# Contributing to Zstandard

We want to make contributing to this project as easy and transparent as

possible.

## Our Development Process

New versions are being developed in the "dev" branch,

or in their own feature branch.

When they are deemed ready for a release, they are merged into "master".

As a consequences, all contributions must stage first through "dev"

or their own feature branch.

## Pull Requests

We actively welcome your pull requests.

1. Fork the repo and create your branch from `dev`.

2. If you've added code that should be tested, add tests.

3. If you've changed APIs, update the documentation.

4. Ensure the test suite passes.

5. Make sure your code lints.

6. If you haven't already, complete the Contributor License Agreement ("CLA").

## Contributor License Agreement ("CLA")

In order to accept your pull request, we need you to submit a CLA. You only need

to do this once to work on any of Facebook's open source projects.

Complete your CLA here: <https://code.facebook.com/cla>

## Workflow

Zstd uses a branch-based workflow for making changes to the codebase. Typically, zstd

will use a new branch per sizable topic. For smaller changes, it is okay to lump multiple

related changes into a branch.

Our contribution process works in three main stages:

1. Local development

\* Update:

\* Checkout your fork of zstd if you have not already

```

git checkout https://github.com/<username>/zstd

cd zstd

```

\* Update your local dev branch

```

git pull https://github.com/facebook/zstd dev

git push origin dev

```

\* Topic and development:

\* Make a new branch on your fork about the topic you're developing for

```

# branch names should be consise but sufficiently informative

git checkout -b <branch-name>

git push origin <branch-name>

```

\* Make commits and push

```

# make some changes =

git add -u && git commit -m <message>

git push origin <branch-name>

```

\* Note: run local tests to ensure that your changes didn't break existing functionality

\* Quick check

```

make shortest

```

\* Longer check

```

make test

```

2. Code Review and CI tests

\* Ensure CI tests pass:

\* Before sharing anything to the community, make sure that all CI tests pass on your local fork.

See our section on setting up your CI environment for more information on how to do this.

\* Ensure that static analysis passes on your development machine. See the Static Analysis section

below to see how to do this.

\* Create a pull request:

\* When you are ready to share you changes to the community, create a pull request from your branch

to facebook:dev. You can do this very easily by clicking 'Create Pull Request' on your fork's home

page.

\* From there, select the branch where you made changes as your source branch and facebook:dev

as the destination.

\* Examine the diff presented between the two branches to make sure there is nothing unexpected.

\* Write a good pull request description:

\* While there is no strict template that our contributors follow, we would like them to

sufficiently summarize and motivate the changes they are proposing. We recommend all pull requests,

at least indirectly, address the following points.

\* Is this pull request important and why?

\* Is it addressing an issue? If so, what issue? (provide links for convenience please)

\* Is this a new feature? If so, why is it useful and/or necessary?

\* Are there background references and documents that reviewers should be aware of to properly assess this change?

\* Note: make sure to point out any design and architectural decisions that you made and the rationale behind them.

\* Note: if you have been working with a specific user and would like them to review your work, make sure you mention them using (@<username>)

\* Submit the pull request and iterate with feedback.

3. Merge and Release

\* Getting approval:

\* You will have to iterate on your changes with feedback from other collaborators to reach a point

where your pull request can be safely merged.

\* To avoid too many comments on style and convention, make sure that you have a

look at our style section below before creating a pull request.

\* Eventually, someone from the zstd team will approve your pull request and not long after merge it into

the dev branch.

\* Housekeeping:

\* Most PRs are linked with one or more Github issues. If this is the case for your PR, make sure

the corresponding issue is mentioned. If your change 'fixes' or completely addresses the

issue at hand, then please indicate this by requesting that an issue be closed by commenting.

\* Just because your changes have been merged does not mean the topic or larger issue is complete. Remember

that the change must make it to an official zstd release for it to be meaningful. We recommend

that contributers track the activity on their pull request and corresponding issue(s) page(s) until

their change makes it to the next release of zstd. Users will often discover bugs in your code or

suggest ways to refine and improve your initial changes even after the pull request is merged.

## Static Analysis

Static analysis is a process for examining the correctness or validity of a program without actually

executing it. It usually helps us find many simple bugs. Zstd uses clang's `scan-build` tool for

static analysis. You can install it by following the instructions for your OS on https://clang-analyzer.llvm.org/scan-build.

Once installed, you can ensure that our static analysis tests pass on your local development machine

by running:

```

make staticAnalyze

```

In general, you can use `scan-build` to static analyze any build script. For example, to static analyze

just `contrib/largeNbDicts` and nothing else, you can run:

```

scan-build make -C contrib/largeNbDicts largeNbDicts

```

### Pitfalls of static analysis

`scan-build` is part of our regular CI suite. Other static analyzers are not.

