DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript

Alexi Turcotte
turcotte.al@northeastern.edu
Northeastern University
Boston, MA, USA

Michael D. Shah
mikeshah@northeastern.edu
Northeastern University
Boston, MA, USA

Mark W. Aldrich
mark.aldrich@tufts.edu
Tufts University
Medford, MA, USA

Frank Tip
f.tip@northeastern.edu
Northeastern University
Boston, MA, USA

ABSTRACT

Promises and async/await have become popular mechanisms for implementing asynchronous computations in JavaScript, but despite their popularity, programmers have difficulty using them. This paper identifies 8 anti-patterns in promise-based JavaScript code that are prevalent across popular JavaScript repositories. We present a light-weight static analysis for automatically detecting these anti-patterns. This analysis is embedded in an interactive visualization tool that additionally relies on dynamic analysis to visualize promise lifetimes and instances of anti-patterns executed at run time. By enabling the user to navigate between promises in the visualization and the source code fragments that they originate from, problems and optimization opportunities can be identified.

We implement this approach in a tool called *DrAsync*, and found 2.6K static instances of anti-patterns in 20 popular JavaScript repositories. Upon examination of a subset of these, we found that the majority of problematic code reported by DrAsync could be eliminated through refactoring. Further investigation revealed that, in a few cases, the elimination of anti-patterns reduced the time needed to execute the refactored code fragments. Moreover, DrAsync's visualization of promise lifetimes and relationships provides additional insight into the execution behavior of asynchronous programs and helped identify further optimization opportunities.

KEYWORDS

JavaScript, asynchronous programming, program analysis, visualization

ACM Reference Format:
Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. 2022. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510097

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510097

1 INTRODUCTION

The `async/await` feature [15, Section 15.8] was added to the JavaScript programming language in 2017 to facilitate asynchronous programming with convenient syntax and error handling. Programmers can designate a function as `async` to indicate that it performs an asynchronous computation, and `await`-expressions may be used in these functions to await the result of other asynchronous computations. The JavaScript community has enthusiastically embraced this feature, as it is less error-prone than event-driven programming and syntactically much less cumbersome than the promises feature [15, Section 27.2] on which it builds. However, many JavaScript programmers are still unfamiliar with asynchronous programming, and particularly with `async/await` and how it interacts with promises. As a result, they sometimes produce code creating redundant promises, or code that performs poorly because the ordering of asynchronous computations is constrained unnecessarily [11].

We identify 8 anti-patterns involving the use of promises and `async/await` that commonly occur in JavaScript programs. These anti-patterns reflect designs that are likely to be suboptimal because they may create promises unnecessarily, perform synchronization that is redundant, or cause code to become needlessly complicated. Examples of these anti-patterns include redundant uses of `await`, the use of `await` in loops over arrays, and explicit creation of new promises where none are needed. In many cases, these anti-patterns can be refactored into code that is more concise or more efficient.

We developed a lightweight static analysis to detect these anti-patterns directly in source code, and implemented this analysis as a set of CodeQL queries [4, 13]. Furthermore, to help programmers understand the run-time impact of the anti-patterns, we developed *DrAsync*, a profiling tool that visualizes the lifetime of the promises created by an application, and that highlights the run-time instances of each anti-pattern. This enables programmers to focus their attention on anti-patterns in frequently-executed code and provides valuable insights into where performance bottlenecks occur.

In an experimental evaluation, *DrAsync*'s static analysis detected 2.6K instances of anti-patterns in 20 JavaScript applications, and *DrAsync*'s dynamic analysis determined that, in the aggregate, these anti-patterns were executed 24K times by the application test suites. To evaluate whether the detected anti-patterns represent actionable findings, we selected 10 instances of each anti-pattern randomly and attempted to manually refactor them to eliminate the anti-pattern. We were able to successfully refactor 65 of these 80 instances, and determined that, in certain cases, these refactorings can have measurable impact on the number of promises created by an application, or the time needed the execute affected code fragments.

In summary, the contributions of this paper are as follows:

- the definition of 8 anti-patterns that commonly occur in asynchronous JavaScript code;
- *DrAsync*, a tool that relies on static and dynamic program analysis to detect anti-patterns and visualize promises and occurrences of anti-patterns during program execution, enabling programmers to quickly identify quality issues and performance bottlenecks;
- an empirical study of 20 JavaScript applications in which *DrAsync* is used to identify 2.6K anti-patterns which are executed 24K times, confirming that they are pervasive; and
- a case study that investigates whether 10 randomly chosen instances of each anti-pattern can be refactored, providing evidence that the majority of anti-patterns reported by *DrAsync* can be eliminated through refactoring. Further analysis of these results suggests that, under certain conditions, eliminating anti-patterns may improve performance.

2 PROMISES AND `ASYNC/AWAIT`

This section reviews promises [15, Section 27.2] and the `async/await` feature [15, Section 15.8] features, which were added to JavaScript in recent years to facilitate asynchronous programming. Readers already familiar with these features may skip this section.

A *promise* is an object that represents the value computed by an asynchronous computation, and is in one of three states: *pending*, *fulfilled*, or *rejected*. Upon construction, a promise is in the *pending* state. If the computation associated with a promise p successfully computes a value v, then p transitions to the *fulfilled* state, and we will say that p is *fulfilled with value* v. If an error e occurs during the computation associated with a promise p, then p transitions to the *rejected* state, and we will say that p is *rejected with value* e. The state of a promise can change at most once; accordingly, we will say that a promise is *settled* if it is *fulfilled* or *rejected*.

Creating promises. Promises can be created by invoking the `Promise` constructor, passing it an *executor function* expecting two arguments, `resolve` and `reject`, for fulfilling or rejecting the newly constructed promise, respectively. E.g., the following code snippet

```
let c = ...
let p1 = new Promise( (resolve, reject) => {
  if (c){ resolve(3) } else { reject("error!") }
})
```

assigns to `p1` a new promise that is fulfilled with the value `3`, or rejected with the value `"error!"`, depending on the value of `c`. Convenience functions `Promise.resolve` and `Promise.reject` accommodate situations where a promise always needs to be fulfilled or rejected with a specified value, respectively. For example, the following code snippet:

```
let p2 = Promise.resolve(4)
let p3 = Promise.reject("error!")
```

assigns to `p2` and `p3` promises that are fulfilled with the value `4` and rejected with the value `"error!"`, respectively.

Reactions. To specify that a designated function should be executed asynchronously upon the settlement of a promise, programmers may register *reactions* on promises using methods `then` and `catch`. Here, a reaction is a function that takes one parameter, which is bound to the value that the promise was fulfilled or rejected with. For example, the following code snippet:

```
p2.then( (v) => console.log(v*v) )
```

extends the previous example by registering a reaction on the promise referenced by variable `p2` to print the value `16`[1]. Similarly, the following code snippet:

```
p3.catch( (e) => console.log("error:␣" + e) )
```

will cause the text "`error: error!`" to be printed.

Promise chains. The `then` method returns a promise. If the reaction that is passed to it returns a (non-promise) value v, then this promise is fulfilled with v. If the reaction that is passed to it throws an exception e, then this promise is rejected with e. Furthermore, if `then` is used to register a reaction f on a promise p, then the rejection of p with a value e will cause the rejection of the promise returned by p.`then`(f) with the same value e. This enables the construction of *chains* of promises. In the following code snippet, a promise chain is created starting with variable `p1` as defined above:

```
p1.then( (v) => v+1 )
  .then( (w) => console.log(w) )
  .catch( (err) => console.log("an␣error␣occurred.") )
```

if `p1` was fulfilled with `3`, then the reaction `(v) => v+1` will be executed asynchronously with `v` bound to the value `3` and return the value `4`, so the promise created by this call to `then` is fulfilled with the value `4` as well. Since a reaction `(w) => console.log(w)` was registered on that promise, the value `4` will be printed. If, on the other hand, `p1` was rejected with the value `"error!"`, the promises created by both calls to `then` will be rejected as well, with the same value, causing the reaction on the last line to execute, which prints `"an␣error␣occurred."`.

