE). In all three cases, the resulting subgraph is `openFile` → `readFile` → `sendFile`.

The first two implementations use AWAIT and THEN. Here, all CGRs are connected by CHAINs. However, in the callback version, according to the presented rules of this first version of the ACG, callback CHAIN heuristics do not apply, and thus, the two edges are FORKs.

1.3 Promise Chain and Nesting Semantics

While a complete discussion of the ADD_EDGE algorithm and its implementation falls out of scope of this document, this subsection aims to better illustrate the CHAIN ruleset (i), as mentioned in the article. In general, that ruleset concerns itself with "promise-based AEs" (AEs that involve promises). It determines `from(e)` by backtracking along the promise tree. To that end, it (i) looks up individual promise chains, (ii) unravels promise nesting relationships, and in case of AWAIT, (iii) looks for previous CGRs of the same async function. Listing 1 illustrates these three types of semantics in three examples.

1.3.1 Promises and CGRs. A CGR is defined as "belonging to", or "owned by" a promise `p`, if the CGR is directly scheduled via `p`:

In case of THEN-type AEs `e`: `p = q.THEN(f)`, `f = from(e) = cgr(p)`. `p` is ensured to always have exactly one CGR.

```
async function send(fpath) {
  const file = await openFile(fpath);

  const cont = await readFile(file);


  await sendFile(cont);


  console.log('File sent!');
}
```

```
function send(fpath) {
  return openFile(fpath).
    then(function (file) {
      return readFile(file);
    }).
    then(function (cont) {
      return sendFile(cont);
    }).
    then(function () {
      console.log('File sent!');
    });
}
```

```
function send(fpath, cb) {
  openFile(fpath, function (file) {

    readFile(file, function (cont) {


      sendFile(cont, function () {
        cb && cb();

        console.log('File sent!');
      });
    });
  });
}
```

Figure 1: Three types of AEs implementing a series of three operations: `openFile` → `readFile` → `sendFile`

```
// ex1: promise chain
p
  .then(() => A)
  .then(() => B);

// ex2: promise nesting
p.then(() => {
    A;
    return Promise.resolve()
      .then(() => B);
  });

// ex3: async function
async function f() {
  await A;
  await B;
}
```

Listing 1: Given some promise `p` and non-promise expressions `A` and `B`, all of the following three examples produce an ACG with at least one CHAIN between `cgr(A)` and `cgr(B)`

.

In case of AWAIT-type AEs `e` (e.g. `p = (async (function g() { A; await x; B; await y; C; })())`, `p` is the promise returned by the async function call `g()`. Asynchronous continuation CGRs (e.g. `cgr(B)` and `cgr(C)`) all belong to `p`. In this case, `cgr(p)` is `cgr(A)`, iff g has no parent, and `cgr(B)` otherwise.

In case of promisified CBs, the asynchronously executed callback's CGR also belongs to its promise. E.g., in `p = new Promise(r => setTimeout(() => r(), delay))`, the CGR representing the execution of `setTimeout`'s callback, `() => r()`, is also `p`'s unique CGR. The executor function itself cannot be a CGR since it is executed synchronously with the promise constructor call.

Note that not all promises have CGRs. For example, `Promise.resolve(x)` has no CGR, and neither does the promise returned by `sleep()` in the first example of Listing 2.

1.3.2 Promise Nesting. In ES2022, there are nine different types of **promise nesting**: (i) `await` expressions, (ii) returning a promise from an async function or a (iii) THEN method's callback (`then`, `catch`, `finally` in the standard implementation), as well as six promise methods: (iv) `resolve`, (v) `reject`, (vi) `all`, (vii) `allSettled`, (viii) `any` and (ix) `race`.

The first three types, (i) through (iii), are the only ones that allow for **dynamic nesting**: an outer promise already exists, independent of the nested promises. Settling of the outer promise can be delayed by dynamically deciding to nest a promise from within the context of the outer promise. The six promise methods, (iv) through (ix), all create new promises, but none of them have their own CGRs. The last four promise methods, (vi) through (ix), constitute synchronization methods[1] between multiple promises: the created promise only settles once some condition applies to the statuses of a set of nested promises.

1.3.3 Promisification. Chaining asynchronous events with callbacks, unlike promises and async functions, requires nesting them to arbitrary depth. That causes, what is commonly known as, "callback hell"[2].

Promisification is the process of using the promise constructor to wrap asynchronous callback operations into promises, as illustrated in Listing 2. We refer to an asynchronous callback scheduled from within the context of a promise constructor's executor function as a **"promisified CB"**.

The resulting promise can be chained and nested with other promises. That, in turn, is captured in form of CHAINs in the ACG. Promisified callbacks, just like other CB-type AEs and unlike

[1] https://en.wikipedia.org/wiki/Synchronization_(computer_science)
[2] http://callbackhell.com

```
// ex1: sleep is setTimeout promisified.
function sleep(delay) {
  return new Promise(
    resolve => setTimeout(resolve, delay)
  );
}

// ex2: generic promisification primitive.
function promisify(operation) {
  return (...args) => {
    new Promise((resolve, reject) => {
      operation(...args, (result, error) => {
        if (error) {
          reject(error);
        }
        else {
          resolve(result);
        }
      });
    });
  };
}
```

Listing 2: The promise constructor takes a single "executor" function argument with two parameters: the `resolve` and `reject` functions are called to settle the promise at a later point in time.

AWAIT- and THEN-type AEs, require manual intervention to propagate errors. However, promisification makes it easy to (manually) lift the error into the promise tree. If done correctly, that error then propagates automatically, i.e. it can be caught by the same error handler as the rest of the promise tree or subtree.

Many asynchronous callback operations in JavaScript follow a common convention of passing two arguments to the callback function: the first encapsulates the `result` of the operation, and the second is an `error` signifying operation failure. Due to that convention, a generic `promisify` primitive, like the one in Listing 2, can be used to automatically convert a conventional callback-based asynchronous operation into one that returns a promise.

1.3.4 The "first CHAIN Problem". Finally, we discuss the last major aspect of the ADD_EDGE algorithm, as it pertains to promises. Consider the three example promises `p` of Fig. 1, and assume that these promises are part of a bigger ACG. Determining how to embed `cgr(p)` into the ACG is called the "first CHAIN problem": given a promise-based AE `e` of promise `p`, find `from(e)` that `cgr(p)` is CHAINed to. We call this the "first CHAIN problem" because it requires finding a possible CHAIN toward the first CGR of a promise.

Such a CHAIN only exists iff `p` is "**chained-to a root**", meaning `p` or some other promise `q`, that transitively nests `p` or a chain that `p` is part of, is nested dynamically (see §1.3.2), at root level. The "at root level" constraint must be added explicitly, since dynamic nesting type (ii) might not nest at root level, if no `await` expression executed in the async function.

```
async function f() {
  FA
  await 0;  // E1: ?
  FB
  await 0;  // E2: CHAIN
  FC
}

// ex1: E1 is FORK
A; f(); B;

// ex2: E1 is CHAIN
A; await f(); B;

// ex3: E1 is FORK
await g();
function g() { f(); }

// ex4: E1 is CHAIN
let p; h(); await p;
function h() { p = f(); }
```

Listing 3: CHAIN vs. FORK: `f` has two AEs `E1` and `E2`. `E2` always induces a CHAIN, but `E1` might induce CHAIN or FORK, depending on the caller. Assume that the example codes ex1-4 execute at root-level and independent of one another.

