# Enabling Additional Parallelism in Asynchronous JavaScript Applications (Artifact)

**Ellen Arteca**
Northeastern University, Boston, MA, USA

**Frank Tip**
Northeastern University, Boston, MA, USA

**Max Schäfer**
GitHub, Oxford, UK

## 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 20 applications that perform file- or network-related I/O. For these applications, we observe average speedups ranging between 0.99% and 53.6% for the tests that execute refactored code (8.1% on average).

**2012 ACM Subject Classification** Software and its engineering → Automated static analysis; Software and its engineering → Concurrent programming structures; Software and its engineering → Software performance

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

**Digital Object Identifier** 10.4230/DARTS.7.2.5

**Funding** E. Arteca and F. Tip were supported in part by the National Science Foundation grants CCF-1715153 and CCF-1907727. E. Arteca was also supported in part by the Natural Sciences and Engineering Research Council of Canada.

**Related Article** Ellen Arteca, Frank Tip, and Max Schäfer, "Enabling Additional Parallelism in Asynchronous JavaScript Applications", in 35th European Conference on Object-Oriented Programming (ECOOP 2021), LIPIcs, Vol. 194, pp. 7:1–7:28, 2021.
`https://doi.org/10.4230/LIPIcs.ECOOP.2021.7`

**Related Conference** 35th European Conference on Object-Oriented Programming (ECOOP 2021), July 12–16, 2021, Aarhus, Denmark (Virtual Conference)

## 1 Scope

This artifact consists of a docker image, the contents of which are described in Section 2. In the docker, you can:

- interact with our data, and reproduce the graphs from the paper (or construct similar graphs we did not include in the paper)
- run *Resynchronizer* on a new project
- check out the transformed projects we tested with and rerun the timing experiments




Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik,
Dagstuhl Publishing, Germany


## 2 Content

The relevant contents of the docker container are as follows:

- `ExperimentalData`: data from our timing experiments; can be used to reproduce the graphs in the paper and supplementary materials
- `DataAnalysis` directory: contains a jupyter notebook for interacting with our data
- `Resynchronizer` directory: contains the code for applying and running *Resynchronizer*
- `Resynchronizer/ReorderingUtils.qll`: static side effect analysis code
- `Resynchronizer/reorder_me.py`: the driving script for applying the reorderings determined through the analysis
- `Resynchronizer/applyResync.sh`: script for applying Resynchronizer to a project `Paper` directory: paper and associated supplementary materials

Instructions for interacting with the container are included in Appendix A.

## 3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the contents of the artifact and instructions for recreating the docker image is also available at: `https://github.com/emarteca/Resynchronizer`.

## 4 Tested platforms

This docker container has been tested on Linux (both Ubuntu and Arch), but it is self-contained and should work on any system where docker is set up. It should run without issue on a standard laptop (it was tested on a ThinkPad P43s with 32GB RAM).

## 5 License

The artifact is available under the MIT License.

## 6 MD5 sum of the artifact

2004edb5b603c4c97d7bffae052939fa

## 7 Size of the artifact

2.34 GiB

## A Usage

Make sure you have docker installed.

Download the image, and load it:

```
docker load -i resynchronizer.tgz # assuming image file resynchronizer.tgz
```

Run the docker:

```
mkdir loc_mount
docker run \
--mount type=bind,source=‘pwd‘/loc_mount,destination=/mount -w
        /home/resynchronizer -it \
--user=0 \
--env="DISPLAY" \
--env="QT_X11_NO_MITSHM=1" \
--volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
-p 8888:8888 \
resynchronizer:latest
```

Now, you'll be in the `/home/resynchronizer` directory of the docker image. In the docker, you can:

- interact with our data, and reproduce the graphs from the paper (or construct similar graphs we did not include in the paper)
- run *Resynchronizer* on a new project
- check out the transformed projects we tested with and rerun the timing experiments

## A.1 Contents of the container

The relevant contents of the docker container are as follows (as listed above in section 2):