Linked promises. So far, we have only considered situations where a function f that is registered as a reaction on a promise returns a non-promise value. However, if f returns a promise p, that promise becomes *linked* with the promise p' created by the call to `then` (or `catch`) that was used to register the reaction. Concretely, this means that p' will be fulfilled with a value v if/when p is fulfilled with v, and p' will be rejected with a value e if p is rejected with e, and if p remains pending then so will p'. Consider the following example:

```
let p4 = Promise.resolve(5);
let p5 = new Promise( (resolve,reject) =>
  setTimeout(() => resolve(6), 1000) )
p4.then( (v) => p5 )
  .then( (w) => console.log(w) ) // prints 6 after one second
```

Here, the promise referenced by `p4` is fulfilled with `5`, and the promise referenced by `p5` is fulfilled with `6` after 1000 milliseconds have elapsed. The reaction `(v) => p5` that is registered on `p4` returns `p5`, so the promise created by this call to `then` becomes linked with `p5`, i.e., it will be fulfilled with `6` after 1000 milliseconds have passed. The last line registers another reaction on this promise, so the value `6` is printed after 1000 milliseconds.

[1]The `then` method optionally accepts a reject-reaction as its second argument.

Synchronization. Several functions are provided for synchronization. The `Promise.all` function takes an array of promises $[p_1, \cdots, p_n]$ as an argument and returns a promise that is either fulfilled with an array $[v_1, \cdots, v_n]$ containing the values that these promises are fulfilled with, or that is rejected with a value e_i, if p_i is the first promise among $p_1, \cdots, p_n$ that is rejected, and e_i is the value that it is rejected with. Other synchronization functions include `Promise.race` and `Promise.any`. For example[2], the following snippet prints `Array [3, 42, "foo"] after 1 second`:

```
let p6 = Promise.resolve(3);
let p7 = 42;
let p8 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'foo');
});
Promise.all([p6, p7, p8])
  .then((vs) => console.log(vs); );
```

Promisification. Promisification is a mechanism for automatically adapting an asynchronous event-driven API into a promise-based API. It assumes that methods in an event-driven API meet two requirements: (i) the callback function is the last parameter, (ii) upon completion of the asynchronous operation, the callback function is invoked with two parameters `err` and `result`, where `err` is a value that indicates whether an error has occurred, and `result` contains the result of the asynchronous computation otherwise. In such cases, an equivalent promise-based API can be derived by creating a new promise that invokes the event-driven API, passing it a callback that rejects the promise with `err` if an error occurred, and fulfills it with `result` otherwise. Promisifying event-driven APIs can be done using the built-in `util.promisify` function.

`async/await`. JavaScript allows a function to be declared as `async` to indicate that it computes its result asynchronously. An `async` function f returns a promise: if no exceptions occur during the execution of f, this promise is fufilled with the returned value, and if an exception e is thrown, then the promise is rejected with e. Inside the body of `async` functions, `await`-expressions may be used to await the settlement of promises, including promises created by calls to other `async` functions. Concretely, when execution encounters an expression `await` x during the execution of an `async` function, control returns to the main event loop. At some later time, when the promise that x evaluates to has settled, execution resumes. If that promise was fulfilled with a value v, then execution resumes with the entire `await`-expression evaluating to v. If the promise was rejected with a value e, then execution resumes with the entire `await`-expression throwing an exception e.

The `async/await` feature has been designed to interoperate with promises, as is illustrated by the example below.

```
24 import fs from 'fs'
25 async function analyzeDir(dName){
26   let fNames = await fs.promises.readdir(dName);
27   let ps = fNames.map( (fName) => fs.promises.stat(fName) );
28   let fStats = await Promise.all(ps);
29   let sum = fStats.reduce((acc,v) => acc + v.size, 0);
30   console.log(sum);
31 }
```

The example shows an `async` function `analyzeDir` that prints the sum of the sizes of the files in the directory identified by its parameter `dName`. On line 26, an await-expression is used to await the results of the built-in `readdir` operation; this operation returns a promise that is eventually fulfilled with an array containing the names of files in the specified directory, which is assigned to `fNames`. On line 27, the `map` operation on arrays is used to map the built-in `fs.stat` operation[3] over this array, resulting in an array `ps` of promises that will eventually resolve to objects containing meta-information for each file. `Promise.all` is used on line 28 to create a promise that is eventually fulfilled with the meta-information objects for each of the files, and an await-expression is used to await this result so that it can be stored in a variable `fStats`. On line 29, the `reduce` operation on arrays is used to compute the sum of the sizes of the files, and this sum is printed on line 30.

JavaScript's async/await feature can be thought of as syntactic sugar for promise-based asynchrony. Consider:

```
function fetchAsynchronously(url) {
  fetch(url)
  .then(response => response.json())
  .then(jsonResponse => {
    // do something
  });
}
```

Here, the function `fetchAsynchronously` takes a `url`, fetches it, converts it to JSON, and then does something with it—all using promises. In this setup, the bulk of the function logic would be in the body of the last callback (`// do something`). Using `async/await`, we can write the function more concisely as:

```
async function fetchAsynchronously(url) {
  const response = await fetch(url);
  const jsonResponse = await response.json();
  // do something
}
```

3 MOTIVATING EXAMPLES

Asynchronous programming is rife with pitfalls. As a first example, consider SAP's *ui5-builder* project, which provides modules for building UI5 projects. *ui5-builder*'s file `ResourcePool.js` contains the following function, which *DrAsync* flagged as an instance of the *promiseResolveThen* anti-pattern that will be presented in Section 4:

```
44 async getModuleInfo(name) {
45   let info = this._dependencyInfos.get(name);
46   if (info == null) {
47     info = Promise.resolve().then(async () => {
48       const resource = await this.findResource(name);
49       return determineDependencyInfo(resource, ... );
50     });
51     this._dependencyInfos.set(name, info);
52   }
53   return info;
54 }
```

On line 47, `Promise.resolve()` is invoked to create a promise that is fulfilled immediately with the value `undefined`[4]. On the same line, an `async` function is registered as a fulfill reaction on this promise, so this reaction is asynchronously invoked with `undefined` as an argument. This means that 3 promises are created when the

[2] Adapted from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all.

[3] `fs.stat` is a library function that returns an object that contains various information about a file, including its size; see https://nodejs.org/api/fs.html#fs_class_fs_stats.

[4] Since no argument is passed in the call to `Promise.resolve`, the value `undefined` is used by default.

Figure 1: An example of the *promiseResolveThen* anti-pattern found in `getModuleInfo`. The user selected one of the promises in a promise chain originating from an empty `Promise.resolve()`, identified by Label A, and the reaction's promise is shown with Label B, and finally the promise belonging to the async function is shown with Label C.

reaction executes: (i) the promise created by `Promise.resolve`, (ii) the promise created by the invocation of `then`, and (iii) the promise created by the invocation of the `async` function. This is manifested in *DrAsync*'s visualization as an extremely short-lived promise linked two other, longer-running promises (see Figure 1).

In this case, the code can be refactored as such:

```
55  async getModuleInfo(name) {
56    let info = this._dependencyInfos.get(name);
57    if (info == null) {
58      info = (async () => {
59        const resource = await this.findResource(name);
60        return determineDependencyInfo(resource, ... );
61      })();
62      this._dependencyInfos.set(name, info);
63    }
64    return info;
65  }
```

Now, only one promise is created (on line 58, by invoking the `async` function). This code is executed 204 times in *ui5-builder*'s test suite, and 2 fewer promises are executed each time. Besides being more efficient, the code is more concise, and easier to understand.

As another example, consider `appcenter-cli`, developed by Microsoft, which implements the Command Line Interface (CLI) for the Visual Studio Code (VSCode) Interactive Development Environment (IDE). Function `cpDir`, defined on lines 89-94 in `src/util/misc/promisified-fs.ts`, implements the copying of a directory:

```
66  async function cpDir(source, target) {
67    // details omitted
68    const files = await readdir(source);
69    for (let i = 0; i < files.length; i++) {
70      const sourceEntry = path.join(source, files[i]);
71      const targetEntry = path.join(target, files[i]);
72      await cp(sourceEntry, targetEntry);
73    }
74  }
```

This code reads the source directory `source` on line 68 and then iterates over the resulting list of file names. In each iteration of the loop, a call to function `cp` is `await`-ed, which copies a file from `sourceEntry` to `targetEntry`. Here, `cp` returns a promise that is fulfilled once `sourceEntry` is successfully copied to `targetEntry`, or rejected if an error occurs. It is important to note that this use of `await` in a loop causes the execution of function `cpDir` to be paused until the promise returned by `cp` is fulfilled, and execution will pass back to the main event loop at this time so that other event handlers can be executed in the meantime. This is manifest in *DrAsync*'s visualization by a "staircase" pattern of promises that have lifetimes that do not overlap (see Figure 2).

