NAME

ccache − a fast C/C++ compiler cache

SYNOPSIS

ccache [*options*]
ccache *compiler* [*compiler options*]
compiler [*compiler options*]   (via symbolic link)

DESCRIPTION

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.

Ccache has been carefully written to always produce exactly the same compiler output that you would get without the cache. The only way you should be able to tell that you are using ccache is the speed. Currently known exceptions to this goal are listed under *CAVEATS*. If you discover an undocumented case where ccache changes the output of your compiler, please let us know.

RUN MODES

There are two ways to use ccache. You can either prefix your compilation commands with **ccache** or you can let ccache masquerade as the compiler by creating a symbolic link (named as the compiler) to ccache. The first method is most convenient if you just want to try out ccache or wish to use it for some specific projects. The second method is most useful for when you wish to use ccache for all your compilations.

To use the first method, just make sure that **ccache** is in your **PATH**.

To use the second method on a Debian system, it’s easiest to just prepend **/usr/lib/ccache** to your **PATH**. **/usr/lib/ccache** contains symlinks for all compilers currently installed as Debian packages.

Alternatively, you can create any symlinks you like yourself like this:

```
ln −s /usr/bin/ccache /usr/local/bin/gcc
ln −s /usr/bin/ccache /usr/local/bin/g++
ln −s /usr/bin/ccache /usr/local/bin/cc
ln −s /usr/bin/ccache /usr/local/bin/c++
```

And so forth. This will work as long as the directory with symlinks comes before the path to the compiler (which is usually in **/usr/bin**). After installing you may wish to run “which gcc” to make sure that the correct link is being used.

Warning

The technique of letting ccache masquerade as the compiler works well, but currently doesn’t interact well with other tools that do the same thing. See *USING CCACHE WITH OTHER COMPILER WRAPPERS*.

Warning

Use a symbolic links for masquerading, not hard links.

COMMAND LINE OPTIONS

These command line options only apply when you invoke ccache as “ccache”. When invoked as a compiler (via a symlink as described in the previous section), the normal compiler options apply and you should refer to the compiler’s documentation.

Common options

−c, **−−cleanup**

Clean up the cache by removing old cached files until the specified file number and cache size limits are not exceeded. This also recalculates the cache file count and size totals. Normally, there is no need to initiate cleanup manually as ccache keeps the cache below the specified limits at runtime and keeps statistics up to date on each compilation. Forcing a cleanup is mostly useful if you manually modify the cache contents or believe that the cache size statistics may be inaccurate.

−C, **−−clear**
Clear the entire cache, removing all cached files, but keeping the configuration file.

−−config−path *PATH*
Let the command line options operate on configuration file *PATH* instead of the default. Using this option has the same effect as setting the environment variable **CCACHE_CONFIGPATH** temporarily.

−d, **−−dir** *PATH*
Let the command line options operate on cache directory *PATH* instead of the default. For example, to show statistics for a cache directory at **/shared/ccache** you can run **ccache −d /shared/ccache −s**. Using this option has the same effect as setting the environment variable **CCACHE_DIR** temporarily.

−−evict−namespace *NAMESPACE*
Remove files created in the given **namespace** from the cache.

−−evict−older−than *AGE*
Remove files older than *AGE* from the cache. *AGE* should be an unsigned integer with a **d** (days) or **s** (seconds) suffix. If combined with **−−evict−namespace**, only remove old files within that namespace.

−h, **−−help**
Print a summary of command line options.

−F *NUM*, **−−max−files** *NUM*
Set the maximum number of files allowed in the cache to *NUM*. Use 0 for no limit. The value is stored in a configuration file in the cache directory and applies to all future compilations.

−M *SIZE*, **−−max−size** *SIZE*
Set the maximum size of the files stored in the cache. *SIZE* should be a number followed by an optional suffix: k, M, G, T (decimal), Ki, Mi, Gi or Ti (binary). The default suffix is G. Use 0 for no limit. The value is stored in a configuration file in the cache directory and applies to all future compilations.

−X *LEVEL*, **−−recompress** *LEVEL*
Recompress the cache to level *LEVEL* using the Zstandard algorithm. The level can be an integer, with the same semantics as the **compression_level** configuration option), or the special value **uncompressed** for no compression. See *CACHE COMPRESSION* for more information. This can potentionally take a long time since all files in the cache need to be visited. Only files that are currently compressed with a different level than *LEVEL* will be recompressed.

−o *KEY=VALUE*, **−−set−config** *KEY=VALUE*
Set configuration option *KEY* to *VALUE*. See *CONFIGURATION* for more information.

−x, **−−show−compression**
Print cache compression statistics. See *CACHE COMPRESSION* for more information. This can potentionally take a long time since all files in the cache need to be visited.

−p, **−−show−config**
Print current configuration options and from where they originate (environment variable, configuration file or compile−time default) in human−readable format.

−−show−log−stats
Print statistics counters from the stats log in human−readable format. See **stats_log**. Use **−v**/**−−verbose** once or twice for more details.

−s, **−−show−stats**
Print a summary of configuration and statistics counters in human−readable format. Use **−v**/**−−verbose** once or twice for more details.

−v, **−−verbose**
Increase verbosity. The option can be given multiple times.

−V, **−−version**
Print version and copyright information.

−z, **−−zero−stats**
Zero the cache statistics (but not the configuration options).

Options for secondary storage

−−trim−dir *PATH*
Remove old files from directory *PATH* until it is at most the size specified by **−−trim−max−size**.

Warning
Don’t use this option to trim the primary cache. To trim the primary cache directory to a certain size, use **CCACHE_MAXSIZE=***SIZE* **ccache −c**.

−−trim−max−size *SIZE*
Specify the maximum size for **−−trim−dir**. *SIZE should be a number followed by an optional suffix: k, M, G, T (decimal), Ki, Mi, Gi or Ti (binary). The default suffix is G.*

−−trim−method *METHOD*
Specify the method to trim a directory with **−−trim−dir**. *Possible values are:*

atime
LRU (least recently used) using the file access timestamp. This is the default.

mtime
LRU (least recently used) using the file modification timestamp.

Options for scripting or debugging

−−checksum−file *PATH*
Print the checksum (128 bit XXH3) of the file at *PATH* (− for standard input).

−−dump−manifest *PATH*
Dump manifest file at *PATH* (− for standard input) in text format to standard output. This is only useful when debugging ccache and its behavior.

−−dump−result *PATH*
Dump result file at *PATH* (− for standard input) in text format to standard output. This is only useful when debugging ccache and its behavior.

−−extract−result *PATH*
Extract data stored in the result file at *PATH* (− for standard input). The data will be written to **ccache−result.*** files in to the current working directory. This is only useful when debugging ccache and its behavior.

−k *KEY*, **−−get−config** *KEY*
Print the value of configuration option *KEY*. See *CONFIGURATION* for more information.

−−hash−file *PATH*
Print the hash (160 bit BLAKE3) of the file at *PATH* (− for standard input). This is only useful when

debugging ccache and its behavior.

−−print−stats
Print statistics counter IDs and corresponding values in machine−parsable (tab−separated) format.

Extra options

When run as a compiler, ccache usually just takes the same command line options as the compiler you are using. The only exception to this is the option **−−ccache−skip**. That option can be used to tell ccache to avoid interpreting the next option in any way and to pass it along to the compiler as−is.

Note
−−ccache−skip currently only tells ccache not to interpret the next option as a special compiler option — the option will still be included in the direct mode hash.