It can be useful to look at additional static analyzers once in a while (and we do), but it's not a good idea to multiply the nb of analyzers run continuously at each commit and PR. The reasons are :

- Static analyzers are full of false positive. The signal to noise ratio is actually pretty low.

- A good CI policy is "zero-warning tolerance". That means that all issues must be solved, including false positives. This quickly becomes a tedious workload.

- Multiple static analyzers will feature multiple kind of false positives, sometimes applying to the same code but in different ways leading to :

+ torteous code, trying to please multiple constraints, hurting readability and therefore maintenance. Sometimes, such complexity introduce other more subtle bugs, that are just out of scope of the analyzers.

+ sometimes, these constraints are mutually exclusive : if one try to solve one, the other static analyzer will complain, they can't be both happy at the same time.

- As if that was not enough, the list of false positives change with each version. It's hard enough to follow one static analyzer, but multiple ones with their own update agenda, this quickly becomes a massive velocity reducer.

This is different from running a static analyzer once in a while, looking at the output, and \_\_cherry picking\_\_ a few warnings that seem helpful, either because they detected a genuine risk of bug, or because it helps expressing the code in a way which is more readable or more difficult to misuse. These kind of reports can be useful, and are accepted.

## Performance

Performance is extremely important for zstd and we only merge pull requests whose performance

landscape and corresponding trade-offs have been adequately analyzed, reproduced, and presented.

This high bar for performance means that every PR which has the potential to

impact performance takes a very long time for us to properly review. That being said, we

always welcome contributions to improve performance (or worsen performance for the trade-off of

something else). Please keep the following in mind before submitting a performance related PR:

1. Zstd isn't as old as gzip but it has been around for time now and its evolution is

very well documented via past Github issues and pull requests. It may be the case that your

particular performance optimization has already been considered in the past. Please take some

time to search through old issues and pull requests using keywords specific to your

would-be PR. Of course, just because a topic has already been discussed (and perhaps rejected

on some grounds) in the past, doesn't mean it isn't worth bringing up again. But even in that case,

it will be helpful for you to have context from that topic's history before contributing.

2. The distinction between noise and actual performance gains can unfortunately be very subtle

especially when microbenchmarking extremely small wins or losses. The only remedy to getting

something subtle merged is extensive benchmarking. You will be doing us a great favor if you

take the time to run extensive, long-duration, and potentially cross-(os, platform, process, etc)

benchmarks on your end before submitting a PR. Of course, you will not be able to benchmark

your changes on every single processor and os out there (and neither will we) but do that best

you can:) We've adding some things to think about when benchmarking below in the Benchmarking

Performance section which might be helpful for you.

3. Optimizing performance for a certain OS, processor vendor, compiler, or network system is a perfectly

legitimate thing to do as long as it does not harm the overall performance health of Zstd.

This is a hard balance to strike but please keep in mind other aspects of Zstd when

submitting changes that are clang-specific, windows-specific, etc.

## Benchmarking Performance

Performance microbenchmarking is a tricky subject but also essential for Zstd. We value empirical

testing over theoretical speculation. This guide it not perfect but for most scenarios, it

is a good place to start.

### Stability

Unfortunately, the most important aspect in being able to benchmark reliably is to have a stable

benchmarking machine. A virtual machine, a machine with shared resources, or your laptop

will typically not be stable enough to obtain reliable benchmark results. If you can get your

hands on a desktop, this is usually a better scenario.

Of course, benchmarking can be done on non-hyper-stable machines as well. You will just have to

do a little more work to ensure that you are in fact measuring the changes you've made not and

noise. Here are some things you can do to make your benchmarks more stable:

1. The most simple thing you can do to drastically improve the stability of your benchmark is

to run it multiple times and then aggregate the results of those runs. As a general rule of

thumb, the smaller the change you are trying to measure, the more samples of benchmark runs

you will have to aggregate over to get reliable results. Here are some additional things to keep in

mind when running multiple trials:

\* How you aggregate your samples are important. You might be tempted to use the mean of your

results. While this is certainly going to be a more stable number than a raw single sample

benchmark number, you might have more luck by taking the median. The mean is not robust to

outliers whereas the median is. Better still, you could simply take the fastest speed your

benchmark achieved on each run since that is likely the fastest your process will be

capable of running your code. In our experience, this (aggregating by just taking the sample

with the fastest running time) has been the most stable approach.

\* The more samples you have, the more stable your benchmarks should be. You can verify

your improved stability by looking at the size of your confidence intervals as you

increase your sample count. These should get smaller and smaller. Eventually hopefully

smaller than the performance win you are expecting.

