Reducing Over-Synchronization in JavaScript Applications

Anonymous author
Anonymous affiliation

Abstract

JavaScript is a single-threaded programming language, so asynchronous programming is practiced out of necessity to ensure that applications remain responsive in the presence of user input or interactions with file systems and networks. However, many JavaScript applications execute in environments that do exhibit concurrency by, e.g., interacting with multiple or concurrent servers, or by using file systems managed by operating systems that support concurrent I/O. In this paper, we demonstrate that JavaScript programmers often schedule asynchronous I/O operations suboptimally, and that reordering such operations may yield significant performance benefits. Concretely, we define a static side-effect analysis that can be used to determine how asynchronous I/O operations can be refactored so that asynchronous I/O-related requests are made as early as possible, and so that the results of these requests are awaited as late as possible. While our static analysis is potentially unsound, we have not encountered any situations where it suggested reorderings that change program behavior. We evaluate the refactoring on 15 applications that perform file-related I/O. For these applications, we observe average speedups ranging between 0.7% and 8.7% for the tests that execute refactored code (4.4% on average).

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases asynchronous programming, refactoring, side-effect analysis, performance optimization, static analysis, JavaScript

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

In JavaScript, asynchronous programming is practiced out of necessity: JavaScript is a single-threaded language and relying on asynchronously invoked functions/callbacks is the only way for applications to remain responsive in the presence of user input and file system or network-related I/O. Originally, JavaScript accommodated asynchrony using event-driven programming, by organizing the program as a collection of event handlers that are invoked from a main event loop when their associated event is emitted. However, event-driven programs suffer from event races [23] and other types of errors [18] and lack adequate support for error handling.

In response to these problems, the JavaScript community adopted promises [9, Section 25.6], which enable programmers to create chains of asynchronous computations with proper error handling. However, promises are burdened by a complex syntax where each element in a promise chain requires a call to a higher-order function. To reduce this burden, the `async`/`await` feature [9, Section 6.2.3.1] was introduced in the ECMAScript 8 version of JavaScript, as syntactic sugar for many use cases of promises. A function designated as `async` can await asynchronous computations (either calls to other `async` functions or promises), enabling programmers to write asynchronous programs with minimal syntactic overhead.

The `async`/`await` feature has quickly become widely adopted, and many libraries have adopted promise-based APIs that enable the use of `async`/`await` in user code. However, many programmers are still unfamiliar with promises and `async`/`await` and are insufficiently aware how careless use of these features may negatively impact performance. In particular, programmers often do not think carefully enough about when to create promises that are

associated with initiating asynchronous I/O operations and when to `await` the resolution of those promises and trigger subsequent computations.

As JavaScript is single-threaded, it does not support multi-threading/concurrency at the language level. However, the placement of promise-creation operations and the awaiting of results of asynchronous operations can have significant performance implications because many JavaScript applications *execute in environments that do feature concurrency*. For example, a JavaScript application can interact with servers, file systems, or databases that can execute multiple operations concurrently. Therefore, in general, it is desirable to trigger asynchronous activities *as early as possible* and await their results *as late as possible*, so that a program can perform useful computations while asynchronous I/O requests are being processed in the environment.

In this paper, we use static interprocedural side-effect analysis [3] to detect situations where oversynchronization occurs in JavaScript applications. For a given statement s, our analysis computes sets $MOD(s)$ and $REF(s)$ of access paths [19] that represent sets of memory locations modified and referenced by s, respectively. We use this analysis to suggest how await-expressions of the form `await` e_{io} can be refactored, where e_{io} is an expression that creates a promise that is settled when an asynchronous I/O operation completes. The refactoring "splits" such await-expressions so that: (i) the promise creation is moved to the earliest possible location within the same scope and (ii) the awaiting of the result of the promise is moved to the latest possible location within the same scope. The static side-effect analysis is unsound, so the programmer needs to ensure that program behavior is preserved.

We implemented the static analysis in QL [2], and incorporated it into a tool called *ReSynchronizer*[1] that automatically refactors I/O-related await-expressions. In an experimental evaluation, we applied *ReSynchronizer* to 15 open-source Node.js applications that perform asynchronous file-system I/O. Our findings indicate that, on these subject applications, our approach yields speedups ranging between 0.7% and 8.7% when running tests that execute refactored code (4.4% on average). We detected no situations where unsoundness in the static analysis resulted in broken tests.

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

- The design of a static side-effect analysis for determining MOD and REF sets of access paths, and the use of this analysis to suggest how I/O-related await-expressions can be refactored to improve performance,
- Implementation of this analysis in a tool called *ReSynchronizer*, and
- An evaluation of *ReSynchronizer* on 15 open-source projects, demonstrating that our approach can produce significant speedups and scales to real-world applications.

The remainder of this paper is organized as follows. Section 2 reviews JavaScript's promises and `async`/`await` features. In Section 3, a real-world example is presented that illustrates how reordering await-expressions may yield performance benefits. Section 4 presents the side-effect analysis that serves as the foundation for our approach. Section 5 presents an evaluation on our approach on open-source JavaScript projects that use `async`/`await`. Related work is discussed in Section 6. Finally, Section 7 concludes and provides directions for future work.

[1] The source code of the tool and all of our data will be made available as an artifact.

2 Review of promises and `async/await`

This section presents a brief review of JavaScript's promises [9, Section 25.6] and the `async`/`await` feature [9, Section 6.2.3.1] for asynchronous programming. Readers already familiar with these concepts may skip this section.

A *promise* represents the result of an asynchronous computation, and is in one of three states. Upon creation, a promise is in the *pending* state, from where it may transition to the *fulfilled* state, if the asynchronous computation completes successfully, or to the *rejected* state, if an error occurs. A promise is *settled* if it is in the fulfilled or rejected state. The state of a promise can change only once, i.e., once a promise is settled, its state will never change again.

Promises are created by invoking the `Promise` constructor, which expects as an argument a function that itself expects two arguments, `resolve` and `reject`. Here, `resolve` and `reject` are functions for fulfilling or rejecting a promise with a given value, respectively. For example, the following code:

```
1 const p = new Promise((resolve, reject) => {
2   setTimeout(function() { resolve(17); }, 1000);
3 });
```

creates a promise that is fulfilled after 1 second with the value `17`.

Once a promise has been created, the `then` method can be used to register *reactions* on it, i.e., functions that are invoked asynchronously from the main event loop when the promise is fulfilled or rejected. Consider extending the previous example as follows:

```
4 p.then(function f(v){ console.log(v); return v+1; });
```

In this case, when the promise assigned to `p` is fulfilled, the value that it was resolved with will be passed as an argument to the resolve-reaction `f`, causing it to print the value `17` and return the value `18`.

The `then` function creates a promise, which is resolved with the value returned by the reaction. This enables the creation of a *promise chain* of asynchronous computations. For example:

```
5 p.then(function(x){ return x+1; })
6  .then(function(y){ return y+2; })
7  .then(function(z){ console.log(z); })
```

results in the value `20` being printed.

The examples given so far only specify resolve-reactions, but in general, care must be taken to handle failures. In particular, the promise implicitly created by calling `then` is rejected if an exception occurs during the execution of the reaction. To this end, the `catch` method can be used to register reject-reactions that are to be executed when a promise is rejected. The `catch` method is commonly used at the end of a promise chain. For example:

```
8  p.then(function(x){ return x+1; })
9  .then(function(y){ throw new Error(); })
10 .then(function(z){ console.log(z); })
11 .catch(function(err){ console.log('error!'); })
```

results in `'error!'` being printed.

Recently, several popular libraries for performing I/O-related operations have adopted promise-based APIs. For example, `fs-extra` is a popular library that provides various file utilities, including a method `copy` for copying files. The `copy` function returns a promise that

is fulfilled when the file-copy operation completes successfully, and that is rejected if an I/O error occurs, enabling programmers to write code such as[2]:

```
const fs = require('fs-extra')
fs.copy('/tmp/myfile', '/tmp/mynewfile')
  .then(() => console.log('success!'))
  .catch(err => console.error(err))
```

JavaScript's `async`/`await` feature builds on promises. A function can be designated as `async` to indicate that it performs an asynchronous computation. An `async` function f returns a promise: if f returns a value, then its associated promise is fulfilled with that value, and if an exception is thrown during execution of f, its associated promise is rejected with the thrown value. The `await` keyword may be used inside the body of `async` functions, to accommodate situations where the function relies on other asynchronous computations. Given an expression e that evaluates to a promise, the execution of an expression `await` e that occurs in the body of an async function f will cause execution of f to be suspended, and control flow will revert to the main event loop. Later, when the promise is fulfilled with a value v, execution of f will resume, and the await-expression will evaluate to v. In the case where the promise that e evaluates to is rejected with a value w, execution will resume and the evaluation of the await-expression will throw w as an exception that can be handled using the standard `try`/`catch` mechanism. Below, we show a variant of the previous example rewritten to use `async`/`await`.

```
async function copyFiles(){
  try {
    await fs.copy('/tmp/myfile', '/tmp/mynewfile')
    console.log('success!')
  } catch (err) {
    console.error(err)
  }
}
```

As is clear from this example, the use of `async`/`await` results in code that is more easily readable. Here, execution of `copyFiles` will be suspended when the await-expression on line 18 is encountered. Later, when the file-copy operation has completed, execution will resume. If the operation completes successfully, line 19 will execute and a message `'success!'` is printed. Otherwise, an exception is thrown, causing the handler on line 20 to execute.