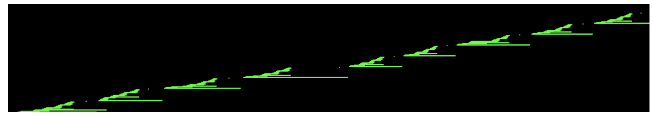

Figure 2: An example of the *loopOverArrayWithAwait* anti-pattern in the visualization, here from a view depicting an overview of all promises. Each loop iteration is clearly separated, with no overlapping promises.

In this case, the copying of file-entries need not be sequential, and we can refactor the above code as follows:

```
75  async function cpDir(source, target) {
76    // details omitted
77    const files = await readdir(source);
78    await Promise.all(files.map(file => {
79      const sourceEntry = path.join(source, file);
80      const targetEntry = path.join(target, file);
81      return cp(sourceEntry, targetEntry);
82    }));
83  }
```

Here, we turn the for-loop into a map over the `files` array, mapping a function that returns the promise associated with `cp`. We then `await` the entire array of promises with `Promise.all` (line 78), which will wait for all these promises to resolve. This refactoring preserves the behavior of *appcenter-cli*'s tests, and enables additional concurrency because, although JavaScript is single-threaded at the language level, it relies on I/O libraries that can execute concurrently [11]. We will report in Section 7 how the refactoring significantly improves the performance of the loop.

These anti-patterns are detected using a simple static analysis. Our *DrAsync* tool additionally relies on dynamic analysis to determine how often each instance of an anti-pattern is executed, and helps programmers prioritize which code should be fixed. For instance, we found many instances of the "await-in-loop" pattern in *appcenter-cli*, but the highlighted `cpDir` example was by far the most frequently executed while running the application's tests.

4 ANTI-PATTERNS

This section defines a set of anti-patterns that occur frequently in asynchronous JavaScript applications. We identified most of these through manually inspecting JavaScript source code[5], and inspecting visual profiles produced by *DrAsync* for noteworthy patterns (e.g., repetitive structures or promises that are very short-lived). In addition, a search for issues related to promises and `async`/`await` on the popular stackoverflow forum turned up the *explicitPromiseConstructor* [6] and *customPromisification* [7] anti-patterns.

It is important to note that an occurrence of one of these anti-patterns is not necessarily a reflection that a design is "wrong" or "inefficient", but it indicates that it is likely that the code can be improved to make it more efficient by creating fewer promises or enabling additional parallelism, or to make it more concise. Section 6 presents a case study that investigates, for a representative subset of instances of these anti-patterns, how often we were able to refactor

[5] Section 7.1 provides further detail on the process for selecting subject applications.
[6] https://stackoverflow.com/questions/23803743
[7] https://www.grouparoo.com/blog/promisifying-node-functions

$$
\begin{aligned}
\mathit{asyncFunctionNoAwait} &= \{\ f \mid f\ \mathit{async} \land (\exists e_0, e_1 : e_0 = \texttt{await}\ e_1 \Rightarrow e_0 \not\lhd f)\ \} \\
\mathit{asyncFunctionAwaitedReturn} &= \{\ f \mid f\ \mathit{async} \land (\exists e_0, e_1 : e_0 = \texttt{return}\ e_1 \land e_0 \lhd f) \Rightarrow \exists e_2 : e_1 = \texttt{await}\ e_2\ \} \\
\mathit{loopOverArrayWithAwait} &= \{\ s_0 \mid \exists e_0, e_1, e_2, e_3, s_1 : s_0 = \texttt{for}(e_0, e_1, e_2)\{s_1\} \land \textbf{isArrayTest}(e_1) \land \texttt{await}\ e_3 \lhd s_1\ \} \\
\mathit{promiseResolveThen} &= \{\ e_0 \mid \exists e_1, f : e_0 = \texttt{Promise.resolve}(e_1)\texttt{.then}(f)\ \} \\
\mathit{executorOneArgUsed} &= \{\ e_0 \mid e_0 = \exists f, v_0, v_1 : \texttt{new Promise}(f) \land v_0 = \textbf{arg}(f, 0)\ \land v_1 = \textbf{arg}(f, 1) \land \\
&\qquad (\exists e_1, e_2 : e_1, e_2 \lhd f \land e_1, e_2 \in \{v_0, v_1\} \Rightarrow e_1 = e_2)\} \\
\mathit{reactionReturnsPromise} &= \{\ e_0 \mid \exists e_1, e_2, f : e_0 = e_1\texttt{.then}(f) \land \texttt{return}\ e_2 \lhd f \land \\
&\qquad (e_2 = \texttt{Promise.resolve}(\cdots) \lor e_2 = \texttt{Promise.reject}(\cdots))\ \} \\
\mathit{customPromisification} &= \{\ e_0 \mid \exists f_0, f_1, f_2, s_0, s_1, v_0, v_1 : e_0 = \texttt{new Promise}(f_0) \land f_1(..., f_2) \lhd f_0\ \land \texttt{if}\ (\cdots)\ \{\texttt{s}_0\}\ \texttt{else}\ \{\texttt{s}_1\} \lhd f_2 \land \\
&\qquad v_0 = \textbf{arg}(f_0, 0)\ \land v_1 = \textbf{arg}(f_0, 1)\ \land ((v_0 \lhd s_0 \land v_1 \lhd s_1) \lor (v_0 \lhd s_1 \land v_1 \lhd s_0))\ \} \\
\mathit{explicitPromiseConstructor} &= \{\ e_0 \mid \exists e_1, f_0, f_1, f_2, v_0, v_1, v_2, v_3 : e_0 = \texttt{new Promise}(f_0) \land e_1\texttt{.then}(f_1)\texttt{.catch}(f_2) \lhd f_0 \land v_0 = \textbf{arg}(f_0, 0) \land \\
&\qquad v_1 = \textbf{arg}(f_0, 1)\ \land v_2 = \textbf{arg}(f_1, 0)\ \land v_0(v_2) \lhd f_1\ \land v_3 = \textbf{arg}(f_2, 0)\ \land v_1(v_3) \lhd f_2\ \}
\end{aligned}
$$

Figure 3: Anti-patterns that commonly occur in asynchronous JavaScript code.

them manually. Section 7 presents an empirical evaluation that reports on the prevalence of each of the anti-patterns.

Figure 3 defines each anti-pattern as a set of AST nodes that meet some specified criteria. In the figure, we use f to represent functions (including arrow functions and class methods), e to represent expressions, and s to represent statements. Subscripts are used in cases where a predicate refers to multiple program elements of the same kind. Furthermore, f *async* denotes that f is an `async` function, and $e \lhd f$ (read as: "f contains expression e") indicates that f is the innermost function declaration or function expression such that e syntactically occurs within the body of function f.

asyncFunctionNoAwait. This anti-pattern is defined as any function f such that: (i) f is an `async` function and (ii) for any expression e_0 = `await` e_1, e_0 does not occur in the body of f. In other words, the pattern identifies `async` functions that do not contain any `await` expressions. As we will discuss in Section 6, such functions can often be refactored into functions that are not `async`, to avoid the creation of a promise each time the function is executed. Note that the scope of this refactoring may expand beyond f itself: functions calling f may no longer need to await the result of the call f.

asyncFunctionAwaitedReturn. This anti-pattern is defined as any function f such that: (i) f is an `async` function and (ii) any return-expression in f is an await-expression. In such cases, the use of `await` is redundant, because the value v that the await-expression evaluates to is immediately used to settle the promise created by the `async` function (which itself would need to be awaited—it is more efficient to return the promise as it will become linked with the promise created by the `async` function).

loopOverArrayWithAwait. This anti-pattern covers `for`-loops of the form `for`$(e_0, e_1, e_2)\{s_1\}$ where (i) the condition e_1 tests that the loop iterates over an array by checking that it refers to the `Array.prototype.length` property (using auxiliary function **isArrayTest**), and (ii) the body s_1 of the loop contains at least one `await`-expression. This situation is well-known in the JavaScript community as being needlessly inefficient in situations where the iterations of the loop are independent of one another, and the ESLint checker [5] has a rule for detecting it. As we will discuss in Section 6, in many cases, such loops can be refactored to use `Promise.all` and `Array.prototype.forEach` to enable additional parallelism.

