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 "An Asynchronous Call Graph for JavaScript" article. It provides extra background information and showcases results. Last modified: 2021/10/27.

Note to the reviewer: we discovered and elaborate on issues that snuck into the submission draft of the paper (mostly in §5).

CCS CONCEPTS

• **Software and its engineering → Concurrent programming structures**.

ACM Reference Format:
Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2021. Technical Report: An Asynchronous Call Graph for JavaScript. In *Proceedings of* . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 A BRIEF INTRODUCTION TO DBUX

1.1 Architecture

Dbux[3] has four applications and several supplementary modules, as depicted in Fig. 1. Several shared modules, such as the `dbux-common` modules, are not shown.

Dbux's three stages *instrument*, *runtime* and *post-processing* are implemented in four collaborating applications:

- `dbux-babel-plugin` instruments the target application and injects the `dbux-runtime`. It requires to be run with Babel[1].
- `dbux-runtime` records the target application's execution trace and streams it to a server in real time.
- `dbux-code` is a one-click-installable extension to VSCode, available on the VSCode Marketplace, complete with extensive documentation. It also has prepared several real-world projects, bugs and experiments to try it out on. Upon activation, it starts a server to wait for the execution data produced by `dbux-runtime`. When received, it *post-processes* it with the help of the `dbux-data` module before presenting it to the user. Data is processed and presented as soon as it is received, meaning that applications can be debugged while they are still running.
- `dbux-cli` is to Dbux, what "nyc" is to the coverage reporter Istanbul[1], that is: a convenient command line tool that makes it easier for developers to execute a JS application with Dbux enabled, without having to prepare a build pipeline. Instead, it uses a modified version of `@babel/register`[2] to inject `dbux-babel-plugin` on the fly.

1.2 Call Graph Assembly

NOTE: In the following, we refer to the "dynamic call graph" just as "call graph". Dbux does not have a static call graph.

We model the call graph as follows:

(1) We refer to a call graph node, that is the recorded execution of a file or function, as an "executionContext", or **context** for short. Given a function f, we denote the context that represents the i'th execution of some function f as f_i.
(2) Edges represent the caller - callee relationship. For uninterruptible functions: if during it's i'th execution, f calls some function g, 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.
(3) Any function execution f_i is considered a Call Graph Root (CGR), if it has no parent caller. This implies that the function was either directly invoked by the JavaScript engine's event queue, or the first invoked function was not recorded.
(4) Traditionally the above rule set was sufficient to build a JavaScript dynamic call graph. However, ES2017[2] 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, a **virtual context** h_i^1 is added as a child to h_i. 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. Once `await` has concluded, that is, after the promise has settled, execution of h_i continues. At this point, our instrumentation adds a new virtual context h_i^k. That context represents the asynchronous continuation of the interrupted real context h_i at a later point in time. We assume that each virtual context $h_i^k, i > 1$ is also a CGR.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://istanbul.js.org/
[2] https://babeljs.io/docs/en/babel-register

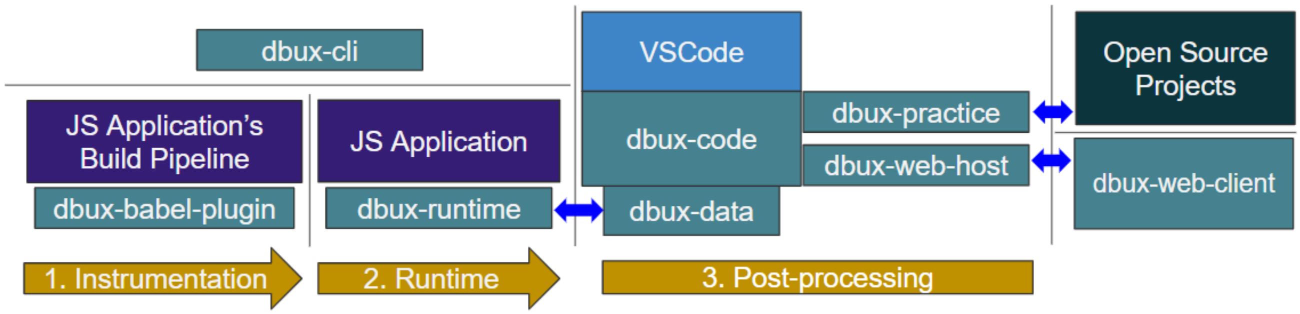

Figure 1: Dbux Architecture.

In general, all events, contexts and roots are ordered by time of occurrence. Dbux's synchronous call graph implementation renders all CGRs linearly in that order.

1.3 Developer Survey

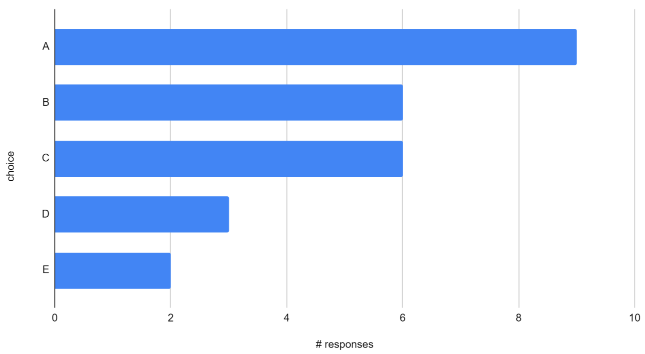

Figure 2: Survey Results: What type of programming problems are the most difficult to deal with? (A) Asynchronous behavior (setTimeout; setInterval; Process.next; promise; async/await etc.) (B) Third-party APIs (e.g. Node API, Browser API, other people's libraries, modules etc.) (C) Programming logic (D) Syntax (E) Events.

During a workshop in summer 2020 that introduced Dbux to 20 TAs of a local JavaScript Bootcamp provider, we asked the participants what type of bugs they found most difficult to deal with. A total of 10 participants filled out our survey. The top choice for "programming problems" (multiple choice) was "asynchronous behavior" with 9 votes, while the second place only received 6.

2 ASYNCHRONOUS SEMANTICS PRIMER

Fig. 3 illustrates the three types of Asynchronous Events (AE). In all three cases, the resulting Asynchronous Call Graph (ACG) should feature three nodes, connected by two CHAINs.

Below are several illustrations of asynchronous programs and their expected conceptual ACG.