The reason this can be important is that ccache does need to parse the command line and determine what is an input filename and what is a compiler option, as it needs the input filename to determine the name of the resulting object file (among other things). The heuristic ccache uses when parsing the command line is that any argument that exists as a file is treated as an input file name. By using **−−ccache−skip** you can force an option to not be treated as an input file name and instead be passed along to the compiler as a command line option.

Another case where **−−ccache−skip** can be useful is if ccache interprets an option specially but shouldn't, since the option has another meaning for your compiler than what ccache thinks.

CONFIGURATION

Ccache's default behavior can be overridden by options in configuration files, which in turn can be overridden by environment variables with names starting with **CCACHE_**. Ccache normally reads configuration from two files: first a system−level configuration file and secondly a cache−specific configuration file. The priorities of configuration options are as follows (where 1 is highest):

1. Environment variables.
2. The primary (cache−specific) configuration file (see below).
3. The secondary (system−wide read−only) configuration file **<sysconfdir>/ccache.conf** (typically **/etc/ccache.conf** or **/usr/local/etc/ccache.conf**).
4. Compile−time defaults.

As a special case, if the the environment variable **CCACHE_CONFIGPATH** is set it specifies the primary configuration file and the secondary (system−wide) configuration file won't be read.

Location of the primary configuration file

The location of the primary (cache−specific) configuration is determined like this:

1. If **CCACHE_CONFIGPATH** is set, use that path.
2. Otherwise, if the environment variable **CCACHE_DIR** is set then use **$CCACHE_DIR/ccache.conf**.
3. Otherwise, if **cache_dir** is set in the secondary (system−wide) configuration file then use **<cache_dir>/ccache.conf**.
4. Otherwise, if there is a legacy **$HOME/.ccache** directory then use **$HOME/.ccache/ccache.conf**.
5. Otherwise, if **XDG_CONFIG_HOME** is set then use **$XDG_CONFIG_HOME/ccache/ccache.conf**.
6. Otherwise, use **%APPDATA%/ccache/ccache.conf** (Windows), **$HOME/Library/Preferences/ccache/ccache.conf** (macOS) or **$HOME/.config/ccache/ccache.conf** (other systems).

Configuration file syntax

Configuration files are in a simple “key = value” format, one option per line. Lines starting with a hash sign are comments. Blank lines are ignored, as is whitespace surrounding keys and values. Example:

```
# Set maximum cache size to 10 GB:
max_size = 10G
```

Boolean values

Some configuration options are boolean values (i.e. truth values). In a configuration file, such values must be set to the string **true** or **false**. For the corresponding environment variables, the semantics are a bit different:

- A set environment variable means “true” (even if set to the empty string).
- The following case−insensitive negative values are considered an error (instead of surprising the user): **0**, **false**, **disable** and **no**.
- An unset environment variable means “false”.

Each boolean environment variable also has a negated form starting with **CCACHE_NO**. For example, **CCACHE_COMPRESS** can be set to force compression and **CCACHE_NOCOMPRESS** can be set to force no compression.

Configuration options

Below is a list of available configuration options. The corresponding environment variable name is indicated in parentheses after each configuration option key.

absolute_paths_in_stderr (**CCACHE_ABSSTDERR**)
This option specifies whether ccache should rewrite relative paths in the compiler’s standard error output to absolute paths. This can be useful if you use **base_dir** with a build system (e.g. CMake with the "Unix Makefiles" generator) that executes the compiler in a different working directory, which makes relative paths in compiler errors or warnings incorrect. The default is false.

base_dir (**CCACHE_BASEDIR**)
This option should be an absolute path to a directory. If set, ccache will rewrite absolute paths into paths relative to the current working directory, but only absolute paths that begin with **base_dir**. Cache results can then be shared for compilations in different directories even if the project uses absolute paths in the compiler command line. See also the discussion under *COMPILING IN DIFFERENT DIRECTORIES*. If set to the empty string (which is the default), no rewriting is done.

A typical path to use as **base_dir** is your home directory or another directory that is a parent of your project directories. Don’t use **/** as the base directory since that will make ccache also rewrite paths to system header files, which typically is contraproductive.

For example, say that Alice’s current working directory is **/home/alice/project1/build** and that she compiles like this:

```
ccache gcc −I/usr/include/example −I/home/alice/project2/include −c /home/alic
```

Here is what ccache will actually execute for different **base_dir** values:

```
# Current working directory: /home/alice/project1/build

# With base_dir = /:
gcc −I../../../../usr/include/example −I../../project2/include −c ../src/examp

# With base_dir = /home or /home/alice:
```

```
gcc −I/usr/include/example −I../../project2/include −c ../src/example.c

# With base_dir = /home/alice/project1 or /home/alice/project1/src:
gcc −I/usr/include/example −I/home/alice/project2/include −c ../src/example.c
```

If Bob has put **project1** and **project2** in **/home/bob/stuff** and both users have set **base_dir** to **/home** or **/home/$USER**, then Bob will get a cache hit (if they share ccache directory) since the actual command line will be identical to that of Alice:

```
# Current working directory: /home/bob/stuff/project1/build

# With base_dir = /home or /home/bob:
gcc −I/usr/include/example −I../../project2/include −c ../src/example.c
```

Without **base_dir** there will be a cache miss since the absolute paths will differ. With **base_dir** set to **/** there will be a cache miss since the relative path to **/usr/include/example** will be different. With **base_dir** set to **/home/bob/stuff/project1** there will a cache miss since the path to project2 will be a different absolute path.

cache_dir (**CCACHE_DIR**)
This option specifies where ccache will keep its cached compiler outputs. The default is **$XDG_CACHE_HOME/ccache** if **XDG_CACHE_HOME** is set, otherwise **$HOME/.cache/ccache**. Exception: If the legacy directory **$HOME/.ccache** exists then that directory is the default.

See also *Location of the primary configuration file*.

If you want to use another **CCACHE_DIR** value temporarily for one ccache invocation you can use the **−d**/**−−dir** command line option instead.

compiler (**CCACHE_COMPILER** or (deprecated) **CCACHE_CC**)
This option can be used to force the name of the compiler to use. If set to the empty string (which is the default), ccache works it out from the command line.

compiler_check (**CCACHE_COMPILERCHECK**)
By default, ccache includes the modification time (“mtime”) and size of the compiler in the hash to ensure that results retrieved from the cache are accurate. This option can be used to select another strategy. Possible values are:

content
Hash the content of the compiler binary. This makes ccache very slightly slower compared to **mtime**, but makes it cope better with compiler upgrades during a build bootstrapping process.

mtime
Hash the compiler’s mtime and size, which is fast. This is the default.

none
Don’t hash anything. This may be good for situations where you can safely use the cached results even though the compiler’s mtime or size has changed (e.g. if the compiler is built as part of your build system and the compiler’s source has not changed, or if the compiler only has changes that don’t affect code generation). You should only use **none** if you know what you are doing.

string:value
Hash **value**. This can for instance be a compiler revision number or another string that the build

system generates to identify the compiler.

a command string

Hash the standard output and standard error output of the specified command. The string will be split on whitespace to find out the command and arguments to run. No other interpretation of the command string will be done, except that the special word **%compiler%** will be replaced with the path to the compiler. Several commands can be specified with semicolon as separator. Examples:

```
%compiler% −v

%compiler% −dumpmachine; %compiler% −dumpversion
```

You should make sure that the specified command is as fast as possible since it will be run once for each ccache invocation.

Identifying the compiler using a command is useful if you want to avoid cache misses when the compiler has been rebuilt but not changed.

