**DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript**

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

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

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

Frank Tip  
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

**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 26K 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

```javascript
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:

```javascript
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:

```javascript
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. Similarly, the following code snippet:

```javascript
p3.catch((e) => console.log("error!": e))
```

will cause the text "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:

```javascript
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:

```javascript
let p4 = Promise.resolve(5);
let p5 = new Promise( (resolve, reject) =>
  setTimeout(() => resolve(6), 1000));
p4.then((v) => p5 + v)
  .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.

1The 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 \([p_1, \ldots, p_n]\) as an argument and returns a promise that is either fulfilled with an array \([q_1, \ldots, q_n]\) containing the values that these promises are fulfilled with, or that is rejected with a value \(e\), if \(p_i\) is the first promise among \(p_1, \ldots, p_n\) that is rejected, and \(e\) is the value that it is rejected with. Other synchronization functions include Promise.race and Promise.any. For example, the following snippet prints \([3, 42, "foo"]\) after 1 second:

```javascript
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 fulfilled 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.

```javascript
import fs from ‘fs’;
async function analyzeDir(dName) {
  let fNames = await fs.promises.readdir(dName);
  let ps = fNames.map((fName) => fs.promises.stat(fName));
  let fStats = await Promise.all(ps);
  let sum = fStats.reduce((acc, v) => acc + v.size, 0);
  console.log(sum);
}
```

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 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:

```javascript
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:

```javascript
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:

```javascript
async getModuleInfo(name) {
  let info = this._dependencyInfos.get(name);
  if (info == null) {
    info = Promise.resolve().then(async () => {
      const resource = await this.findResource(name);
      return determineDependencyInfo(resource, ...);
    });
    this._dependencyInfos.set(name, info);
  }
  return info;
}
```

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

\(^{1}\)Adapted from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all.

\(^{2}\)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.

\(^{3}\)Since no argument is passed in the call to Promise.resolve, the value undefined is used by default.
that do not overlap (see Figure 2).

back to the main event loop at this time so that other event handlers
until the promise returned by
rejected if an error occurs. It is important to note that this use of
lled once

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.

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

```javascript
async getModuleInfo(name) {
  let info = this._dependencyInfos.get(name);
  if (info == null) {
    info = (async () => {
      const resource = await this.findResource(name);
      return determineDependencyInfo(resource, ...);
    })();
  } else {
    return info;
  }
}
```

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:

```javascript
async function cpDir(source, target) {
  const files = await readdir(source);
  await Promise.all(files.map(file => {
    const sourceEntry = path.join(source, file);
    const targetEntry = path.join(target, file);
    return cp(sourceEntry, targetEntry);
  }));
}
```

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 code3, 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

3Section 7.1 provides further detail on the process for selecting subject applications.

4https://www.stackoverflow.com/questions/23683743

5https://www.grouparoo.com/blog/promisifying-node-functions
asyncFunctionNoAwait = \{ f \mid f \text{ async} \land (\exists e_0, e_1 : e_0 = await e_1 \Rightarrow e_0 \neq f) \}

asyncFunctionAwaitedReturn = \{ f \mid f \text{ async} \land (\exists e_0, e_1 : e_0 = return e_1 \land e_0 < f) \Rightarrow \exists e_2 : e_1 = await e_2 \}

loopOverArrayWithAwait = \{ s \mid \exists e_0, e_1, e_2, e_3, s_0 : s_0 = for(e_0, e_1, e_2)(s_1) \land isArrayTest(e_1) \land await e_3 < s_1 \}

promiseResolveThen = \{ e_0 \mid \exists e_1, f : e_0 = Promise.resolve(e_1).then(f) \}

executorOneArgUsed = \{ e_0 \mid e_0 = \exists f, e_0, e_1 : \text{ new Promise}(f) \land e_0 = \text{ arg}(f, 0) \land e_1 = \text{ arg}(f, 1) \land (e_1, e_2 : e_1 < e_2 \land e_1, e_2 \in \{ e_0, e_1 \} \Rightarrow e_1 = e_2) \}

reactionReturnsPromise = \{ e_0 \mid \exists e_1, e_2, f : e_0 = e_1.then(f) \land return e_2 < f \land (e_2 = \text{ Promise.resolve}(\cdots) \lor e_2 = \text{ Promise.reject}(\cdots)) \}

customPromisification = \{ e_0 \mid \exists f_0, f_1, f_2, s_0, s_1, s_2, e_0 : \text{ new Promise}(f_0) \land f_1(\text{ ... }, f_2) < f_0 \land if (\cdots) \{ s_0 \} \text{ else } \{ s_1 \} < f_2 \land e_0 = \text{ arg}(f_0, 0) \land e_1 = \text{ arg}(f_0, 1) \land \{ (e_0 < s_0 \land e_1 < s_1) \lor (e_0 < s_1 \land e_1 < s_0) \} \}

explicitPromiseConstructor = \{ e_0 \mid \exists e_1, f_0, f_1, f_2, e_0, e_1, e_2, e_3 : \text{ new Promise}(f_0) \land e_1, \text{ then}(f_1).\text{ catch}(f_2) < f_0 \land e_0 = \text{ arg}(f_0, 0) \land e_1 = \text{ arg}(f_0, 1) \land e_2 = \text{ arg}(f_1, 0) \land e_3 = \text{ arg}(f_2, 0) \land e_0(e_2) < f_1 \land e_3 < f_2 \}

Figure 3: Anti-patterns that commonly occur in asynchronous JavaScript code.
resolve) and \( s_2 \) calls the function passed as the second parameter to the executor (usually called \texttt{reject} or vice versa. In such cases, it is often possible to utilize the \texttt{util.promisify} promiscification 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.