As a final comment, we remark on the fact that it is straightforward to convert an existing event-based API into an equivalent promise-based API, by creating a promise that is settled when an event arrives. Various utility libraries exist for "promisification" of event-driven APIs, e.g., `util.promisify` [12] and `universalify` [28].

3 Motivating Example

We now present a motivating example that illustrates the performance benefits that may result from reordering await-expressions. The example was taken from Kactus[3], a git-based version control tool for design sketches. Figure 1(a) shows a function `getStatus` that is defined in the file `status.ts`[4]. As an `async` function, `getStatus` may depend on the values

[2] Example taken from https://www.npmjs.com/package/fs-extra.
[3] See https://kactus.io/.
[4] Some details not pertinent to the program transformation under consideration have been elided here. The complete source code can be found at https://github.com/kactus-io/kactus.

```
24 export async function getStatus(repository) {
25     const stdout = await gitMergeTree(repository)
26     const parsed = parsePorcelainStatus(stdout) (A)
27     const entries = parsed.filter(isStatusEntry) (B)
28
29     const hasMergeHead = await fs.pathExists(getMergeHead(repository))
30     const hasConflicts = entries.some(isConflict) (C)
31
32     const state = await getRebaseInternalState(repository)
33
34     const conflictDetails = await getConflictDetails(repository, hasMergeHead,
35                                                       hasConflicts, state)
36
37     buildStatusMap(conflictDetails) (G)
38 }
```

(a)

```
39 async function getRebaseInternalState(repository) {
40     let targetBranch = await fs.readFile(getHeadName(repository))
41     if (targetBranch.startsWith('refs/heads/'))
42       targetBranch = targetBranch.substr(11).trim() (D)
43
44     let baseBranchTip = await fs.readFile(getOnto(repository))
45     baseBranchTip = baseBranchTip.trim() (E)
46
47     return { targetBranch, baseBranchTip } (F)
48 }
```

(b)

Figure 1 Example.

computed by other `async` functions, by awaiting such values in await-expressions. The code shown in Figure 1(a) contains four such await-expressions, on lines 25, 29, 32, and 34, which we now consider in some detail:

- The await-expression on line 25 invokes an async function `gitMergeTree` (omitted for brevity) that relies on the `dugite` and `child_process` libraries to execute a `git merge-tree` command in a separate process.
- The await-expression on line 29 calls an `async` function `pathExists` from the `fs-extra` package mentioned above, to check if a file `MERGE_HEAD` exists in the `.git` directory. `pathExists` is implemented in terms of the function `access` from the built-in `fs` package provided by the Node.js platform, which in turn triggers the execution of an OS-level file-read operation.
- The await-expression on line 32 calls an `async` function `getRebaseInternalState`, of which we show some relevant fragments in Figure 1(b). Note in particular that two asynchronous file-read operations are performed on lines 40 and 44, using the `readFile` function from `fs-extra`. Each of these calls causes the execution of an OS-level file-read operation.
- The await-expression on line 34 invokes an `async` utility function `getConflictDetails` (omitted for brevity) to gather information about files that have merge conflicts.

Figure 2 shows a UML Sequence Diagram[5] that visualizes the flow of control during the execution of `getStatus`. In this diagram, labels (A) – (G) inside timelines indicate when code

[5] To prevent clutter, the diagram only shows asynchronous calls and returns and elides details that are not relevant to the example under consideration.

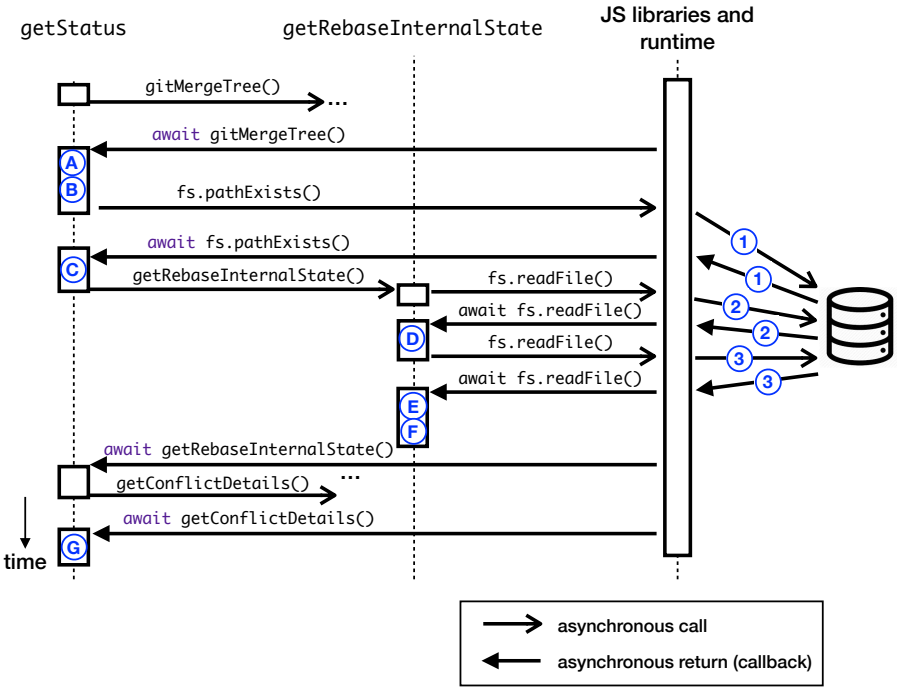

Figure 2 Visualization of the execution of `getStatus`.

fragments labeled similarly in Figure 1 execute. Furthermore, labels ① – ③ indicate when file I/O operations associated with the call to `fs.pathExists` on line 29 and with the two calls to `fs.readFile` in function `getRebaseInternalState` execute.

The leftmost timeline in the diagram depicts the execution of code fragments in the `getStatus` function itself. The middle timeline depicts the execution of function `getRebaseInternalState`. The timeline on the right, labeled 'JS libraries and runtime' visualizes the execution of functions in JavaScript libraries such as `fs-extra` and other libraries that the application relies on such as `universalify` [28], `graceful-fs` [25], and libraries such as the `fs` file-system package that are included with the JS runtime.

Taking a closer look at the diagram, we can observe that the code fragments Ⓐ and Ⓑ will run before I/O operation ① is initiated. Then, after I/O operation ① has completed, code fragment Ⓒ is evaluated. Next, when `getRebaseInternalState` is invoked, I/O operation ② is initiated. After it has completed, code fragment Ⓓ executes, which is followed in turn by I/O operation ③. When that operation completes, code fragments Ⓔ and Ⓕ execute, and finally code fragment Ⓖ executes. Crucially, the use of `await` on lines 29, 32, 40, and 44 ensures that each file I/O operation must complete before execution can proceed. As a result, *the file I/O operations ① – ③ execute in a strictly sequential order, where each operation must complete before the next one is dispatched.*

However, most JavaScript runtimes are capable of processing multiple asynchronous I/O requests concurrently. In this paper, we demonstrate that it is often possible to refactor JavaScript code in a way that enables for multiple I/O requests to be processed concurrently with the main program. The refactoring that we propose targets expressions of the form `await` e_{io}, where e_{io} is an expression that creates a promise that is settled when an asynchronous I/O operation completes. The expressions `await fs.pathExists(getMergeHead(repository))` on line 29 and `await getRebaseInternalState (repository)` on line 32 are examples of such expressions, as are the await-expressions on lines 40 and 44 in Figure 1(b).

```
49 export async function getStatus(repository) {
50     const stdout = await gitMergeTree(repository)
51     const parsed = parsePorcelainStatus(stdout) (A)
52
53     var T1 = fs.pathExists(getMergeHead(repository))
54     var T2 = getRebaseInternalState(repository)
55
56     const entries = parsed.filter(isStatusEntry) (B)
57     const hasConflicts = entries.some(isConflict) (C)
58
59     const state = await T2
60     const hasMergeHead = await T1
61
62     const conflictDetails = await getConflictDetails(repository, hasMergeHead,
63                                                       hasConflicts, state)
64
65     buildStatusMap(conflictDetails) (G)
66 }
```

(a)

```
67 async function getRebaseInternalState(repository) {
68     var T3 = fs.readFile(getHeadName(repository))
69     var T4 = fs.readFile(getOnto(repository))
70     let targetBranch = await T3
71     if (targetBranch.startsWith('refs/heads/'))
72       targetBranch = targetBranch.substr(11).trim() (D)
73
74     let baseBranchTip = await T4
75     baseBranchTip = baseBranchTip.trim() (E)
76
77     return { targetBranch, baseBranchTip } (F)
78 }
```

(b)

Figure 3 Example, reordered.