promiseResolveThen. An expression e_0 = `Promise.resolve(e`$_1$`)` `.then`(f) is constructed, i.e., a new promise is constructed on which a fulfill-reaction is registered immediately. Note that entire expression e_0 may form the beginning of a longer chain of promises. In such cases, it is often possible to shorten the length of the promise-chain by refactoring e_0, e.g., to `Promise.resolve(f(e`$_1$`)}`, to reduce the number of created promises. Section 3 discussed a slightly more complex instance of this anti-pattern.

executorOneArgUsed. This anti-pattern targets expressions of the form `new Promise`(f) where a promise is constructed using an executor function f that has formal parameters v_0 and v_1 (usually the parameters of executor functions are called `resolve` and `reject` but programmers may choose different names). Furthermore, an additional constraint is imposed that if the body of f contains expressions e_1 and e_2 that refer to v_0 or v_1, then they must both refer to the same variable. In other words, the anti-pattern targets executor functions that either resolve or reject the promise, but not both. In such cases, it may be possible to refactor the code to use `Promise.resolve` or `Promise.reject` instead.

reactionReturnsPromise. In this scenario, a reaction f that is registered on a promise in an expression of the form e_1`.then`(f) returns an expression e_2 that consists of either a call to `Promise.resolve` or a call to `Promise.reject`. In such cases, it is often possible to avoid the explicit construction of a promise because the reaction already creates a promise that is fulfilled or rejected with the reaction's return value.

customPromisification. This anti-pattern aims to detect situations where a programmer has written a custom function for promisifying an event-based API call. It targets expressions of the form `new Promise`(f_0) where the Promise constructor is invoked with an executor function that contains a call $f_1(..., f_2)$, that passes a callback function f_2 to some API function f_1. Moreover, f_2 contains a statement `if` $(\cdots)$ $\{\texttt{s}_0\}$ `else` $\{\texttt{s}_1\}$, where either s_1 calls the function passed as the first parameter to the executor (usually called `resolve`) and s_2 calls the function passed as the second parameter to the executor (usually called `reject`), or vice versa. In such cases, it is often possible to utilize the `util.promisify` promisification function instead. While this does not reduce the number of promises created, it avoids the pitfalls of accidentally introducing bugs when re-implementing functionality that is available in standard libraries.

explicitPromiseConstructor. This anti-pattern occurs when a new promise is constructed that is fulfilled when some existing promise is fulfilled, and that is rejected when that promise is rejected. Concretely, we say that an instance of this pattern occurs when the promise constructor is invoked with an executor function f_0 that has parameters v_0 and v_1. In addition, the body of f_0 contains an expression e_1`.then(`f_1`).catch(`f_2`)`, where f_1 has a parameter v_2 and f_2 has a parameter v_3. Lastly, f_1 is required to contain a call $v_0(v_2)$ and f_2 is required to contain a call $v_1(v_3)$. Occurrences of this anti-pattern can often be refactored to avoid the creation of a new promise, e.g., by returning the promise e_1.

5 IMPLEMENTATION

DrAsync consists of three components: (i) a static analysis for detecting anti-patterns, (ii) a dynamic analysis for gathering information about the lifetimes of promises and detecting run-time instances of anti-patterns, and (iii) an interactive profiling tool that visualizes the lifetimes of promises and instances of anti-patterns, and that provides additional features for understanding execution behavior. Our code is open-source and publicly available [8].

5.1 Static Analysis

The static analysis uses CodeQL [4, 13] to implement the anti-patterns of Figure 3 as a set of QL queries. These queries follow the logic of the definition closely. For example, the query that is used to find the *promiseResolveThen* anti-pattern looks as follows:

```
predicate promiseDotResolveDotThen(MethodCallExpr c) {
  c.getMethodName() = "then" and
  c.getReceiver() instanceof MethodCallExpr and
  ((MethodCallExpr) c.getReceiver()).getMethodName() = "resolve"
}
```

In two cases, we extended the queries with special handling of corner cases. Our implementation of *executorOneArgUsed* was extended to exclude cases where calls to `resolve` are passed as an argument to `setTimeout` as we found that such occurrences of the anti-pattern are generally not amenable to refactoring. We also extended *loopOverArrayWithAwait* to handle `for-of` and `for-in` loops. All QL queries can be found in the supplemental materials.

5.2 Dynamic Analysis

DrAsync relies on the Node.js Async hooks API [3] to instrument source code to log the creation and settlement of promises, to record when `await`-expressions are first encountered and when their execution is resumed, and to determine run-time instances of anti-patterns. The instrumentation distinguishes different run-time instances of promises that are created at the same location (e.g., promises created during multiple executions of the same promise constructor or of the same `async` function), enabling us to calculate how often each anti-pattern is executed.

[8] Artifact link: https://doi.org/10.5281/zenodo.5915257

Furthermore, information is recorded about dependencies between promises: the Async hooks API provides a unique `asyncId` for each promise, as well as a `triggerAsyncId`, which is the `asyncId` of the promise that triggered it (i.e., the promise that it depends on). Moreover, the dynamic analysis determines whether promises are related to I/O operations through simple heuristics (if a promise originates from a function from a predefined list of I/O functions from the `util` Node.js library), and whether they originate from user code or from library code. This information is used in the interactive visualization to enable programmers to filter promises based on their origin, and quickly hone-in on relevant promises.

The results of the static analysis and a dynamic analysis are aggregated into a single trace file that is used in *DrAsync*'s interactive visualization component.

5.3 Interactive Visualization

The visualization helps with exploring the execution behavior of asynchronous JavaScript code and enables one to identify certain anti-patterns visually. The visualization also shows the number of runtime occurrences for each instance of an anti-pattern, enabling programmers to prioritize those anti-pattern instances that may impact execution behavior the most.

DrAsync's interactive visualization tool was developed using the P5.js framework [26]. Figure 4 shows a screenshot of a visualization produced by *DrAsync*, which follows the standard information taxonomy by providing: a high level overview, filters, and details on demand [29]. We briefly discuss *DrAsync*'s different views.

Promise Lifetime View and Source Code View. This view (labeled A in the figure) is organized as a Gantt Chart [20]. Here the x-axis represents time, and the y-axis shows the created promises as a series of stacked bars, so each promise is represented by one line that starts at the time when the promise was created, and that ends when it was settled. Users can pan and zoom through the promise lifetime view, and hovering on a promise shows a fragment of the source code responsible for creating the promise, along with some meta-information. Furthermore, clicking on one of the promises opens the associated source code in tab B for further inspection.

Mini Display View. This view (green bars in the view labeled C at the bottom of the figure) shows the general 'shape' of the promises created during execution; clicking here enables the user to quickly navigate to areas of interest in the promise lifetime view (e.g., staircase patterns corresponding to instances of *loopOverArrayWithAwait* that may benefit from refactoring).

Metrics View. This view, labeled D, summarizes metrics: how many promises were created, the total elapsed time, the average duration of promises, and counts for detected anti-patterns. These can be compared before and after refactoring to see if redundant promises have been eliminated, or if performance has changed.

Summary View and Filters. This view, labeled E, shows of all promises and anti-pattern instances; clicking on these will navigate to the associated promise in the promise lifetime view, and will display the associated source code. For realistic applications, the number of promises created at run-time can quickly become overwhelming, so *DrAsync* provides various filtering facilities to

