

Speeding up static analysis with clang-tidy-cache

January 21, 2021

Introduction

clang-tidy

- “a clang-based C++ static analysis tool¹”
 - an extensible framework for diagnosing and fixing typical programming errors,
 - has a comprehensive suite of built-in checks, for
 - style violations,
 - language misuse,
 - anti-patterns,
 - common bugs,
 - etc.
 - provides a convenient interface for writing new checks,
 - is configurable, with a large set of options.

¹<https://clang.llvm.org/extra/clang-tidy/>

Using clang-tidy with cmake

- `cmake` has some built-in support for `clang-tidy`:
 - the `CXX_CLANG_TIDY` target property².
 - the generated build system code includes instructions to run `clang-tidy`, typically chained with the compilation commands.

²<https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html> □ ▸ ◀ ◻ ▸ ◀ ≡ ▸ ◀ ≡ ▸ ≡ 🔍 ↺

Find the clang-tidy command:

```
find_program(  
  CLANG_TIDY_COMMAND  
  clang-tidy  
)
```

Add executable;

```
add_executable(my_target my_target.cpp)
```

or library target:

```
add_library(my_target my_target.cpp)
```

If `clang-tidy` was found, tell `cmake` to check the sources as a part of compilation:

```
if(CLANG_TIDY_COMMAND)
  set_target_properties(
    my_target PROPERTIES
    CXX_CLANG_TIDY ${CLANG_TIDY_COMMAND}
  )
endif()
```

Motivation

The downsides of using clang-tidy

- Running static analysis takes time, a lot of time.
 - Often more time than the actual compilation.
 - Unacceptable increase in build times.
 - Switching static analysis on/off:
 - Not ideal.
 - When do we flip the switch?
 - Typically ends up always off.

Switching clang-tidy checks on/off with cmake

Add a cmake option:

```
option(  
  WITH_STATIC_ANALYSIS  
  "Enable static analysis" ON  
)
```

Add analysis-related target properties only when switched on:

```
if(WITH_STATIC_ANALYSIS and CLANG_TIDY_COMMAND)  
  set_target_properties(  
    my_target PROPERTIES  
    CXX_CLANG_TIDY ${CLANG_TIDY_COMMAND}  
  )  
endif()
```

The reason for build slowdowns

- Typically most of the code that is analysed doesn't change.
 - It is re-checked with the same result over and over.
 - Unless you change something in the core sources included everywhere.

The goal

- Have static analysis always on during development.
- Don't wait for rechecking of unchanged code.

Solution

The solution – analysis result caching

- If we can uniquely identify a static-analysis tool invocation we can store the result and retrieve it when the same invocation is repeated.
 - Similar to compilation-caching³.
 - Track all inputs of the analysis:
 - configuration options,
 - command-line arguments,
 - the source files,
 - etc.

³<https://ccache.dev/>

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ↺ 🔍 ↻

How does it work?

- Scans the command-line arguments of `clang-tidy` and its input:
 - configuration files,
 - analysed source files.
- Makes a hash uniquely identifying the invocation from the above.

How does it work? (cont.)

- Checks if the hash is in the cache database:
 - if it is
 - don't run `clang-tidy` and return immediately,
 - this is typically much faster.
 - otherwise
 - run `clang-tidy` and if successful⁴ store the hash.
 - This means that sources with findings keep being re-checked and the findings are shown.

⁴if there are no warnings or errors reported

How is it used?

Create⁵ a wrapper script, like:

```
#!/bin/bash  
REAL_CT=/full/path/to/clang-tidy  
  
/path/to/clang-tidy-cache \  
"${REAL_CT}" "${@}"
```

Put it into a directory in search `PATH`, before real `clang-tidy`.

⁵or use the one in the repository

Modes of operation

- Local
- Client / Server

Local mode

- Stores the database in a local directory hierarchy.
- Location determined by the `CTCACHE_DIR` environment variable.
- By default a sub-tree in the temporary directory.
- If you want persistence specify a directory in a disk-based file system.

Client / server mode

- `clang-tidy-cache-server`
 - HTTP server exposing a REST API.
 - Can be used to store and retrieve hashes from the client.
 - The client (`clang-tide-cache`) can query the server – still way faster than running `clang-tidy`.