Conceptually, the refactoring involves splitting an expression `await` e_{io} occurring in an async function f into two parts:

1. a local variable declaration `var` $t = e_{io}$ that starts the asynchronous I/O operation and that is placed *as early as possible* in the control-flow graph of f, and
2. an expression `await` t where the result of the asynchronous I/O operation is awaited and that is placed *as late as possible* in the control-flow graph of f.

We will make the notions "as early as possible" and "as late as possible" more precise in Section 4, but intuitively, the idea is that we want to move the expression e_{io} *before* any statement that precedes it provided that this does not change the values computed or side-effects created at any program point. Likewise, we want to move the expression `await` t *after* any statement that follows it provided that this does not alter the values computed or side-effects created at any program point. Section 4 will present a static data flow analysis for determining when statements can be reordered.

Figure 3(a) shows how the `getStatus` function is refactored by our technique. As can be seen in the figure, the await-expression that occurred on line 29 in Figure 1(a) is split into the declaration of a variable `T1` on line 53 and an await-expression on line 60 in Figure 3(a). Likewise, the await-expression that occurred on line 32 in Figure 1(a) is split into the declaration of a variable `T2` on line 54 and an await-expression on line 59 in Figure 3(a).

The await-expression on line 25 cannot be split because it relies on `process.spawn` to

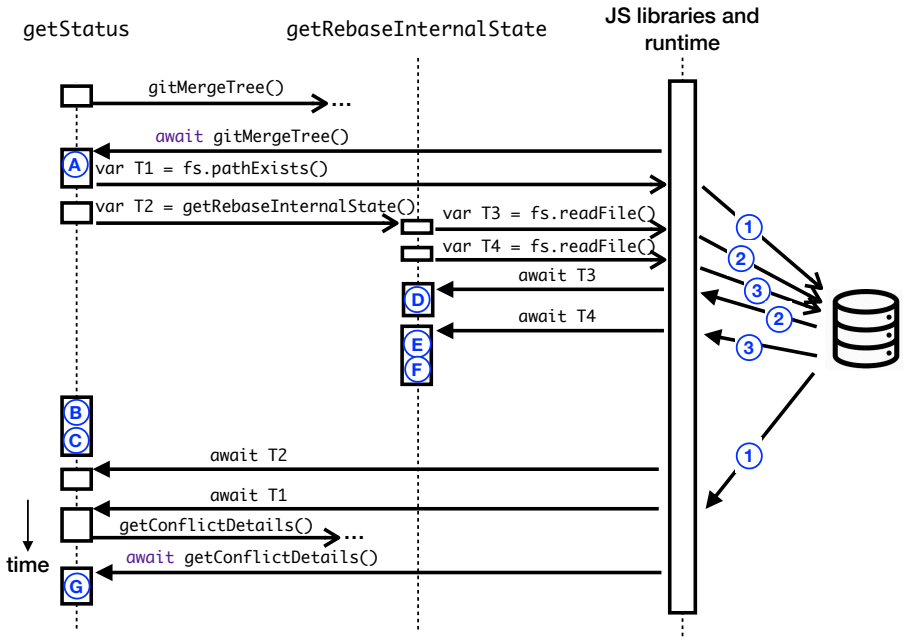

Figure 4 Visualization of the execution of `getStatus` after reordering.

execute a `git merge-tree` command in a separate process, and our analysis conservatively assumes that statements that spawn new processes have side-effects and thus cannot be reordered (this is discussed in detail in Section 4.4). Furthermore, the await-expression on line 34 was not reordered because it references the variable `state` defined on the previous line, and it defines a variable `conflictDetails` that is referenced in the subsequent statement, so any reordering might cause different values to be computed at those program points. Figure 3(b) shows how the two await-expressions in Figure 1(b) have been split similarly.

Figure 4 shows a UML Sequence diagram that visualizes the execution of the refactored `getStatus` method. As can be seen in the figure, the I/O operation labeled (1) is now initiated after code fragment (A) has been executed. However, since the result of this I/O operation is not needed until after code fragment (C) has executed, this I/O operation can now execute *concurrently* with I/O operations (2) and (3). Additional potential for concurrency is enabled by starting I/O operation (3) before awaiting the result of I/O operation (2). Note that, as a result of splitting await-expressions and reordering statements, the labeled code fragments now execute in a slightly different order: (A), (D), (E), (F), (B), (C), (G). Our static analysis, defined in Section 4 inspects the MOD and REF sets of memory locations modified and referenced by statements to determine when reordering is safe. The analysis abstractly represents memory locations using *access paths*, and is presented in Section 4.

At this point, the reader may wonder whether the additional concurrency enabled by the transformation results in performance improvements. For the Kactus project from which the example was taken, a total of 72 I/O-related await-expressions were reordered by our technique, including the ones discussed above. Of the 799 tests associated with Kactus, 172 execute at least one reordered await-expression. For these impacted tests, we observed an average speedup of 7.1%. A detailed discussion of our experimental results follows in Section 5.

4 Approach

This section presents a static analysis for determining how await-expressions can be reordered to reduce over-synchronization. The analysis determines whether reordering adjacent statements may impact program behavior by determining the side-effects of each statement. Here, the *side-effects* of statements are defined in terms of MOD and REF sets [3] of access paths [19]. Below, we will present these concepts before defining predicates that specify when statements can be reordered.

4.1 Access paths

An *access path* represents a set of memory locations referred to by an expression in a program. The access path representation that we use in this paper is based on the work by Mezzetti et al. [19]: starting from a root, the access path records a sequence of property reads, method calls and function parameters that need to be traversed to arrive at the designated locations. It is often also useful to view access paths as representing a set of values, namely those values that are stored in these locations at runtime. Access paths a conform to the following grammar:

a	::=	**root**	a root of an access path
	|	$a.f$	a property f of an object represented by a
	|	$a()$	values returned from a function represented by a
	|	$a(i)$	the i^{th} parameter of a function represented by a
	|	$a_{\textbf{new}}()$	instances of a class represented by a

Mezzetti et al. developed access paths to abstractly represent objects originating from a particular API. As such, their **root** was always of the form **require**(m). We additionally allow variables as roots, including both global variables and local variables, with the latter also covering function parameters including the implicit receiver parameter `this`.

Example 4.1: We give a few examples of access paths:

- The local variable `targetBranch` declared on line 40 in Figure 1 is represented by the access path `targetBranch`.
- The argument `'refs/heads/'` in the method call `targetBranch.startsWith('refs/heads/')` on line 41 is represented by the access path `targetBranch.startsWith(1)`.
- The property-access expression `fs.pathExists` on line 29 is represented by the access path `require(fs-extra).pathExists`.

Note that access paths are not canonical: due to aliasing, it is possible for multiple access paths to represent the same memory locations.

4.2 MOD and REF

Intuitively, for a given statement or expression s, $MOD(s)$ is a set of access paths representing locations modified by s and $REF(s)$ is a set of access paths representing locations referenced by s. Note that in cases where s is a compound statement or expression such as a block, if-statement, or while-statement $MOD(s)$ and $REF(s)$ include all access paths modified/referenced in any component of s, respectively. Furthermore, if s includes a function call $e.f(\cdots)$,

$MOD(s)$ and $REF(s)$ include all access paths modified/referenced in any statement in any function transitively invoked from this call site[6].

When a statement s contains an assignment to an access path a, the set $MOD(s)$ contains a and all access paths that are rooted in a. However, note that we limit the set of access paths in $MOD(s)$ to those that are explicitly referenced in the program. To understand why this must be the case, consider a scenario where a is a string object. Such an object has all properties that are defined on strings[7]. As one particular example, consider the `toString` function defined on strings. Since $a.toString()$ is rooted in a, $MOD(s)$ should include $a.toString()$. The result of $a.toString()$ is also a string, which means that $a.toString().toString()$ is another valid access path rooted in a, and should be included in $MOD(s)$. This could be repeated ad infinitum, and is only one possible example of such an infinite recursive process. So, to ensure that $MOD(s)$ and $REF(s)$ are always finite sets, they only include access paths that are actually accessed in the program.