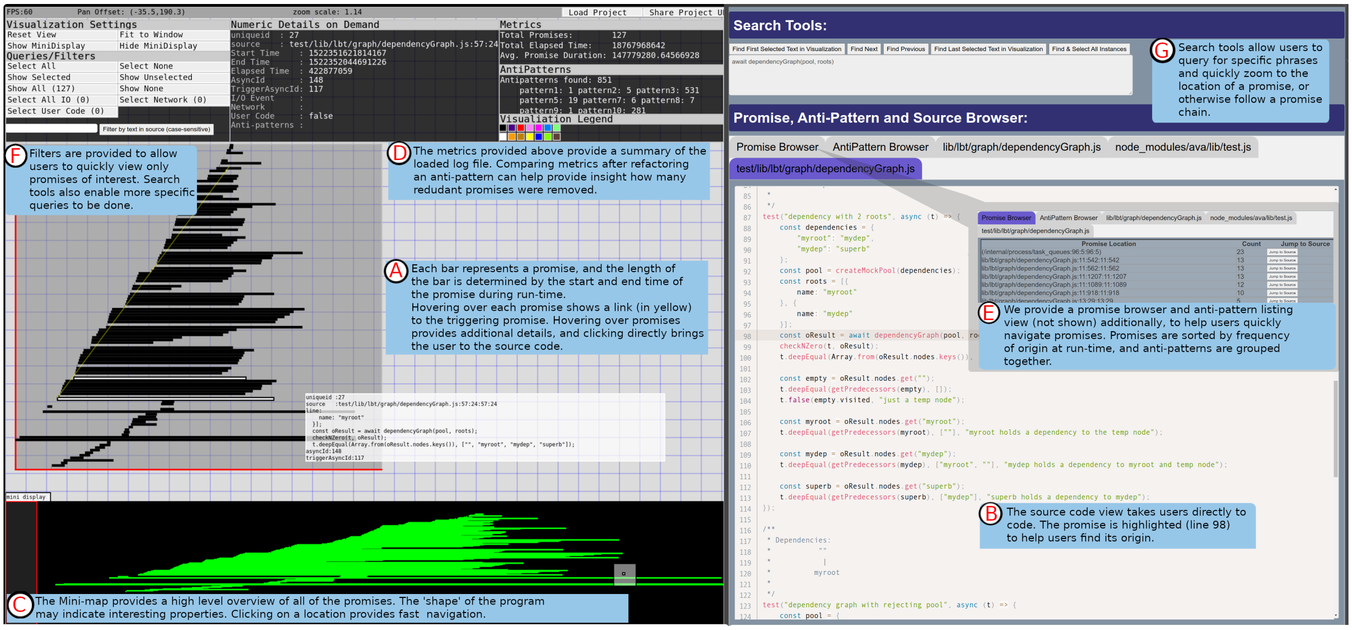

Figure 4: The interactive visualization displays the run-times of each promise as well as visually summarizes the data capture by DrAsync. Users can filter particular promises and directly investigate the source code for more details on demand.

Table 1: Summary of Case Study

Anti-Pattern	# Successful	# Unsuccessful
asyncFunctionNoAwait	9	1
asyncFunctionAwaitedReturn	9	1
loopOverArrayWithAwait	7	3
promiseResolveThen	9	1
executorOneArgUsed	6	4
reactionReturnsPromise	9	1
customPromisification	9	1
explicitPromiseConstructor	7	3

focus on promises of interest. In particular, users can focus on those promises that are related to file I/O or network I/O (see view labeled Ⓕ), or on promises whose creation site matches a specified text string (see view labeled Ⓖ).

6 CASE STUDY

To evaluate if the anti-patterns reported by *DrAsync* represent useful information, we randomly selected 10 instances of each anti-pattern and attempted to refactor them manually. These 10 instances were chosen from the 20 subject applications that we will report further on in Section 7. To ensure that our findings are not biased towards a particular programming style, no more than three instances of each pattern were chosen from a single application, and we only selected anti-pattern instances that *DrAsync* reported as being executed by the application's test suite, so that we could check that the refactoring did not cause behavioral changes.

An overview of our findings can be found in Table 1. Below we report on some noteworthy situations that we encountered. Many refactorings were simple and quick, though others took more considerable time (e.g., some loop refactorings took >15 minutes in order to understand possible data dependencies). Further details for all 80 cases can be found in the supplemental materials.

asyncFunctionAwaitedReturn. As discussed in Section 4, this anti-pattern reflects inefficient code as it involves waiting for a promise to settle with some value v, and then creating a new promise that is settled with the same value. The following function in file `/src/utils/readSpec.ts` in *openapi-typescript-codegen* was flagged by *DrAsync* as an instance of this anti-pattern:

```
export async function readSpec(input: string): Promise<string> {
  if (input.startsWith('https://')) {
    return /*await*/ readSpecFromHttps(input);
  }
  if (input.startsWith('http://')) {
    return await readSpecFromHttp(input); // not executed
  }
  return /*await*/ readSpecFromDisk(input);
}
```

Here, `await` is redundantly used on each of the return paths and *DrAsync* informed us that the first and third of these await-expressions were executed by the test suite. We confirmed that the tests still passed after removing the `await` keywords.

loopOverArrayWithAwait. Section 3 already discussed an instance of this anti-pattern in *appcenter-cli* that we were able to refactor successfully. However, some of the instances reported by *DrAsync* could not be refactored, such as the the following code snippet on lines 159–162 in file `/src/TemplateLayout.js` in *eleventy*:

```
for (let fn of fns) {
  templateContent = await fn(data);
  data = TemplateLayout
           .augmentDataWithContent(data, templateContent);
}
```

Here, each loop iteration awaits the result of the call to `fn(data)` and then re-assigns `data` on the next line. Since each loop iteration depends on a value computed in the previous iteration, we are unable to parallellize the loop using `Promise.all`.

executorOneArgUsed. An interesting case of this anti-pattern occurs on lines 39-56 in `src/streaming/utils/CapabilitiesFilter.js` in *dash.js*:

```
return new Promise((resolve) => {
  const promises = // details omitted
  Promise.all(promises)
    .then(() => { /* details omitted */ resolve(); })
    .catch(() => { resolve(); });
});
```

Here, a new promise is created that is fulfilled (with the value `undefined` since no argument is passed to `resolve`) in reactions on a promise that is created by a call to `Promise.all`. The creation of a new promise can be avoided by refactoring the above code to:

```
const promises = // details omitted
return Promise.all(promises)
    .then(() => { /* details omitted */ return; })
    .catch(() => { return; });
```

After this refactoring, it is evident that the resulting code lacks proper error handling, given that `catch` is used to register a no-op function to "absorb" errors that cause the previous reaction in the promise chain to be rejected.

customPromisification. For this anti-pattern, we found that we could successfully refactor 9 of 10 instances highlighted by the tool using the `util.promisify` library function. The remaining case involved the use of an event handler with complex control flow.

In all but one of the successful cases, using `promisify` and refactoring the inner logic of the callback into a reaction on a call to the promisified function was sufficient. For a more interesting case, consider the following snippet on lines 467-475 in file `src/Engines/Nunjucks.js` in `eleventy`:

```
return async function (data) {
  /* return new Promise(function (resolve, reject) {
    tmpl.render(data, function (err, res) {
      if (err) {
        reject(err);
      } else {
        resolve(res);
      }
    });
  }); */
  const tmplRenderProm = util.promisify(tmpl.render);
  return tmplRenderProm.call(tmpl, data);
};
```

Here, `tmplRenderProm` must be invoked with `Function.prototype.call` to preserve the correct value for `this` during its execution.

reactionReturnsPromise. For this anti-pattern, 9 of the 10 cases we examined could be refactored; the one unsuccessful case involved a promise reaction with complex event-handlers, where the returned promise was fulfilled or rejected in response to external events.

For an example of a successful refactoring, consider this snippet from `netlify-cms`, lines 428-433 of `packages/netlify-cms-core/src/backend.ts`:

```
const publishedEntry = await this.implementation
  .getEntry(path)
  .then(({ data }) => data)
  .catch(() => {
    // return Promise.resolve(false);
    return false;
  });
```

Here, `.catch` and `.then` return promises anyway, so explicitly returning a promise that is immediately fulfilled or rejected is needless.

7 EVALUATION

This evaluation aims to answer the following research questions:

RQ1: How often do the anti-patterns of Figure 3 occur in practice?
RQ2: How often can anti-patterns reported by *DrAsync* be eliminated using refactoring?
RQ3: Can the elimination of anti-patterns improve performance?
RQ4: What is the performance of *DrAsync*?

7.1 Experimental Setup

To identify a set of candidate projects, we first ran a CodeQL query (on a large set of JavaScript GitHub repositories available to the CodeQL team) to find projects containing promise-related features[9]. Of the >100K projects that this turned up, we used the *npm-filter* [12] tool to discard projects that did not have running test suites, resulting in 450 projects with at least one running test command. Of those projects, we randomly selected 20 projects meeting the following criteria: the project (i) was edited in the last year, (ii) had over 20 stars, (iii) contained over 20 instances of promise-related features, and (iv) running the application's test suite results in the creation of at least 40 promises.

All experiments were performed on a CentOS Linux 7.8.2003 (Core) server, with 2x 32-core 2.35GHz processors, and 128GB RAM.

7.2 RQ1: How often do anti-patterns occur?

After discounting anti-patterns occurring in test code, compiled TypeScript, and distributions, we found 2.6k anti-patterns instances in the 20 projects selected for evaluation. Moreover, *DrAsync*'s dynamic analysis detected that a total of 24K instances of these anti-patterns were executed by the applications' test suites. These results are tabulated in Table 3, and provide strong evidence that anti-patterns commonly occur. The first cells of the table read: `appcenter-cli` has 23 instances of the *asyncFunctionNoAwait* pattern in its code (S), 1 instance is executed in the tests (E), and 42 runtime promises are associated with this anti-pattern (D).

