Speeding up static analysis with `clang-tidy-cache`

January 28, 2021

Introduction

clang-tidy

- "a `clang`-based C++ static analysis tool[1]"
  - 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 built-in support for `clang-tidy`:
  - the `CXX_CLANG_TIDY` target property[2].
  - 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 / executable path:

```
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()
```

`clang-tidy` is configured[3] by creating a file called `.clang-tidy`, typically in the project root directory:

```
Checks:              'clang-diagnostic-*,clang-analyzer-*,bugprone-*,\
                      -bugprone-branch-clone,-bugprone-macro-parentheses,\
                      -bugprone-exception-escape,cert-*,hicpp-*'
WarningsAsErrors: '*'
HeaderFilterRegex: '*'
AnalyzeTemporaryDtors: false
FormatStyle:      file
CheckOptions:
...
```

[3] https://clang.llvm.org/extra/clang-tidy/

Checking this *bad* C++ code with `clang-tidy`:

```
int main(int argc, const char** argv) {
  const char str[] = "This is bad C++";
  std::cout << str << std::endl;
  return 0;
}
```

we will get the following analysis findings:

```
.../bad01.cpp:4:11: error: do not declare C-style arrays,
use std::array<> instead [hicpp-avoid-c-arrays,-warnings-as-errors]
  const char str[] = "This is bad C++";
          ^
.../bad01.cpp:5:18: error: do not implicitly decay an array into a pointer;
consider using gsl::array_view or an explicit cast instead
[hicpp-no-array-decay,-warnings-as-errors]
  std::cout << str << std::endl;
```

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.
  - Especially in CI pipelines doing full rebuilds.
  - 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

- Often 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.
  - Happens typically in CI pipelines, but occasionally also on developers' machines.

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-caching[4].
  - Track all inputs of the analysis:
    - configuration options,
    - command-line arguments,
    - the source files,
    - etc.

[4] `https://ccache.dev/`

`clang-tidy-cache!`

How does it work?

- Scans the inputs of `clang-tidy`:
  - command-line arguments,
  - configuration files,
  - analysed source files[5].
- Makes a hash uniquely identifying the invocation from the above.

[5] preprocessed by the C/C++ preprocessor

How does it work? (cont.)

- Checks if the hash is in the cache database:
  - if it is
    - doesn't run `clang-tidy` and returns immediately,
    - this is typically **much** faster.
  - otherwise
    - runs `clang-tidy`, and if successful[6] stores the hash.
      - This means that sources with findings keep being re-checked and the findings are shown.

[6] if there are no warnings or errors reported

How do I use it?

Create[7] a wrapper script, called `clang-tidy`, like:

```
#!/bin/bash
REAL_CT=/full/path/to/clang-tidy

/path/to/clang-tidy-cache \
  "${REAL_CT}" "${@}"
```

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

```
export PATH="/path/to/wrapper-script-dir:${PATH}"
```

[7] or 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

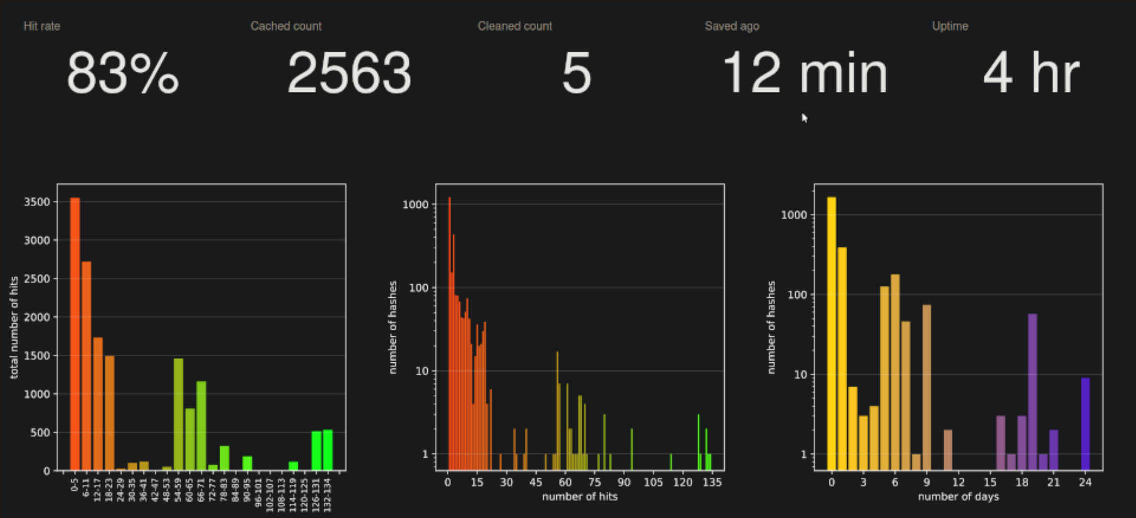
- `http://ctcache:5000/...`
  - `/cache/<hash>` – insert `<hash>` into cache.
  - `/is_cached/<hash>` – tests if `<hash>` is cached.
  - `/purge_cache` – remove all cached hashes.
  - `/info` – static configuration information[8].
  - `/stats` – server run-time status information[9].
  - `/stats/*` – individual server status readouts[10].
  - `/stats/ctcache.json` – long-term persistent server status information[11].
  - `/images/*` – status chart images[12].