\textit{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 \( e_1 \), \( \text{then}(f_1).\text{catch}(f_2) \), 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 \( e_1 \).

\section{IMPLEMENTATION}

\textit{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.\footnote{Artifact link: https://doi.org/10.5281/zenodo.5915257}

\subsection{Static Analysis}

The static analysis uses CodeQL \cite{Java, C} 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 \texttt{promiseResolveThen} anti-pattern looks as follows:

\begin{verbatim}
predicate promiseDotResolveDotThen(MethodCallExpr c) {
  c.getMethodName() = "then" and
  c.getReceiver() instanceof MethodCallExpr and
  ((MethodCallExpr c).getReceiver()).getMethodName = "resolve"
}
\end{verbatim}

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

\subsection{Dynamic Analysis}

\textit{DrAsync} relies on the Node.js Async hooks API \cite{NodeJS} to instrument source code to log the creation and settlement of promises, to record when \texttt{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 \texttt{async} function), enabling us to calculate how often each anti-pattern is executed.

Furthermore, information is recorded about dependencies between promises: the Async hooks API provides a unique \texttt{asyncId} for each promise, as well as a \texttt{triggerAsyncId}, which is the \texttt{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 \texttt{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 \textit{DrAsync}'s interactive visualization component.

\subsection{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.

\textit{DrAsync}'s interactive visualization tool was developed using the P5.js framework \cite{P5.js}. Figure 4 shows a screenshot of a visualization produced by \textit{DrAsync}, which follows the standard information taxonomy by providing: a high level overview, filters, and details on demand \cite{ICSE20}. We briefly discuss \textit{DrAsync}'s different views.

\textit{Promise Lifetime View and Source Code View}. This view (labeled \( A \) in the figure) is organized as a Gantt Chart \cite{GanttChart}. 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.

\textit{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 \texttt{loopOverArrayWithAwait} that may benefit from refactoring).

\textit{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.

\textit{Summary View and Filters}. This view, labeled \( E \), shows 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 \textit{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

<table>
<thead>
<tr>
<th>Anti-Pattern</th>
<th># Successful</th>
<th># Unsuccessful</th>
</tr>
</thead>
<tbody>
<tr>
<td>asyncFunctionAwaitedReturn</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>loopOverArrayWithAwait</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>promiseResolveThen</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>reactionReturnsPromise</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>executorOneArgUsed</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>customPromisification</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>explicitPromiseConstructor</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

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 $\mathcal{F}$), or on promises whose creation site matches a specified text string (see view labeled $\mathcal{G}$).

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:

```typescript
export async function readSpec(input: string): Promise<string> {
    if (input.startsWith('https://')) {
        return await readSpecFromHttp(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.

loopOverArrayAwait. 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 following code snippet on lines 159–162 in file `src/TemplateLayout.js` in `eleventy`:

```javascript
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 parallelize the loop using `Promise.all`. 
After this refactoring, it is evident that the resulting code lacks a promise that is immediately fulfilled or rejected in response to external events. 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:

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

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. In all but one of the successful cases, using反应ReturnsPromise. For this anti-pattern, 9 of the 10 cases we have additional insights enabling them to refactor the code. Each of the refactorings is reported on in the supplementary materials.

```
return async function (data) {
  /* return new Promise((resolve, reject) {} */
  tmpl.render(data, function (err, res) {
    if (err) {
      reject(err);
    } else {
      resolve(res);
    }
  });
  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/backends.ts:

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

Here, catch and .then return promises anyway, so explicitly returning a promise that is immediately fulfilled or rejected is needless.
which we crafted experiments that emphasize the performance of

Table 3: Anti-pattern stats. Legend: P1 = explicitPromiseConstructor, P2 = customPromiseFication, P3 = promiseResolveThen, P4 = 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td>appcenter-cli</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boostnote</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>browserTime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codecept15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dash.js</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eleventy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ergos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fastify</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flow-issuer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mercuris</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>netlify-cms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>openapi-typescript-codegen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remk-tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stencil</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strapi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>treeherder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ui5-builder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vscode-javascript</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vuepress</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 (0.53s vs 0.34s), leading to more predictable performance.

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

```javascript
for (const { 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%.

This instance of the promiseResolveThen anti-pattern occurs in the strapi application:

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

We selected an instance of this anti-pattern to assess the performance impact of eliminating more than just the `loopOverArrayWith-Await` 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 × 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, visual-
ing 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 ex-
tends 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 StackOver-
flow. Alimadadi et al. [9] present PromiseKeeper, a tool that con-
structs 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
predate JavaScript’s `async/await` feature. While our work and
PromiseKeeper are concerned with the visualization of execution be-
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 per-
formance aspects of promises and await-expressions.

Arteca et al. [11] present a static analysis and refactoring for
enabling additional parallelism in JavaScript applications by split-
ting 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 in-
troducing `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 Wa-
heed and Rover [32] considered techniques for visualizing the per-
formance of parallel programs at the processor level, using tech-
niques from the scientific visualization community. Miller et al. [27]
present Paradyne, a tool for measuring and visualizing the perform-
ce of large-scale parallel programs using an adaptive instru-
mentation targeted at long-running applications. Paradyne 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 bot-
tlenecks 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 com-
prehension 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 asyn-
chronous execution and primarily focus on when specific events
happen during the asynchronous flow of execution in Node.js ap-
lications 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 Zen-
odo: 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