Anti-patterns commonly occur in asynchronous JavaScript code. We found a total of 2.6K anti-patterns in 20 subject applications.

7.3 RQ2: Can detected anti-patterns be refactored?

Section 6 summarized findings of a case study wherein we tried to refactor 80 instances of anti-patterns flagged by *DrAsync*. Of these 80 cases, we were able to successfully refactor 65. For the 15 that we were unable to refactor, not all are necessarily false positive, because developers with more expert knowledge may have additional insights enabling them to refactor the code. Each of the refactorings is reported on in the supplementary materials.

A case study involving 80 anti-patterns in real-world code suggests that the majority of anti-patterns detected by *DrAsync* can be eliminated through refactoring.

[9]This includes: references to the `Promise` constructor, references to `Promise.resolve`, `Promise.reject`, `Promise.all`, `Promise.race`, and `Promise.any`, references to methods with names `then` or `catch`, `async` functions, and `await` expressions.

Table 2: Subject Applications

Project *(links to repos @ SHA)*	**SHA**	**KLOC**	**Anti-Patterns #**	**Anti-Patterns / KLOC**	**# Files**	**# Funs**	**Tests**	**QLDB Build Time (s)**	**Test Time (Before/After) Mean**	**StDev**	**Mean**	**StDev**	**Overhead of Instrumentation**
appcenter-cli	2109d1	96	73	0.76	2645	8406	434	126.172	31.45	1.05	34.29	0.86	9.03%
Boostnote	58c4a7	32	29	0.92	276	4572	81	40.069	41.50	0.80	43.59	2.23	5.03%
browsertime	648e16	223	134	0.60	197	17557	13	29.61	0.55	0.01	0.66	0.01	20.59%
CodeceptJS	68ad16	19	398	21.5	180	3583	34	57.448	2.83	0.02	3.16	0.02	11.62%
dash.js	996e21	20	70	3.5	123	3598	18	59.681	4.12	0.16	5.77	0.26	39.79%
eleventy	6776e8	53	65	1.2	358	5532	1070	34.446	21.62	0.27	50.93	0.36	135.6%
erpjs	5ddcb7	30	139	4.6	295	4509	973	106.687	19.0	0.23	21.15	0.27	11.37%
fastify	aee28e	136	2	0.01	108	20461	54	42.472	118.58	0.67	127.43	1.02	7.47%
flowcrypt-browser	bc0d348	41	296	7.1	240	7119	5394	1064.285	1.77	0.03	2.27	0.05	1.28%
media-stream-library-js	4dd02a	37	184	5.0	117	4754	154	63.543	122.88	0.85	131.52	1.36	7.03%
mercurius	97ee14	60	22	0.37	220	4969	959	42.099	55.17	0.51	65.44	0.65	18.62%
netlify-cms	071b05	12	77	6.6	118	4009	73	94.35	504.42	2.30	605.48	1.69	20.04%
openapi-typescript-codegen	715ddc	34	9	0.27	180	4529	1092	45.618	45.56	0.62	56.10	0.57	23.04%
rmrk-tools	64c8cf	36	334	9.2	301	7916	247	326.839	38.01	0.32	41.42	0.39	8.97%
stencil	0c2e95	193	265	1.4	326	45025	1619	823.68	453.33	1.67	484.40	5.89	6.85%
strapi	1fe4b5e	80	198	2.5	292	4875	982	77.734	164.89	0.95	195.13	2.06	18.33%
treeherder	b70d3b	37	50	1.4	154	4004	300	43.12	209.79	1.06	229.93	2.60	9.60%
ui5-builder	7490fb	44	77	1.8	216	4802	741	44.462	31.82	0.23	69.14	0.49	117.31%
vscode-js-debug	2af8cb	78	150	1.9	300	11496	186	127.798	1.39	0.02	2.31	0.06	65.48%
vuepress	f077f7	14	19	1.3	276	7736	104	81.301	6.97	0.20	22.64	0.96	224.79%

Table 3: Anti-pattern stats. Legend: P1 = *asyncFunctionNoAwait*, **P2** = *loopOverArrayWithAwait*, **P3** = *asyncFunctionAwaitedReturn*, **P4** = *explicitPromiseConstructor*, **P5** = *customPromisification*, **P6** = *promiseResolveThen*, **P7** = *reactionReturnsPromise*, **P8** = *executorOneArgUsed*. **"S" stands for *static occurrences*; "E" stands for *static occurrences that are dynamically executed*; "D" stands for *the total number of runtime promises associated with this anti-pattern.***

Project	**P1** S (E)	D	**P2** S (E)	D	**P3** S (E)	D	**P4** S (E)	D	**P5** S (E)	D	**P6** S (E)	D	**P7** S (E)	D	**P8** S (E)	D
appcenter-cli	23 (1)	42	11 (0)	0	18 (0)	0	1 (0)	0	14 (3)	446	1 (0)	0	4 (1)	4	1 (0)	0
Boostnote	1 (0)	0	0 (0)	0	0 (0)	0	3 (3)	6	9 (5)	18	5 (2)	7	5 (0)	0	6 (1)	1
browsertime	105 (1)	3	21 (1)	47	0 (0)	0	1 (0)	0	1 (0)	0	2 (0)	0	0 (0)	0	4 (0)	0
CodeceptJS	357 (3)	39	33 (0)	0	1 (0)	0	0 (0)	0	1 (0)	0	3 (3)	1125	0 (0)	0	3 (0)	0
dash.js	0 (0)	0	0 (0)	0	0 (0)	0	23 (8)	224	2 (2)	55	0 (0)	0	27 (0)	0	18 (10)	188
eleventy	39 (24)	4416	10 (10)	884	9 (7)	1271	0 (0)	0	1 (1)	244	0 (0)	0	5 (4)	31	1 (1)	6
erpjs	40 (0)	0	12 (0)	0	66 (1)	36	0 (0)	0	14 (0)	0	6 (0)	0	0 (0)	0	1 (0)	0
fastify	0 (0)	0	0 (0)	0	0 (0)	0	0 (0)	0	0 (0)	0	0 (0)	0	0 (0)	0	2 (2)	25
flowcrypt-browser	79 (0)	0	50 (0)	0	150 (0)	0	2 (0)	0	3 (0)	0	0 (0)	0	0 (0)	0	12 (0)	0
media-stream-library-js	56 (0)	0	3 (0)	0	121 (1)	1	0 (0)	0	2 (0)	0	0 (0)	0	0 (0)	0	2 (1)	2
mercurius	14 (3)	72	4 (3)	322	0 (0)	0	0 (0)	0	3 (3)	409	1 (1)	10	0 (0)	0	0 (0)	0
netlify-cms	45 (3)	1261	8 (0)	0	5 (0)	0	0 (0)	0	0 (0)	0	4 (1)	10	10 (2)	14	5 (1)	2286
openapi-typescript-codegen	2 (1)	2	3 (0)	0	2 (2)	28	0 (0)	0	1 (0)	0	0 (0)	0	0 (0)	0	1 (1)	4
rmrk-tools	241 (0)	0	43 (0)	0	18 (0)	0	0 (0)	0	8 (0)	0	2 (0)	0	0 (0)	0	22 (0)	0
stencil	123 (1)	74	33 (3)	217	20 (2)	35	1 (0)	0	17 (1)	3	21 (0)	0	1 (0)	0	49 (0)	0
strapi	81 (5)	179	45 (6)	100	26 (0)	0	4 (0)	0	19 (0)	0	8 (1)	20	12 (5)	5	3 (0)	0
treeherder	43 (7)	211	2 (2)	10	0 (0)	0	0 (0)	0	2 (0)	0	0 (0)	0	3 (3)	61	0 (0)	0
ui5-builder	51 (25)	1510	5 (3)	373	1 (1)	23	2 (2)	69	5 (2)	56	5 (5)	896	2 (2)	310	6 (2)	50
vscode-js-debug	94 (2)	84	7 (0)	0	20 (3)	749	1 (0)	0	4 (0)	0	0 (0)	0	2 (0)	0	22 (2)	42
vuepress	7 (0)	0	3 (2)	3448	1 (1)	1910	1 (0)	0	0 (0)	0	5 (0)	0	1 (0)	0	1 (0)	0
Summary	1401 (76)	7893	293 (30)	5401	458 (18)	4053	39 (13)	299	106 (17)	1231	63 (13)	2068	72 (17)	425	159 (21)	2604