Listing 3 illustrates the "first CHAIN problem" in four examples: the first CGR of promise `p = f()` (`FB`) is CHAINed or FORKed iff `p` is chained-to a root.

2 RESULTS: DETAILS

We used the ACG on eleven real-world projects, with three goals in mind:

(1) Verify that the ACG can run on real-world projects.
(2) Explore real-world asynchronous control flow patterns.
(3) Analyze debugging journeys in order to test the ACG's utility in dealing with real-world bugs.

This section is organized as follows: §2.1 and §2.2 provide more details of the async-js and sequelize exploratory case studies, respectively. §2.3 depicts all ACGs omitted from the article.

2.1 Async-js Bug #1729

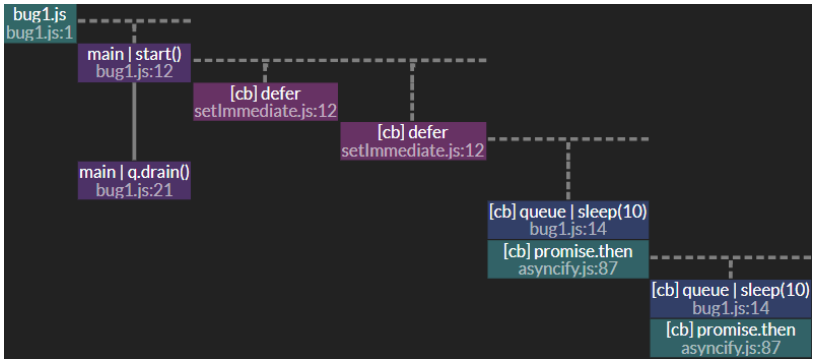
Our async-js sample code, shown in Listing 2a, is a slight modification of the buggy code provided in issue #1729[3]. The code queues two tasks in an asynchronous queue and then waits for the queue to finish those tasks, via the `await q.drain()`. The `drain` event should activate after all tasks have finished, but the bug defeats that assumption.

2.1.1 Debugging Journey. The bug reveals itself in the console. The order of console log messages indicate that the promise `queue.drain()` settled too early. It is probably an order violation bug.

[3] https://github.com/caolan/async/issues/1729

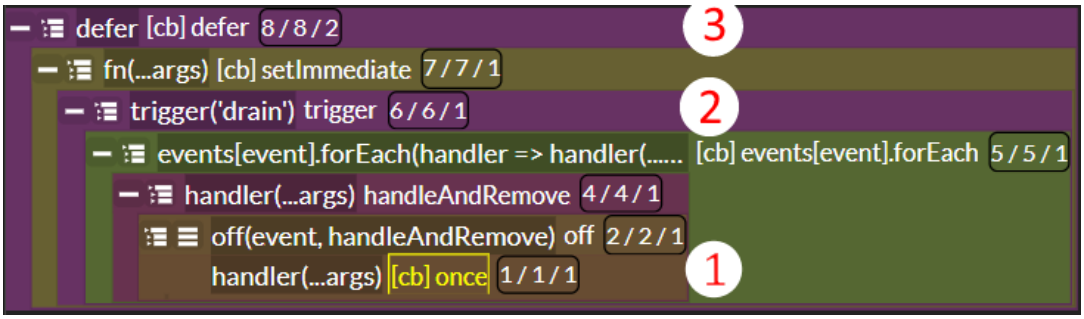
```
await start();
const q = queue(async (task) => {
  await sleep(10);
  console.log("task done");
});

q.push([]); // this triggers the bug
q.push([1, 2]);

await q.drain();
console.log("all tasks completed");
```

(a) `main` function for the async-js bug

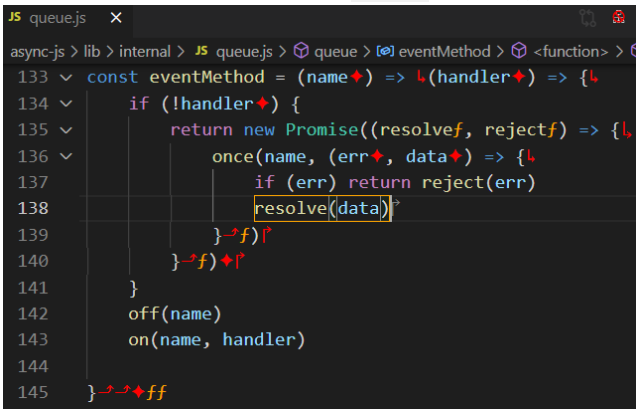


(b) ACG of the buggy execution (details enabled)

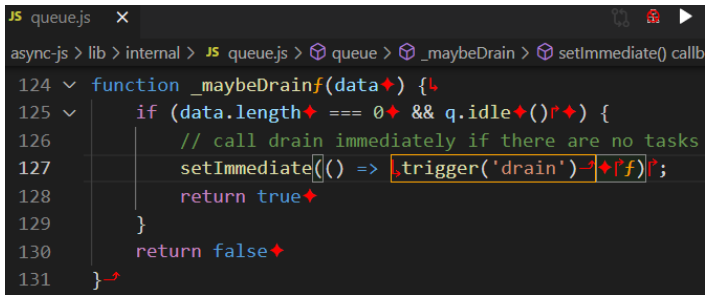
Figure 2: code vs. ACG



(a) Synchronous call graph of the `[cb] defer` CGR. (1) Context of the callback to the `once` function; selected after we jumped to the promise's `resolve` call (see Fig. 3b). (2) Context of the `trigger` function. (3) Context of the `[cb] defer` CGR.



(b) The `resolve` call is selected after following the promise returned from `drain()` in Debugging Step 2.



(c) The `trigger` call is selected after clicking into it in Debugging Step 5.

Figure 3: Other snapshots of the async-js debugging process.

The following is one (of many possible) series of steps that lead from discovering the symptom to finding the cause. While only the first three steps relate to the ACG, all steps of the journey are provided for completeness.

(1) Investigate the ACG. It is depicted with `details` enabled in Fig. 2b. The bug is visually obvious: the `drain` event (second thread from the left) is triggered before the two tasks (two right-most threads) finish. We click into the `main | queue.drain()` node, which takes us to the `await queue.drain()` expression in the code.
(2) Find out who settled the promise and why. We select the expression of the nested promise `queue.drain()` and use

Dbux's "Trace Details"[4] tool to jump to the event that settled it. This takes us to the `resolve` call of a promisified callback belonging to a generic event handler function, depicted in Fig. 3b.

(3) Re-orient (1). Now that we have jumped an arbitrary distance to another place in the unknown codebase, we lost our bearings. The ACG helps us verify that we are now in the first "`[cb] defer`" CGR, FORKed from and executed right after the first `main` CGR (see Fig. 2b).

(4) Re-orient (2). We switch to the Synchronous Call Graph (Fig. 3a) which serves as an interactive version of a traditional dynamic call graph. It can also be used to read the synchronous part of the call stack. We find: (1) the currently selected trace is in the callback of the `once` function (shown as `[cb] once` in the graph). (2) We also see the CGR *defer* function. (3) Between the call to `once` and its CGR, we see `trigger('drain')` in the middle of the stack. This is most likely what we are looking for: the code that actually triggers the early `drain` event.

