# Productivity Notes

## Table of Contents



## General

### Cache compilations with `ccache`

The easiest way to faster compile times is to cache compiles. `ccache` is a way to do so, from its description at the time of writing:

> *ccache is a compiler cache. It speeds up recompilation by caching the result of previous compilations and detecting when the same compilation is being done again. Supported languages are C, C++, Objective-C and Objective-C++.*

Install `ccache` through your distribution's package manager, and run `./configure` with your normal flags to pick it up.

To use ccache for all your C/C++ projects, follow the symlinks method here to set it up.

To get the most out of ccache, put something like this in `~/.ccache/ccache.conf`:

```
max_size = 50.0G  # or whatever cache size you prefer; default is 5G; 0 means
unlimited
base_dir = /home/yourname  # or wherever you keep your source files
```

Note: base_dir is required for ccache to share cached compiles of the same file across different repositories / paths; it will only do this for paths under base_dir. So this option is required for effective use of ccache with git worktrees (described below).

You *must not* set base_dir to "/", or anywhere that contains system headers (according to the ccache docs).

### Disable features with `./configure`

After running `./autogen.sh`, which generates the `./configure` file, use `./configure --help` to identify features that you can disable to save on compilation time. A few common flags:

```
--without-miniupnpc
--without-natpmp
--disable-bench
--disable-wallet
--without-gui
```

If you do need the wallet enabled, it is common for devs to add `--with-incompatible-bdb`. This uses your system bdb version for the wallet, so you don't have to find a copy of bdb 4.8. Wallets from such a build will be incompatible with any release binary (and vice versa), so use with caution on mainnet.

### Make use of your threads with `make -j`

If you have multiple threads on your machine, you can tell `make` to utilize all of them with:

```
make -j"$((($(nproc)+1))"
```

### Only build what you need

When rebuilding during development, note that running `make`, without giving a target, will do a lot of work you probably don't need. It will build the GUI (unless you've disabled it) and all the tests (which take much longer to build than the app does).

Obviously, it is important to build and run the tests at appropriate times -- but when you just want a quick compile to check your work, consider picking one or a set of build targets relevant to what you're working on, e.g.:

```
make src/bitcoind src/bitcoin-cli
make src/qt/bitcoin-qt
make -C src bitcoin_bench
```

(You can and should combine this with `-j`, as above, for a parallel build.)

### Multiple working directories with `git worktrees`

If you work with multiple branches or multiple copies of the repository, you should try `git worktrees`.

To create a new branch that lives under a new working directory without disrupting your current working directory (useful for creating pull requests):

```
git worktree add -b my-shiny-new-branch ../living-at-my-new-working-directory based-
on-my-crufty-old-commit-ish
```

To simply check out a commit-ish under a new working directory without disrupting your current working directory (useful for reviewing pull requests):

```
git worktree add --checkout ../where-my-checkout-commit-ish-will-live my-checkout-
commit-ish
```

### Interactive "dummy rebases" for fixups and execs with `git merge-base`

When rebasing, we often want to do a "dummy rebase," whereby we are not rebasing over an updated master but rather over the last common commit with master. This might be useful for rearranging commits, `rebase --autosquash`ing, or `rebase --exec`ing without introducing conflicts that arise from an updated master. In these situations, we can use `git merge-base` to identify the last common commit with master, and rebase off of that.

To squash in `git commit --fixup` commits without rebasing over an updated master, we can do the following:

```
git rebase -i --autosquash "$(git merge-base master HEAD)"
```

To execute `make check` on every commit since last diverged from master, but without rebasing over an updated master, we can do the following:

```
git rebase -i --exec "make check" "$(git merge-base master HEAD)"
```

---

This synergizes well with `ccache` as objects resulting from unchanged code will most likely hit the cache and won't need to be recompiled.

You can also set up upstream refspecs to refer to pull requests easier in the above `git worktree` commands.

## Writing code

### Format C/C++ diffs with `clang-format-diff.py`

See contrib/devtools/README.md.

### Format Python diffs with `yapf-diff.py`

Usage is exactly the same as `clang-format-diff.py`. You can get it here.

## Rebasing/Merging code

### More conflict context with `merge.conflictstyle diff3`

For resolving merge/rebase conflicts, it can be useful to enable diff3 style using `git config merge.conflictstyle diff3`. Instead of

```
<<<
yours
===
theirs
>>>
```

you will see

```
<<<
yours
|||
```

```
original
===
theirs
>>>
```

This may make it much clearer what caused the conflict. In this style, you can often just look at what changed between *original* and *theirs*, and mechanically apply that to *yours* (or the other way around).

## Reviewing code

### Reduce mental load with `git diff` options

When reviewing patches which change indentation in C++ files, use `git diff -w` and `git show -w`. This makes the diff algorithm ignore whitespace changes. This feature is also available on github.com, by adding `?w=1` at the end of any URL which shows a diff.

When reviewing patches that change symbol names in many places, use `git diff --word-diff`. This will instead of showing the patch as deleted/added *lines*, show deleted/added *words*.

When reviewing patches that move code around, try using `git diff --patience commit~:old/file.cpp commit:new/file/name.cpp`, and ignoring everything except the moved body of code which should show up as neither `+` or `-` lines. In case it was not a pure move, this may even work when combined with the `-w` or `--word-diff` options described above. `--color-moved=dimmed-zebra` will also dim the coloring of moved hunks in the diff on compatible terminals.

### Reference PRs easily with `refspec`s

When looking at other's pull requests, it may make sense to add the following section to your `.git/config` file:

```
[remote "upstream-pull"]
        fetch = +refs/pull/*/head:refs/remotes/upstream-pull/*
        url = git@github.com:bitcoin/bitcoin.git
```

This will add an `upstream-pull` remote to your git repository, which can be fetched using `git fetch --all` or `git fetch upstream-pull`. It will download and store on disk quite a lot of data (all PRs, including merged and closed ones). Afterwards, you can use `upstream-pull/NUMBER/head` in arguments to `git show`, `git checkout` and anywhere a commit id would be acceptable to see the changes from pull request NUMBER.

### Diff the diffs with `git range-diff`

It is very common for contributors to rebase their pull requests, or make changes to commits (perhaps in response to review) that are not at the head of their branch. This poses a problem for reviewers as when the contributor force pushes, the reviewer is no longer sure that his previous reviews of commits are still valid (as the commit hashes can now be different even though the diff is semantically the same). git range-diff (Git >= 2.19) can help solve this problem by diffing the diffs.

For example, to identify the differences between your previously reviewed diffs P1-5, and the new diffs P1-2,N3-4 as illustrated below:

```
       P1--P2--P3--P4--P5   <-- previously-reviewed-head
      /
...--m   <-- master
```

```
      \
       P1--P2--N3--N4--N5   <-- new-head (with P3 slightly modified)
```

You can do:

```
git range-diff master previously-reviewed-head new-head
```

Note that `git range-diff` also work for rebases:

```
       P1--P2--P3--P4--P5   <-- previously-reviewed-head
      /
...--m--m1--m2--m3   <-- master
                 \
                  P1--P2--N3--N4  <-- new-head (with P3 modified, P4 & P5 squashed)

PREV=P5 N=4 && git range-diff `git merge-base --all HEAD $PREV`...$PREV HEAD~$N...HEAD
```

Where `P5` is the commit you last reviewed and `4` is the number of commits in the new version.

---

`git range-diff` also accepts normal `git diff` options, see Reduce mental load with `git diff` options for useful `git diff` options.

You can also set up upstream refspecs to refer to pull requests easier in the above `git range-diff` commands.