7.4 RQ3: Can the elimination of anti-patterns improve performance?

Generally speaking, we would expect the elimination of an anti-pattern to impact performance only in significant ways if the anti-pattern is executed many times, if the refactoring results in the elimination of a large number of promises at run-time, or if the refactoring enables additional concurrency. We examined three refactorings in our case study that meet some of these criteria, for which we crafted experiments that emphasize the performance of the code fragment in question.

appcenter-cli/`cpDir`. This particular instance of the *loopOverArrayWithAwait* anti-pattern was previously discussed in Section 3 and involves a function that copies one directory to another. We chose this anti-pattern instance as the correctness of the refactoring was easy to confirm, and we could easily craft a controlled experiment; in this experiment, we executed `cpDir` 50 times on a large directory of 7.8G with 37 files, and found that the refactored version ran 16.4% (4.8s vs 5.8s) faster on average than the original, and that the variance between run times was 37.9% smaller (0.33s vs 0.54s), leading to more predictable performance.

vuepress/`apply`. This function contains a loop exhibiting the *loopOverArrayWithAwait* anti-pattern:

```
for (const { value, name: pluginName } of this.appliedItems) {
  // details omitted
  await ctx.writeTemp(`${dirname}/${name}`, ... );
```

We chose to focus on this anti-pattern instance because the correctness of the refactoring was easy to check, and the code is frequently invoked by the test suite, so we can observe performance in a realistic use-case. After refactoring this code fragment to use `Promise.all`, we ran the application's test suite 50 times on the versions before and after the refactoring. The results show that the refactoring reduced the time needed to execute this code fragment by 36.1% on average, and that run time variability was reduced by 16%.

strapi/`evaluate`. This instance of the *promiseResolveThen* anti-pattern occurs in the `strapi` application:

```
// const evaluatedConditions = await Promise.resolve(conditions)
//     .then(resolveConditions)
//     .then(filterValidConditions)
//     .then(evaluateConditions)
//     .then(filterValidResults);
const evaluatedConditions = filterValidResults(await
  evaluateConditions(filterValidConditions(
  resolveConditions(conditions))));
```

We selected an instance of this anti-pattern to assess the performance impact of eliminating more than just the *loopOverArrayWithAwait* anti-pattern, and we selected this instance specifically as it is frequently executed by the test suite and involves many chained promises (our refactoring eliminates 5 runtime promises per execution of this snippet). We refactored this fragment to instead call the functions directly (the code exhibiting the anti-pattern is commented). We ran the `strapi` test suite 50 times and observed that the refactoring reduced the average time needed to execute this code fragment by 4%, and the standard deviation by 7.4%.

Full Test Suite Refactorings. We refactored every executed instance of an anti-pattern in the `eleventy` project, and timed the execution of the test suite before and after. We found that roughly 1.1k fewer user promises (39,978 to 38,748) were created, and found no meaningful change in the run time of the test suite. We performed a similar case study with `vuepress`. We again found no meaningful change in test suite execution time, and found roughly 1.2k fewer user promises (32,264 to 31,021).

Note that we chose these projects to fully refactor as they had a few anti-patterns that had many associated dynamic promises, and the refactorings were simple enough such that we could verify their correctness.

Discussion. Overall, it is difficult to measure the effect of the removal of runtime promises on the overall performance of applications, due mostly to their asynchronous nature. Even if thousands of redundant promises are eliminated, it is possible that the application was waiting on another operation which takes longer than the sum total of the lifetimes of the eliminated promises.

The elimination of anti-patterns reduces the number of promises created and enables additional parallelism, which may speed up the execution of the affected code fragments.

7.5 RQ4: What is the performance of *DrAsync*?

There are three main components to the run time of *DrAsync*.

First, the time to build the QL databases is reported in column **"QLDB Build Time"** in Table 2—the build times vary, but are only exceptionally high for `flowcrypt-browser` and `rmrk-tools`. Note that this only needs to be done once per project (it needs to be rebuilt when code changes, however), and the database can be reused for other CodeQL queries; linting, by comparison, would be much faster but cannot detect all of the anti-patterns detected by *DrAsync*. To put this number into context, the mean run time of the test suites are found under the first **Mean** column.

Second, the time to run the anti-pattern detection queries is quite low: we ran 160 queries (8 anti-patterns $\times$ 20 projects) in sequence, and only 14 of the 160 query/project combinations took over 30s, and the mean run time was 18.4s. The full query run times are available in supplemental material.

Finally, *DrAsync*'s dynamic analysis adds roughly 27% performance overhead (harmonic mean from column **Overhead of Instrumentation**). Note that, for the **Mean** columns under **Test Time (Before/After)**, the means reported are taken over 20 test suite executions, and the standard deviation of those runs is reported in the **StDev** columns. The overhead was calculated by dividing the mean test suite execution time with instrumentation by the mean test suite execution time without instrumentation. Importantly, note that the subject applications vary wildly in size, and *DrAsync*'s run time is reasonable in all cases.

DrAsync runs quickly, and the performance of the tool scales well as code size increases.

8 THREATS TO VALIDITY

There are several factors that threaten the validity of our results. First, the selection of subject applications used for our evaluation may not be representative. We attempted to mitigate this by randomly selecting applications that met specified criteria that made them suitable subjects for analysis. Also, note that the subject applications include popular and well-maintained projects from major vendors such as Microsoft and SAP. Second, the anti-pattern instances selected in our case study may not be representative. We attempted to mitigate this by randomly selecting these instances, and selected no more than three instances from any one project. Third, our experiences in manually refactoring the anti-pattern instances may be subject to bias and errors. To mitigate the risk of mistakes in the manual refactorings, we focused on anti-pattern instances that are executed by the application's test suite so that we could check for behavioral differences by running the tests. As for bias, we were unfamiliar with the source code for the subject applications, we made an effort to randomly select subjects for the case study, and we highlighted both positive and negative refactoring experiences. Finally, regarding the performance implications of eliminating anti-patterns, one may object that the observed speedups are small and only apply to code fragments in three selected subject applications, under idealized conditions. This is correct, and we do not make broader claims in this regard.

9 RELATED WORK

Several categories of related work can be distinguished: detection of anti-patterns in JavaScript software, profiling concurrent applications, and performance visualization.

JavaScript Anti-Patterns. The detection and remediation of anti-patterns in software has long been a part of good software development practices. Chapter 3 in Fowler's seminal book on refactoring [17] enumerates a number of "code smells" that can be addressed using the refactorings presented in the later chapters.

Several tools for static analysis and style have been developed [2, 5, 6] that check a broad range of rules for identifying potential quality issues in JavaScript software. ESLint [5] supports several rules concerned with `async/await` such as `no-await-in-loop` for detecting the use of `await` in loops. Our research goes beyond ESLint by considering a broader range of asynchronous anti-patterns, visualizing the behavior of asynchronous applications, and combining more sophisticated static analysis and dynamic analysis. Further, ESLint only detects three of the eight anti-patterns reported in this paper: *loopOverArrayWithAwait*, *asyncFunctionAwaitedReturn*, and *asyncFunctionNoAwait* (ESLint flags *any* loop with an await inside, while our anti-pattern is specific to loops over arrays, which in our experience is more likely to amenable to refactoring). ESLint also currently does not support the data-flow analysis required to detect several anti-patterns described in the paper.

Madsen et al. [25] defined the *event-based call graph*, which extends the traditional notion of a call graph with nodes and edges that reflect the flow of control due to event-handling. Recently, Arteca et al. [10] presented a statistical analysis for detecting event listeners that are likely to be dead code due to bugs in event-handling code.

Madsen et al. [24] presented a formal semantics for JavaScript promises, and defined the *promise graph* capturing relationships between promises, and use it to identify bugs found on StackOverflow. Alimadadi et al. [9] present PromiseKeeper, a tool that constructs promise graphs using dynamic analysis, defining a number of dynamic anti-patterns in promise graphs such as unhandled promise rejections. The work by Madsen et al. and Alimadadi et al. predates JavaScript's `async/await` feature. While our work and PromiseKeeper are concerned with the visualization of execution behavior of promise-based code, the visualizations are very different: PromiseKeeper provides a fine-grained visualization of promises and the functions and values they interact with, whereas our work is focused on a large-scale visualization that is focused on the performance aspects of promises and await-expressions.