(5) Investigate the `trigger` call. We click into the caller `trigger('drain')`, which takes us to the `_maybeDrain` function, as depicted in Fig. 3c.

(6) The bug is in line 127 of that file. We see a common bug pattern here: `setImmediate` is called in a given state (i.e. the queue being empty). In this particular bug, that state is encoded in the condition of the `if` statement on line 126. However, due to lack of atomicity, when the callback gets executed at a later point in time, that constraint on state is not ensured to hold. One possible solution is to explicitly check again whether the required state constraint still holds after event activation and before committing the requested operation. That is also the solution we have detailed in our bug report[3].

2.1.2 Asynchronous Control Flow Patterns. The third goal of this study is to use the ACG to explore asynchronous control flow patterns in real-world application. We first summarize the overall structure of the ACG and then investigate several subpatterns in more detail.

In an ideal world, the amount of the ACG's threads should reflect the real degree of concurrency of the application. However, due to semantics not clearly signaling intent and limitations of the ACG's construction algorithm, that is rarely the case. That discrepancy between ideal world expectations and real world limitations often offers interesting insights. That is why we start our exploration with an assessment of the application's concurrency, and then compare that to the amount of virtual threads uncovered by the ACG.

The test case has no actual concurrency, in that, everything is supposed to execute sequentially. That means that ideally, there should only be one thread. However, our ACG, as shown in Fig. 2b, found six, caused by five FORKs. From left to right, the threads represent:

(1) The entry point (`bug1.js`).
(2) The async main function.
(3) The buggy `drain` event handler.
(4) The queue driver function `queue.process()`, triggered asynchronously.
(5) The final two threads each represent execution of one of the two queued task.

Next, we investigate each FORK and determine why it is not a CHAIN:

(1) The `main` function is FORKed, because its call is not `await`ed in the entry point file.
(2) The queue's `drain` event handler and driver CGRs are both scheduled as asynchronous callbacks via `setImmediate`. No callback CHAIN heuristic applies.
(3) The two task functions are async functions, but their returned promises are not chained-to the CGR that calls them.

We found one scenario deserving special attention: `queue.drain()` returns a promise and is also chained-to the (virtual) asynchronous continuation CGR of the `main` function. Despite that, its CGR seems to be missing a CHAIN. The promise, as depicted in Fig. 3b, schedules an event handler callback in its executor function via `once`. The promise is settled from within the `once` callback, but that callback is not a CGR. Instead, the `once` callback is called from a CGR representing a `setImmediate` callback (Fig. 3c), which was already FORKed due to the rules applied to the `setImmediate` CB-type AE.

In short: the promise does not "own" a CGR. Instead, it acts as a **synchronization link** between two otherwise independent CGRs.

2.1.3 Async-js Case Study Summary. In our biased opinion, while other Dbux features were also vital in finding the bug, the ACG did provide crucial clues that enabled this straight-forward series of debugging steps. Specifically, we found that the ACG (i) provided an obvious entry point for the debugging process and (ii) served as a high-level map to assist orientation while jumping through an unknown codebase.

While using the ACG is not always as easy as in this example, it generally provides great value to the analysis process. In the next subsection, a more complex scenario is investigated.

[4] https://domiii.github.io/dbux/runtime-analysis/trace-details

2.2 Sequelize Bug #13554

```
try {
  await sequelize.sync();
  await Promise.all([
    User.findOrCreate({
      where: { id: 1 },
      defaults: { x: 1 }
    }),
    User.findOrCreate({
      where: { id: 1 },
      defaults: { x: 2 }
    })
  ]);
}
catch (err) {
  console.error(err);
  await User.findAll({ where: { id: 1 } });
}
```

Listing 4: Simplified Pseudocode for the sequelize bug

In this second case study, we decided to explore sequelize because it is a popular ORM library with all its database operations wrapped into asynchronous operations. We set out to follow up on an already fixed atomicity violation bug #1831[5] from 2014: back then, the composite `findOrCreate` operation did not use a transaction to protect atomicity between its two child operations `find` and `create`. That has been fixed since. However, when we tested multiple concurrent `findOrCreate` operations in the most recent version, we discovered two new problems that we also reported on their GitHub page[6]. Listing 4 shows an approximation of the code we ran in two slightly different scenarios. The resulting ACGs are shown in Fig. 4.

2.2.1 Converting FORKs to CHAINs. We started this exploration by assessing the application's real degree of concurrency. We estimated (and later verified) it to be 2, due to the two concurrent `findOrCreate` calls. Everything beside those two queries is serially connected to them. At first inspection, we found it curious that despite primarily being connected by ASYNC and THEN type of AEs, the ACG, depicted in Fig. 4a, boasted 18 FORKs, 17 more than expected.

When investigating, we found that 11 (the majority) of those forks were due to the "retry-as-promised" library. That is because the library employs a type of promisification that the ACG currently does not properly recognize which leads to a FORK. The library promisifies an async function call, which is an unnecessary promisification since async function calls already produce a promise. Fig. 4b (right) shows the same program after editing 13 lines of that library to remove the unnecessary promisification. In the new version, only 7 FORKs remain while the 11 false FORKs have been eliminated. The resulting ACG captures asynchronous control flow much more clearly, with less clutter. For example, the initialization routine before line (2) is now accurately represented by a single thread. The same goes for the other seven queries.

2.2.2 Asynchronous Control Flow Patterns: "Landmarks". The ACGs, depicted in Fig. 4, both have over a hundred CGRs, more than ten times the size of the previous sample. Before proceeding with debugging, we first deciphered some major "landmarks" of the sample application and how they show up in the ACG:

The four lines (1a-d) show a somewhat repeating pattern. Upon clicking into the relevant nodes, one can quickly learn that each of these groups represents a query to the database. Horizontal line (2) shows the end of the `sync` function, sequelize's initialization routine. Right below that, both ACGs show a divergence: the two `findOrCreate` operations start running in parallel. The flame icon at horizontal line (3) marks the occurrence of the first bug that is also visible in the console: the second `findOrCreate` operation tries to create a second transaction while there is already one active, which the DBMS denies. Line (4) marks the settling of the `Promise.all` promise. However, that is not the end of our journey. The ACG reveals a classic misconception of asynchronous JavaScript: `Promise.all` (unlike `allSettled`) does not wait until all promises have settled. It can also settle early, if any nested promise failed. In that case, all remaining unsettled promises are now "dangling", without anyone awaiting their settling. In this case, the failure of the second operation lead to the `Promise.all` promise settling early, and the first operation "dangling". That's why we can see its thread growing past line (4). Line (5) marks the spot where the first operation errors out (flame icon next to the node). We explain why in the next section.