\* Most processors will take some time to get `hot` when running anything. The observations

you collect during that time period will very different from the true performance number. Having

a very large number of sample will help alleviate this problem slightly but you can also

address is directly by simply not including the first `n` iterations of your benchmark in

your aggregations. You can determine `n` by simply looking at the results from each iteration

and then hand picking a good threshold after which the variance in results seems to stabilize.

2. You cannot really get reliable benchmarks if your host machine is simultaneously running

another cpu/memory-intensive application in the background. If you are running benchmarks on your

personal laptop for instance, you should close all applications (including your code editor and

browser) before running your benchmarks. You might also have invisible background applications

running. You can see what these are by looking at either Activity Monitor on Mac or Task Manager

on Windows. You will get more stable benchmark results of you end those processes as well.

\* If you have multiple cores, you can even run your benchmark on a reserved core to prevent

pollution from other OS and user processes. There are a number of ways to do this depending

on your OS:

\* On linux boxes, you have use https://github.com/lpechacek/cpuset.

\* On Windows, you can "Set Processor Affinity" using https://www.thewindowsclub.com/processor-affinity-windows

\* On Mac, you can try to use their dedicated affinity API https://developer.apple.com/library/archive/releasenotes/Performance/RN-AffinityAPI/#//apple\_ref/doc/uid/TP40006635-CH1-DontLinkElementID\_2

3. To benchmark, you will likely end up writing a separate c/c++ program that will link libzstd.

Dynamically linking your library will introduce some added variation (not a large amount but

definitely some). Statically linking libzstd will be more stable. Static libraries should

be enabled by default when building zstd.

4. Use a profiler with a good high resolution timer. See the section below on profiling for

details on this.

5. Disable frequency scaling, turbo boost and address space randomization (this will vary by OS)

6. Try to avoid storage. On some systems you can use tmpfs. Putting the program, inputs and outputs on

tmpfs avoids touching a real storage system, which can have a pretty big variability.

Also check our LLVM's guide on benchmarking here: https://llvm.org/docs/Benchmarking.html

### Zstd benchmark

The fastest signal you can get regarding your performance changes is via the in-build zstd cli

bench option. You can run Zstd as you typically would for your scenario using some set of options

and then additionally also specify the `-b#` option. Doing this will run our benchmarking pipeline

for that options you have just provided. If you want to look at the internals of how this

benchmarking script works, you can check out programs/benchzstd.c

For example: say you have made a change that you believe improves the speed of zstd level 1. The

very first thing you should use to asses whether you actually achieved any sort of improvement

is `zstd -b`. You might try to do something like this. Note: you can use the `-i` option to

specify a running time for your benchmark in seconds (default is 3 seconds).

Usually, the longer the running time, the more stable your results will be.

```

$ git checkout <commit-before-your-change>

$ make && cp zstd zstd-old

$ git checkout <commit-after-your-change>

$ make && cp zstd zstd-new

$ zstd-old -i5 -b1 <your-test-data>

1<your-test-data> : 8990 -> 3992 (2.252), 302.6 MB/s , 626.4 MB/s

$ zstd-new -i5 -b1 <your-test-data>

1<your-test-data> : 8990 -> 3992 (2.252), 302.8 MB/s , 628.4 MB/s

```

Unless your performance win is large enough to be visible despite the intrinsic noise

on your computer, benchzstd alone will likely not be enough to validate the impact of your

changes. For example, the results of the example above indicate that effectively nothing

changed but there could be a small <3% improvement that the noise on the host machine

obscured. So unless you see a large performance win (10-15% consistently) using just

this method of evaluation will not be sufficient.

### Profiling

There are a number of great profilers out there. We're going to briefly mention how you can

profile your code using `instruments` on mac, `perf` on linux and `visual studio profiler`

on windows.

Say you have an idea for a change that you think will provide some good performance gains

for level 1 compression on Zstd. Typically this means, you have identified a section of

code that you think can be made to run faster.

The first thing you will want to do is make sure that the piece of code is actually taking up

a notable amount of time to run. It is usually not worth optimzing something which accounts for less than

0.0001% of the total running time. Luckily, there are tools to help with this.

Profilers will let you see how much time your code spends inside a particular function.

If your target code snippit is only part of a function, it might be worth trying to

isolate that snippit by moving it to its own function (this is usually not necessary but

might be).

Most profilers (including the profilers dicusssed below) will generate a call graph of

functions for you. Your goal will be to find your function of interest in this call grapch

and then inspect the time spent inside of it. You might also want to to look at the

annotated assembly which most profilers will provide you with.

#### Instruments

We will once again consider the scenario where you think you've identified a piece of code

whose performance can be improved upon. Follow these steps to profile your code using

Instruments.