Another case is when the compiler (as seen by ccache) actually isn't the real compiler but another compiler wrapper — in that case, the default **mtime** method will hash the mtime and size of the other compiler wrapper, which means that ccache won't be able to detect a compiler upgrade. Using a suitable command to identify the compiler is thus safer, but it's also slower, so you should consider continue using the **mtime** method in combination with the **prefix_command** option if possible. See *USING CCACHE WITH OTHER COMPILER WRAPPERS*.

compiler_type (**CCACHE_COMPILERTYPE**)

Ccache normally guesses the compiler type based on the compiler name. The **compiler_type** option lets you force a compiler type. This can be useful if the compiler has a non−standard name but is actually one of the known compiler types. Possible values are:

auto

Guess one of the types below based on the compiler name (following symlinks). This is the default.

clang

Clang−based compiler.

gcc

GCC−based compiler.

nvcc

NVCC (CUDA) compiler.

other

Any compiler other than the known types.

pump

distcc's "pump" script.

compression (**CCACHE_COMPRESS** or **CCACHE_NOCOMPRESS**, see *Boolean values* above)

If true, ccache will compress data it puts in the cache. However, this option has no effect on how files are retrieved from the cache; compressed and uncompressed results will still be usable regardless of this option. The default is true.

Compression is done using the Zstandard algorithm. The algorithm is fast enough that there should be little reason to turn off compression to gain performance. One exception is if the cache is located on a compressed file system, in which case the compression performed by ccache of course is redundant.

Compression will be disabled if file cloning (the **file_clone** option) or hard linking (the **hard_link** option) is enabled.

compression_level (**CCACHE_COMPRESSLEVEL**)
This option determines the level at which ccache will compress object files using the real−time compression algorithm Zstandard. It only has effect if **compression** is enabled (which it is by default). Zstandard is extremely fast for decompression and very fast for compression for lower compression levels. The default is 0.

Semantics of **compression_level**:

> **0**
A positive value corresponds to normal Zstandard compression levels. Lower levels (e.g. **1**) mean faster compression but worse compression ratio. Higher levels (e.g. **19**) mean slower compression but better compression ratio. The maximum possible value depends on the libzstd version, but at least up to 19 is available for all versions. Decompression speed is essentially the same for all levels. As a rule of thumb, use level 5 or lower since higher levels may slow down compilations noticeably. Higher levels are however useful when recompressing the cache with command line option **−X**/**−−recompress**.

< **0**
A negative value corresponds to Zstandard's "ultra−fast" compression levels, which are even faster than level 1 but with less good compression ratios. For instance, level **−3** corresponds to **−−fast=3** for the **zstd** command line tool. In practice, there is little use for levels lower than **−5** or so.

0 (default)
The value **0** means that ccache will choose a suitable level, currently **1**.

See the Zstandard documentation <http://zstd.net> for more information.

cpp_extension (**CCACHE_EXTENSION**)
This option can be used to force a certain extension for the intermediate preprocessed file. The default is to automatically determine the extension to use for intermediate preprocessor files based on the type of file being compiled, but that sometimes doesn't work. For example, when using the "aCC" compiler on HP−UX, set the cpp extension to **i**.

debug (**CCACHE_DEBUG** or **CCACHE_NODEBUG**, see *Boolean values* above)
If true, enable the debug mode. The debug mode creates per−object debug files that are helpful when debugging unexpected cache misses. Note however that ccache performance will be reduced slightly. See *CACHE DEBUGGING* for more information. The default is false.

debug_dir (**CCACHE_DEBUGDIR**)
Specifies where to write per−object debug files if the debug mode is enabled. If set to the empty string, the files will be written next to the object file. If set to a directory, the debug files will be written with full absolute paths in that directory, creating it if needed. The default is the empty string.

For example, if **debug_dir** is set to **/example**, the current working directory is **/home/user** and the object file is **build/output.o** then the debug log will be written to **/example/home/user/build/output.o.ccache−log**. See also *CACHE DEBUGGING*.

depend_mode (**CCACHE_DEPEND** or **CCACHE_NODEPEND**, see *Boolean values* above)
If true, the depend mode will be used. The default is false. See *The depend mode*.

direct_mode (**CCACHE_DIRECT** or **CCACHE_NODIRECT**, see *Boolean values* above)
If true, the direct mode will be used. The default is true. See *The direct mode*.

disable (**CCACHE_DISABLE** or **CCACHE_NODISABLE**, see *Boolean values* above)
When true, ccache will just call the real compiler, bypassing the cache completely. The default is false.

extra_files_to_hash (**CCACHE_EXTRAFILES**)
This option is a list of paths to files that ccache will include in the the hash sum that identifies the build. The list separator is semicolon on Windows systems and colon on other systems.

file_clone (**CCACHE_FILECLONE** or **CCACHE_NOFILECLONE**, see *Boolean values* above)
If true, ccache will attempt to use file cloning (also known as "copy on write", "CoW" or "reflinks") to store and fetch cached compiler results. **file_clone** has priority over **hard_link**. The default is false.

Files stored by cloning cannot be compressed, so the cache size will likely be significantly larger if this option is enabled. However, performance may be improved depending on the use case.

Unlike the **hard_link** option, **file_clone** is completely safe to use, but not all file systems support the feature. For such file systems, ccache will fall back to use plain copying (or hard links if **hard_link** is enabled).

hard_link (**CCACHE_HARDLINK** or **CCACHE_NOHARDLINK**, see *Boolean values* above)
If true, ccache will attempt to use hard links to store and fetch cached object files. The default is false.

Files stored via hard links cannot be compressed, so the cache size will likely be significantly larger if this option is enabled. However, performance may be improved depending on the use case.

Warning
Do not enable this option unless you are aware of these caveats:

- If the resulting file is modified, the file in the cache will also be modified since they share content, which corrupts the cache entry. As of version 4.0, ccache makes stored and fetched object files read−only as a safety measure guard. Furthermore, a simple integrity check is made for cached object files by verifying that their sizes are correct. This means that mistakes like **strip file.o** or **echo >file.o** will be detected even if the object file is made writeable, but a modification that doesn't change the file size will not.
- Programs that don't expect that files from two different identical compilations are hard links to each other can fail.
- Programs that rely on modification times (like **make**) can be confused if several users (or one user with several build trees) use the same cache directory. The reason for this is that the object files share i−nodes and therefore modification times. If **file.o** is in build tree **A** (hard−linked from the cache) and **file.o** then is produced by ccache in build tree **B** by hard−linking from the cache, the modification timestamp will be updated for **file.o** in build tree **A** as well. This can retrigger relinking in build tree **A** even though nothing really has changed.

hash_dir (**CCACHE_HASHDIR** or **CCACHE_NOHASHDIR**, see *Boolean values* above)
If true (which is the default), ccache will include the current working directory (CWD) in the hash that is used to distinguish two compilations when generating debug info (compiler option **−g** with variations). Exception: The CWD will not be included in the hash if **base_dir** is set (and matches the CWD) and the compiler option **−fdebug−prefix−map** is used. See also the discussion under *COMPILING IN DIFFERENT DIRECTORIES*.

The reason for including the CWD in the hash by default is to prevent a problem with the storage of the current working directory in the debug info of an object file, which can lead ccache to return a cached object file that has the working directory in the debug info set incorrectly.