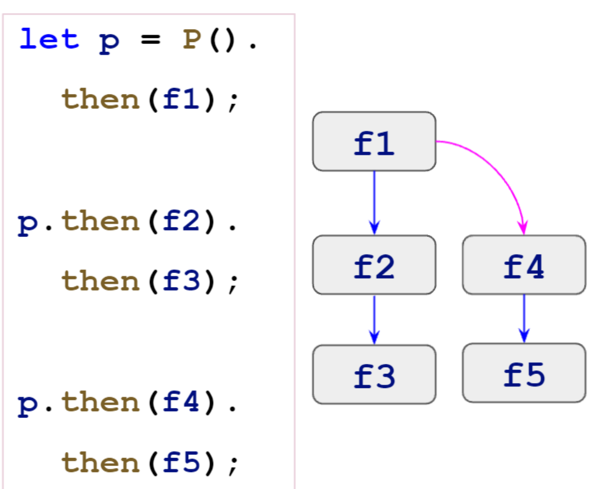

Figure 4: Promises (CHAIN vs. FORK)

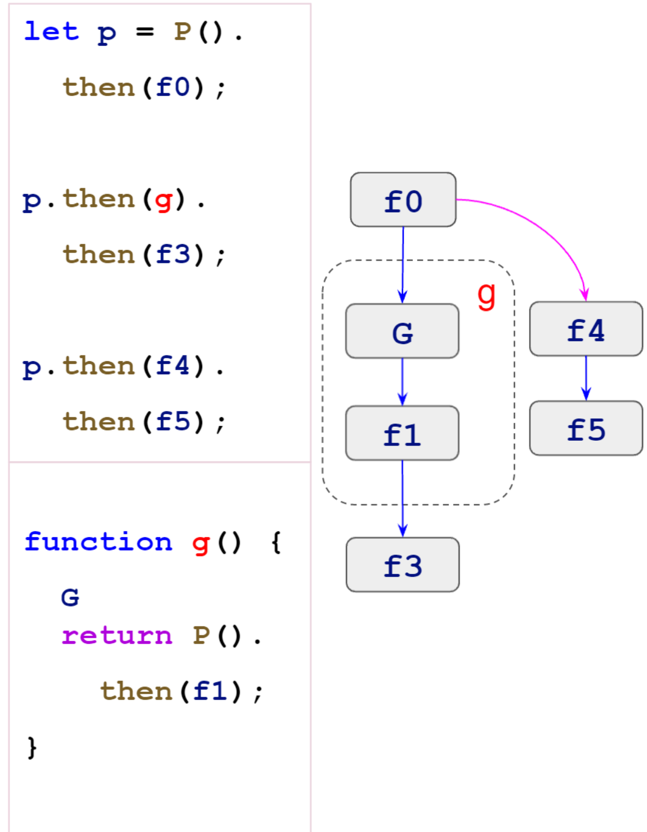

Figure 5: Nested Promises (CHAIN vs. FORK)

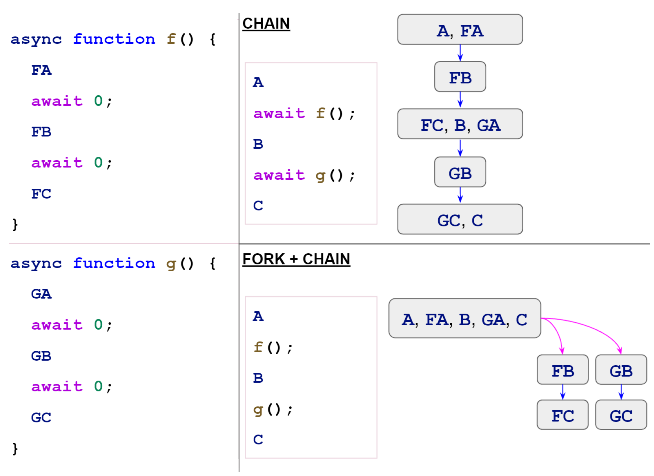

Figure 6: AWAIT (CHAIN vs. FORK)

2.1 Promise Creation Semantics

In JavaScript, promises can be created in four ways. Somewhat counter-intuitively, (i), (ii) and (iii) do *not* cause an asynchronous event on their own. However, all of them can nest promises. Most of these nesting relationships are captured by the ACG's PromiseLinks:

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

The (i) Promise constructor takes an executor function which in turn is provided two parameters: the `resolve` and `reject` functions which are to be called to fulfill the promise. The executor function is called synchronously from the constructor. The Promise constructor is commonly used to wrap asynchronous callbacks into promises. This process is commonly referred to as "promisification".

(ii) `Promise.resolve(x)` and `Promise.reject(x)` are equivalent to using the (i) Promise constructor and synchronously calling `resolve` or `reject` respectively. `Promise.all` and `Promise.race` work similar to `resolve` but allow nesting multiple promises. They fulfil once all or the first nested promise fulfil, respectively.

When (iii) an async function is called, the runtime environment creates a new promise. Its call expression value is set to that promise. Async functions execute synchronously until the first `await` is encountered. This means that if an async function concluded without explicitly invoking an `await` expression or any of the three other types of events, it does not trigger an asynchronous event. Await expressions can nest promises. Furthermore, promises can be nested by returning them from an async function.

(iv) Promise chaining (`then`, `catch`, `finally`) allow for promise nesting by returning a promise from their respective fulfillment and rejection handler callbacks.