2.2.3 Debugging Journey. Since this is a more convoluted journey, we don't provide all steps, and instead only summarize findings that are relevant to the ACG:

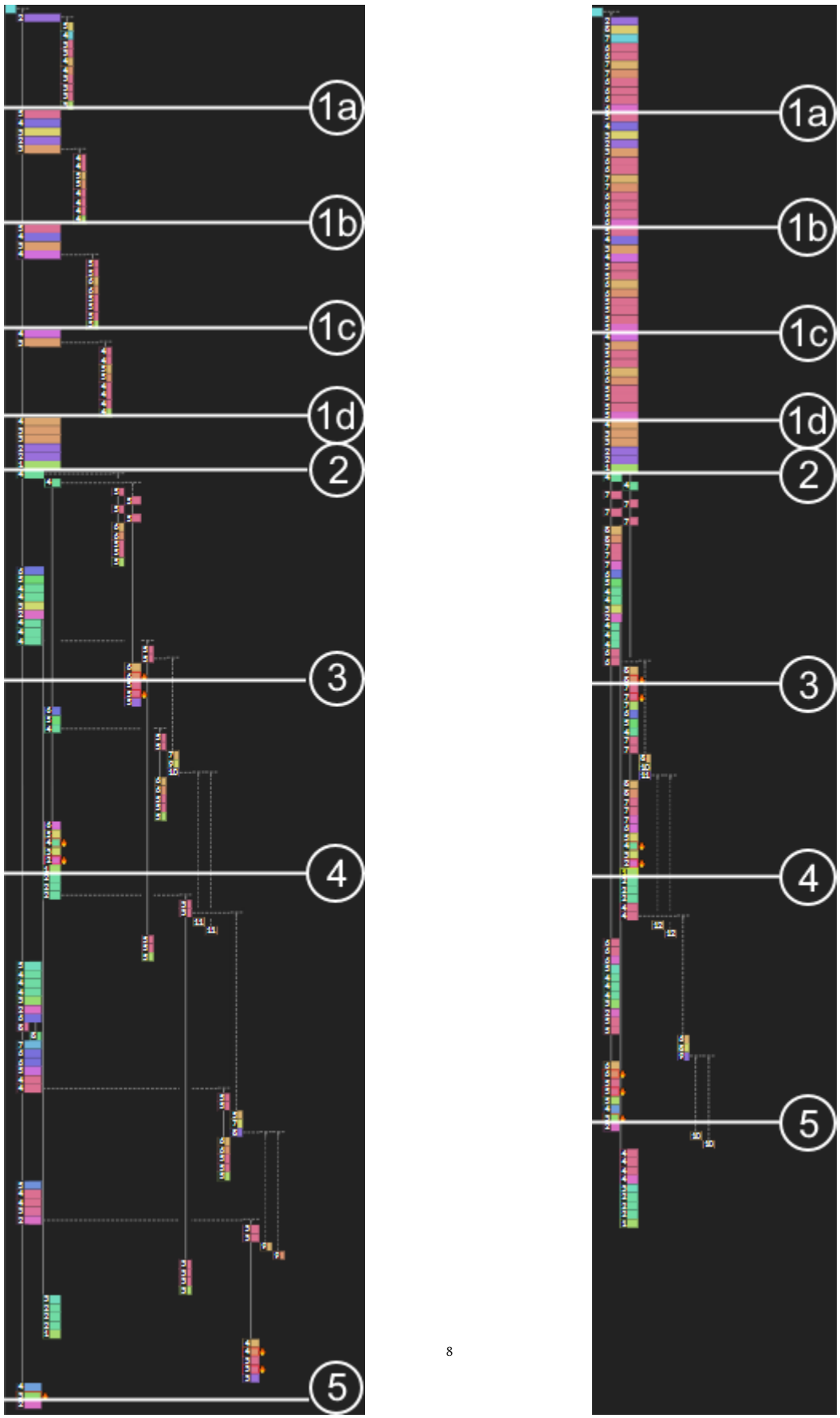
(1) After having already investigated two ACGs and established some of the "landmarks" in the codebase, as discussed in §2.2.2, we started debugging by looking at each of the four errors that are visible in the ACG and their place of occurrence.
(2) We determined that the first error can be ignored. It is internal to some library that catches it.
(3) Already being aware of the "landmarks", we can quickly verify that the second error (line (3) of Fig. 4) belongs to the second `findOrCreate` operation. It represents the first bug.
(4) The third visible error is a re-throw of the first.
(5) The fourth visible error was not reported in the console. Its message is "SQLITE_ERROR: cannot commit - no transaction is active". This hints at a second bug.
(6) We used Dbux's other tools, complemented by the ACG to help with orientation, recall and control flow comprehension, to verify that this is indeed another bug: the second operation's failure lead to the first operation's transaction getting "cleaned up". Our findings are detailed in our bug report[6].

This time, the ACG provided the same two benefits it did before: (1) It, again, provided a small set of feasible starting points for our

[5] https://github.com/sequelize/sequelize/issues/1831
[6] https://github.com/sequelize/sequelize/issues/13554

investigation: the four error CGRs (with flame icons next to them). (2) It also helped with orientation.

In this case study, we experienced three further benefits that the ACG brings to the debugging process, that were not as obvious previously: specifically, it allowed us to (3) easily find and reason about an unhandled rejection that did not get reported. Furthermore, the ACG aided (4) **recall** and (5) **control flow comprehension**. (3) If an error occurs that was not handled properly, the JS engine should report it. However, that does not happen in some rare circumstances. E.g. the second error (Debugging Step 5) is not reported because it is the second rejection from a promise wrapped with `Promise.all`. This is one of multiple fallacies of `Promise.all`. The ACG helped us quickly see the error clearly, contextualize it and even find out why it was not reported. (4) When investigating and jumping between different places in the code, having access to a spatial layout of the entire execution makes it significantly easier to remember places we visited in the code, when compared to the traditional alternatives of memorizing symbol names, file names and line numbers. (5) The ACG's ability to uncover the connections between the CGRs and their asynchronous events eases one major source of frustration of the investigative process. Being able to not only see, but also navigate directly to the code that created, triggered or is otherwise related to, each CGR makes navigating complex asynchronous control flows a much more straight-forward and interactive endeavor. This is also the reason we, with relative ease, were able to identify and recognize important "landmarks", as shown in Fig. 4 and explained in §2.2.2. These "landmarks" helped us contextualize the code that we investigated, every step of the way, and allowed us to answer crucial high-level questions, such as: Which query or operation does this piece of code, execution path or value belong to? Where is it in relation to which errors?



(a) sequelize(1)

(b) sequelize(2)

Figure 4: The ACGs of the two different sequelize implementations. The two versions observe the same semantics. Both have the same bug. Both show seven error indicators (little flame icons) next to a node that executed a throw statement or had an

2.3 More Result ACGs

In the following, we list depictions of ACGs of the remaining nine projects that we had to omit from the article for brevity. Contrast was enhanced for clarity.