You can disable this option to get cache hits when compiling the same source code in different directories if you don’t mind that CWD in the debug info might be incorrect.

ignore_headers_in_manifest (**CCACHE_IGNOREHEADERS**)
This option is a list of paths to files (or directories with headers) that ccache will **not** include in the manifest list that makes up the direct mode. Note that this can cause stale cache hits if those headers do indeed change. The list separator is semicolon on Windows systems and colon on other systems.

ignore_options (**CCACHE_IGNOREOPTIONS**)
This option is a space−delimited list of compiler options that ccache will exclude from the hash. Excluding a compiler option from the hash can be useful when you know it doesn’t affect the result (but ccache doesn’t know that), or when it does and you don’t care. If a compiler option in the list is suffixed with an asterisk (*****) it will be matched as a prefix. For example, **−fmessage−length=*** will match both **−fmessage−length=20** and **−fmessage−length=70**.

inode_cache (**CCACHE_INODECACHE** or **CCACHE_NOINODECACHE**, see *Boolean values* above)
If true, enables caching of source file hashes based on device, inode and timestamps. This will reduce the time spent on hashing included files as the result can be resused between compilations.

The feature is still experimental and thus off by default. It is currently not available on Windows.

The feature requires **temporary_dir** to be located on a local filesystem.

keep_comments_cpp (**CCACHE_COMMENTS** or **CCACHE_NOCOMMENTS**, see *Boolean values* above)
If true, ccache will not discard the comments before hashing preprocessor output. This can be used to check documentation with **−Wdocumentation**.

limit_multiple (**CCACHE_LIMIT_MULTIPLE**)
Sets the limit when cleaning up. Files are deleted (in LRU order) until the levels are below the limit. The default is 0.8 (= 80%). See *Automatic cleanup* for more information.

log_file (**CCACHE_LOGFILE**)
If set to a file path, ccache will write information on what it is doing to the specified file. This is useful for tracking down problems.

If set to **syslog**, ccache will log using **syslog()** instead of to a file. If you use rsyslogd, you can add something like this to **/etc/rsyslog.conf** or a file in **/etc/rsyslog.d**:

```
# log ccache to file
:programname, isequal, "ccache"         /var/log/ccache
# remove from syslog
& ˜
```

max_files (**CCACHE_MAXFILES**)
This option specifies the maximum number of files to keep in the cache. Use 0 for no limit (which is the default). See also *CACHE SIZE MANAGEMENT*.

max_size (**CCACHE_MAXSIZE**)
This option specifies the maximum size of the cache. Use 0 for no limit. The default value is 5G.

Available suffixes: k, M, G, T (decimal) and Ki, Mi, Gi, Ti (binary). The default suffix is G. See also *CACHE SIZE MANAGEMENT*.

namespace (**CCACHE_NAMESPACE**)

If set, the namespace string will be added to the hashed data for each compilation. This will make the associated cache entries logically separate from cache entries with other namespaces, but they will still share the same storage space. Cache entries can also be selectively removed from the primary cache with the command line option **−−evict−namespace**, potentially in combination with **−−evict−older−than**. . For instance, if you use the same primary cache for several disparate projects, you can use a unique namespace string for each one. This allows you to remove cache entries that belong to a certain project if stop working with that project.

path (**CCACHE_PATH**)

If set, ccache will search directories in this list when looking for the real compiler. The list separator is semicolon on Windows systems and colon on other systems. If not set, ccache will look for the first executable matching the compiler name in the normal **PATH** that isn't a symbolic link to ccache itself.

pch_external_checksum (**CCACHE_PCH_EXTSUM** or **CCACHE_NOPCH_EXTSUM**, see *Boolean values* above)

When this option is set, and ccache finds a precompiled header file, ccache will look for a file with the extension ".sum" added (e.g. "pre.h.gch.sum"), and if found, it will hash this file instead of the precompiled header itself to work around the performance penalty of hashing very large files.

prefix_command (**CCACHE_PREFIX**)

This option adds a list of prefixes (separated by space) to the command line that ccache uses when invoking the compiler. See also *USING CCACHE WITH OTHER COMPILER WRAPPERS*.

prefix_command_cpp (**CCACHE_PREFIX_CPP**)

This option adds a list of prefixes (separated by space) to the command line that ccache uses when invoking the preprocessor.

read_only (**CCACHE_READONLY** or **CCACHE_NOREADONLY**, see *Boolean values* above)

If true, ccache will attempt to use existing cached results, but it will not add new results to any cache backend. Statistics counters will still be updated, though, unless the **stats** option is set to **false**.

If you are using this because your ccache directory is read−only, you need to set **temporary_dir** since ccache will fail to create temporary files otherwise. You may also want to set **stats** to **false** make ccache not even try to update stats files.

read_only_direct (**CCACHE_READONLY_DIRECT** or **CCACHE_NOREADONLY_DIRECT**, see *Boolean values* above)

Just like **read_only** except that ccache will only try to retrieve results from the cache using the direct mode, not the preprocessor mode. See documentation for **read_only** regarding using a read−only ccache directory.

recache (**CCACHE_RECACHE** or **CCACHE_NORECACHE**, see *Boolean values* above)

If true, ccache will not use any previously stored result. New results will still be cached, possibly overwriting any pre−existing results.

reshare (**CCACHE_RESHARE** or **CCACHE_NORESHARE**, see *Boolean values* above)

If true, ccache will write results to secondary storage even for primary storage cache hits. The default is false.

run_second_cpp (**CCACHE_CPP2** or **CCACHE_NOCPP2**, see *Boolean values* above)

If true, ccache will first run the preprocessor to preprocess the source code (see *The preprocessor mode*) and then on a cache miss run the compiler on the source code to get hold of the object file. This is the default.

If false, ccache will first run preprocessor to preprocess the source code and then on a cache miss run the compiler on the *preprocessed source code* instead of the original source code. This makes cache misses slightly faster since the source code only has to be preprocessed once. The downside is that some compilers won't produce the same result (for instance diagnostics warnings) when compiling preprocessed source code.

A solution to the above mentioned downside is to set **run_second_cpp** to false and pass **−fdirectives−only** (for GCC) or **−frewrite−includes** (for Clang) to the compiler. This will cause the compiler to leave the macros and other preprocessor information, and only process the **#include** directives. When run in this way, the preprocessor arguments will be passed to the compiler since it still has to do *some* preprocessing (like macros).

secondary_storage (**CCACHE_SECONDARY_STORAGE**)
This option specifies one or several storage backends (separated by space) to query after the primary cache storage. See *SECONDARY STORAGE BACKENDS* for documentation of syntax and available backends.

Examples:

- **file:/shared/nfs/directory**
- **file:///shared/nfs/one|read−only file:///shared/nfs/two**
- **http://example.com/cache**
- **redis://example.com**

sloppiness (**CCACHE_SLOPPINESS**)
By default, ccache tries to give as few false cache hits as possible. However, in certain situations it's possible that you know things that ccache can't take for granted. This option makes it possible to tell ccache to relax some checks in order to increase the hit rate. The value should be a comma−separated string with one or several of the following values:

clang_index_store
Ignore the Clang compiler option **−index−store−path** and its argument when computing the manifest hash. This is useful if you use Xcode, which uses an index store path derived from the local project path. Note that the index store won't be updated correctly on cache hits if you enable this sloppiness.

file_stat_matches
Ccache normally examines a file's contents to determine whether it matches the cached version. With this sloppiness set, ccache will consider a file as matching its cached version if the mtimes and ctimes match.

file_stat_matches_ctime
Ignore ctimes when **file_stat_matches** is enabled. This can be useful when backdating files' mtimes in a controlled way.

include_file_ctime
By default, ccache will not cache a file if it includes a header whose ctime is too new. This sloppiness disables that check. See also *HANDLING OF NEWLY CREATED HEADER FILES*.