3 CONCURRENT DATA FLOW: RESULTS

These are the results from the “Concurrent Data Flow” extension on the three producer-consumer problems (§4.1):

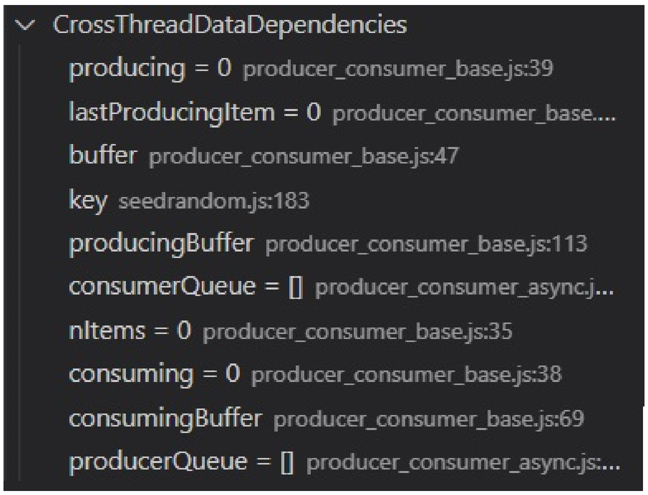

Figure 7: Version 1: Async Function Implementation

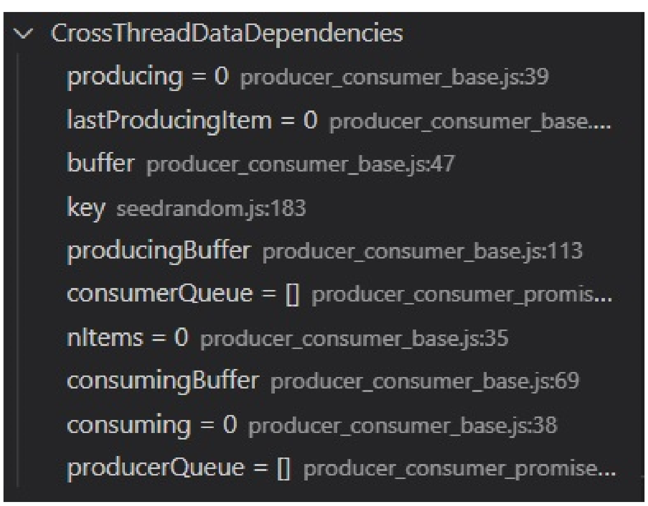

Figure 8: Version 2: Promise Implementation

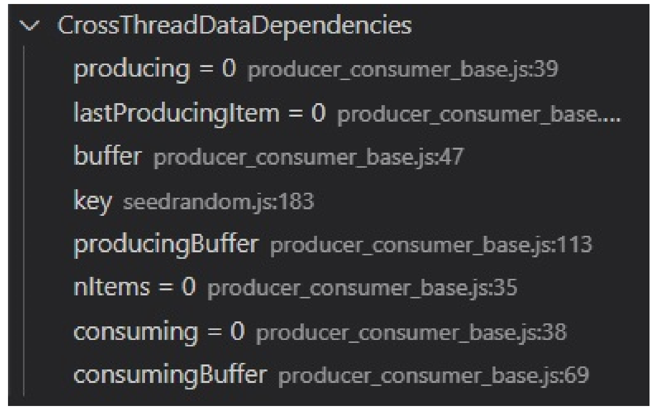

Figure 9: Version 3: Callback Implementation

4 PROJECT RESULTS

4.1 Project ACS

Fig. 10 shows the asynchronous call stack of the sequelize bug. The stack prominently features the sequelize API call that caused it: `findOrCreateCall`. For contrast: `err.stack` is empty. `formatError` returns a new `Error` object with its `stack` only containing the functions within its current CGR, up to `Query.afterExecute`.

4.2 Project ACGs

In the following, we list the raw ACG results of the nine projects that we had to omit from the article for brevity (enhanced for contrast).

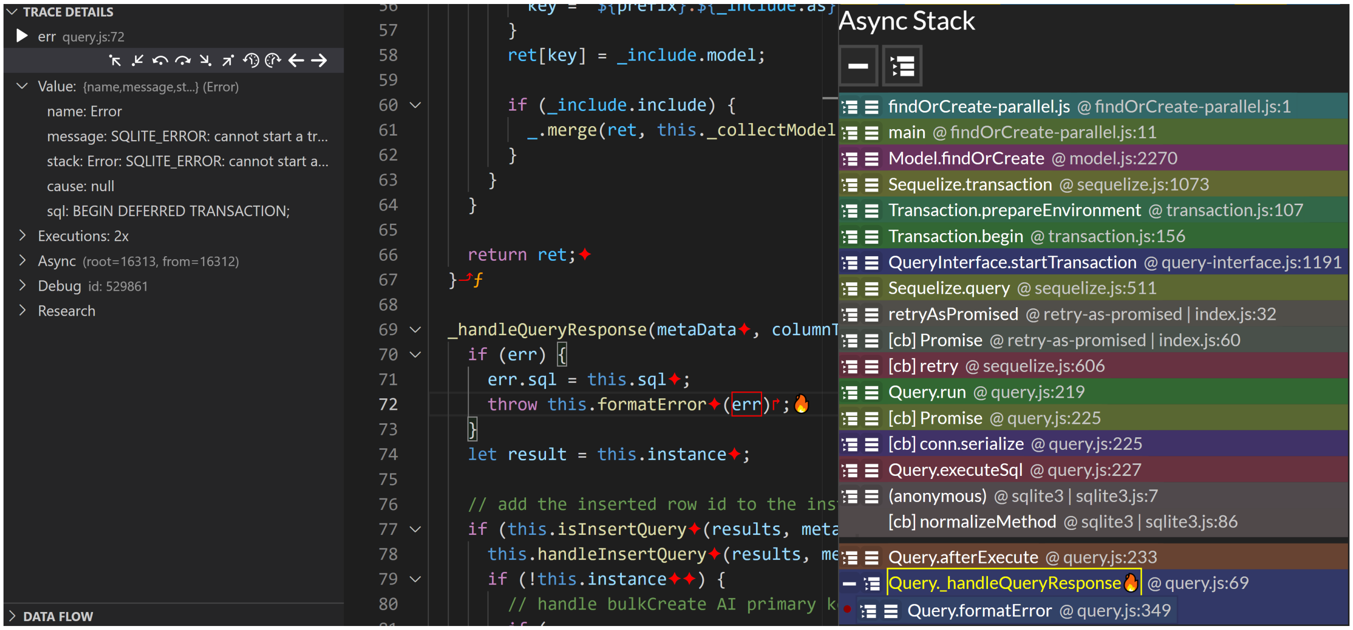

Figure 10: The sequelize ACS when the first thrown error is selected in the code.