Service – Rest API

- `http://ctcache:5000/...`
 - `/cache/<hash>` – insert `<hash>` into cache.
 - `/is_cached/<hash>` – tests if `<hash>` is cached.
 - `/purge_cache` – remove all cached hashes.
 - `/info` – static configuration information⁶.
 - `/stats` – server run-time status information⁷.
 - `/stats/*` – individual server status readouts⁸.
 - `/stats/ctcache.json` – long-term persistent server status information⁹.
 - `/images/*` – status chart images¹⁰.

⁶JSON object

⁷JSON object

⁸JSON values

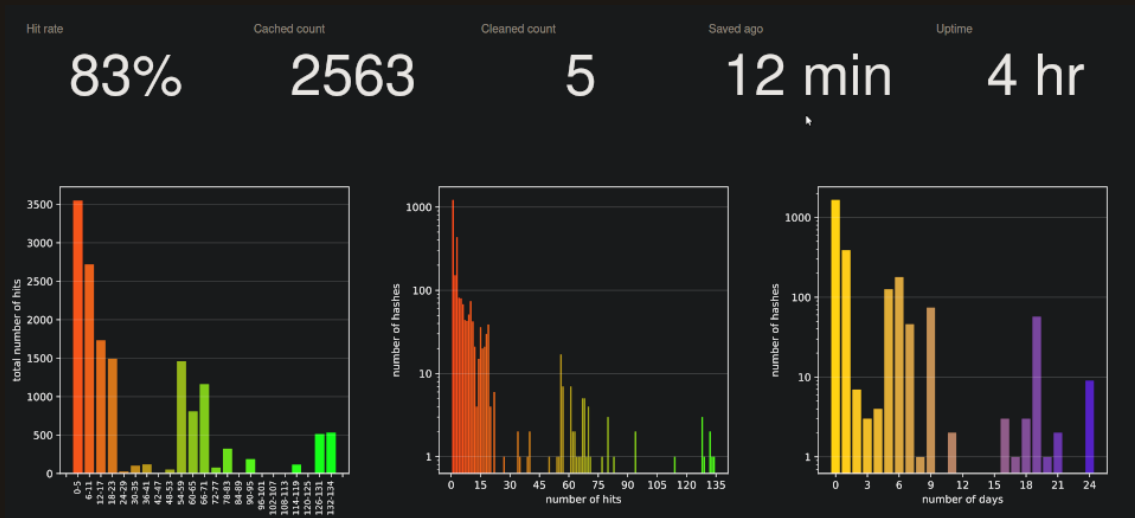
⁹JSON file

¹⁰SVG

Server pages

- The clang tidy-cache's HTTP server also serves several web pages that are designed to be viewed in a browser:
 - the *dashboard* – the main one,
 - SVG plots showing various server statistics.

Server dashboard



Deployment

Deploying the server

- There are several ways how to run the server:
 - just run it in Python,
 - as a *systemd* service,
 - in a *docker* container.

Using python3

If you want to try it out:

```
python3 ./clang-tidy-cache-server
```

Check command-line arguments:

```
python3 ./clang-tidy-cache-server --help
```

```
usage: clang-tidy-cache-server
  [-h] [--debug] [--port NUMBER]
  [--save-path FILE-PATH.gz]
  [--save-interval NUMBER]
  [--stats-save-interval NUMBER]
  [--cleanup-interval NUMBER]
  [--stats-path DIR-PATH]
  [--chart-path DIR-PATH]
```

Using systemd – installation

Install server as user service to home directory:

```
cd path/to/ctcache_repo  
./install-user-service
```

BTW: install client to user's home directory:

```
cd path/to/ctcache_repo  
./install-user-client
```

Installed files:

```
~/local/bin/clang-tidy # default wrapper script  
~/local/bin/clang-tidy-cache  
~/local/bin/clang-tidy-cache-server  
~/local/share/ctcache/static/* # static web files  
~/config/ctcache/systemd_env # systemd environment  
~/config/systemd/user/ctcache.service
```

Using systemd – service start/stop

Reload the user service files:

```
systemctl --user daemon-reload
```

Start the service:

```
systemctl --user start ctcache.service
```

Stop the service:

```
systemctl --user stop ctcache.service
```

Using systemd

Permanently enable automatic start of the service:

```
systemctl --user enable ctdcache.service
```

Permanently disable automatic start of the service:

```
systemctl --user disable ctdcache.service
```

Using docker

Build the image:

```
docker build -t ctcache .
```

Basic usage:

```
docker run \  
  -e CTCACHE_PORT=5000 \  
  -p "80:5000" \  
  -it --rm \  
  --name ctcache ctcache
```