include_file_mtime

By default, ccache will not cache a file if it includes a header whose mtime is too new. This sloppiness disables that check. See also *HANDLING OF NEWLY CREATED HEADER FILES*.

ivfsoverlay
Ignore the Clang compiler option **−ivfsoverlay** and its argument. This is useful if you use Xcode, which uses a virtual file system (VFS) for things like combining Objective−C and Swift code.

locale
Ccache includes the environment variables **LANG**, **LC_ALL**, **LC_CTYPE** and **LC_MESSAGES** in the hash by default since they may affect localization of compiler warning messages. Set this sloppiness to tell ccache not to do that.

pch_defines
Be sloppy about **#define** directives when precompiling a header file. See *PRECOMPILED HEADERS* for more information.

modules
By default, ccache will not cache compilations if **−fmodules** is used since it cannot hash the state of compiler's internal representation of relevant modules. This sloppiness allows caching in such a case. See *C++ MODULES* for more information.

system_headers
By default, ccache will also include all system headers in the manifest. With this sloppiness set, ccache will only include system headers in the hash but not add the system header files to the list of include files.

time_macros
Ignore **__DATE__**, **__TIME__** and **__TIMESTAMP__** being present in the source code.

See the discussion under *TROUBLESHOOTING* for more information.

stats (**CCACHE_STATS** or **CCACHE_NOSTATS**, see *Boolean values* above)
If true, ccache will update the statistics counters on each compilation. The default is true.

stats_log (**CCACHE_STATSLOG**)
If set to a file path, ccache will write statistics counter updates to the specified file. This is useful for getting statistics for individual builds. To show a summary of the current stats log, use **ccache −−show−log−stats**.

Note
Lines in the stats log starting with a hash sign (**#**) are comments.

temporary_dir (**CCACHE_TEMPDIR**)
This option specifies where ccache will put temporary files. The default is **/run/user/<UID>/ccache−tmp** if **/run/user/<UID>** exists, otherwise **<cache_dir>/tmp**.

Note
In previous versions of ccache, **CCACHE_TEMPDIR** had to be on the same filesystem as the **CCACHE_DIR** path, but this requirement has been relaxed.

umask (**CCACHE_UMASK**)
This option (an octal integer) specifies the umask for files and directories in the cache directory. This is mostly useful when you wish to share your cache with other users.

SECONDARY STORAGE BACKENDS

The **secondary_storage** option lets you configure ccache to use one or several other storage backends in addition to the primary cache storage located in **cache_dir**. Note that cache statistics counters will still be kept in the primary cache directory — secondary storage backends only store cache results and manifests.

A secondary storage backend is specified with a URL, optionally followed by a pipe (|) and a pipe−separated list of attributes. An attribute is *key*=*value* or just *key* as a short form of *key*=**true**. Attribute values must be percent−encoded <https://en.wikipedia.org/wiki/Percent−encoding> if they contain percent, pipe or space characters.

Attributes for all backends

These optional attributes are available for all secondary storage backends:

- **read−only**: If **true**, only read from this backend, don't write. The default is **false**.
- **shards**: A comma−separated list of names for sharding (partitioning) the cache entries using Rendezvous hashing <https://en.wikipedia.org/wiki/Rendezvous_hashing>, typically to spread the cache over a server cluster. When set, the storage URL must contain an asterisk (*****), which will be replaced by one of the shard names to form a real URL. A shard name can optionally have an appended weight within parentheses to indicate how much of the key space should be associated with that shard. A shard with weight **w** will contain **w**/**S** of the cache, where **S** is the sum of all shard weights. A weight could for instance be set to represent the available memory for a memory cache on a specific server. The default weight is **1**.

  Examples:

  - **redis://cache−*.example.com|shards=a(3),b(1),c(1.5)** will put 55% (3/5.5) of the cache on **redis://cache−a.example.com**, 18% (1/5.5) on **redis://cache−b.example.com** and 27% (1.5/5.5) on **redis://cache−c.example.com**.
  - **http://example.com/*|shards=alpha,beta** will put 50% of the cache on **http://example.com/alpha** and 50% on **http://example.com/beta**.
- **share−hits**: If **true**, write hits for this backend to primary storage. The default is **true**.

Storage interaction

The table below describes the interaction between primary and secondary storage on cache hits and misses:

Primary storage	Secondary storage	What happens
miss	miss	Compile, write to primary, write to secondary[1]
miss	hit	Read from secondary, write to primary[2]
hit	−	Read from primary, don't write to secondary[3]

[1] Unless secondary storage has attribute **read−only=true**.
[2] Unless secondary storage has attribute **share−hits=false**.
[3] Unless primary storage is set to share its cache hits with the **reshare** option.

File storage backend

URL format: **file:DIRECTORY** or **file://DIRECTORY**

This backend stores data as separate files in a directory structure below **DIRECTORY** (an absolute path), similar (but not identical) to the primary cache storage. A typical use case for this backend would be sharing a cache on an NFS directory.

Important

ccache will not perform any cleanup of the storage — that has to be done by other means, for instance by running **ccache −−trim−dir** periodically.

Examples:

- **file:/shared/nfs/directory**
- **file:///shared/nfs/directory|umask=002|update−mtime=true**

Optional attributes:

- **layout**: How to store file under the cache directory. Available values:
  - **flat**: Store all files directly under the cache directory.
  - **subdirs**: Store files in 256 subdirectories of the cache directory.

  The default is **subdirs**.
- **umask**: This attribute (an octal integer) overrides the umask to use for files and directories in the cache directory.
- **update−mtime**: If **true**, update the modification time (mtime) of cache entries that are read. The default is **false**.

HTTP storage backend

URL format: **http://HOST[:PORT][/PATH]**

This backend stores data in an HTTP−compatible server. The required HTTP methods are **GET**, **PUT** and **DELETE**.

Important

ccache will not perform any cleanup of the storage — that has to be done by other means, for instance by running **ccache −−trim−dir** periodically.

Note

HTTPS is not supported.

Tip

See How to set up HTTP storage <https://ccache.dev/howto/http−storage.html> for hints on how to set up an HTTP server for use with ccache.

Examples:

- **http://localhost**
- **http://someusername:p4ssw0rd@example.com/cache/**
- **http://localhost:8080|layout=bazel|connect−timeout=50**

Optional attributes:

- **connect−timeout**: Timeout (in ms) for network connection. The default is 100.
- **keep−alive**: If **true**, keep the HTTP connection to the storage server open to avoid reconnects. The default is **false**.

  Note

  Connection keep−alive is disabled by default because with the current HTTP implementation uploads to the remote end might fail in case the server closes the connection due to a keep−alive timeout. If the general case with short compilation times should be accelerated or the server is configured with a long−enough timeout, then connection keep−alive could be enabled.

- **layout**: How to map key names to the path part of the URL. Available values:
  - **bazel**: Store values in a format compatible with the Bazel HTTP caching protocol. More specifically, the entries will be stored as 64 hex digits under the **/ac/** part of the cache.

    Note

    You may have to disable verification of action cache values in the server for this to work since ccache entries are not valid action result metadata values.
  - **flat**: Append the key directly to the path part of the URL (with a leading slash if needed).
  - **subdirs**: Append the first two characters of the key to the URL (with a leading slash if needed), followed by a slash and the rest of the key. This divides the entries into 256 buckets.

  The default is **subdirs**.
- **operation−timeout**: Timeout (in ms) for HTTP requests. The default is 10000.