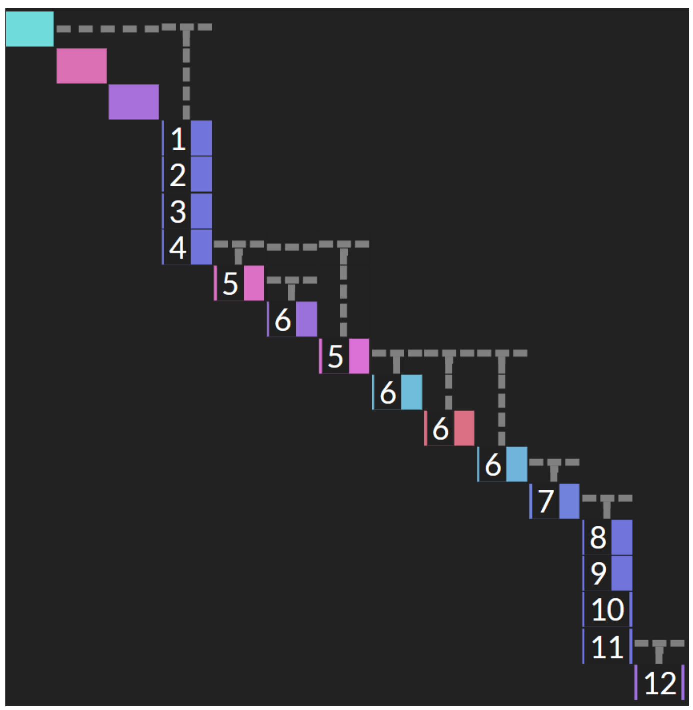

Figure 11: express

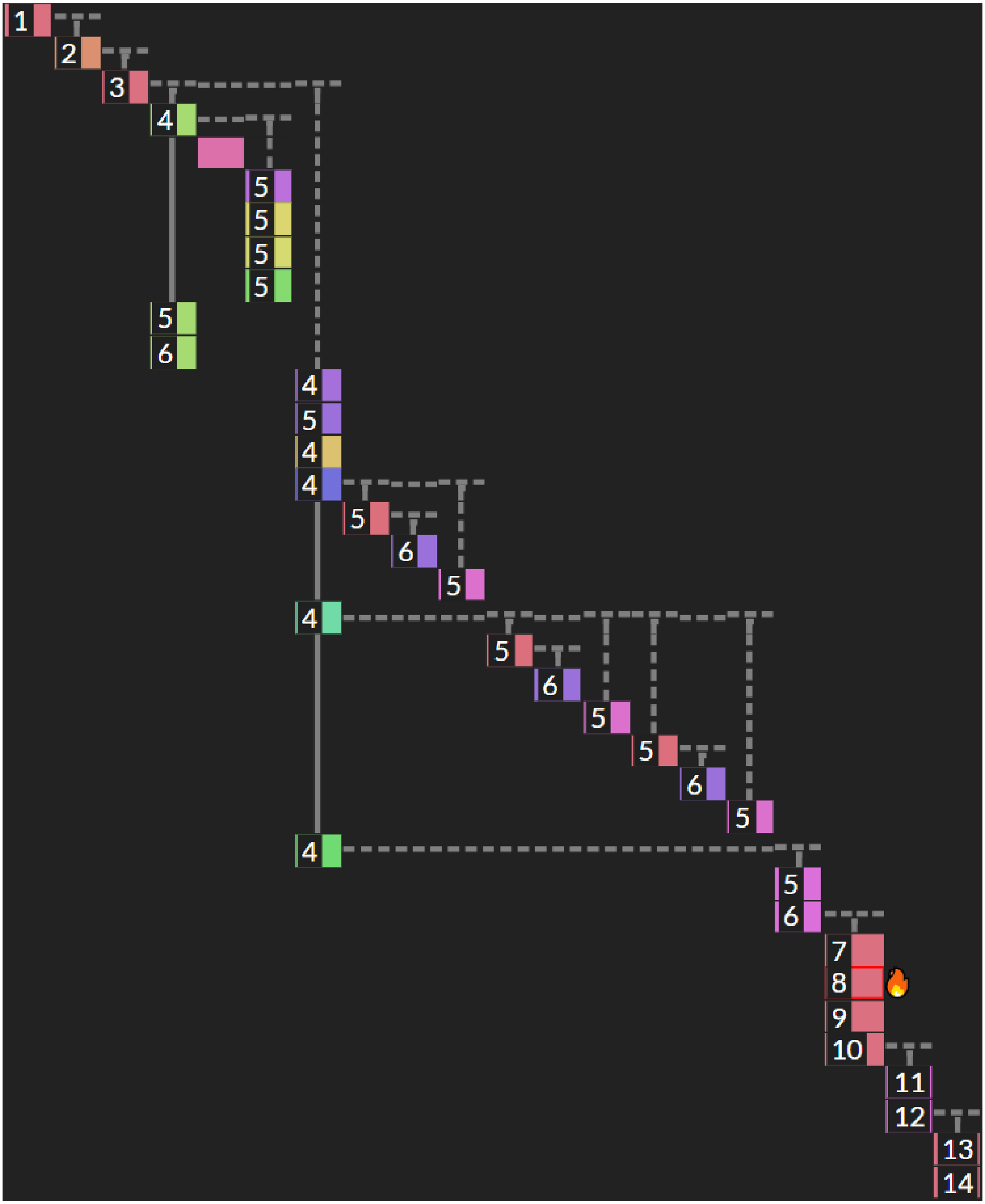

Figure 12: hexo

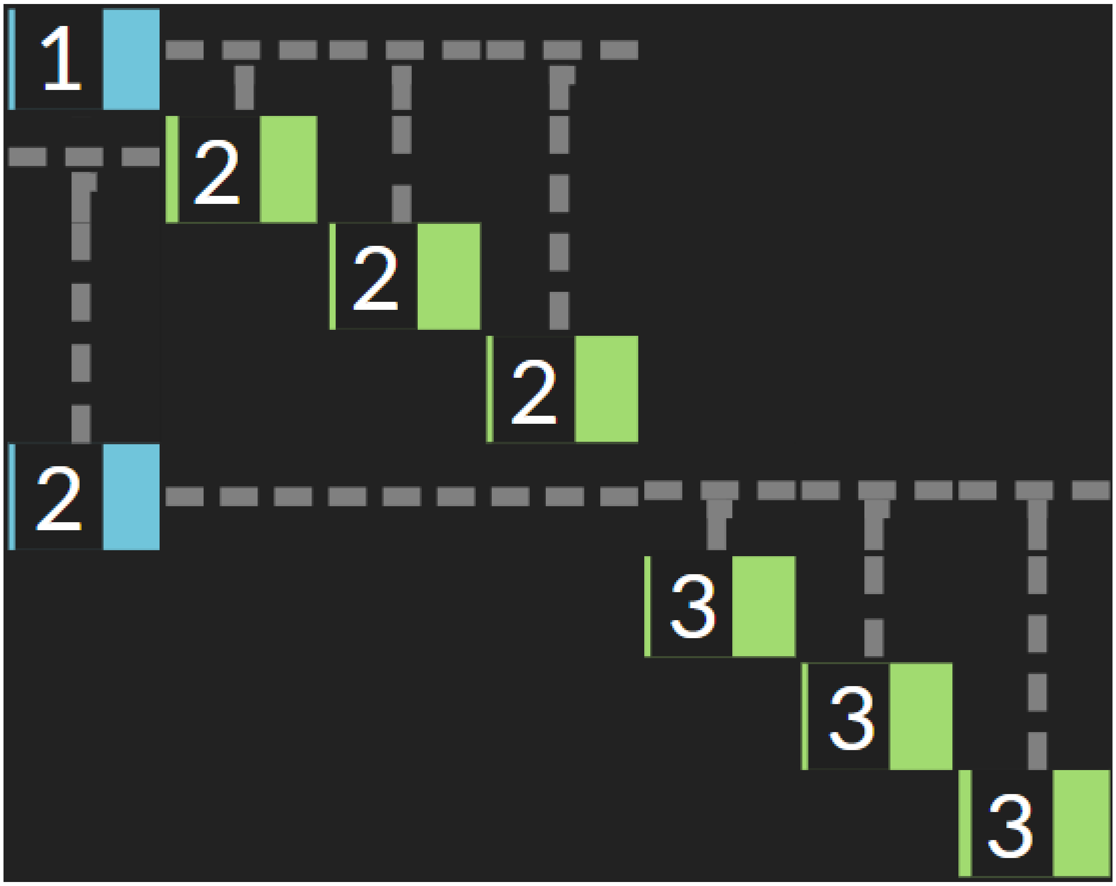

Figure 13: bluebird

Figure 14: node-fetch

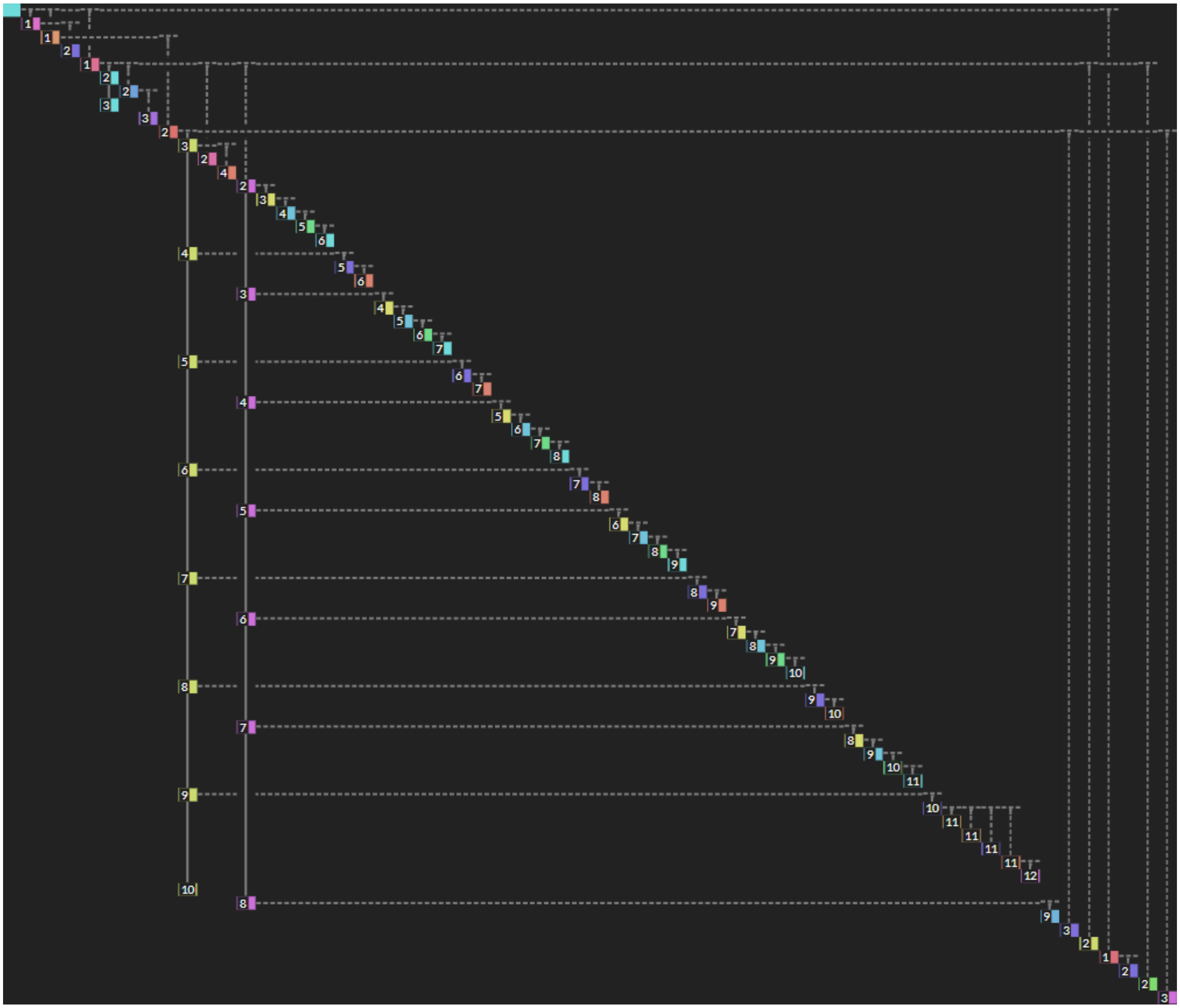

Figure 15: socket.io

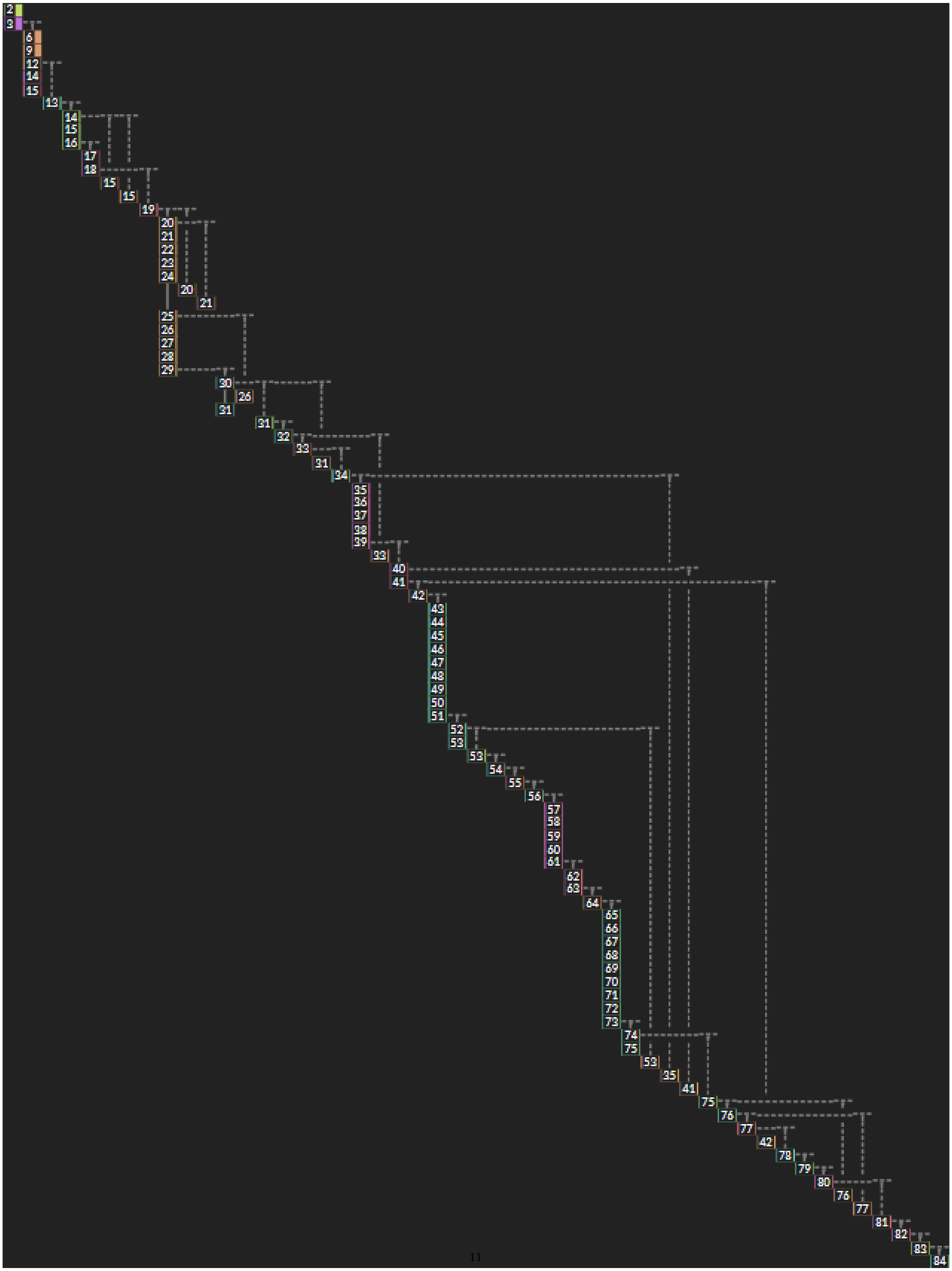

Figure 16: webpack

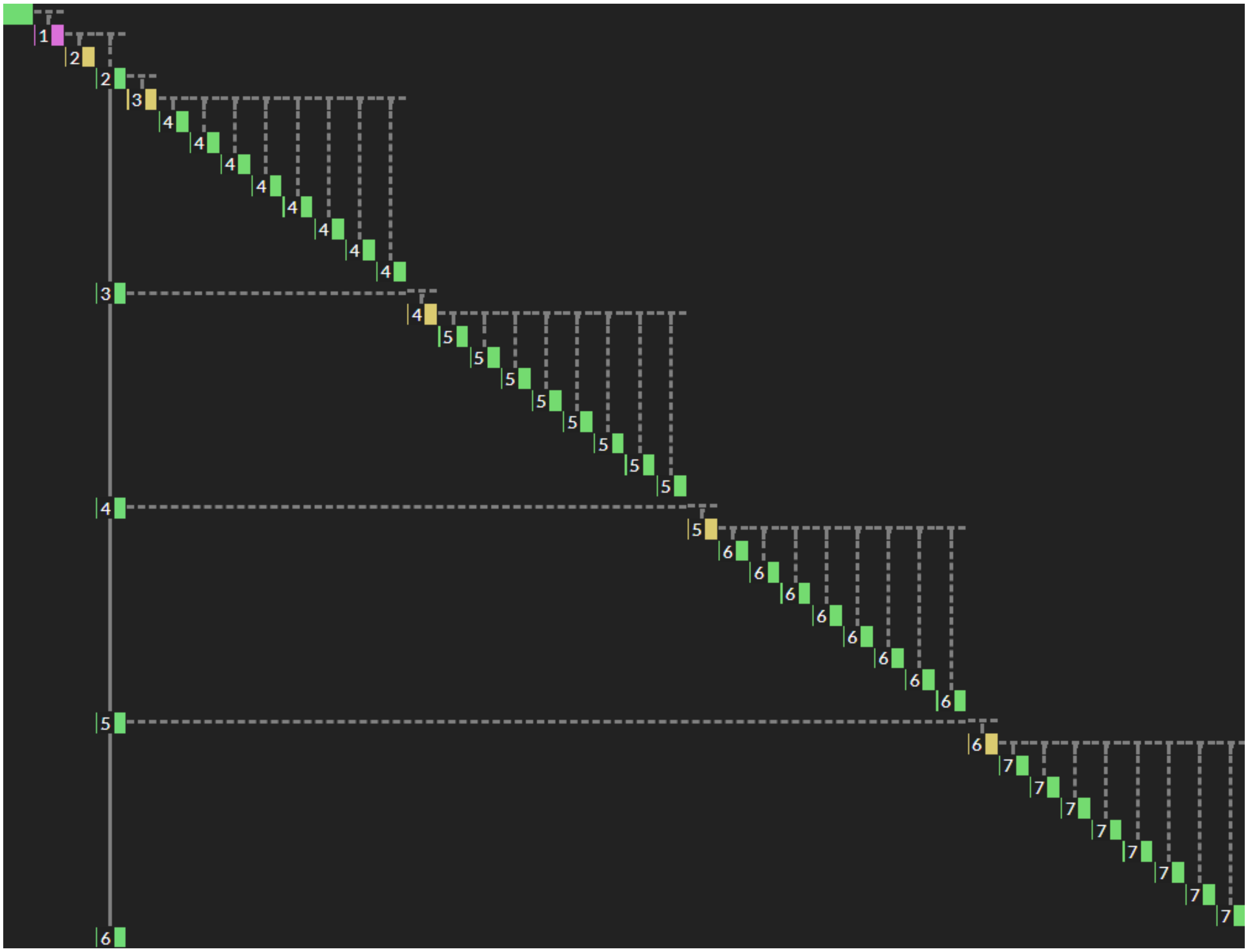

Figure 17: 2048

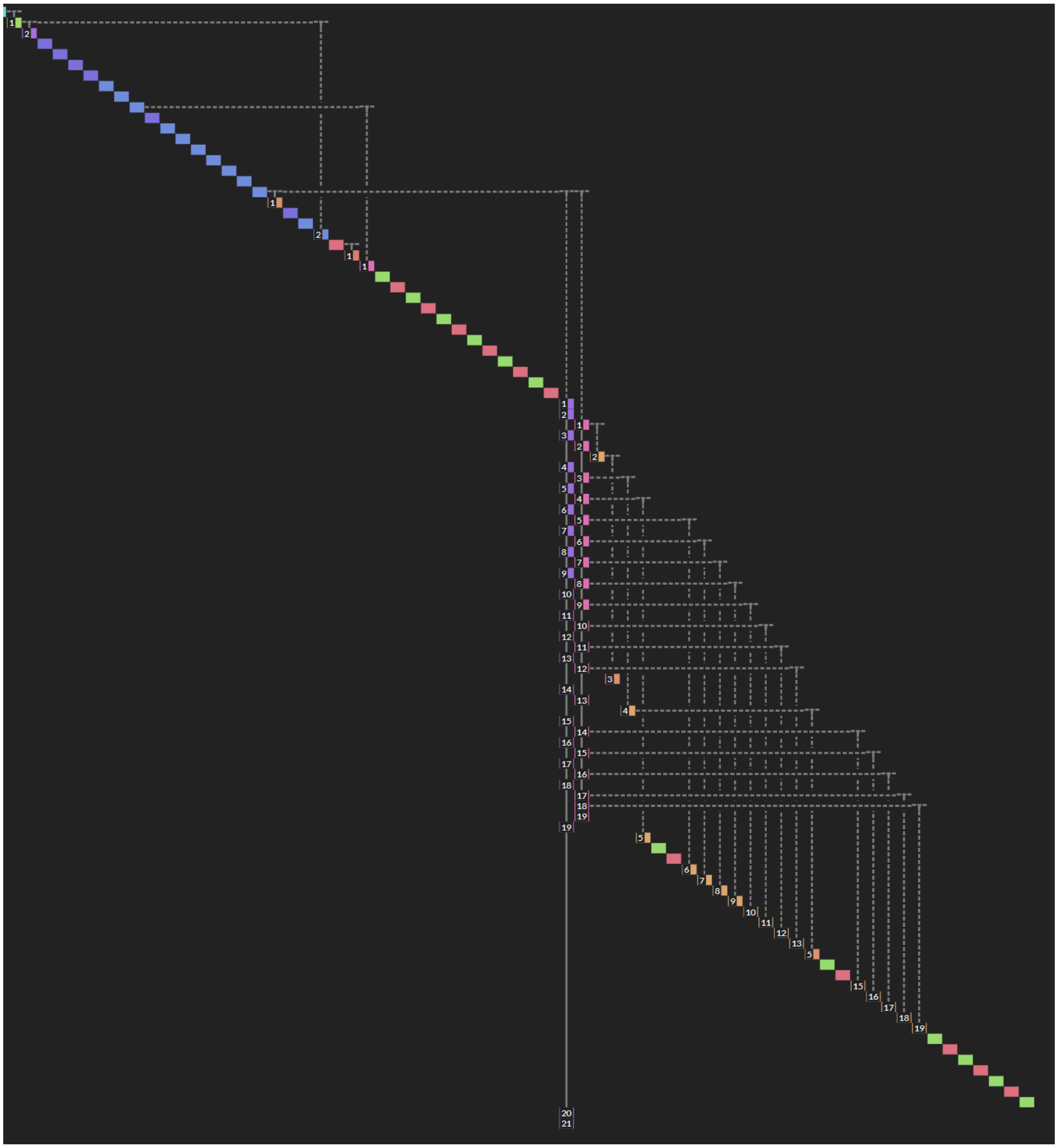

Figure 18: Editor.md

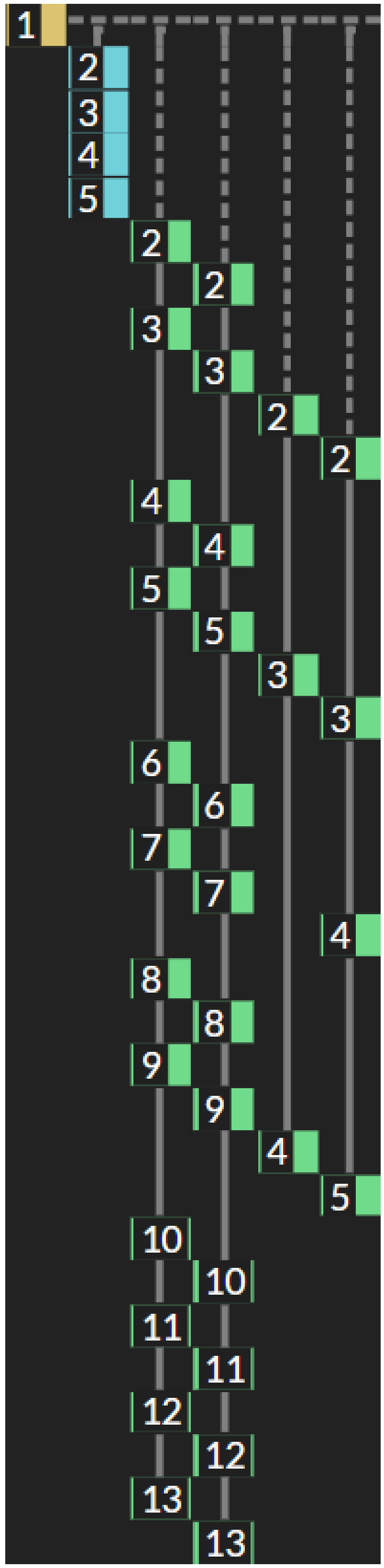

Figure 19: todomvc

5 CORRECTIONS

This section lists several corrections of the submission draft (submitted 2021/10/15). This section will be removed in the final version. Corrections are typeset in *italic, red*.

5.1 Data Corrections

L78 incorrectly states: "10 out of 11 respondents". Correction: *9 out of 10*.

Table 1 (page 7) requires four corrections. (i) A proper definition of "accuracy" is missing. (ii) It should be mentioned that the fork count (F) does not include orphans (forks without a parent). (iii) Orphan nodes should be properly accounted for. (iv) The Editor.md sample required a re-run due to a bug.