Note that in JavaScript it is also possible to access properties dynamically, with expressions of the form $e[p]$, where p is a value computed at run time. In such cases, our analysis cannot statically determine which of e's properties is specified by p, and so we conservatively assume that all properties of e are accessed (i.e., all access paths rooted in e).

Example 4.2: Consider the assignment statement

```
let targetBranch = await fs.readFile(getHeadName(repository))
```

on line 40 in Figure 1.

Since we are assigning to `targetBranch`, this statement modifies `targetBranch` *and* all access paths rooted in `targetBranch`. From a quick glance at the code, we can see that two properties of `targetBranch` are accessed (`startsWith` and `substr`) and called as methods, and the `trim` method is called on the result of calling `substr` (and none of these has any further properties accessed). The assignment also contains a call to `getHeadName` – the function body is elided for brevity, but suffice it to say that `getHeadName` does not modify its `repository` argument or any global variables. Taking these considerations into account, the following MOD set is computed for the statement on line 40:

{ `targetBranch`, `targetBranch.startsWith`, `targetBranch.startsWith()`, `targetBranch.substr`, `targetBranch.substr()`, `targetBranch.substr().trim`, `targetBranch.substr().trim()` }

The REF set includes all access paths referenced in the assignment, which includes the call to `fs.readFile` that is represented by the access path **require**(`fs-extra`).`readFile`(), the function `getHeadName`, and the variable `repository`. In the implementation of function `getHeadName`, there is a call to `fs.pathExists`, another to `Path.join`, and an access to the `path` property of the `repository` object. Therefore, the REF set for the statement is:

{ **require**(`fs-extra`), **require**(`Path`), **require**(`fs-extra`).`readFile`, **require**(`fs-extra`).`readFile`(), **require**(`fs-extra`).`pathExists`, **require**(`fs-extra`).`pathExists`(), **require**(`Path`).`join`, **require**(`Path`).`join`(), `repository`, `repository.path` }

[6] Note that for brevity, when describing modification/reference of the locations abstractly represented by an access path, we refer to it as modification/reference of the access path itself.

[7] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String.

Note that, for a given statement s, $MOD(s)$ and $REF(s)$ do not include access paths rooted in local variables, parameters or `this` parameters in scopes disjoint from the scope of s. For example, for the statement on line 32 where we see a call to `getRebaseInternalState`, the MOD set does not include an access path `targetBranch` for the local variable `targetBranch` modified in that function because it has no effect on the calling statement.

4.3 Determining whether statements can be reordered

In order to determine whether two adjacent statements s_1 and s_2 can be reordered, we need to determine whether doing so might change the values computed at either statement. We consider statements s_1 and s_2 *data-independent* if all of the following criteria are satisfied:

1. $MOD(s_1) \cap MOD(s_2) = \emptyset$
2. $MOD(s_1) \cap REF(s_2) = \emptyset$
3. $REF(s_1) \cap MOD(s_2) = \emptyset$

If s_1 and s_2 are not data-independent, then we will say that they are *data-conflicting*.

Example 4.3:

We discussed the MOD set for the statement at line 40 in Figure 1 in Example 4.2. Similarly, the statement on line 44 is an assignment to variable `baseBranchTip`, whose MOD set consists of {`baseBranchTip`, `baseBranchTip.trim`, `baseBranchTip.trim()`}. Since neither of these statements is modifying data that the other is modifying or referencing, these statements are *data-independent*. Note that they do have an overlap in the REF sets: both statements include calls to `fs.readFile`, and access the variable `repository`. However, since these accesses are read-only, the order in which they execute is not required to be preserved. Indeed, in Figure 3, we see that, in the reordered code, the `await` for the `targetBranch` assignment is moved *after* the `baseBranchTip` assignment.

Still considering the statement on line 44, note that this statement *data-conflicts* with the statement on line 45, which uses the value of variable `baseBranchTip`. Since the statement on line 44 has `baseBranchTip` in its MOD set, it data-conflicts with the statement on line 45, indicating that these statements cannot be reordered. Indeed, in Figure 3, we see that the `await` for the assignment of `baseBranchTip` remains *before* the reference to `baseBranchTip` on line 75.

Note that due to the non-canonicity of access paths data independence is not, strictly speaking, a sound criterion for reorderability: if two statements modify the same location under different access paths, we will consider them to be data independent, but it may not be safe to reorder them. We will revisit this issue in Section 5.5.

4.4 Environmental side effects

So far, we have only considered side-effects consisting of referencing and modification of locations through variables and object properties. However, statements may also have side-effects beyond the state of the program itself, such as modifications to file systems, databases, or the environment in which the program is being executed. Our approach to handling such side-effects is to model them in terms of MOD and REF sets for (pseudo-)variables. We distinguish two types of special side effects: global side-effects and environment-specific side-effects, which we discuss below.

4.4.0.1 Global environmental side-effects.

We say that a statement s has a *global side-effect* if it could affect any of the data in the program or its environment. In such cases, our analysis infers that $MOD(s) = \top$ and $REF(s) = \top$, where $\top$ is the set containing all access paths computed for the program. Currently, our analysis flags the following functions as having global side-effects: `eval`, `exec`, `spawn`, `fork`, `run`, and `setTimeout`. All but the last of these functions may execute arbitrary code and `setTimeout` is often used to explicitly force a specific execution order[8].

4.4.0.2 Specific environmental side-effects.

We say a statement has an *environment-specific side-effect* if it can affect a specific aspect of the program's run-time environment, such as the file system or a database. Environment-specific side-effects are modeled in terms of MOD and REF sets for pseudo-variables that are introduced for the aspect of the environment under consideration.

The experiments reported on in this paper focus on applications that access the file system – we model this environment using a single pseudo-variable `__FILESYSTEM__`. Our current implementation flags a statement as having a MOD side-effect specific to the file system if it consists of a call to any of the following functions:

- `fs.write` (or any function starting with `write`, such as `writeSync`, `writeFile`, etc)
- `fs.unlink`, `fs.remove`, `fs.rename`, or `fs.move`
- `fs.copy`
- `fs.mkdir` or `fs.rmdir`
- `fs.output` (or any function starting with `output`, such as `outputFile`, `outputFileSync`, etc.[9])
- `fs.dir`, `fs.file`, `fs.dirSync`, or `fs.fileSync`[10]
- `process.chdir`

For each of these operations, the MOD sets will include `__FILESYSTEM__`. Any other operations that reference the file system (e.g., `fs.readFile`) will have their REF set include `__FILESYSTEM__`.

The result is that no file operations can be reordered around a call with `__FILESYSTEM__` in its MOD set. This means that no file read will ever be reordered around a file write. We have taken this conservative approach because, in many cases, it is not possible to determine precisely which files are being accessed because names of accessed files are specified with string values that may be computed at run time. As future work, we plan to explore approaches for computing environmental side-effects more precisely when the file being accessed is given as a string literal.

Note that any two file reads can be reordered (as seen in our motivating example), since there will never be a data conflict between read-only operations.

---

[8] While conducting our experiments, we ran into cases where reordering awaits around a call to `setTimeout` caused changes in program behavior because the execution order was modified.

[9] These are part of the `fs-extra` package, which is a wrapper for `fs`

[10] These are part of the `tmp` package, another wrapper for `fs`

Input: s statement and a access path
Result: `True` if s modifies a, `False` otherwise

predicate $MOD(s, a)$
  // (i) base case: direct modification of a
  (s has environmental side-effect a $\vee$ s declares or assigns to a)
  $\vee$ // recursive cases...
    // (ii) check if there's a statement nested in s (in the AST) that modifies a
    $\exists\ s_{in},\ nestedIn(s_{in}, s) \wedge MOD(s_{in}, a)$
    // (iii) check if s modifies a base path of a
    $\vee\ \exists\ b,\ b.p == a \wedge MOD(s, b)$
    // (iv) check if s modifies a property of a using a dynamic property expression
    $\vee$ s assigns to $a[p]$
    // (v) check if s contains a call to a function that modifies a
    $\vee\ \exists\ f,\ calledIn(f, s) \wedge \exists\ s_f \in f_{body},$
      // direct modification of a in the function
      $MOD(s_f, a)$
      $\vee$ // parameter alias to a is modified in the function
        a is the i^{th} argument to f $\wedge$ $\exists$ a_{pi}, $MOD(s_f, a_{pi})$ $\wedge$ a_{pi} is the i^{th} parameter of f
end predicate

Figure 5 Predicate for determining if an access path a is modified by a statement s

4.5 Computing MOD and REF sets

Figure 5 shows our algorithm for computing MOD sets[11], expressed as a predicate *MOD*. The *MOD* predicate states that statement s modifies access path a if one of the following conditions holds: (i) s modifies a directly in an assignment or in the initializer associated with a declaration, or via an environment-specific side effect, (ii) there is a statement nested inside s that modifies a, (iii) s modifies a base path of a (i.e., $a == b.p$, and s modifies b), (iv) s modifies a property of a using a dynamic property expression p, or (v) s consists of a call to a function f, the body of f contains a statement s_f, and either s_f modifies a or s_f modifies a parameter of f that is bound to a.