[8] as JSON object
[9] as JSON object
[10] as JSON values
[11] as JSON file
[12] as SVG

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 ctcache.service
```

Permanently disable automatic start of the service:

```
systemctl --user disable ctcache.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
```

Start container using the volume:

```
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:

```yaml
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 the 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_DUMP	✓		enables dumping of everything that is hashed into a file
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/oglplu2`,
  - ≈ 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,
  - ≈ 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 / ≈ 650Mbps[13]).

[13] according to `iwconfig`

Test configurations

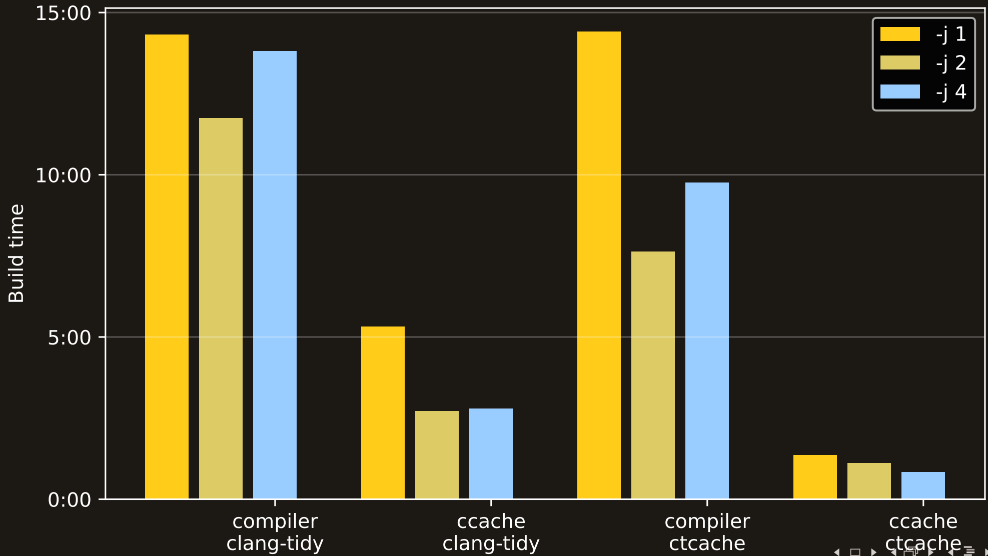
- "compiler & clang-tidy"
  - `export CCACHE_DISABLE=1`
  - `export CTCACHE_DISABLE=1`
- "ccache & clang-tidy"
  - `unset CCACHE_DISABLE`
  - `export CTCACHE_DISABLE=1`
- "compiler & ctcache"
  - `export CCACHE_DISABLE=1`
  - `unset CTCACHE_DISABLE`
- "ccache & ctcache"
  - `unset CCACHE_DISABLE`
  -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

Test execution

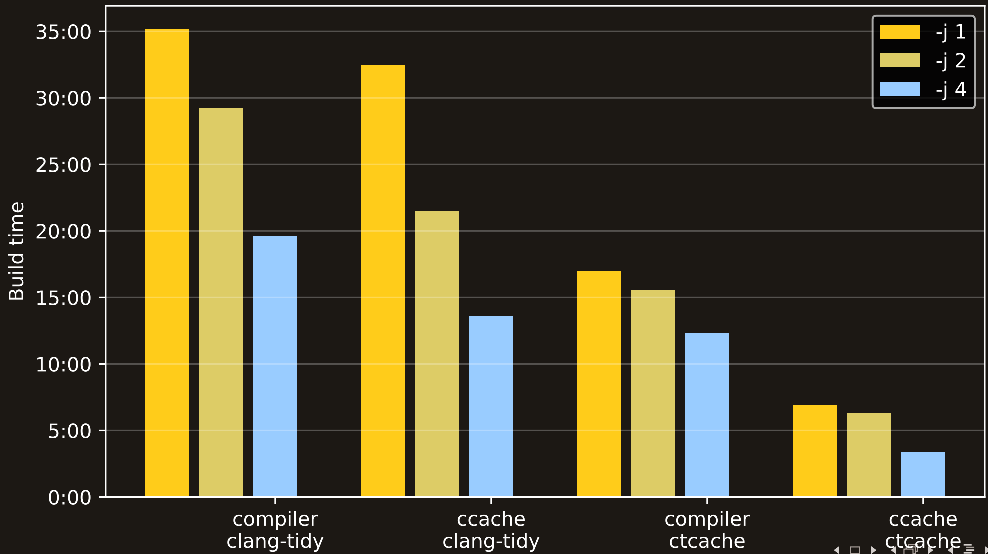