- `ExperimentalData`: data from our timing experiments; can be used to reproduce the graphs in the paper and supplementary materials
- `DataAnalysis` directory: contains a jupyter notebook for interacting with our data
- `Resynchronizer` directory: contains the code for applying and running *Resynchronizer*
- `Resynchronizer/ReorderingUtils.qll`: static side effect analysis code
- `Resynchronizer/reorder_me.py`: the driving script for applying the reorderings determined through the analysis
- `Resynchronizer/applyResync.sh`: script for applying Resynchronizer to a project `Paper` directory: paper and associated supplementary materials

## A.2 Interacting with data: graph reproduction

The `ExperimentalData` directory contains all the raw timing data from our experiments. To reproduce the graphs in the paper and further explore the data, go into the `DataAnalysis` directory and open the jupyter notebook:

```
cd DataAnalysis
jupyter notebook --ip 0.0.0.0 --no-browser --allow-root
```

This will produce some output, the last line of which will be of the form:

```
http://127.0.0.1:8888/?token=<some string of chars>
```

Then, on your machine, you can access the notebook by copy pasting that path into your browser and opening `data_analysis.ipynb` . Alternatively, you can access the notebook by going to `http://127.0.0.1:8888/notebooks/data_analysis.ipynb` on your browser, and then entering the string of characters following `token=` from the docker output, when prompted for a token.

In the notebook, there are the following example commands to recreate the graphs from the paper. For example:

```
pkgname = "kactus"
load_data_for_package(pkgname)
# regenerate Figure 10 in the paper
scatterplot_test_speedup(comp_mean_table, pkgname)
# regenerate Figure 11 in the paper
plot_test_times( bothswap_jest_tests, noswap_jest_tests, 117)
```

Calling `load_data_for_package` with the name of another project will allow you to interact with that data instead. Looking at the supplementary materials, here are a few other examples:

```
# second graph in Supplementary materials: section 3
load_data_for_package("webdriverio")
scatterplot_test_speedup(comp_mean_table, "webdriverio")
# first graph in Supplementary materials: section 4
load_data_for_package("kactus")
plot_test_times( bothswap_jest_tests, noswap_jest_tests, 22)
```

When you're done looking at the data, exit the notebook to try the rest of the artifact.

## A.3 Run *Resynchronizer* on a new project

To use Resynchronizer, first enter the `Resynchronizer` directory in the container home. Demonstrative example of applying resynchronizer to the version of `mattermost-redux` used in our experiments:

```
git clone https://github.com/mattermost/mattermost-redux
    Playground/mattermost-redux
cd Playground/mattermost-redux
# checkout the specific commit we tested on
# (was most recent commit at the time)
git checkout dd44020f008f6e1955709ff7fc3e1c8c42388396
cd ../..

# set up the project
# note: this can differ per project, we have a general script that works
# in most cases but depending on the build configuration of the project
# you want to test with, this may differ
./resetProject.sh Playground/mattermost-redux

# now, apply resynchronizer
./applyResync.sh Playground/mattermost-redux/ QLDBs/mattermost-redux
```

The `mattermost-redux` with the reorderings applied is now saved in the directory `reordered_proj` in `/home/resynchronizer` (your current directory).

To see the effect of the transformations, `grep` for the temporary variables:

```
cd reordered_proj
grep -rn "TEMP_VAR_AUTOGEN"
```