4.6 Determining whether statements can be exchanged

As a first step towards determining reordering opportunities, Figure 6 defines a predicate for determining if two statements are data-independent, by checking that they do not have conflicting side-effects. This predicate operationalizes the condition that was specified in Section 4.3. However, data-independence is by itself not a sufficient condition for statements being exchangeable. Figure 7 shows a predicate *exchangeable* that checks if two statements s_1 and s_2 are exchangeable by checking that: (i) they are data independent, (ii) neither is a control-flow construct such as `return` or the test condition of an `if` or loop, and (iii) they occur in the same basic block. The last condition was introduced to avoid problems related to scoping. As part of future work, we plan to explore ways to move statements into different scopes by relaxing this condition.

[11] REF sets are computed analogously. Pseudocode of the REF algorithm is included in the supplementary material.

Input: s_1 and s_2 statements
Result: `True` if s_1 and s_2 are data-independent, `False` otherwise

predicate $dataIndependent(s_1, s_2)$
$\quad \forall a,\ MOD(s_1, a) \implies \neg MOD(s_2, a)$
$\quad \wedge\ \forall a,\ MOD(s_1, a) \implies \neg REF(s_2, a)$
$\quad \wedge\ \forall a,\ REF(s_1, a) \implies \neg MOD(s_2, a)$
end predicate

Figure 6 Predicate for determining if two statements have overlapping MOD/REF sets.

Input: s_1 and s_2 statements
Result: `True` if the statements can be exchanged, `False` otherwise

predicate $exchangeable(s_1, s_2)$
$\quad dataIndependent(s_1, s_2)$
$\quad \wedge\ \neg isControlFlowStmt(s_1) \wedge \neg isControlFlowStmt(s_2)$
$\quad \wedge\ inSameBasicBlock(s_1, s_2)$
end predicate

Figure 7 Predicate for determining if two statements can be swapped.

Input: s and s_{up} statements
Result: `True` if s can be reordered above s_{up}, `False` otherwise

predicate $stmtCanSwapUpTo(s, s_{up})$
$\quad$ `// base case`
$\quad\quad s == s_{up}$
$\quad \vee$ `// recursive case`
$\quad\quad \exists\ s_{mid},\ (\ stmtCanSwapUpTo(s, s_{mid}) \wedge$
$\quad\quad\quad\quad s_{up}.nextStmt == s_{mid} \wedge$
$\quad\quad\quad\quad exchangeable(s, s_{up})\ )$
end predicate

Figure 8 Predicate for determining if statement s can be reordered above another statement s_{up}.

Input: s and **result** statements
Result: `True` if **result** is the earliest statement above which s can be swapped, `False` otherwise

predicate $earliestStmtToSwapWith(s,$ **result**$)$
$\quad$ `// find the earliest statement s can swap above (min by source code location)`
$\quad$ **result** $==$ $min($ all stmts s_i where $inSameBasicBlock(s, s_i) \wedge stmtCanSwapUpTo(s, s_i))$
end predicate

Figure 9 Predicate for finding the earliest statement above which s can be placed.

4.7 Identifying reordering opportunities

We are now in a position to present our algorithm for identifying reordering opportunities. The analysis for determining earliest point above which a statement can be placed, or the latest point below which a statement can be placed is symmetric, so without loss of generality we will focus on the case of determining the earliest point. Our solution for this problem takes the form of two predicates, *stmtCanSwapUpTo* and *earliestStmtToSwapWith* [12].

Figure 8 defines a predicate *stmtCanSwapUpTo* that associates a statement s with an earlier statement s_{up} above which it can be reordered. This predicate relies on the predicate *exchangeable* to determine if it can be swapped with each statement in between s and s_{up}. If one of these intermediate statements data-conflicts with s then reordering is not possible.

The predicate *earliestStmtToSwapWith* defined in Figure 9 uses *stmtCanSwapUpTo* to find the earliest statement above which a statement can be placed.

By applying this predicate to statements containing I/O-dependent await-expressions, we can identify reordering opportunities that can enable concurrent I/O. Here, we consider an await-expression to be *I/O-dependent* if it (transitively) invokes functions originating from one of the (many) npm packages that make use of the file system. I/O dependency is determined through a call graph analysis – much like how we compute MOD and REF sets, for statement s we look for calls to I/O-related package functions explicitly in s, or in a function transitively called by s. In terms of access paths, these calls correspond to function call access paths rooted in a **require**(m) for some I/O-dependent package m.

4.8 Program transformation

As discussed in Section 3, the execution of an await-expression `await` e_{io} involves two key steps: the *creation of a promise*, and *awaiting its resolution*. The creation of the promise kicks off an asynchronous computation, and our goal is to move it as *early* as possible, so as to maximize the amount of time where it can run concurrently with the main program or other concurrent I/O. On the other hand, we want to await the resolution of the promise *as late as possible*, for the same reason. We achieve this objective by splitting the original await-expression into two statements `var t =` e_{io} and `await t`, and using our analysis to move the former as early as possible, and the latter as late as possible. The example given previously in Section 3 illustrates an application of this refactoring to a real application.

4.9 Implementation

We implemented our approach in a tool named *ReSynchronizer*. The static analysis algorithm, as presented in Section 4, is implemented using approximately 1,600 lines of QL [2], building on extensive libraries for writing static analyzers in QL that are available from [11]. In particular, we rely on existing frameworks for dataflow analysis and call graphs, and on an implementation of access paths that we extended to suit our analysis, as discussed. Note that the QL implementation caps access paths at a maximum length of 10; this could lead to MOD/REF for very long paths not being accounted for, which is a source of potential unsoundness and is discussed in Section 5.4. The QL representation of local variables also relies on single static assignment (SSA), enabling us to regain some precision that would be lost in a purely flow-insensitive analysis.

[12] Pseudocode for *stmtCanDownUpTo* and *latestStmtToSwapWith* are included in the supplementary material.

Once *ReSynchronizer* has determined the await-expressions that are to be reordered and where they should be moved to, the next stage of the tool is the program transformation itself. The actual reordering is done by splitting and moving nodes around in a parse tree representation of the program. We implemented this in Python, and make use of the pandas library to store our list of statements to reorder in a dataframe over which we can efficiently apply transformations.

All the code for our implementation of *ReSynchronizer* will be made available as an artifact.

5 Evaluation

In this section, we apply our technique to a collection of open-source JavaScript applications to answer the following research questions:

RQ1. How many await-expressions are identified as candidates for reordering?

RQ2. How much time does *ReSynchronizer* take to analyze applications?

RQ3. What is the impact of reordering await-expressions on run-time performance?

RQ4. How often does *ReSynchronizer* produce reordering suggestions that are not behaviour-preserving?

5.1 Experimental Methodology

To answer the above research questions, we applied *ReSynchronizer* to 15 open-source JavaScript applications that are available from GitHub. Then, we conducted performance measurements on tests associated with these applications before and after reordering await-expressions.

5.1.0.1 Selecting subject applications.

To be a suitable candidate for our technique, an application needs to apply the `async`/`await` feature to promises that are associated with I/O. In this evaluation, we decided to focus on applications that perform file I/O, as it would be difficult to conduct performance measurements on applications relying on network I/O. Furthermore, to conduct performance measurements, we need to be able to observe executions in which the reordered await-expressions are evaluated. To this end, we focus on applications that have a test suite that we can execute, and monitor test coverage to observe whether await-expressions are executed.

To identify projects that satisfy these requirements, we wrote a QL query that identifies projects that contain await-expressions in files that import a file system I/O-related package[13], and ran it over all 85k JavaScript projects available on GitHub's `LGTM.com` site. This resulted in a list of 42,378 candidate projects. To further narrow the list, we filtered for projects that contain at least 50 await-expressions in files that import a file system I/O-related package. This left us with 490 candidate projects.

From these candidates, we then randomly selected a project, cloned its repository, and attempted to build the project by running the setup code. If the build was successful, we ran the project's tests and made sure they all passed. Projects with broken builds, with failing tests or with fewer than 30 passing tests were discarded. These steps were applied

[13] The specific I/O-related packages our test projects rely on are: `fs`, `fs-admin`, `fs-extra`, `fs-tree-utils`, `fs-exists-cached`, `mock-fs`, `cspell-io`, `path-env`, and `tmp`.