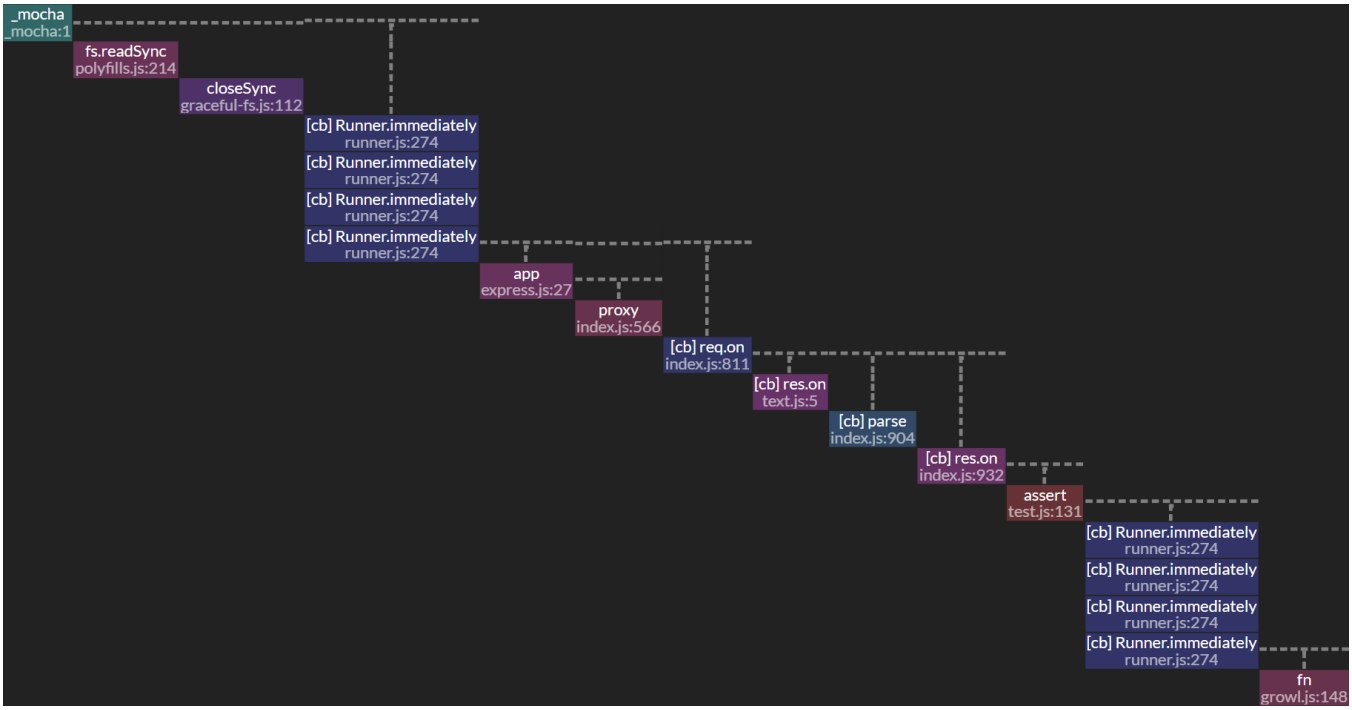


Figure 5: express (`detail` disabled)

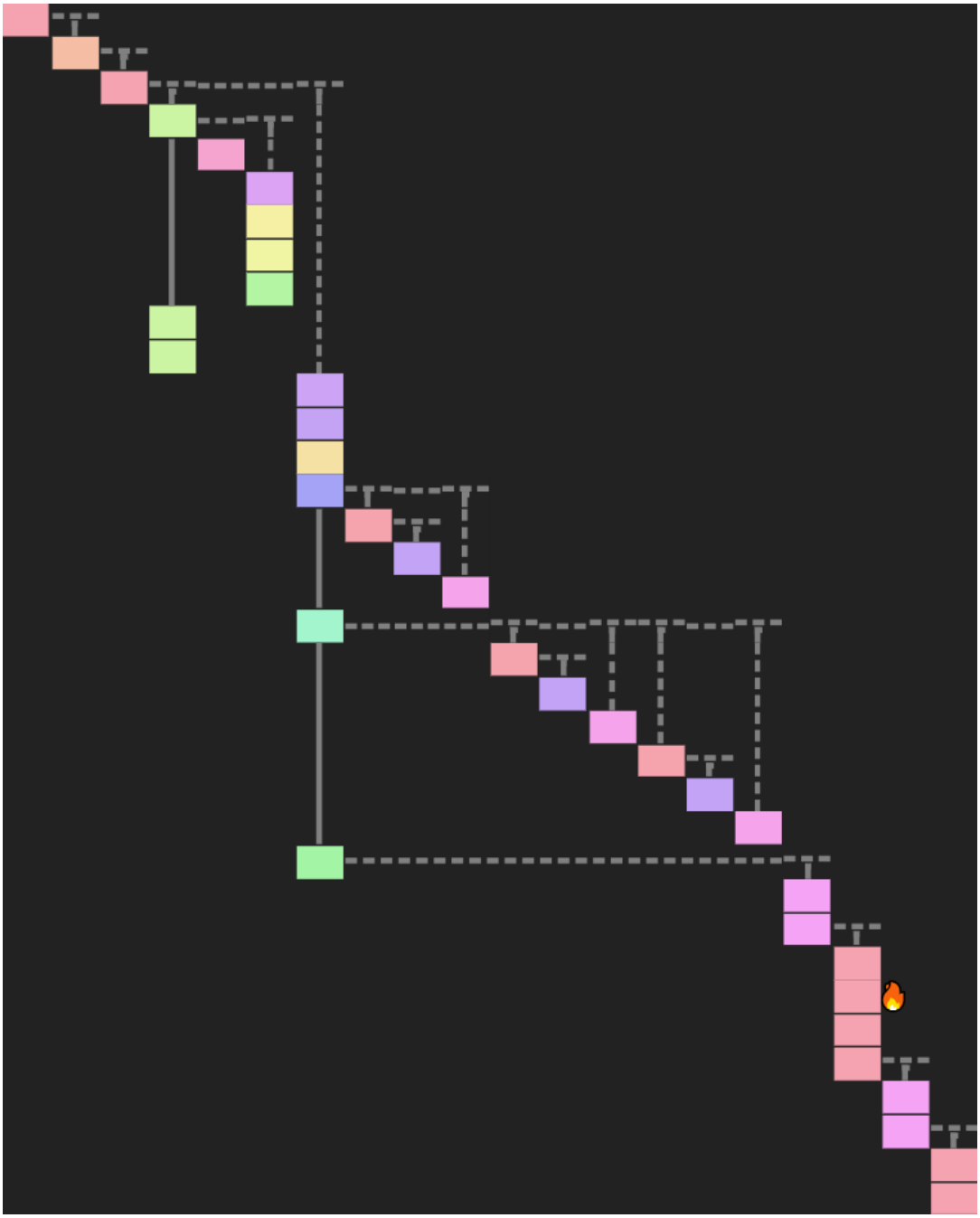


Figure 6: hexo (`detail` disabled)

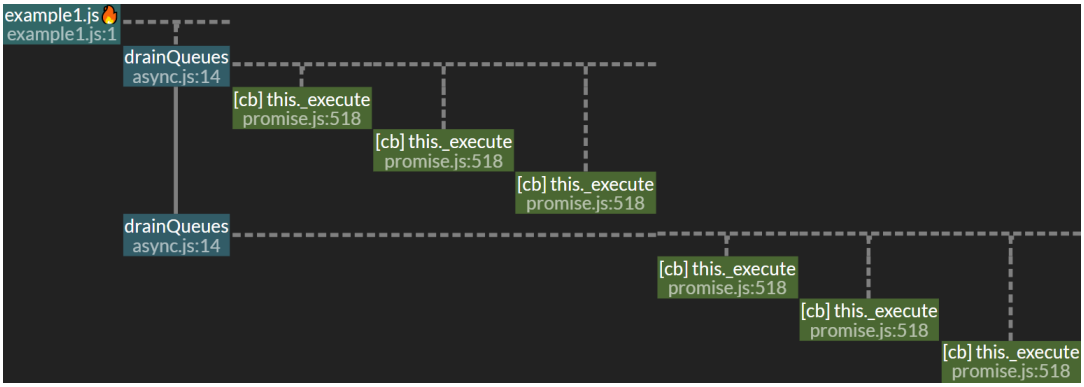


Figure 7: bluebird (`detail` enabled)