You should see the following output:

```
src/client/client4.ts:1047:var TIMING_TEMP_VAR_AUTOGEN287__RANDOM =
    perf_hooks.performance.now();
src/client/client4.ts:1048: var AWAIT_VAR_TIMING_TEMP_VAR_
    AUTOGEN287__RANDOM = await this.doFetchWithResponse(
src/client/client4.ts:1052:console.log("/home/resynchronizer/
    reordered_proj/src/client/client4.ts& [719, 8; 722, 10]&
    TEMP_VAR_AUTOGEN287__RANDOM& " +
    (perf_hooks.performance.now() - TIMING_TEMP_VAR_AUTOGEN287__RANDOM));
src/client/client4.ts:1053: const {response} =
    AWAIT_VAR_TIMING_TEMP_VAR_AUTOGEN287__RANDOM
src/actions/admin.ts:1007:var TEMP_VAR_AUTOGEN263__RANDOM =
    Client4.sendWarnMetricAck(warnMetricId, forceAck);
src/actions/admin.ts:1012:var TIMING_TEMP_VAR_AUTOGEN263__RANDOM =
    perf_hooks.performance.now();
src/actions/admin.ts:1013: await TEMP_VAR_AUTOGEN263__RANDOM
src/actions/admin.ts:1014:console.log("/home/resynchronizer/reordered_
    proj/src/actions/admin.ts& [656, 12; 656, 68]& TEMP_VAR_
    AUTOGEN263__RANDOM& " + (perf_hooks.performance.now() -
    TIMING_TEMP_VAR_AUTOGEN263__RANDOM));
src/actions/search.ts:511:var TEMP_VAR_AUTOGEN152__RANDOM =
    Client4.searchPosts(teamId, terms, true);
src/actions/search.ts:516:var TIMING_TEMP_VAR_AUTOGEN152__RANDOM =
    perf_hooks.performance.now();
src/actions/search.ts:517: var AWAIT_VAR_TIMING_TEMP_VAR_
    AUTOGEN152__RANDOM = await TEMP_VAR_AUTOGEN152__RANDOM
src/actions/search.ts:518:console.log("/home/resynchronizer/
    reordered_proj/src/actions/search.ts& [298, 12; 298, 67]&
    TEMP_VAR_AUTOGEN152__RANDOM& " +
    (perf_hooks.performance.now() - TIMING_TEMP_VAR_AUTOGEN152__RANDOM));
src/actions/search.ts:519: posts =
    AWAIT_VAR_TIMING_TEMP_VAR_AUTOGEN152__RANDOM
```

Here you see the newly introduced variables assigned to the computation that was originally being awaited, with:

```
var TEMP_VAR_AUTOGEN<number> = ...
```

You can also see where the results are awaited:

```
await TEMP_VAR_AUTOGEN<number>
```

The other results of the `grep` are the `TIMING_TEMP` variables, which are only introduced for the purposes of logging how long the awaited computations are taking (you see these variables in the `console.log` calls).

If you want to run the tests of `mattermost-redux` and observe the printing of the timing tracking statements:

```
npm run test
```

Then, go back to the `/home/resynchronizer` directory to go to the next step.

## A.4 Rerunning timing experiments

The transformed projects are available on github: we forked them and created branches with the reorderings applied (called ReorderingApplied).

For example, to check out the reordered version of `mattermost-redux` that contains the experiment scripts:

```
git clone --branch ReorderingApplied
    https://github.com/emarteca/mattermost-redux.git
```

This version of the repo contains the reorderings, all the scripts required to run the experiments, and the list of tests affected by the reorderings.

Before running experiments, you must set up and build the project:

```
./resetProject.sh mattermost-redux

# standardize the paths of the tests to match those in
# the docker container (when cloned, they match the original
# paths on the computer where I ran the experiments
./dockerize_paths.sh mattermost-redux
```

Then, you can run the experiments.

```
cd mattermost-redux
./batchListOfTests.sh 50 test_list.txt raw_output.out
    test_times_bothswap_50times.out 5
```

The parameters are:

- `50` : the number of test iterations
- `test_list.txt` : the pre-generated list of tests affected by the reorderings
- `raw_output.out` : the raw logged output of all the tests, that gets processed into the next file
- `test_times_bothswap_50times.out` : the file where the processed test output gets dumped; this matches `ExperimentalData/mattermost-redux/test_times_bothswap_50times.out` (although of course the exact numbers will differ since they are test runtimes)
- `5 : the number of warmup runs`

If you want to only run a few test iterations to make sure it's working, I would recommend setting a smaller number of test iterations (maybe 10) and omitting the warmup runs (if the warmup run argument is omitted it defaults to 0).

You can also run the experiments on the non-reordered code by checking out the `JustTiming` branch (where all awaits that will be reordered are timed):

```
git checkout JustTiming
```

Then, rerun the experiments the same way as above. Change the output filename to `test_times_noswap_50times.out` to emulate the experiments we performed.

**Note:** the timing values will be different running here than in the reported results in the paper, since those were not run inside a docker container.

## A.5 Thanks!

Let us know if you run into any issues or have any questions! PRs or issues on the associated GitHub repo are more than welcome.