Project	LOC	Fcts (Async)	Awaits (IO)	Tests	IO dep.	Brief description
kactus	145k	15178 (444)	3946 (1829)	799	FS, FSA, FSE	Version control for designers (git for sketch)
webdriverio	40k	4878 (274)	3899 (126)	1884	FSE	WebDriver automated testing for Node.js
desktop	157k	15741 (397)	4065 (1463)	837	FSA, FSE	Github desktop app
fiddle	19k	3808 (153)	865 (108)	609	FSE	Tool for creating small Electron experiments
nodemonorepo	14k	3660 (61)	350 (266)	499	FSE, FSTU, PE	Management of nodejs env/packages
zapier-...	15k	3652 (29)	144 (59)	36	FSE, MFS	CLI tool for zapier applications
wire-desktop	14k	3628 (41)	553 (236)	37	FSE	Desktop app for wire messenger
cspell	17k	3745 (70)	371 (226)	954	FSE, CIO	Spell checker for code
sourcecred	39k	5031 (186)	844 (191)	1824	FSE	Reputation networks for opensource projects
bit	58k	7981 (251)	2526 (2144)	405	FSE	Platform for collaborating on components
vscode-psl	17k	8313 (87)	695 (406)	450	FSE	Profile Scripting Language plugin for VSCode
gatsby	86k	5542 (598)	4177 (821)	2708	FSE, FSEC	Web framework built on React
jamserve	39k	8133 (4019)	10825 (1067)	3883	FSE, T	Audio library server
get	10k	3471 (38)	120 (107)	50	FSE	Download Electron release artifacts
cucumber-js	73k	9407 (115)	577 (31)	445	FSE, T	Cucumber for JS

Table 1 Summary of GitHub projects we're using for experiments

repeatedly until we identified 15 projects, listed in Table 1. The columns in this table state the following characteristics for these projects:

- **LOC:** total lines of JavaScript/TypeScript code in the project being analyzed (not including packages imported by the project).
- **Fcts (Async):** total number of functions in the project; the number between the parentheses gives the number of `async` functions.
- **Awaits (IO):** total number of await-expressions in the project; the number between parentheses gives the number that are dependent on file-I/O (as described in Section 4.7).
- **Tests:** the number of tests associated with the project.
- **IO dep.:** the npm packages through which the file system is accessed. Here, 'FS' denotes `fs`, 'FSA' is `fs-admin`, 'FSE' is `fs-extra`, 'FSTU' is `fs-tree-utils`, 'FSEC' is `fs-exists-cached`, 'MFS' is `mock-fs`, 'CIO' is `cspell-io`, 'PE' is `path-env`, and 'T' is `tmp`.
- **Brief description:** of the project (taken from the repository's README file).

5.1.0.2 Measuring run-time performance.

To determine the impact of reordering await-expressions, we measure the execution time of those tests that execute at least one await-expression that was reordered. Tests that only execute unmodified code are not affected by our transformation, so their execution time is unaffected. We constructed a simple coverage tool that instruments the code to enable us to determine which tests are affected by the reordering of await-expressions.

Performance improvements are measured by comparing runtimes of each affected test before and after the reordering transformation. For our timed experiments, we ran the tests 50 times, and calculated the running time for each test as the average over those 50 runs. This procedure was followed both for the original version of the project, and for the version with the reorderings.

We took several steps to minimize potential bias or inconsistencies in our experimental results. First, we minimized contention for resources by running all experiments on a "quiet" machine where no other user programs were running. For our OS we chose Arch linux: as a bare-bones linux distribution, this minimizes competing resource use between the tests and the OS itself (since there are less processes running in the background than would be the

Project	Awaits Reordered (%)	Tests Affected (%)	Analysis Runtime (s)
kactus	72 (3.9%)	172 (21.5%)	121
webdriverio	9 (7.1%)	12 (0.6%)	19
desktop	67 (4.6%)	187 (22.3%)	177
fiddle	3 (2.8%)	2 (0.3%)	8
nodemonorepo	22 (8.3%)	15 (3.0%)	7
zapier-platform-cli	16 (27.1%)	2 (5.6%)	5
wire-desktop	31 (13.1%)	14 (37.8%)	6
cspell	22 (9.7%)	26 (2.7%)	8
sourcecred	22 (11.5%)	29 (1.6%)	14
bit	116 (5.4%)	8 (2.0%)	204
vscode-psl	19 (4.7%)	116 (25.8%)	8
gatsby	103 (12.5%)	43 (1.6%)	30
jamserve	59 (5.5%)	272 (7.0%)	62
get	6 (5.6%)	3 (6.0%)	5
cucumber-js	13 (41.9%)	17 (3.1%)	64

Table 2 Number and percentage of `await`s reordered, per test project.

case with most other OSs). We also configured each project's test runner so that tests were executed sequentially[14], removing the possibility for resource contention between tests.

During our initial experiments we observed that the first few runs of test suites were always slower, and determined this was due to some files remaining in cache between test runs, reducing the time needed to read them as compared to the first runs which read them directly from disk. To prevent such effects from skewing the results of our experiments, we ran the tests 5 times before taking performance measurements. We also decided to run the tests for the version with reorderings applied *before* the original version. Hence, if there is any caching bias resulting from the order of the experiments it would just make our results worse.

The hardware used for experiments was a Thinkpad P43s, with core i7 processor and 32GB RAM.

5.2 RQ1 and RQ2

To answer RQ1, we ran *ReSynchronizer* on each of the projects described in Table 1. Table 2 displays some metrics on the results, namely:

- **Awaits Reordered (%):** the absolute number of await-expressions reordered, with the parenthetical giving what fraction this is of the project's total I/O-dependent `await`s
- **Tests Affected (%):** the total number of affected tests (i.e., the number of tests that execute at least one reordered await-expression), with the parenthetical giving the percentage of the project's total tests this represents. For example: for the Kactus project there are 172 impacted tests, which is 21.5% of the 799 tests associated with the project.
- **Analysis Runtime (s):** the time taken to run the static analysis over the project (in seconds)

[14] Some of the projects we tested relied on jest for their testing, while others used mocha. By default, jest runs tests concurrently, so we relied on its command-line argument `runInBand` to execute tests sequentially. This issue does not arise in the case of mocha, which runs tests sequentially by default.

	Impacted Tests		
Project	**Avg Speedup (%)**	**Max Speedup (%)**	**% Sig Speedup (%)**
`kactus`	7.1%	32.4%	80.2%
`webdriverio`	1.4%	5.4%	16.7%
`desktop`	8.1%	35.4%	90.9%
`fiddle`	8.7%	16.6%	50.0%
`nodemonorepo`	3.4%	10.5%	86.7%
`zapier-platform-cli`	5.6%	8.9%	100.%
`wire-desktop`	3.5%	17.3%	50.0%
`cspell`	3.2%	14.1%	50.0%
`sourcecred`	4.0%	20.2%	48.3%
`bit`	3.3%	16.7%	15.4%
`vscode-psl`	2.5%	75.0%	8.6%
`gatsby`	6.0%	52.2%	44.2%
`jamserve`	0.7%	23.1%	12.9%
`get`	1.3%	3.4%	33.3%
`cucumber-js`	7.2%	62.5%	17.6%

Table 3 Results of performance experiments on github projects – Tests

From this table, it can be seen that our analysis reorders between 2.8% and 41.9% of the I/O-dependent await-expressions (10.9% on average). While the number of reorderings strongly depends on the nature of the project being analyzed, it is clear that a nontrivial number of await-expressions can be reordered[15].

Looking at the **Tests Affected** column in this table, we see that between 0.3% and 37.8% of the projects' tests execute code affected by our reordering (9.4% on average). This is also a huge range. We note that the number of affected tests is not necessarily correlated with the number of `await`s reordered either: indeed, `cucumber-js`, the project with the highest fraction of `await`s reordered, has one of the lowest fractions of affected tests at only 3.1%. Clearly, the number of affected tests depends strongly on the way the developers structured their tests and on the relative spread of the reorderings themselves across the project. This wide range of fractions of affected tests shows how important it is to only consider the affected tests when measuring the effect of the reordering on performance, to avoid the results being skewed by tests that are unaffected by the refactoring.

Table 2 also shows the time required by *ReSynchronizer* to process each of the subject projects, which range from 10k-160k lines of code. As can be seen from the table, the longest analysis time was 204 seconds. Applying the program transformation took less than 5 seconds for each project tested. Hence, our analysis scales to large applications.

5.3 RQ3

Table 3 shows the results of our experiments, with the following columns:

- **Avg Speedup (%):** the average percentage speedup over all affected tests for the project. The percentage speedup for each test is computed as $1 - \frac{\text{average time with reordering}}{\text{average time with original code}}$; we report the average of these over all tests. If there was a slowdown this value would be

[15] Note that these results only reflect await-expressions related to file I/O, and it is possible that opportunities may exist for reordering await-expressions that are associated with other types of I/O, such as network-dependent code.