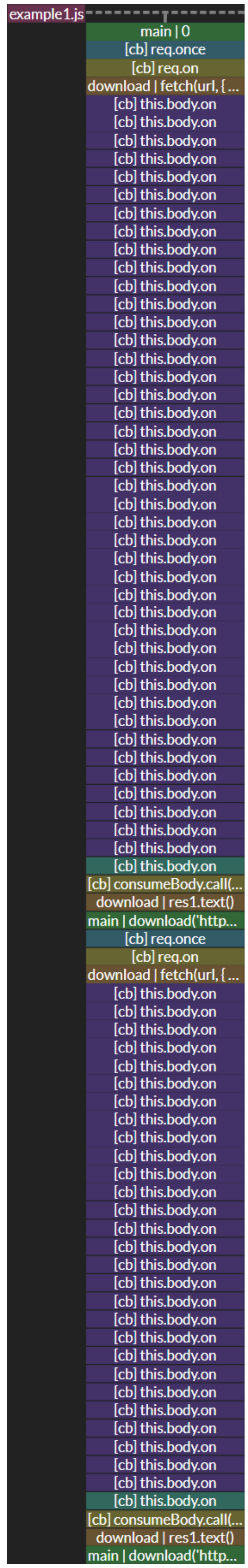


Figure 8: node-fetch (`detail` enabled)

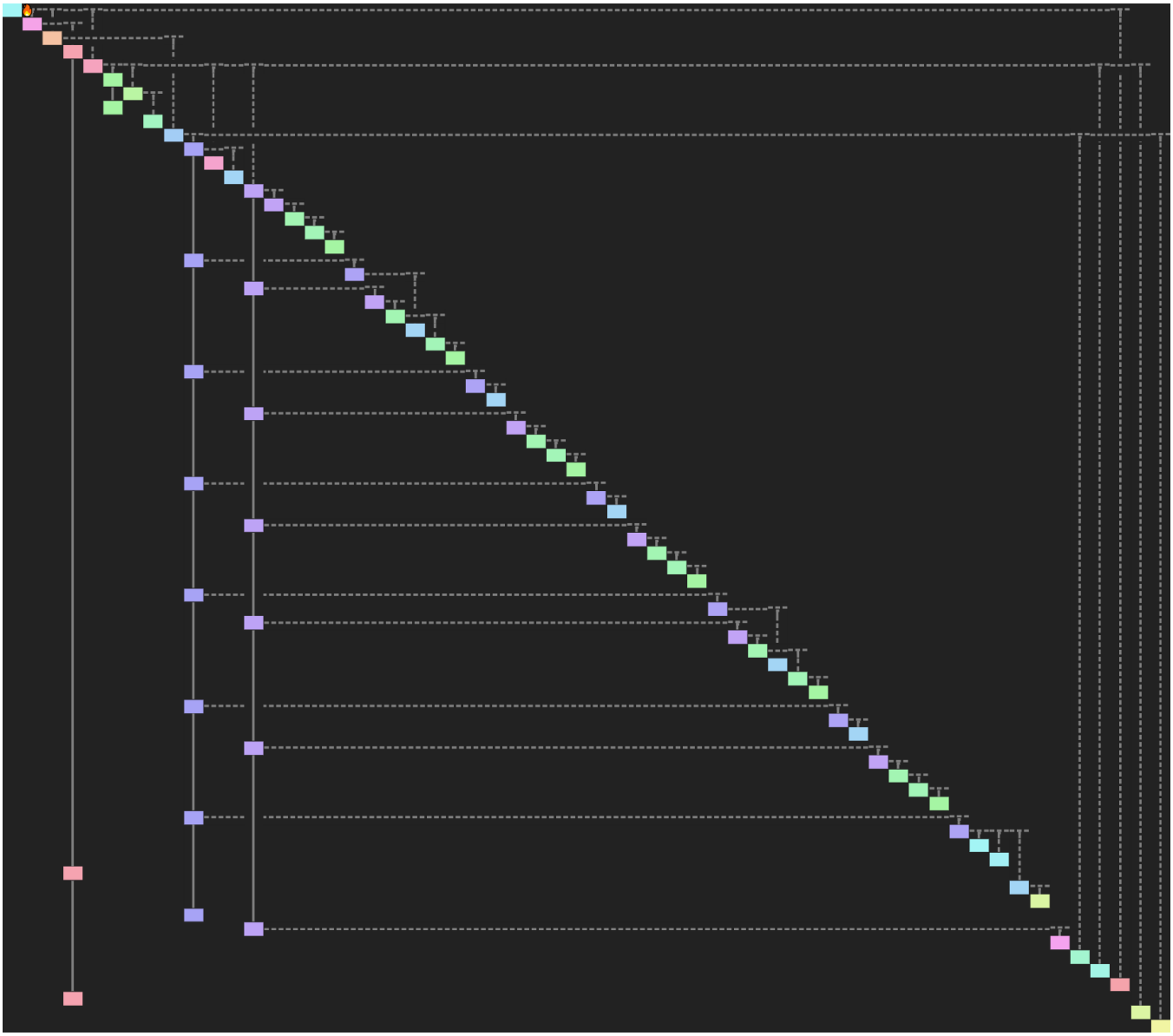


Figure 9: socket.io (`detail` disabled)