Redis storage backend

URL format: **redis://[[USERNAME:]PASSWORD@]HOST[:PORT][/DBNUMBER]**

This backend stores data in a Redis <https://redis.io> (or Redis−compatible) server. There are implementations for both memory−based and disk−based storage. **PORT** defaults to **6379** and **DBNUMBER** defaults to **0**.

Note

ccache will not perform any cleanup of the Redis storage, but you can configure LRU eviction <https://redis.io/topics/lru−cache>.

Tip

See How to set up Redis <https://ccache.dev/howto/redis−storage.html> storage" for hints on setting up a Redis server for use with ccache.

Tip

You can set up a cluster of Redis servers using the **shards** attribute described in *SECONDARY STORAGE BACKENDS*.

Examples:

- **redis://localhost**
- **redis://p4ssw0rd@cache.example.com:6379/0|connect−timeout=50**

Optional attributes:

- **connect−timeout**: Timeout (in ms) for network connection. The default is 100.
- **operation−timeout**: Timeout (in ms) for Redis commands. The default is 10000.

CACHE SIZE MANAGEMENT

By default, ccache has a 5 GB limit on the total size of files in the cache and no limit on the number of files. You can set different limits using the command line options **−M**/**−−max−size** and **−F**/**−−max−files**. Use the **−s**/**−−show−stats** option to see the cache size and the currently configured limits (in addition to other various statistics).

Cleanup can be triggered in two different ways: automatic and manual.

Automatic cleanup

Ccache maintains counters for various statistics about the cache, including the size and number of all cached files. In order to improve performance and reduce issues with concurrent ccache invocations, there is one statistics file for each of the sixteen subdirectories in the cache.

After a new compilation result has been written to the cache, ccache will update the size and file number statistics for the subdirectory (one of sixteen) to which the result was written. Then, if the size counter for said subdirectory is greater than **max_size / 16** or the file number counter is greater than **max_files / 16**, automatic cleanup is triggered.

When automatic cleanup is triggered for a subdirectory in the cache, ccache will:

1. Count all files in the subdirectory and compute their aggregated size.
2. Remove files in LRU (least recently used) order until the size is at most **limit_multiple * max_size / 16** and the number of files is at most **limit_multiple * max_files / 16**, where **limit_multiple**, **max_size** and **max_files** are configuration options.
3. Set the size and file number counters to match the files that were kept.

The reason for removing more files than just those needed to not exceed the max limits is that a cleanup is a fairly slow operation, so it would not be a good idea to trigger it often, like after each cache miss.

Manual cleanup

You can run **ccache −c/−−cleanup** to force cleanup of the whole cache, i.e. all of the sixteen subdirectories. This will recalculate the statistics counters and make sure that the configuration options **max_size** and **max_files** are not exceeded. Note that **limit_multiple** is not taken into account for manual cleanup.

CACHE COMPRESSION

Ccache will by default compress all data it puts into the cache using the compression algorithm Zstandard <http://zstd.net> (zstd) using compression level 1. The algorithm is fast enough that there should be little reason to turn off compression to gain performance. One exception is if the cache is located on a compressed file system, in which case the compression performed by ccache of course is redundant. See the documentation for the configuration options **compression** and **compression_level** for more information.

You can use the command line option **−x/−−show−compression** to print information related to compression. Example:

```
Total data:           14.8 GB (16.0 GB disk blocks)
Compressed data:      11.3 GB (30.6% of original size)
  Original size:      36.9 GB
  Compression ratio: 3.267 x  (69.4% space savings)
Incompressible data:   3.5 GB
```

Notes:

- The “disk blocks” size is the cache size when taking disk block size into account. This value should match the “Cache size” value from “ccache −−show−stats”. The other size numbers refer to actual content sizes.
- “Compressed data” refers to result and manifest files stored in the cache.
- “Incompressible data” refers to files that are always stored uncompressed (triggered by enabling **file_clone** or **hard_link**) or unknown files (for instance files created by older ccache versions).
- The compression ratio is affected by **compression_level**.

The cache data can also be recompressed to another compression level (or made uncompressed) with the command line option **−X/−−recompress**. If you choose to disable compression by default or to use a low compression level, you can (re)compress newly cached data with a higher compression level after the build or at another time when there are more CPU cycles available, for instance every night. Full recompression potentially takes a lot of time, but only files that are currently compressed with a different level than the target level will be recompressed.

CACHE STATISTICS

ccache −−show−stats shows a summary of statistics, including cache size, cleanups (number of performed cleanups, either implicitly due to a cache size limit being reached or due to explicit **ccache −c** calls), overall hit rate, hit rate for direct/preprocessed modes and hit rate for primary and secondary storage.

The summary also includes counters called “Errors” and “Uncacheable”, which are sums of more detailed counters. To see those detailed counters, use the **−v**/**−−verbose** flag. The verbose mode can show the following counters:

HOW CCACHE WORKS

The basic idea is to detect when you are compiling exactly the same code a second time and reuse the previously produced output. The detection is done by hashing different kinds of information that should be unique for the compilation and then using the hash sum to identify the cached output. Ccache uses BLAKE3, a very fast cryptographic hash algorithm, for the hashing. On a cache hit, ccache is able to supply all of the correct compiler outputs (including all warnings, dependency file, etc) from the cache. Data stored in the cache is checksummed with XXH3, an extremely fast non−cryptographic algorithm, to detect corruption.

Ccache has two ways of gathering information used to look up results in the cache:

- the **preprocessor mode**, where ccache runs the preprocessor on the source code and hashes the result
- the **direct mode**, where ccache hashes the source code and include files directly

The direct mode is generally faster since running the preprocessor has some overhead.

If no previous result is detected (i.e., there is a cache miss) using the direct mode, ccache will fall back to the preprocessor mode unless the **depend mode** is enabled. In the depend mode, ccache never runs the preprocessor, not even on cache misses. Read more in *The depend mode* below.

Common hashed information

The following information is always included in the hash:

- the extension used by the compiler for a file with preprocessor output (normally **.i** for C code and **.ii** for C++ code)
- the compiler's size and modification time (or other compiler−specific information specified by **compiler_check**)
- the name of the compiler
- the current directory (if **hash_dir** is enabled)
- contents of files specified by **extra_files_to_hash** (if any)

The preprocessor mode

In the preprocessor mode, the hash is formed of the common information and:

- the preprocessor output from running the compiler with **−E**
- the command line options except those that affect include files (**−I**, **−include**, **−D**, etc; the theory is that these command line options will change the preprocessor output if they have any effect at all)
- any standard error output generated by the preprocessor

Based on the hash, the cached compilation result can be looked up directly in the cache.

The direct mode

In the direct mode, the hash is formed of the common information and:

- the input source file
- the compiler options

Based on the hash, a data structure called "manifest" is looked up in the cache. The manifest contains:

- references to cached compilation results (object file, dependency file, etc) that were produced by previous compilations that matched the hash
- paths to the include files that were read at the time the compilation results were stored in the cache
- hash sums of the include files at the time the compilation results were stored in the cache

The current contents of the include files are then hashed and compared to the information in the manifest. If there is a match, ccache knows the result of the compilation. If there is no match, ccache falls back to running the preprocessor. The output from the preprocessor is parsed to find the include files that were read. The paths and hash sums of those include files are then stored in the manifest along with information about the produced compilation result.

There is a catch with the direct mode: header files that were used by the compiler are recorded, but header files that were **not** used, but would have been used if they existed, are not. So, when ccache checks if a result can be taken from the cache, it currently can't check if the existence of a new header file should invalidate the result. In practice, the direct mode is safe to use in the absolute majority of cases.