1. Open Instruments

2. Select `Time Profiler` from the list of standard templates

3. Close all other applications except for your instruments window and your terminal

4. Run your benchmarking script from your terminal window

\* You will want a benchmark that runs for at least a few seconds (5 seconds will

usually be long enough). This way the profiler will have something to work with

and you will have ample time to attach your profiler to this process:)

\* I will just use benchzstd as my bencharmking script for this example:

```

$ zstd -b1 -i5 <my-data> # this will run for 5 seconds

```

5. Once you run your benchmarking script, switch back over to instruments and attach your

process to the time profiler. You can do this by:

\* Clicking on the `All Processes` drop down in the top left of the toolbar.

\* Selecting your process from the dropdown. In my case, it is just going to be labled

`zstd`

\* Hitting the bright red record circle button on the top left of the toolbar

6. You profiler will now start collecting metrics from your bencharking script. Once

you think you have collected enough samples (usually this is the case after 3 seconds of

recording), stop your profiler.

7. Make sure that in toolbar of the bottom window, `profile` is selected.

8. You should be able to see your call graph.

\* If you don't see the call graph or an incomplete call graph, make sure you have compiled

zstd and your benchmarking scripg using debug flags. On mac and linux, this just means

you will have to supply the `-g` flag alone with your build script. You might also

have to provide the `-fno-omit-frame-pointer` flag

9. Dig down the graph to find your function call and then inspect it by double clicking

the list item. You will be able to see the annotated source code and the assembly side by

side.

#### Perf

This wiki has a pretty detailed tutorial on getting started working with perf so we'll

leave you to check that out of you're getting started:

https://perf.wiki.kernel.org/index.php/Tutorial

Some general notes on perf:

\* Use `perf stat -r # <bench-program>` to quickly get some relevant timing and

counter statistics. Perf uses a high resolution timer and this is likely one

of the first things your team will run when assessing your PR.

\* Perf has a long list of hardware counters that can be viewed with `perf --list`.

When measuring optimizations, something worth trying is to make sure the handware

counters you expect to be impacted by your change are in fact being so. For example,

if you expect the L1 cache misses to decrease with your change, you can look at the

counter `L1-dcache-load-misses`

\* Perf hardware counters will not work on a virtual machine.

#### Visual Studio

TODO

## Setting up continuous integration (CI) on your fork

Zstd uses a number of different continuous integration (CI) tools to ensure that new changes

are well tested before they make it to an official release. Specifically, we use the platforms

travis-ci, circle-ci, and appveyor.

Changes cannot be merged into the main dev branch unless they pass all of our CI tests.

The easiest way to run these CI tests on your own before submitting a PR to our dev branch

is to configure your personal fork of zstd with each of the CI platforms. Below, you'll find

instructions for doing this.

### travis-ci

Follow these steps to link travis-ci with your github fork of zstd

1. Make sure you are logged into your github account

2. Go to https://travis-ci.org/

3. Click 'Sign in with Github' on the top right

4. Click 'Authorize travis-ci'

5. Click 'Activate all repositories using Github Apps'

6. Select 'Only select repositories' and select your fork of zstd from the drop down

7. Click 'Approve and Install'

8. Click 'Sign in with Github' again. This time, it will be for travis-pro (which will let you view your tests on the web dashboard)

9. Click 'Authorize travis-pro'

10. You should have travis set up on your fork now.

### circle-ci

TODO

### appveyor

Follow these steps to link circle-ci with your girhub fork of zstd

1. Make sure you are logged into your github account

2. Go to https://www.appveyor.com/

3. Click 'Sign in' on the top right

4. Select 'Github' on the left panel

5. Click 'Authorize appveyor'

6. You might be asked to select which repositories you want to give appveyor permission to. Select your fork of zstd if you're prompted

7. You should have appveyor set up on your fork now.

### General notes on CI

CI tests run every time a pull request (PR) is created or updated. The exact tests

that get run will depend on the destination branch you specify. Some tests take

longer to run than others. Currently, our CI is set up to run a short

series of tests when creating a PR to the dev branch and a longer series of tests

when creating a PR to the master branch. You can look in the configuration files

of the respective CI platform for more information on what gets run when.

Most people will just want to create a PR with the destination set to their local dev

branch of zstd. You can then find the status of the tests on the PR's page. You can also

re-run tests and cancel running tests from the PR page or from the respective CI's dashboard.

## Issues

We use GitHub issues to track public bugs. Please ensure your description is

clear and has sufficient instructions to be able to reproduce the issue.

Facebook has a [bounty program](https://www.facebook.com/whitehat/) for the safe

disclosure of security bugs. In those cases, please go through the process

outlined on that page and do not file a public issue.

## Coding Style

\* 4 spaces for indentation rather than tabs

## License

By contributing to Zstandard, you agree that your contributions will be licensed

under both the [LICENSE](LICENSE) file and the [COPYING](COPYING) file in the root directory of this source tree.