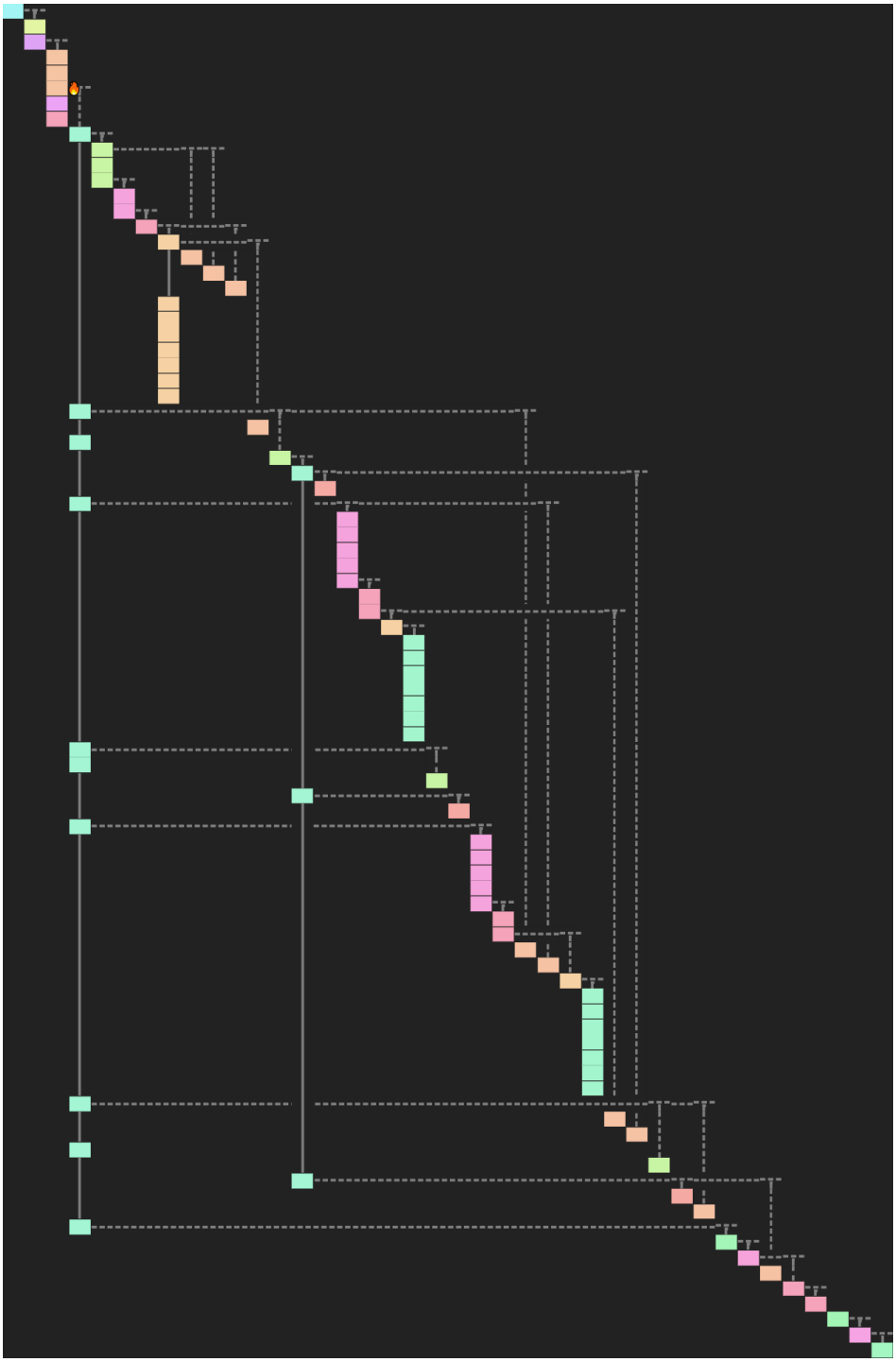


Figure 10: webpack (`detail` disabled)

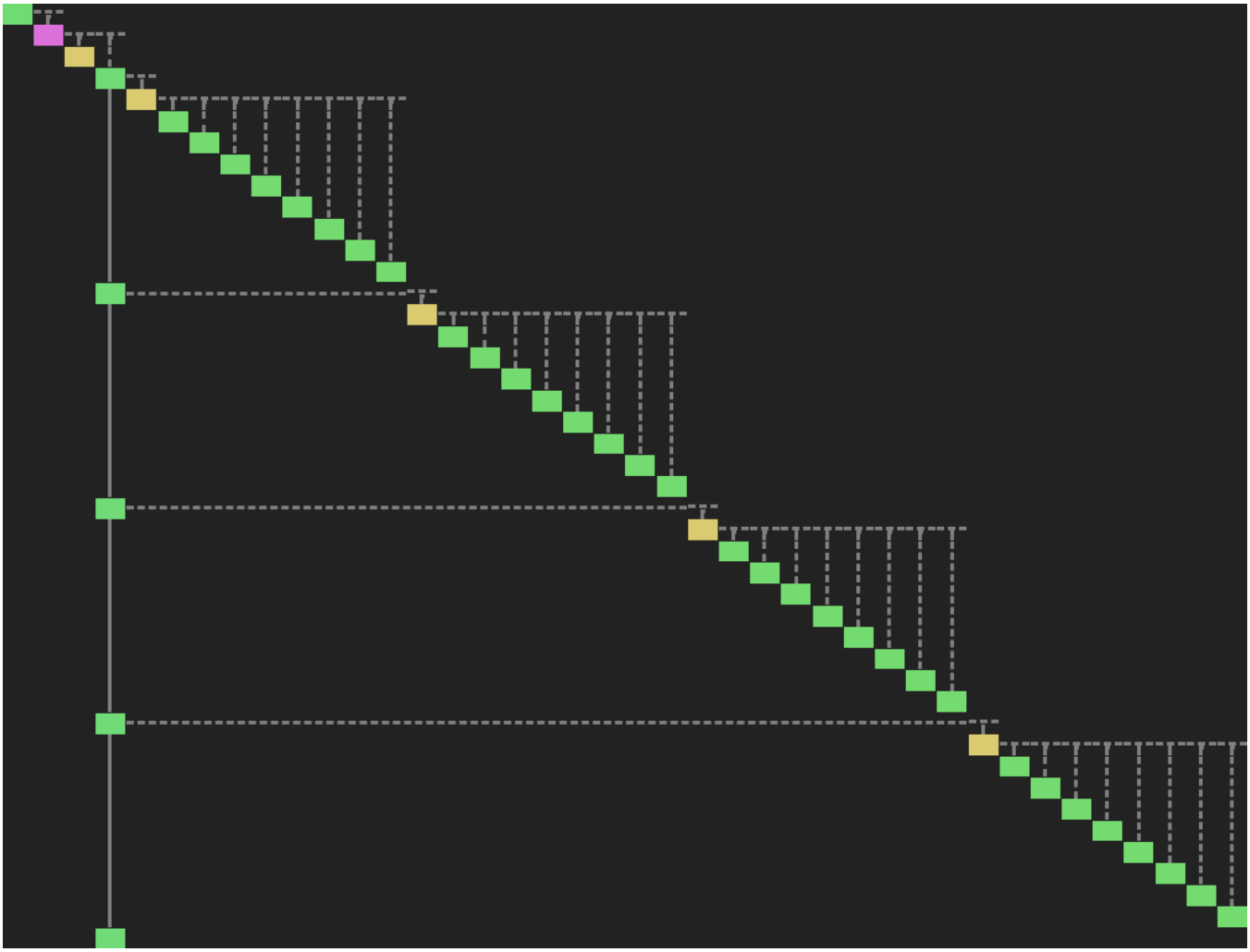


Figure 11: 2048 (`detail` disabled)