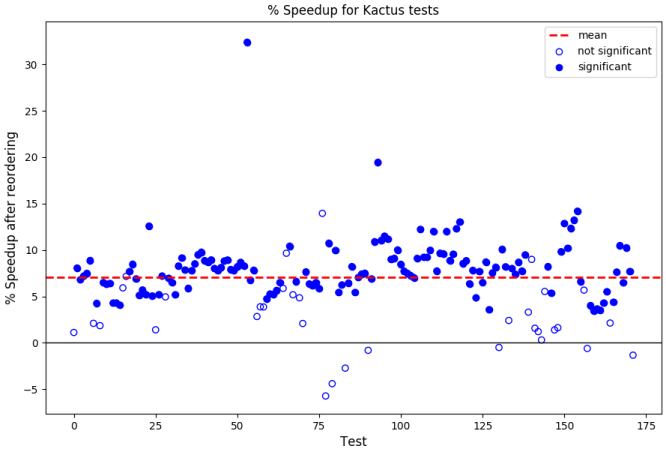

Figure 10 Average percentage speedups for all Kactus tests

negative.

- **Max Speedup (%):** the maximum percentage speedup (i.e., the speedup for the test which was most improved by our reordering).
- **% Sig Speedup (%):** the percentage of tests for which there was a *statistically significant* speedup. We want to count how many of the tests were sped up by our reordering; but if we just counted how many tests had an average speedup after reordering, this would not account for the variance of our data – a speedup could be reported but be within the error bars of the data. To address this, we performed a standard two-tailed t-test with the timings for each test with and without the reorderings. The t-test indicates a significant result only when the measured difference in timing is large with respect to the variability of the data, with "how large" being controlled by the confidence level (here, we chose 90% confidence). This is a measure of the proportion of the affected tests that our technique actually improved (with 90% confidence).

From Table 3, we see that the average speedups for the affected tests ranges from 0.7% to 8.7% for the projects under consideration, whereas maximum speedups range from 3.4% to 75.0%, suggesting that there is a large amount of variability in the performance improvements. As a result, one might wonder what effect these tests with huge improvements have on the average speedup, and whether a few outliers are significantly skewing the data. We address this with our last column, which shows the proportion of the tests for which we see a statistically significant speedup. Here too, we see a big variance, with 8.6% to 100% of the affected tests seeing statistically significant speedups.

To better understand the variability in our experimental results, we decided to take a closer look at the observed average speedups for all individual tests for the Kactus project[16], shown in Figure 10. This chart shows the percentage speedup as a result of reordering 72 await-expressions in Kactus, for each of Kactus's 172 impacted tests. Here, results for tests for which the reordering has a statistically significant effect on the runtime are depicted as

[16] Refer to the supplemental materials for results from similar experiments with the other 14 subject applications.

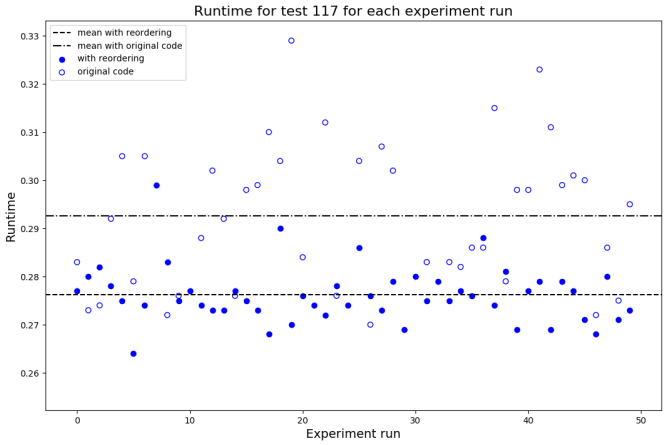

Figure 11 Runtimes for all experiment runs of Kactus test 117

colored circles, whereas the results for those where the effect is not significant are shown as empty circles.

From Table 3 we recall that 80.2% of Kactus's affected tests are statistically significantly sped up, and indeed on this graph the vast majority of the tests experience a significant effect. From this graph we also get some information that is not available in the table: looking at the distribution of test speedups, we see that indeed the test with the maximum speedup of 32.4% is an outlier. We also see that most of the tests have speedups clustered fairly closely around the average of 7.1% (indicated by the dashed line on the graph). This is encouraging, as it means our reordering has a fairly consistent positive effect on the performance of Kactus. Finally, we see that although there are a few tests that see a slowdown, none of these indicate a significant effect.

Prompted by these results, we decided to take an even closer look at the variability in our results. To this end, we created Figure 11, which shows the individual runtimes for each experiment run of one specific test of Kactus. For this, we chose as representative test #117, which executes the code in the motivating example presented in Section 3, and for which we observed an average speedup of 9.5%, which is fairly close to the mean of 7.1%. The figure displays the runtimes for this test both with the original version of Kactus and with the version with all reorderings applied. The mean of each of these runtimes is indicated using dot-dashed and dashed lines respectively.

From Figure 11, we observe that there is *less variation in the running time of the test after reordering*. This same pattern is seen with other tests[17]. Our conjecture is that this reduction in variability of running times occurs because, before reordering, a test will experience the sum of the times needed to access multiple files, each of which may exhibit worst-case access time behavior. However, after reordering, when files are being accessed concurrently, the test execution experiences the maximum of these file-access times, i.e., experiencing the sum of the worst-case file access behaviors no longer occurs. This reduction in runtime variability is a positive side effect of the transformation, as it makes application runtime more stable and

[17] The supplementary materials include similar graphs for a few other select tests, all of which follow the same trend.

predictable.

5.4 RQ4

As discussed, there are several factors that contribute to unsoundness in our side-effect analysis. The call graph that we rely on may be unsound, which could cause missing access paths in MOD or REF sets. Moreover, our access path representation is not canonical, i.e., multiple access paths may refer to the same location(s). And, since the implementation caps access path length at 10, very long access paths will not be represented. These potential sources of unsoundness have the same potential risk, of suggesting reorderings that do not preserve program behaviour.

We conducted an experiment in which we made the analysis more conservative, by assuming that every call to an unknown function modifes and references all access paths (i.e., that every unanalyzable call has global environmental side-effects). With this approach, the only project for which any reorderings were still suggested was Kactus, for which the number of reordered await-expressions was reduced from 72 to 1, rendering *ReSynchronizer* essentially useless. This suggests that a fully sound side-effect analysis is impractical as the basis for implementing the refactoring we consider in this paper.

In practice, we have not observed any situations where unsoundness manifests itself via invalid reorderings. In the 15 subject applications, we did not observe a single case where reordering await-expressions caused a test failure. While this observation is no guarantee that *ReSynchronizer* always suggests valid reorderings, it suggests that *ReSynchronizer* is not significantly less robust than many state-of-the-art in refactoring tools.

5.5 Threats to Validity

Beyond the risks caused by the unsoundness of the static analysis that we already discussed, we consider the following threats to validity.

The projects we used in our evaluation could not be representative of JavaScript projects using `async`/`await`, so our results might not generalize beyond these projects. However, the 15 projects we chose for our evaluation were chosen at random, and we observed the same trends among them.

In designing our performance evaluations, we were mindful of potential sources of bias to our results. We described the reasoning behind our design and how we mitigated bias in Section 5.1. In the case of caching bias, we ran our tests with reordered code *before* the tests for the original code, so that any bias would be against us.

Finally, our results might not generalize to I/O other than the file system such as database I/O and network I/O. We conjecture that they will, as the logic of splitting an await-expression to maximize concurrency is an environment-agnostic transformation, but have not yet tested this assumption.

6 Related Work

This section covers related work on side-effect analysis and on refactorings related to asynchrony and concurrency.

6.0.0.1 Side-Effect Analysis.

Our paper relies on interprocedural side-effect analysis to determine whether statements can be reordered without changing program behavior. Work on side-effect analysis started in

the early 1970s, with the objective of computing dataflow facts that can be used to direct compiler optimizations.

[26] presents a side-effect analysis for the PL/I programming language that computes the expressions whose value may change as a result of assignments to variables. Spillman's analysis accounts for aliasing induced by pointers and parameter-passing, and is specified operationally as a procedure that creates a matrix that associates each variable with all expressions whose value would be impacted by an assignment to that variable. Procedure invocations are represented by additional rows in the matrix and side-effects for such invocations are computed in invocation order, using a fixpoint procedure to handle recursion.

[1] presents an interprocedural data flow analysis in which a simple intraprocedural analysis first identifies definitions that may affect uses outside a block, and uses in a block that may be affected by definitions outside the block. An interprocedural analysis then traverses a call graph in reverse invocation order to combine the facts computed for the individual procedures. Allen's algorithm does not handle recursive procedures.