Make cache data persistent with `docker volumes`

Create the volume:

```
docker volume create ctcache
```

Basic usage:

```
docker run \  
  -e CTCACHE_PORT=5000 \  
  -p "80:5000" \  
  -v "ctcache:/var/lib/ctcache" \  
  -it --rm \  
  --name ctcache ctcache
```

Using docker-compose

The `docker-compose.yaml` file:

```
version: "3.6"
services:
  ctcache:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - "ctcache:/var/lib/ctcache"
volumes:
  ctcache:
```


Using docker-compose

Start the service container using docker-compose:

```
docker-compose up
```

as a daemon:

```
docker-compose up -d
```

Stop the running daemon and cleanup container:

```
docker-compose down
```

Environment variables

variable	client	server	meaning
CTCACHE_CLANG_TIDY	✓		path to the clang-tidy executable to be used
CTCACHE_DISABLE	✓		disables cache, always runs clang-tidy
CTCACHE_SKIP	✓		disables analysis, client returns "OK" immediately
CTCACHE_STRIP	✓		list of strings stripped from hashed inputs
CTCACHE_DIR	✓		the cache directory in local mode
CTCACHE_HOST	✓	✓	hostname or IP address of the server
CTCACHE_PORT	✓	✓	port number on which the server accepts connections
CTCACHE_WEBROOT		✓	directory where static served files are located

Measurements

Test projects

- Project 1 (small):
 - proprietary C++ training examples,
 - some use **heavy** template meta-programming,
 - part of the code is not analyzed,
 - ≈ 6300 LOC, *tens* of build targets.
- Project 2 (medium):
 - open-source C++ wrapper for EGL, OpenGL, OpenAL, ...,
 - <https://github.com/matus-chochlik/oglpplu2>,
 - $\approx 135k$ LOC, *hundreds* of build targets.
- Project 3 (large):
 - proprietary, production code for embedded HW,
 - there are some unfixed findings – some sources are re-checked,
 - $\approx 750k$ LOC, *thousands* of build targets.

Test hardware setup

- development laptop:
 - i5-7200U @ 2.50GHz (4 cores),
 - 16GB RAM,
 - 250GB SSD.
- ctcache server:
 - RPi 3B,
 - BCM2837 64bit @ 1.20GHz (4 cores),
 - 1GB RAM,
 - 1TB USB HDD.
- connected over WiFi (5GHz / $\approx 650\text{Mbps}^{11}$).

¹¹according to `iwconfig`

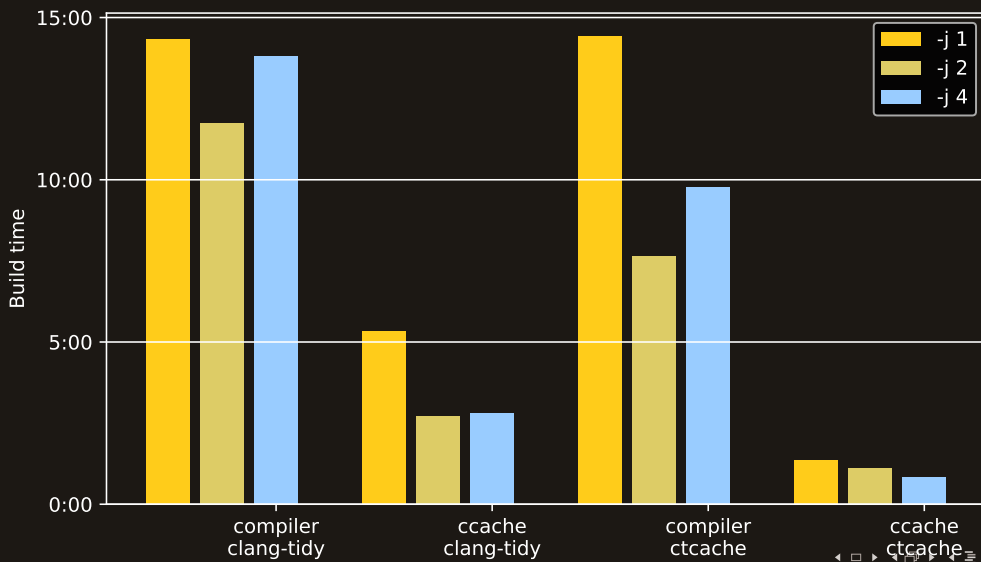
Test configurations

- “compiler & clang-tidy”
 - `export CCACHE_DISABLE=1`
 - `export CTCACHE_DISABLE=1`
- “ccache & clang-tidy”
 - `unset CCACHE_DISABLE`
 - `export CTCACHE_DISABLE=1`
- “compiler & ctcache”
 - `export CCACHE_DISABLE=1`
 - `unset CTCACHE_DISABLE`
- “ccache & ctcache”
 - `unset CCACHE_DISABLE`
 - `unset CTCACHE_DISABLE`