The corrected table follows:

Table 1: Projects sequelize and async-js were each ran in two modified versions. *Trace*: number of recorded trace events. A: AWAIT events. T: THEN events. CB: asynchronous callback events. C: chains. F: forks *and multi-chains (excl. orphans)*. *O: orphan CGRs (excl. file load orphans).* TT: total threads *= F+O*. RT: real threads. Acc: accuracy *= RT/TT*. N: average nesting depth of all CGRs *(excl. file load orphans)*.

name	*Trace*	A	T	CB	C	F	*O*	TT	RT	Acc	N
express	240,475	0	0	16	6	10	*2*	*12*	1	*0.08*	6
hexo	909,886	0	13	22	16	*19*	*2*	*21*	2	*0.10*	6
async-js(1)	3,068	5	3	2	4	6	*0*	6	2	0.33	3
async-js(2)	3,068	5	3	2	4	6	*0*	6	2	0.33	3
bluebird	24,350	0	3	6	1	8	*0*	8	3	0.38	2
node-fetch	13,991	7	2	82	90	1	*0*	1	1	1.00	3
sequelize(1)	554,586	123	11	27	140	*21*	*0*	21	2	0.10	4
sequelize(2)	539,370	104	10	25	130	*9*	*0*	9	2	0.22	5
socket.io	150,009	12	0	61	14	59	*0*	59	1	0.02	6
webpack	1,326,471	7	0	88	46	49	*0*	49	1-3	0.04	44
2048	37,957	0	0	43	4	39	*0*	39	5	0.13	5