- For each test project:
  - `cd /path/to/build/dir`
  - `rm -rf ./`
  - `cmake ... /path/to/project/source`
  - do initial build with caches enabled[14] (no measurements),
  - for each test configuration:
    - setup environment variables,
    - `make clean`
    - `time make -j N ...`

[14] to fill them

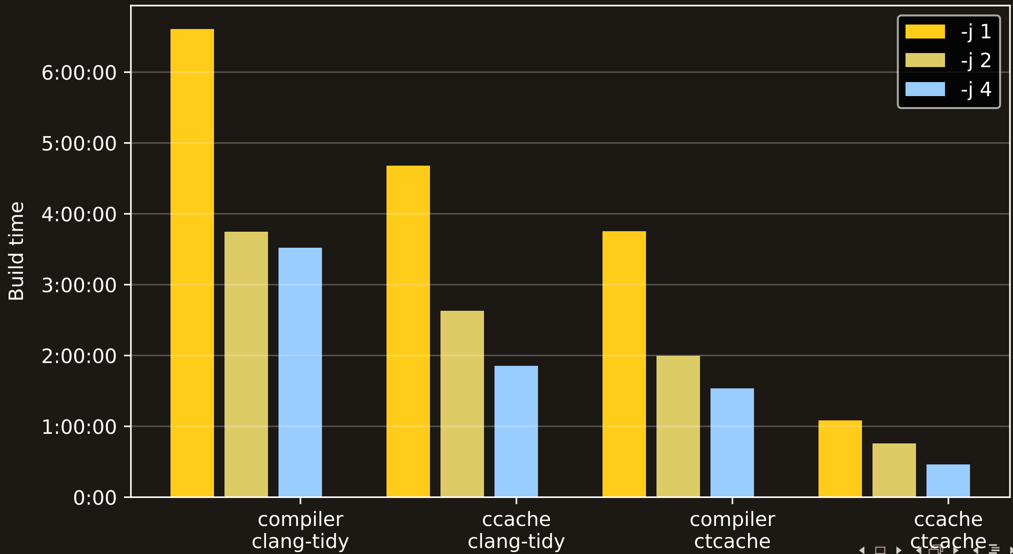
Project 1 – clean build times (compilation caused some swapping on disk)



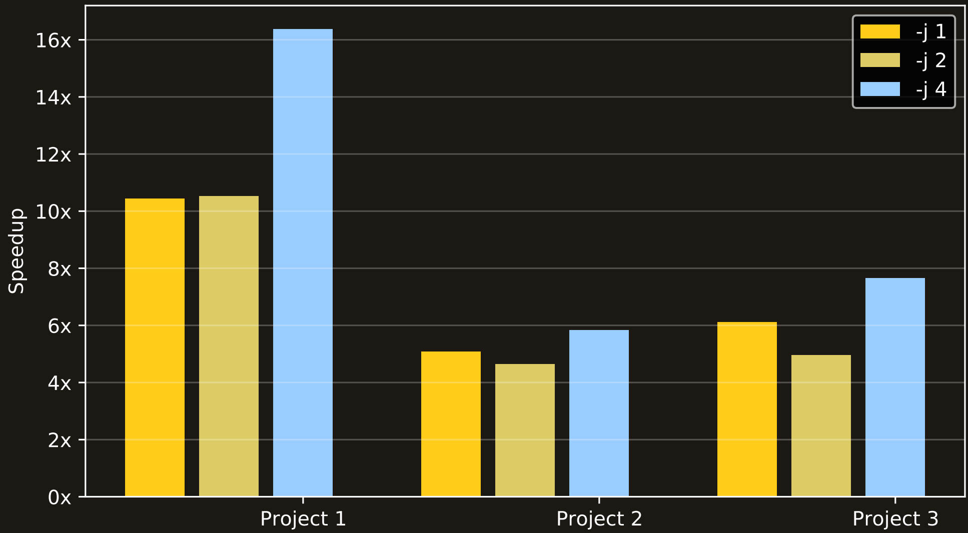
Project 2 – clean build times



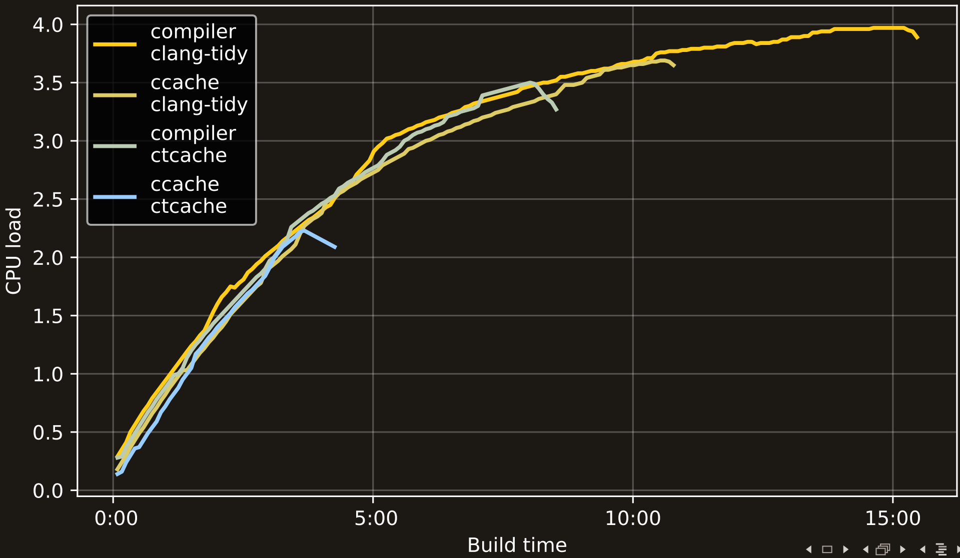
Project 3 – clean build times



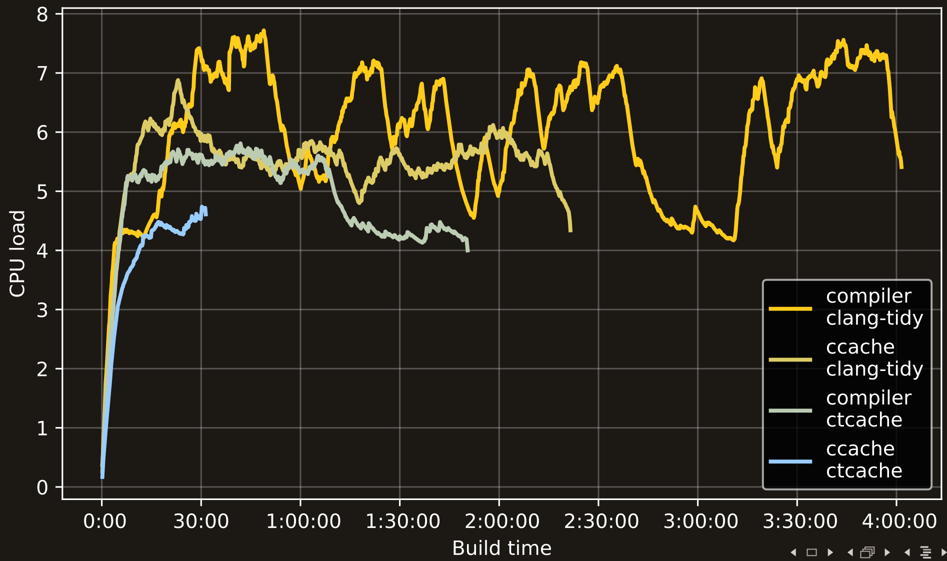
Clean build speedups – $t_{uncached} / t_{cached}$



Project 2 – system load (-j 4)



Project 3 – system load (-j 4)



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 times[15] for projects of various sizes.
- The achieved speedups[16] are in the range of $\approx$ 5x – 15x.

[15] especially clean rebuild times
[16] when using both `ccache` and `ctcache`

Overview

- `clang-tidy-cache` (client and server)
  - are reasonably simple to setup,
  - provide many configuration options,
  - provide several deployment options,
  - can be integrated into CI pipelines[17],
  - work well together with `ccache`,
  - can be shared among multiple users[18].

[17] Jenkins, Travis, etc.
[18] more in a moment

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.
- Number of requests 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 sources[19],
- and they have reasonably similar development environments[20],
- 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.

[19] they are working on the same project

[20] same 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 **“same” check**.
- Between users with similar environments, the analysis inputs typically differ only in **username-dependent** sub-strings[21].

[21] i.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 hashed inputs.
  - For example `CTCACHE_STRIP="/home/user/myproject:/opt/custom/libs"`.
- If the “right” strings are stripped by every user $\longrightarrow$ the same hashes are generated.
- May be somewhat tricky to setup properly.
- If set-up incorrectly may lead to false positives in some cases!

The `CTCACHE_DUMP` variable

- If defined in the environment where `clang-tidy-cache` is executed, then everything that is hashed, is dumped into a file.
  - `/tmp/ctcache.dump`
  - in append mode.
- This can help tweaking the content of the `CTCACHE_STRIP` variable and help to achieve the same hashes for the "same" builds in various users' environments.