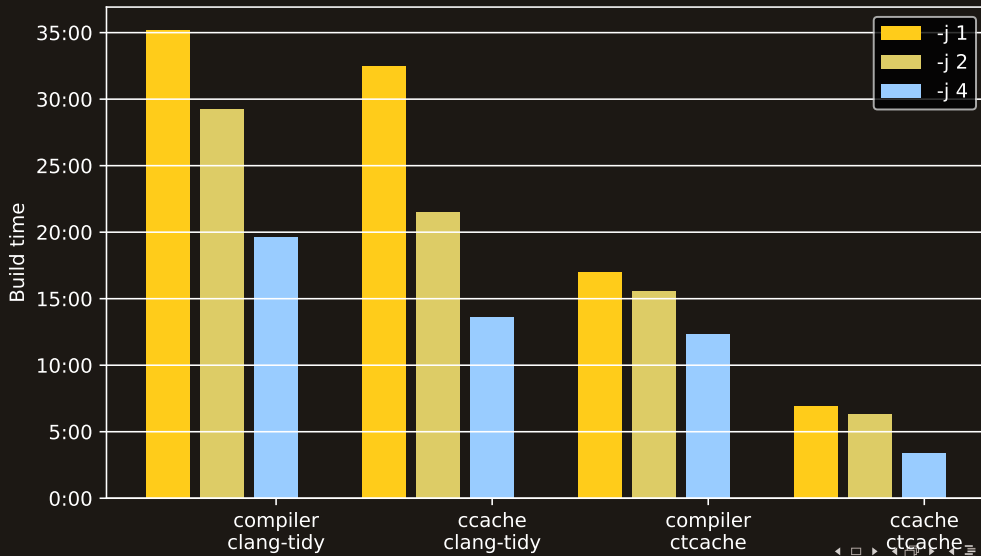
Test execution

- For each test project:
 - `cd /path/to/build/dir`
 - `rm -rf ./`
 - `cmake ... /path/to/project/source`
 - do initial build with caches enabled (no measurements),
 - for each test configuration:
 - setup environment variables,
 - `make clean`
 - `time make -j N ...`

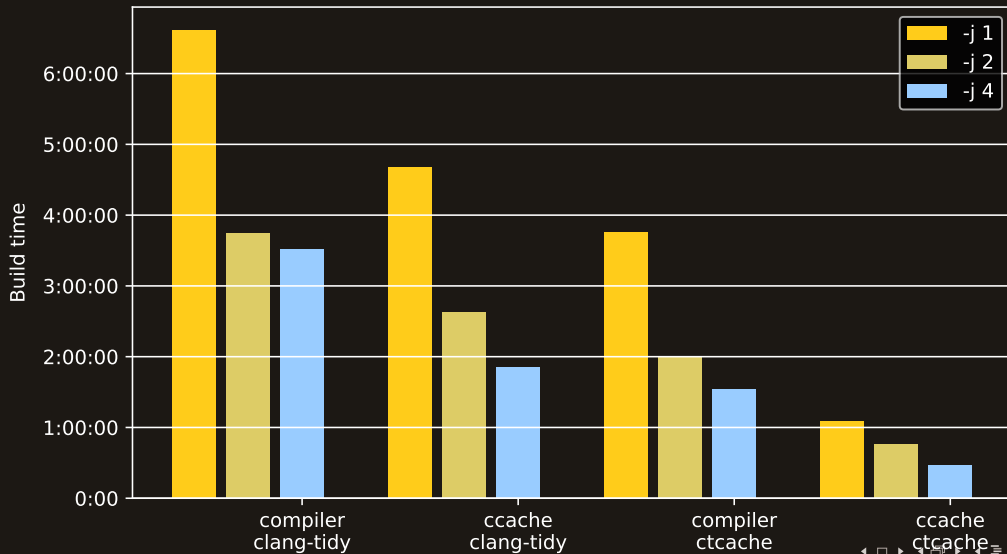
Project 1 – clean build times (compilation caused some swapping on disk)