The direct mode will be disabled if any of the following holds:

- **direct_mode** is false
- a modification time of one of the include files is too new (needed to avoid a race condition)
- a compiler option not supported by the direct mode is used:
  - a **−Wp,*** compiler option other than **−Wp,−MD,<path>**, **−Wp,−MMD,<path>** and **−Wp,−D<define>**
  - **−Xpreprocessor**
- the string **__TIME__** is present in the source code

The depend mode

If the depend mode is enabled, ccache will not use the preprocessor at all. The hash used to identify results in the cache will be based on the direct mode hash described above plus information about include files read from the dependency file generated by the compiler with **−MD** or **−MMD**.

Advantages:

- The ccache overhead of a cache miss will be much smaller.
- Not running the preprocessor at all can be good if compilation is performed remotely, for instance when using distcc or similar; ccache then won't make potentially costly preprocessor calls on the local machine.

Disadvantages:

- The cache hit rate will likely be lower since any change to compiler options or source code will make the hash different. Compare this with the default setup where ccache will fall back to the preprocessor mode, which is tolerant to some types of changes of compiler options and source code changes.
- If −MD is used, the manifest entries will include system header files as well, thus slowing down cache hits slightly, just as using −MD slows down make.
- If −MMD is used, the manifest entries will not include system header files, which means ccache will ignore changes in them.

The depend mode will be disabled if any of the following holds:

- **depend_mode** is false.
- **run_second_cpp** is false.
- The compiler is not generating dependencies using **−MD** or **−MMD**.

HANDLING OF NEWLY CREATED HEADER FILES

If modification time (mtime) or status change time (ctime) of one of the include files is the same second as the time compilation is being done, ccache disables the direct mode (or, in the case of a precompiled header, disables caching completely). This done as a safety measure to avoid a race condition (see below).

To be able to use a newly created header files in direct mode (or use a newly precompiled header), either:

- create the include file earlier in the build process, or
- set **sloppiness** to **include_file_ctime,include_file_mtime** if you are willing to take the risk, for instance if you know that your build system is robust enough not to trigger the race condition.

For reference, the race condition mentioned above consists of these events:

1. The preprocessor is run.
2. An include file is modified by someone.
3. The new include file is hashed by ccache.
4. The real compiler is run on the preprocessor's output, which contains data from the old header file.
5. The wrong object file is stored in the cache.

CACHE DEBUGGING

To find out what information ccache actually is hashing, you can enable the debug mode via the configuration option **debug** or by setting **CCACHE_DEBUG** in the environment. This can be useful if you are investigating why you don't get cache hits. Note that performance will be reduced slightly.

When the debug mode is enabled, ccache will create up to five additional files next to the object file:

Filename	**Description**
<objectfile>.ccache−input−c	Binary input hashed by both the direct mode and the preprocessor mode.
<objectfile>.ccache−input−d	Binary input only hashed by the direct mode.
<objectfile>.ccache−input−p	Binary input only hashed by the preprocessor mode.
<objectfile>.ccache−input−text	Human−readable combined diffable text version of the three files above.
<objectfile>.ccache−log	Log for this object file.

If **config_dir** (environment variable **CCACHE_DEBUGDIR**) is set, the files above will be written to that directory with full absolute paths instead of next to the object file.

In the direct mode, ccache uses the 160 bit BLAKE3 hash of the "ccache−input−c" + "ccache−input−d" data (where **+** means concatenation), while the "ccache−input−c" + "ccache−input−p" data is used in the preprocessor mode.

The "ccache−input−text" file is a combined text version of the three binary input files. It has three sections ("COMMON", "DIRECT MODE" and "PREPROCESSOR MODE"), which is turn contain annotations that say what kind of data comes next.

To debug why you don't get an expected cache hit for an object file, you can do something like this:

1. Build with debug mode enabled.
2. Save the **<objectfile>.ccache−*** files.
3. Build again with debug mode enabled.
4. Compare **<objectfile>.ccache−input−text** for the two builds. This together with the **<objectfile>.ccache−log** files should give you some clues about what is happening.

COMPILING IN DIFFERENT DIRECTORIES

Some information included in the hash that identifies a unique compilation can contain absolute paths:

- The preprocessed source code may contain absolute paths to include files if the compiler option **−g** is used or if absolute paths are given to **−I** and similar compiler options.
- Paths specified by compiler options (such as **−I**, **−MF**, etc) on the command line may be absolute.
- The source code file path may be absolute, and that path may substituted for **__FILE__** macros in the source code or included in warnings emitted to standard error by the preprocessor.

This means that if you compile the same code in different locations, you can’t share compilation results between the different build directories since you get cache misses because of the absolute build directory paths that are part of the hash.

Here’s what can be done to enable cache hits between different build directories:

- If you build with **−g** (or similar) to add debug information to the object file, you must either:
  - use the compiler option **−fdebug−prefix−map=<old>=<new>** for relocating debug info to a common prefix (e.g. **−fdebug−prefix−map=$PWD=.**); or
  - set **hash_dir = false**.
- If you use absolute paths anywhere on the command line (e.g. the source code file path or an argument to compiler options like **−I** and **−MF**), you must set **base_dir** to an absolute path to a “base directory”. Ccache will then rewrite absolute paths under that directory to relative before computing the hash.

PRECOMPILED HEADERS

Ccache has support for GCC’s precompiled headers. However, you have to do some things to make it work properly:

- You must set **sloppiness** to **pch_defines,time_macros**. The reason is that ccache can’t tell whether **__TIME__**, **__DATE__** or **__TIMESTAMP__** is used when using a precompiled header. Further, it can’t detect changes in **#define**s in the source code because of how preprocessing works in combination with precompiled headers.
- You may also want to include **include_file_mtime,include_file_ctime** in **sloppiness**. See *HANDLING OF NEWLY CREATED HEADER FILES*.
- You must either:
  - use the compiler option **−include** to include the precompiled header (i.e., don’t use **#include** in the source code to include the header; the filename itself must be sufficient to find the header, i.e. **−I** paths are not searched); or
  - (for the Clang compiler) use the compiler option **−include−pch** to include the PCH file generated from the precompiled header; or
  - (for the GCC compiler) add the compiler option **−fpch−preprocess** when compiling.

  If you don’t do this, either the non−precompiled version of the header file will be used (if available) or ccache will fall back to running the real compiler and increase the statistics counter “Preprocessing failed” (if the non−precompiled header file is not available).

C++ MODULES

Ccache has support for Clang’s **−fmodules** option. In practice ccache only additionally hashes **module.modulemap** files; it does not know how Clang handles its cached binary form of modules so those are ignored. This should not matter in practice: as long as everything else (including **module.modulemap** files) is the same the cached result should work. Still, you must set **sloppiness** to **modules** to allow caching.

You must use both **direct mode** and **depend mode**. When using the preprocessor mode Clang does not provide enough information to allow hashing of **module.modulemap** files.

SHARING A CACHE

A group of developers can increase the cache hit rate by sharing a cache directory. To share a cache without unpleasant side effects, the following conditions should to be met:

- Use the same cache directory.
- Make sure that the configuration option **hard_link** is false (which is the default).
- Make sure that all users are in the same group.
- Set the configuration option **umask** to **002**. This ensures that cached files are accessible to everyone in the group.
- Make sure that all users have write permission in the entire cache directory (and that you trust all users of the shared cache).
- Make sure that the setgid bit is set on all directories in the cache. This tells the filesystem to inherit group ownership for new directories. The following command might be useful for this:

  ```
  find $CCACHE_DIR −type d | xargs chmod g+s
  ```

The reason to avoid the hard link mode is that the hard links cause unwanted side effects, as all links to a cached file share the file’s modification timestamp. This results in false dependencies to be triggered by timestamp−based build systems whenever another user links to an existing file. Typically, users will see that their libraries and binaries are relinked without reason.