Editor.md	*208,937*	0	0	*64*	*38*	*26*	*41*	*67*	*32*	*0.48*	10
todomvc	19,610	0	0	36	30	6	*0*	6	6	1.00	6

Changes in the table are due to the following reasons:

- The text (L748) refers to an order of projects ("first", "next", "last") that did not match the ordering of the table.
- express, hexo and Editor.md had unaccounted orphan nodes.
- F of sequelize now also accounts for MC (multi-chain) nodes (the columns were merged due to space reasons).
- We had to run a new sample for Editor.md due to a bug[3]. Numbers are generally similar to the previous sample because we interacted with it according to the same protocol. However, newly accounted orphan nodes (i) reduced the Acc value and (ii) revealed previously unaccounted "parallel task" and "event stream" type of concurrency patterns which increased RT (real thread count).

We propose further explanation in the text in three places:

- L805 (Sec 5.2): Add *To simplify, we exclude initial file execution CGRs from our concurrency analysis. All such nodes would contribute 1 RT and 1 TT per sample. They currently show up as orphans in their own thread, but we are considering other rendering options, such as moving them all into the first column. Fig. 1 shows a graph with a single file node.*
- L844: change "Those timers are the reason why" → *Those timers are one of the reasons why*.
- L853: Add *A different dimension of inaccuracy is illustrated by Editor.md which observes a high orphan count for two reasons. Firstly, Editor.md registers some callbacks via DOM event handler property assignment (e.g. `el.onload = cb`) which the ACG currently does not identify as schedulers. Secondly, Editor.md uses the `JQuery.bind` function which takes an object-of-callback argument. Since our dynamic callback patcher does not search objects and arrays for callbacks, those callback invocations do not know their scheduler. That leads to multiple orphans being added whenever the user moves the mouse, thus decreasing "Acc".*

5.2 Other Corrections

Furthermore, we (thus far) identified the following places needing corrections:

- L274: "edge" → *event*.
- L487: invalid reference to `f`. → Change to *`p`*.
- L726: "a given CGR `t` is identified as synchronizing against CGR `s` if `s` is `t`'s ancestor" → *a given CGR `t` is identified as synchronizing against CGR `s` if `s` happened before `t`, `s` is not an ancestor of `t`*.
- § 4.1: missing reference to result images in Technical Report, Sec. 3.
- L748: "The next five" → *The next six*.
- L750: "The last four" → *The last three*.
- L1037: "include them in for" → *include them in case of*.

Lastly, we found several places where the text is not incorrect, but can use improvement, such as L478 or the paragraph symbol (§) followed by empty space.

REFERENCES

[1] (accessed in 12/2020). *Babel.* https://babeljs.io/docs/en/index.html

[2] Ecma International. 2021. *ECMAScript® 2017 Language Specification.* Retrieved 10/2021 from https://262.ecma-international.org/8.0/

[3] Dominik Seifert and Michael Wan. 2019. *Dbux.* Retrieved 10/2021 from https://github.com/Domiii/dbux

[3]The runtime did not send all data (only in case of Editor.md). We have since fixed the bug and added data verification.