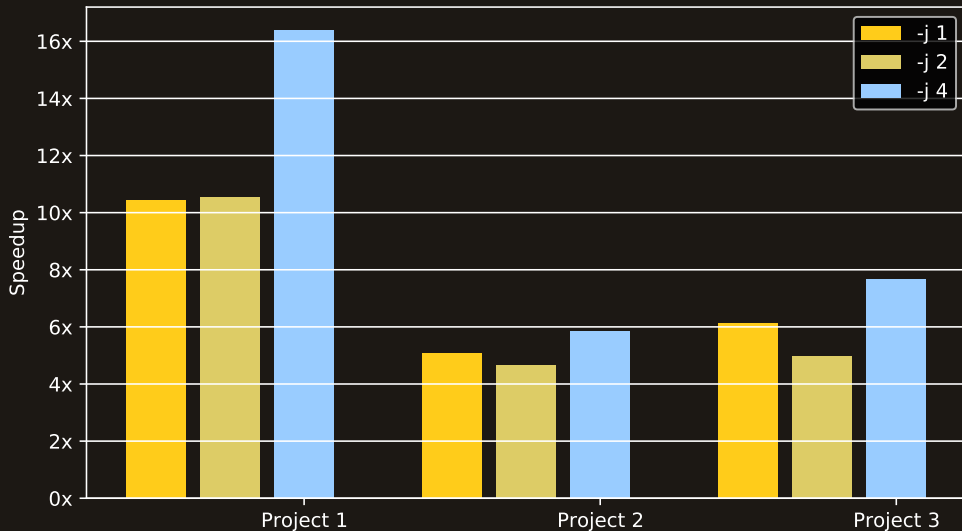
Project 2 – clean build times



Project 3 – clean build times



Clean build speedups – $t_{uncached}/t_{cached}$



Conclusions

Conclusions

- Having static analysis (almost) always on is useful during development.
- Unnecessary delays from re-checking of unchanged code can be avoided by caching.
- Using `clang-tidy-cache` (and `ccache`) significantly improves build times¹² for projects of various sizes.
- The achieved speedups¹³ are in the range of $\approx 5x - 15x$.

¹²especially clean rebuild times

¹³when using both `ccache` and `ctcache`

Overview

- clang-tidy-cache (client & server)
 - are reasonably simple to setup,
 - provide many configuration options,
 - provide several deployment options,
 - can be shared among multiple users,
 - can be integrated into CI pipelines¹⁴,
 - work well together with ctcache.

¹⁴Jenkins, Travis, etc.

Personal experience

- Did development with `clang-tidy` checks enabled on “Project 2” and “Project 3” for more than a year.
- This helps to find and fix many bugs early.
- Also helps to enforce coding guidelines.
- With `clang-tidy-cache` the build times are kept to very reasonable levels.
- Build times increase only in rare cases when one of the “core” headers are changed.

Possible future improvements

- Support other command-line static analysis tools.
- Support for *HTTPS* in the server.
- Request count over time statistics chart.
- Additional command-line options in the client:
 - Display server cache statistics on the command-line.
 - Reset server cache from the command-line.
- ...

Thank you!

Questions?

<https://github.com/matus-chochlik/ctcache>

<https://github.com/matus-chochlik/ctcache/doc/overview.pdf>

Extras

Sharing cache server among multiple users

- If multiple developers are building the same set of sources¹⁵,
- and they have reasonably similar development environments¹⁶,
- they have access to the same `clang-tidy-cache-server` instance,
- and they setup the `CTCACHE_STRIP` variable properly,
- they can share each others, cached static analysis results.

¹⁵they are working on the same project

¹⁶same platform, same compiler, STL and library versions

What prevents analysis result sharing

- The cache works by hashing input strings – command-line arguments, pre-processed source file lines, configuration file lines, etc.
- The goal is to get the same hash for the larger “same” check.
- Between users with similar environments, the analysis inputs typically differ only in username-dependent sub-strings¹⁷.

¹⁷i.e. paths in the `-L`, `-I`, etc. compiler options

The CTCACHE_STRIP variable

- `CTCACHE_STRIP` – List of colon-separated strings, which are removed from the input strings.

- For example

```
CTCACHE_STRIP="/home/user/myproject:/opt/custom/libs" .
```

- If the “right” strings are stripped by every user, same hashes are generated.
 - May be somewhat tricky to setup properly.
 - If set-up incorrectly may lead to false positives!