[3] presents an interprocedural side-effect analysis that accounts for parameter-induced aliasing in a language with nested procedures, and defines notions MOD and REF for flow-insensitive side-effects, and USE and DEF for flow-sensitive side-effects. Banning's flow-insensitive technique determines the set of variables immediately modified by a procedure and assumes the availability of a call graph to map variables in a callee to variables in a caller. The side-effect of a procedure call is then computed by way of a meet-over-all-paths solution. Our analysis follows Banning's approach but defines MOD and REF in terms of access paths [19] instead of names of variables, and relies on SSA form for improved precision (for access paths rooted in local variables).

[4] present a faster algorithm for solving the same problem of alias-free flow-insensitive side-effect analysis discussed by [3]. To improve the performance of the algorithm, they divide the problem into two distinct cases: side-effects to *reference parameters* (i.e., interprocedural function parameter aliasing), and *global variables*. They introduce a new data structure, the binding multigraph, for side-effect tracking through reference parameters, and a new, linear algorithm for side-effect tracking through global variables.

A few years later, [14] present work on computing MOD sets for languages with general-purpose pointers. The introduction of pointers introduced another type of aliases to the problem of computing side effects, and Landi et al. extended previous work on computing MOD sets, by adapting and incorporating an existing algorithm for approximating pointer-based aliases.

Since their introduction by [3], MOD and REF algorithms have also been adopted for use as parts of other dataflow analyses. [15] present an algorithm for computing SSA numbering for languages with pointer indirection, which relies on MOD/REF side-effect analysis to track when the variable referred to by an SSA representation is being reassigned (in order to signal the need for a new SSA number).

[5] also present an algorithm for computing SSA form which makes use of the MOD and REF side-effect analysis in order to determine when a variable could be modified indirectly by a statement. This work does not consider aliasing through pointers, and just uses the reference parameter and global variable aliasing as presented by Banning.

6.0.0.2 Refactorings related to Asynchrony and Concurrency.

[10] present a refactoring for converting event-driven code into promise-based code. They assume that event-driven APIs conform to the error-first protocol (i.e., the first parameter of the callback functions is assumed to be a flag indicating whether an error occurred) and

consider two strategies: "direct modification" and "wrap-around", where the latter approach is similar to "promisification" performed by libraries such as `universalify`. Their work predates the wide-spread adoption of `async`/`await` and does not show how to introduce these features, though there is a brief discussion how some of the presented mechanisms provide a first step towards refactorings for introducing `async`/`await`.

[6] presented an overview of the challenges associated with refactorings related to the introduction and use of asynchronous programming features for Android and C# applications. [17] present *Asynchronizer*, a refactoring tool that enables developers to extract long-running Android operations into an `AsyncTask`. Since Java is multi-threaded, Android applications may exhibit real concurrency, so (unlike with the JavaScript applications that we consider in our work) care must be taken to prevent data races that may cause nondeterministic failures. To this end, Lin et al. extend a previously developed static data race detector [22]. In later work, [16] study the use of Android's three mechanisms for asynchronous programming: `AsyncTask`, `IntentService`, and `AsyncTaskLoader` and the scenarios for which each of these mechanisms is well-suited. They observe that developers commonly misuse `AsyncTask` for long-running tasks that it is not suitable for, and present a refactoring tool, *AsyncDroid*, that assists with the migration to `IntentService`.

[21] studied the use of asynchronous programming in C#, soon after that language added an `async`/`await` feature in 2012. At the time of this study, callback-based asynchronous programming was still dominant, although `async`/`await` was starting to be adopted widely. To facilitate the transition, Okur et al. created a refactoring tool, *Asyncifier* for automatically converting C# applications to use `async`/`await`. Okur et al. also observed several common anti-patterns involving the misuse of `async`/`await`, including unnecessary use of `async`/`await` and using long-running synchronous operations inside of `async` methods, and developed another tool, *Corrector* for detecting and fixing some of these issues.

Several other projects are concerned with refactorings for introducing and manipulating concurrency. [8] presented *Relooper*, a refactoring tool for converting sequential loops into parallel loops in Java programs. [27] presented *Reentrancer*, a refactoring tool for making existing Java applications reentrant, so that they can be deployed on parallel machines without concurrency control. [7] presented *Concurrencer*, a refactoring tool that supports three refactorings for introducing AtomicInteger, ConcurrentHashMap, and FJTask data structures from the `java.util.concurrent` library. [20] presented two refactoring tools for C#, *Taskifier* and *Simplifier*, for transforming Thread and ThreadPool abstractions into Task abstractions, and for transforming Task abstractions into higher-level design patterns.

[24] present a framework of synchronization dependences that refactoring engines must respect in order to maintain the correctness of a number of commonly used refactorings in the presence of concurrency. [13] present a refactoring for migrating between sequential and parallel streams in Java 8 programs.

7 Conclusions and Future Work

The changing landscape of asynchronous programming in JavaScript makes it all too easy for programmers to schedule asynchronous I/O operations suboptimally. In this paper, we show that refactoring I/O-related await-expressions can yield significant performance benefits. To identify situations where this refactoring can be applied, we rely on an interprocedural side-effect analysis that computes, for a statement s, sets $MOD(s)$ and $REF(s)$ of access paths that represent sets of memory locations modified and referenced by s, respectively. We

implemented the analysis using QL, and incorporated it into a tool, *ReSynchronizer*, that automatically applies the suggested refactorings. In an experimental evaluation, we applied *ReSynchronizer* to 15 open-source JavaScript applications that rely on file-system I/O, and observe average speedups of between 0.7% and 8.7% (4.4% on average) when running tests that execute refactored code. While the analysis is potentially unsound, we did not encounter any situations where applying the refactoring causes test failures.

As future work, we plan to extend our implementation to handle sources of I/O other than the file system, such as databases and networks. In addition, we plan to explore the possibility of splitting/reordering await-expressions across scopes, and implement more complex optimizations such as factoring `await`s out of loops. Finally, we are also interested in making our side-effect analysis more precise. For example, for cases where file names are specified as string literals, one could create MOD and REF sets representing just a single file as opposed to the entire file system.

References

1. Frances E. Allen. Interprocedural data flow analysis. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 398–402. North-Holland, 1974.
2. Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 2:1–2:25, 2016. doi:10.4230/LIPIcs.ECOOP.2016.2.
3. John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 29–41. ACM Press, 1979. doi:10.1145/567752.567756.
4. Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 57–66, 1988. doi:10.1145/53990.53996.
5. Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
6. Danny Dig. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 32(6):52–61, 2015. doi:10.1109/MS.2015.133.
7. Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 397–407, 2009. doi:10.1109/ICSE.2009.5070539.
8. Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph E. Johnson. Relooper: refactoring for loop parallelism in Java. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 793–794, 2009. doi:10.1145/1639950.1640018.
9. ECMA. Ecmascript 2019 language specification, 2010. Available from http://www.ecma-international.org/ecma-262/.
10. Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring asynchrony in javascript. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 353–363. IEEE Computer Society, 2017. doi:10.1109/ICSME.2017.83.
11. GitHub. Codeql. https://github.com/github/codeql, 2020. Accessed: 2020-05-13.

12 Jordan Harband. util.promisify. https://github.com/ljharb/util.promisify, 2020. Accessed: 2020-05-14.

13 Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of java 8 streams. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 619–630. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00072.

14 William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

15 Christopher Lapkowski and Laurie J Hendren. Extended ssa numbering: Introducing ssa properties to languages with multi-level pointers. In *International Conference on Compiler Construction*, pages 128–143. Springer, 1998.

16 Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 224–235, 2015. doi:10.1109/ASE.2015.50.

17 Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 341–352, 2014. doi:10.1145/2635868.2635903.

18 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2015.

19 Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 7:1–7:24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

20 Semih Okur, Cansu Erdogan, and Danny Dig. Converting parallel code from low-level abstractions to higher-level abstractions. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 515–540, 2014. doi:10.1007/978-3-662-44202-9_21.

21 Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in C#. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1117–1127, 2014. doi:10.1145/2568225.2568309.

22 Cosmin Radoi and Danny Dig. Practical static race detection for Java parallel loops. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 178–190, 2013. doi:10.1145/2483760.2483765.

23 Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, volume 48, pages 151–166. ACM, 2013.

24 Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent java code. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 225–249, 2010. doi:10.1007/978-3-642-14107-2_11.

25 Isaac Z. Schlueter. graceful-fs. https://www.npmjs.com/package/graceful-fs, 2020. Accessed: 2020-05-14.

26 Thomas C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, pages 376–381, 1971.

27 Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 173–182, 2009. doi:10.1145/1595696.1595723.

28 Ryan Zim. universalify. https://github.com/RyanZim/universalify, 2020. Accessed: 2020-05-14.