You may also want to make sure that a base directory is set appropriately, as discussed in a previous section.

SHARING A CACHE ON NFS

It is possible to put the cache directory on an NFS filesystem (or similar filesystems), but keep in mind that:

- Having the cache on NFS may slow down compilation. Make sure to do some benchmarking to see if it’s worth it.
- Ccache hasn’t been tested very thoroughly on NFS.

A tip is to set **temporary_dir** to a directory on the local host to avoid NFS traffic for temporary files.

It is recommended to use the same operating system version when using a shared cache. If operating system versions are different then system include files will likely be different and there will be few or no cache hits between the systems. One way of improving cache hit rate in that case is to set **sloppiness** to **system_headers** to ignore system headers.

An alternative to putting the main cache directory on NFS is to set up a secondary storage file cache.

USING CCACHE WITH OTHER COMPILER WRAPPERS

The recommended way of combining ccache with another compiler wrapper (such as “distcc”) is by letting ccache execute the compiler wrapper. This is accomplished by defining **prefix_command**, for example by setting the environment variable **CCACHE_PREFIX** to the name of the wrapper (e.g. **distcc**). Ccache will then prefix the command line with the specified command when running the compiler. To specify several prefix commands, set **prefix_command** to a colon−separated list of commands.

Unless you set **compiler_check** to a suitable command (see the description of that configuration option), it is not recommended to use the form **ccache anotherwrapper compiler args** as the compilation command. It’s also not recommended to use the masquerading technique for the other compiler wrapper. The reason is that by default, ccache will in both cases hash the mtime and size of the other wrapper instead of the real compiler, which means that:

- Compiler upgrades will not be detected properly.
- The cached results will not be shared between compilations with and without the other wrapper.

Another minor thing is that if **prefix_command** is used, ccache will not invoke the other wrapper when running the preprocessor, which increases performance. You can use **prefix_command_cpp** if you also want to invoke the other wrapper when doing preprocessing (normally by adding **−E**).

CAVEATS

- The direct mode fails to pick up new header files in some rare scenarios. See *The direct mode* above.

TROUBLESHOOTING

General

A general tip for getting information about what ccache is doing is to enable debug logging by setting the configuration option **debug** (or the environment variable **CCACHE_DEBUG**); see *CACHE DEBUGGING* for more information. Another way of keeping track of what is happening is to check the output of **ccache −s**.

Performance

Ccache has been written to perform well out of the box, but sometimes you may have to do some adjustments of how you use the compiler and ccache in order to improve performance.

Since ccache works best when I/O is fast, put the cache directory on a fast storage device if possible. Having lots of free memory so that files in the cache directory stay in the disk cache is also preferable.

A good way of monitoring how well ccache works is to run **ccache −s** before and after your build and then compare the statistics counters. Here are some common problems and what may be done to increase the hit rate:

- If the counter for preprocessed cache hits has been incremented instead of the one for direct cache hits, ccache has fallen back to preprocessor mode, which is generally slower. Some possible reasons are:
  - The source code has been modified in such a way that the preprocessor output is not affected.
  - Compiler arguments that are hashed in the direct mode but not in the preprocessor mode have changed (**−I**, **−include**, **−D**, etc) and they didn’t affect the preprocessor output.
  - The compiler option **−Xpreprocessor** or **−Wp,*** (except **−Wp,−MD,<path>**, **−Wp,−MMD,<path>**, and **−Wp,−D<define>**) is used.
  - This was the first compilation with a new value of the base directory.
  - A modification or status change time of one of the include files is too new (created the same second as the compilation is being done). See *HANDLING OF NEWLY CREATED HEADER FILES*.
  - The **__TIME__** preprocessor macro is (potentially) being used. Ccache turns off direct mode if **__TIME__** is present in the source code. This is done as a safety measure since the string indicates that a **__TIME__** macro *may* affect the output. (To be sure, ccache would have to run the preprocessor, but the sole point of the direct mode is to avoid that.) If you know that **__TIME__** isn’t used in practise, or don’t care if ccache produces objects where **__TIME__** is expanded to something in the past, you can set **sloppiness** to **time_macros**.

    - The **__DATE__** preprocessor macro is (potentially) being used and the date has changed. This is similar to how **__TIME__** is handled. If **__DATE__** is present in the source code, ccache hashes the current date in order to be able to produce the correct object file if the **__DATE__** macro affects the output. If you know that **__DATE__** isn't used in practise, or don't care if ccache produces objects where **__DATE__** is expanded to something in the past, you can set **sloppiness** to **time_macros**.
    - The **__TIMESTAMP__** preprocessor macro is (potentially) being used and the source file's modification time has changed. This is similar to how **__TIME__** is handled. If **__TIMESTAMP__** is present in the source code, ccache hashes the string representation of the source file's modification time in order to be able to produce the correct object file if the **__TIMESTAMP__** macro affects the output. If you know that **__TIMESTAMP__** isn't used in practise, or don't care if ccache produces objects where **__TIMESTAMP__** is expanded to something in the past, you can set **sloppiness** to **time_macros**.
    - The input file path has changed. Ccache includes the input file path in the direct mode hash to be able to take relative include files into account and to produce a correct object file if the source code includes a **__FILE__** macro.
- If a cache hit counter was not incremented even though the same code has been compiled and cached before, ccache has either detected that something has changed anyway or a cleanup has been performed (either explicitly or implicitly when a cache limit has been reached). Some perhaps unobvious things that may result in a cache miss are usage of **__TIME__**, **__DATE__** or **__TIMESTAMP__** macros, or use of automatically generated code that contains a timestamp, build counter or other volatile information.
- If "Multiple source files" has been incremented, it's an indication that the compiler has been invoked on several source code files at once. Ccache doesn't support that. Compile the source code files separately if possible.
- If "Unsupported compiler option" has been incremented, enable debug logging and check which compiler option was rejected.
- If "Preprocessing failed" has been incremented, one possible reason is that precompiled headers are being used. See *PRECOMPILED HEADERS* for how to remedy this.
- If "Could not use precompiled header" has been incremented, see *PRECOMPILED HEADERS*.
- If "Could not use modules" has been incremented, see *C++ MODULES*.

Corrupt object files

It should be noted that ccache is susceptible to general storage problems. If a bad object file sneaks into the cache for some reason, it will of course stay bad. Some possible reasons for erroneous object files are bad hardware (disk drive, disk controller, memory, etc), buggy drivers or file systems, a bad **prefix_command** or compiler wrapper. If this happens, the easiest way of fixing it is this:

1. Build so that the bad object file ends up in the build tree.
2. Remove the bad object file from the build tree.
3. Rebuild with **CCACHE_RECACHE** set.

An alternative is to clear the whole cache with **ccache −C** if you don't mind losing other cached results.

There are no reported issues about ccache producing broken object files reproducibly. That doesn't mean it can't happen, so if you find a repeatable case, please report it.

MORE INFORMATION

Credits, mailing list information, bug reporting instructions, source code, etc, can be found on ccache's web site: https://ccache.dev.

AUTHOR

Ccache was originally written by Andrew Tridgell and is currently developed and maintained by Joel Rosdahl. See AUTHORS.txt or AUTHORS.html and https://ccache.dev/credits.html for a list of contributors.