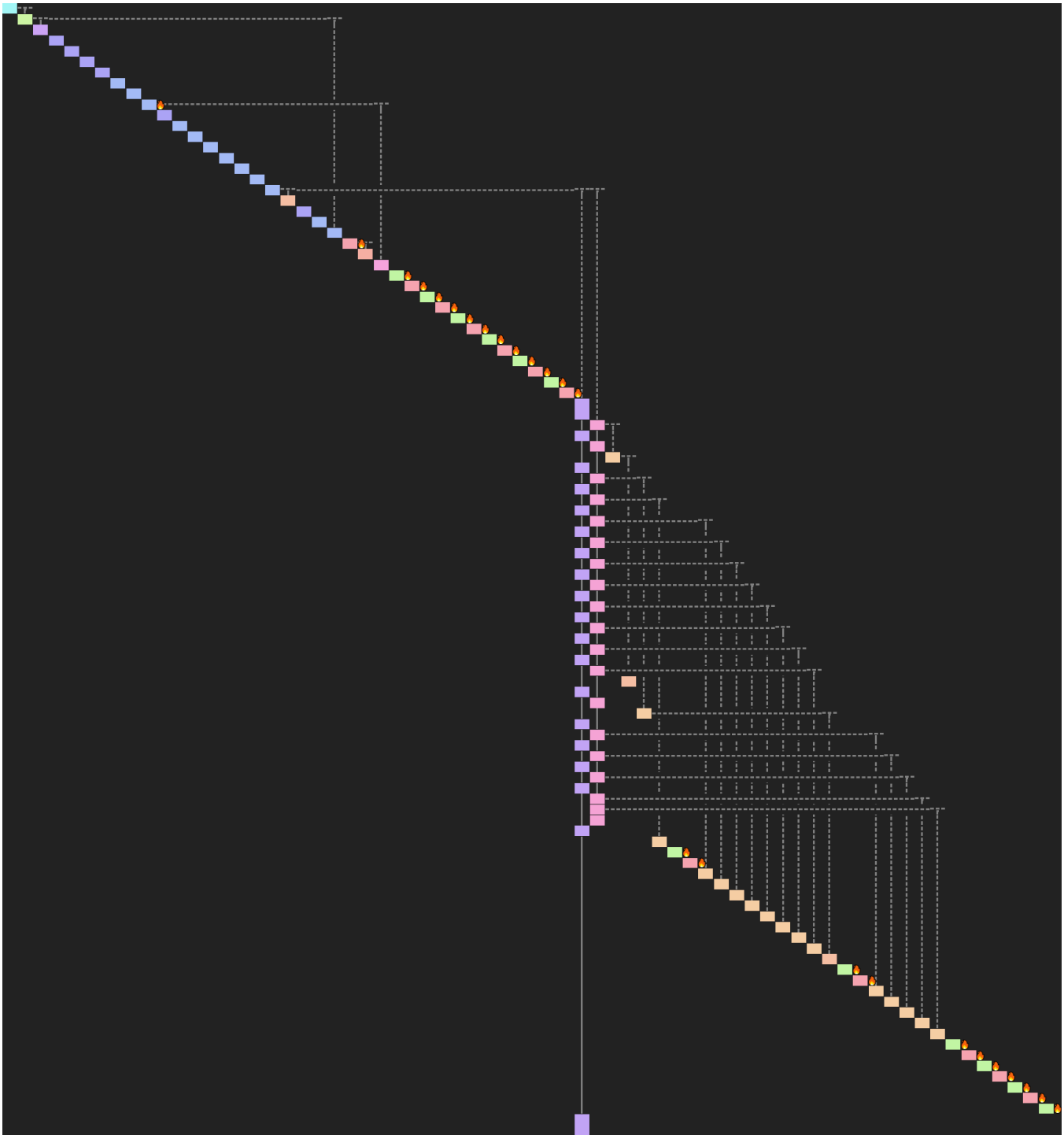


Figure 12: Editor.md (`detail` disabled)

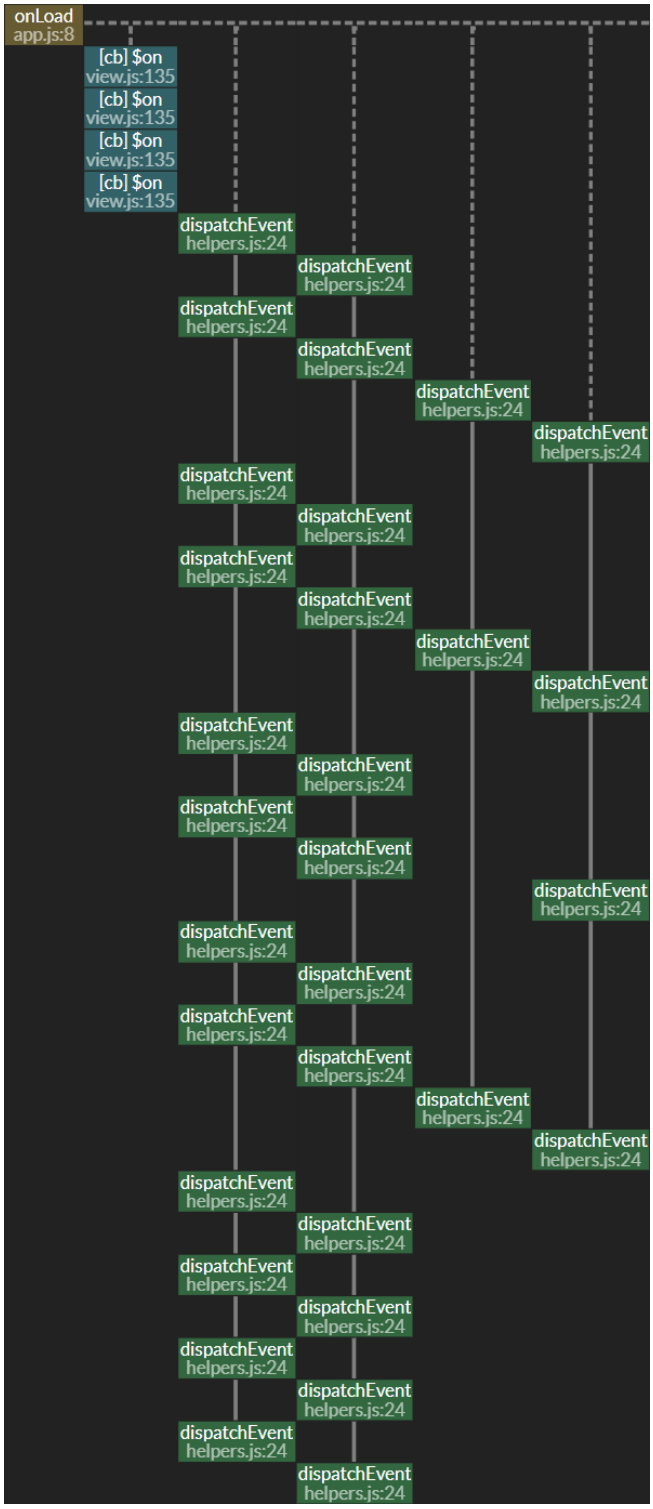


Figure 13: todomvc (`detail` enabled)

3 CONCLUSION

In this work, we provided supplementary material to aid understanding of our original work "An Asynchronous Call Graph for JavaScript". We conclude that in this first exploratory study, the ACG has proven an invaluable addition to debugging and understanding inherently asynchronous codebases. We found that it supported the debugging process in five distinct ways:

(1) **Provide a concise set of debugging entry points.**
(2) **Aid orientation.**
(3) **Visualize errors, even if they were not reported.**
(4) **Aid recall.**
(5) **Aid control flow comprehension.**

Navigating and reading a complex ACG requires more skill than a smaller one. We hypothesize that such skill can be practiced effectively, and not only makes it easier to use the tool, but can also assist the developer in thinking about and comprehending even complex asynchronous control flow of their applications.

A real-world analogy concludes this report: interactive debugging in an unknown codebase is like finding a street address in some city, inside a country one has never been to before, but traditionally without a map. The ACG's ability to reveal the current CGR and its relationship to other CGRs is akin to seeing and contextualizing the location of the "city", in relationship to other "cities" on a map.

In the future, we aim to discuss the ACG's algorithms and its implementation in detail, as well as analysis of the ACG's behavior and limitations when used on real-world projects, including explanations of some of the features seen in §2.3. We also plan to provide several extensions to the ACG in order to address other important relationships between CGRs: capture synchronization between multiple concurrent control flows, provide a proper definition of an Asynchronous Call Stack (ACS), propose possible uses of the ACG for data race detection and experiment with improvements to callback CHAIN heuristics. Finally, user studies will be necessary to more properly evaluate real-world utility of the ACG, and Dbux in general. We look forward to quantifiably verify the preliminary results and hypotheses of this first exploratory self-study.

REFERENCES

[1] Ecma International. 2021. *ECMAScript® 2017 Language Specification.* Retrieved 2/2022 from https://262.ecma-international.org/8.0/