Arteca et al. [11] present a static analysis and refactoring for enabling additional parallellism in JavaScript applications by splitting and reordering `await`-expressions. Gokhale et al. [18] present a static analysis and refactoring for migrating from synchronous to asynchronous APIs in JavaScript applications that involves introducing `async` functions and `await` expressions.

The academic community has also focused on the detection of code smells in JavaScript code that are unrelated to asynchrony. Nguyen et al. [28] present a tool for detecting embedded code smells in web applications using dynamic analysis. Fard and Mesbah [16] identify 13 code smells that commonly arise in JavaScript software and present a technique based on static and dynamic analysis to detect them. Johannes et al. [22] report on a large-scale empirical study that investigates the relation between code smells in JavaScript software and the fault-proneness of the program parts containing the code smells. Gong et al. [19] present DLint, a tool for detecting code quality issues using dynamic analysis rather than the traditional static analysis.

Profiling concurrent applications. Early work in this area by Waheed and Rover [32] considered techniques for visualizing the performance of parallel programs at the processor level, using techniques from the scientific visualization community. Miller et al. [27] present Paradyn, a tool for measuring and visualizing the performance of large-scale parallel programs using an adaptive instrumentation targeted at long-running applications. Paradyn differs from our work in that it selectively instruments code and visualizes the program as a graph using a graph coloring technique. Meira et al. [23] present Carnival, a performance measurement tool for determining the underlying causes for waiting time in distributed memory systems, again at the processor level. Carnival differs in that it measuring wait times that rely on synchronization primitives used on multi-processor (as opposed to single core) systems.

Joao et al. [21] present a technique for detecting performance bottlenecks in multi-threaded applications (critical sections, barriers, and slow pipeline stages) that have the effect of serializing program execution. Unlike [21], our technique is implemented entirely using source code instrumentation and our focus is on visualizing anti-patterns so that users can remedy them manually.

Dutta et al. [14] present a technique for classifying performance bottlenecks in multi-threaded applications, differentiating between *on-chip* and *off-chip* Unlike our approach, Dutta's only provides an overall assessment, and it does not identify specific regions in the code that constitute the most significant performance bottlenecks.

Software Visualization. Recent work by Tominaga et al. [31] built a tool called AwaitViz to capture instances of async/await in order to visualize execution order focus on improving programmer comprehension of the code. Additional visualizations on understanding async/await was done by Sun et al. by generating Async Graphs [30]. The async graphs are used to help identify bugs related to asynchronous execution and primarily focus on *when* specific events happen during the asynchronous flow of execution in Node.js applications for bug detection. Additional concurrency profiling tools with visualizations in IDEs have been created, focusing on multi-threaded applications and resource utilization: JetBrains's PyCharm Thread Concurrency Visualization [7], Visual Studio's Concurrency Visualizer [1], and Intel's VTune [8].

10 CONCLUSION

We identified 8 anti-patterns that commonly occur in JavaScript code that uses promises and `async/await`. We presented *DrAsync*, a tool that relies on a combination of static and dynamic analysis to detect instances of anti-patterns, and that provides an interactive visualization to help programmers quickly diagnose quality issues and performance bottlenecks in their asynchronous applications.

In an empirical evaluation, *DrAsync* detected 2.6K anti-patterns in 20 subject applications, which were executed 24K times in the aggregate. We report on a case study in which we manually attempted to refactor 10 instances of each anti-pattern, concluding that the majority of *DrAsync*'s findings are actionable, and that refactoring anti-patterns may improve the performance of the affected code.

As future work, we plan to grow our catalog of anti-patterns, and refine the existing anti-patterns to exclude corner cases where successful refactoring is unlikely.

11 DATA AVAILABILITY

Experimental data associated with this research is available on Zenodo: https://doi.org/10.5281/zenodo.5428997. A software artifact is also available on Zenodo: https://doi.org/10.5281/zenodo.5915257.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation grants CCF-1715153, CCF-1930604, and CCF-1907727. A. Turcotte was also supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2017. Concurrency Visualizer - Visual Studio (Windows) | Microsoft Docs. https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019. (Accessed on 08/20/2021).

[2] 2019. JSHint: A Static Code Analysis Tool for JavaScript. See https://jshint.com/.

[3] 2020. Async hooks | Node.js v16.6.0 Documentation. https://nodejs.org/api/async_hooks.html. (Accessed on 08/02/2021).

[4] 2021. CodeQL for research | GitHub Security Lab. https://securitylab.github.com/tools/codeql/. (Accessed on 08/10/2021).

[5] 2021. ESLint: Find and fix problems in your JavaScript code. See https://eslint.org/.

[6] 2021. JSLint. See https://www.jslint.com/.

[7] 2021. Thread Concurrency Visualization | PyCharm. https://www.jetbrains.com/help/pycharm/thread-concurrency-visualization.html. (Accessed on 08/20/2021).

[8] 2021. Threading Analysis. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/parallelism-analysis-group/threading-analysis.html. (Accessed on 08/20/2021).

[9] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 162:1–162:26. https://doi.org/10.1145/3276532

[10] Ellen Arteca, Max Schäfer, and Frank Tip. 2022. Learning How to Listen: Automatically Finding Bug Patterns in Event-Driven JavaScript APIs. *IEEE Trans. Software Eng.* (2022). To appear.

[11] Ellen Arteca, Frank Tip, and Max Schäfer. 2021. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs)*, Anders Møller and Manu Sridharan (Eds.), Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.7

[12] Ellen Arteca and Alexi Turcotte. 2022. npm-filter: Automating the mining of dynamic information from npm packages. *arXiv preprint arXiv:2201.08452* (2022).

[13] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy.* 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[14] Sourav Dutta, Sheheeda Manakkadu, and Dimitri Kagaris. 2014. Classifying Performance Bottlenecks in Multi-threaded Applications. In *IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2014, Aizu-Wakamatsu, Japan, September 23-25, 2014.* IEEE Computer Society, 341–345. https://doi.org/10.1109/MCSoC.2014.55

[15] ECMA. 2021. ECMAScript 2021 Language Specification. Available from https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[16] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013.* IEEE Computer Society, 116–125. https://doi.org/10.1109/SCAM.2013.6648192

[17] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[18] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485537

[19] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 94–105. https://doi.org/10.1145/2771783.2771809

[20] Maila Hardin, Daniel Hom, Ross Perez, and Lori Williams. 2012. Which chart or graph is right for you? *Tell Impactful Stories with Data. Tableau Software* (2012).

[21] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012.* 223–234. https://doi.org/10.1145/2150976.2151001

[22] David Johannes, Foutse Khomh, and Giuliano Antoniol. 2019. A large-scale empirical study of code smells in JavaScript projects. *Softw. Qual. J.* 27, 3 (2019), 1271–1314. https://doi.org/10.1007/s11219-019-09442-9

[23] Wagner Meira Jr., Thomas J. LeBlanc, and Alexandros Poulos. 1996. Waiting Time Analysis and Performance Visualization in Carnival. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT'96).* 1–10.

[24] Magnus Madsen, Ondrej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 86:1–86:24. https://doi.org/10.1145/3133910

[25] Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 505–519. https://doi.org/10.1145/2814270.2814272

[26] Lauren McCarthy, Casey Reas, and Ben Fry. 2015. *Getting started with P5. js: Making interactive graphics in JavaScript and processing.* Maker Media, Inc.

[27] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28, 11 (1995), 37–46. https://doi.org/10.1109/2.471178

[28] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2012. Detection of embedded code smells in dynamic web applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, Michael Goedicke, Tim Menzies, and Motoshi Saeki (Eds.). ACM, 282–285. https://doi.org/10.1145/2351676.2351724

[29] Ben Shneiderman. 2003. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization.* Elsevier, 364–371.

[30] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. Reasoning about the Node. js event loop using Async Graphs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 61–72.

[31] Ena Tominaga, Yoshitaka Arahori, and Katsuhiko Gondow. 2019. AwaitViz: a visualizer of JavaScript's async/await execution order. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing.* 2515–2524.

[32] Abdul Waheed and Diane T. Rover. 1993. Performance Visualization of Parallel Programs. In *Proceedings IEEE Visualization '93, San Jose, California, USA, October 25-29, 1993.* 174–182. https://doi.org/10.1109/VISUAL.1993.398866