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. Supported languages are C, C++, Objective−C and Objective−C++.

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 ever discover an undocumented case where ccache changes the output of your compiler, please let us know.

Features

- Keeps statistics on hits/misses.
- Automatic cache size management.
- Can cache compilations that generate warnings.
- Easy installation.
- Low overhead.
- Optionally compresses files in the cache to reduce disk space.

Limitations

- Only knows how to cache the compilation of a single C/C++/Objective−C/Objective−C++ file. Other types of compilations (multi−file compilation, linking, etc) will silently fall back to running the real compiler.
- Only works with GCC and compilers that behave similar enough.
- Some compiler flags are not supported. If such a flag is detected, ccache will silently fall back to running the real compiler.

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 symlinks method, do something like this:

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

Do not use a hard link, use a symbolic link. A hard link will cause "interesting" problems.

OPTIONS

These 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.

−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.

−−dump−manifest=*PATH*
Dump manifest file at PATH in text format. This is only useful when debugging ccache and its behavior.

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

−−hash−file=*PATH*
Print the hash (in format **<MD4>−<size>**) of the file at PATH. This is only useful when debugging ccache and its behavior.

−h, −−help
Print an options summary page.

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

−M, −−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.

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

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

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

−s, −−show−stats
Print a summary of configuration and statistics counters in human−readable format.

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

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

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 configuration file settings, 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 priority of configuration settings is as follows (where 1 is highest):

1. Environment variables.
2. The cache−specific configuration file *<ccachedir>***/ccache.conf** (typically **$HOME/.ccache/ccache.conf**).
3. The system−wide 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 environment variable **CCACHE_CONFIGPATH** is set, ccache reads configuration from the specified path instead of the default paths.

Configuration file syntax

Configuration files are in a simple "key = value" format, one setting 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 settings 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 (rather than surprising the user): **0**, **false**, **disable** and **no**, and 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 settings

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

base_dir (**CCACHE_BASEDIR**)

This setting should be an absolute path to a directory. ccache then rewrites absolute paths into relative paths before computing the hash that identifies the compilation, but only for paths under the specified directory. If set to the empty string (which is the default), no rewriting is done. A typical path to use as the base directory is your home directory or another directory that is a parent of your build directories. Don't use **/** as the base directory since that will make ccache also rewrite paths to system header files, which doesn't gain anything.

See also the discussion under COMPILING IN DIFFERENT DIRECTORIES.

cache_dir (**CCACHE_DIR**)

This setting specifies where ccache will keep its cached compiler outputs. It will only take effect if set in the system−wide configuration file or as an environment variable. The default is **$HOME/.ccache**.

cache_dir_levels (**CCACHE_NLEVELS**)

This setting allows you to choose the number of directory levels in the cache directory. The default is 2. The minimum is 1 and the maximum is 8.

compiler (**CCACHE_COMPILER** or (deprecated) **CCACHE_CC**)

This setting 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 setting 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 the **mtime** setting, 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 the **none** setting if you know what you are doing.

string:value

Use **value** as the string to calculate hash from. This can be the compiler revision number you retrieved earlier and set here via environment variable.

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** setting if possible. See USING CCACHE WITH OTHER COMPILER WRAPPERS.

compression (**CCACHE_COMPRESS** or **CCACHE_NOCOMPRESS**, see Boolean values above)
If true, ccache will compress object files and other compiler output it puts in the cache. However, this setting has no effect on how files are retrieved from the cache; compressed and uncompressed results will still be usable regardless of this setting. The default is false.

compression_level (**CCACHE_COMPRESSLEVEL**)
This setting determines the level at which ccache will compress object files. It only has effect if **compression** is enabled. The value defaults to 6, and must be no lower than 1 (fastest, worst compression) and no higher than 9 (slowest, best compression).

cpp_extension (**CCACHE_EXTENSION**)
This setting 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 debugging for more information. The default is false.

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 setting 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.

hard_link (**CCACHE_HARDLINK** or **CCACHE_NOHARDLINK**, see Boolean values above)
If true, ccache will attempt to use hard links from the cache directory when creating the compiler output rather than using a file copy. Hard links are never made for compressed cache files. This means that you should not enable compression if you want to use hard links. The default is false.

Warning

Do not enable this option unless you are aware of the consequences. Using hard links may be slightly faster in some situations, but there are several pitfalls since the resulting object file will share i−node with the cached object file:

1. If the resulting object file is modified in any way, the cached object file will be modified as well. For instance, if you run **strip object.o** or echo >object.o, you will corrupt the cache.

2. Programs that rely on modification times (like “make”) can be confused since ccache updates the cached files' modification times as part of the automatic cache size management. This will affect object files in the build tree as well, which can retrigger the linking step 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 setting 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 setting 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.

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.

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.

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 object files, but it will not add new results to the cache. 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 = false** to 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.

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).

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 setting 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 options. Available options are:

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 option.

file_stat_matches
ccache normally examines a file's contents to determine whether it matches the cached version. With this option 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 option disables that check.

include_file_mtime
By default, ccache will not cache a file if it includes a header whose mtime is too new. This option disables that check.

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 option to tell ccache not to do that.

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

system_headers
By default, ccache will also include all system headers in the manifest. With this option 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__** and **__TIME__** 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.

temporary_dir (**CCACHE_TEMPDIR**)
This setting specifies where ccache will put temporary files. The default is **<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 setting specifies the umask for ccache and all child processes (such as the compiler). This is mostly useful when you wish to share your cache with other users. Note that this also affects the file permissions set on the object files created from your compilations.

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 **−M/−−max−size** and **−F/−−max−files** options. Use **ccache −s/−−show−stats** 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 settings.
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 **max_size** and **max_files** settings are not exceeded. Note that **limit_multiple** is not taken into account for manual cleanup.

CACHE COMPRESSION

ccache can optionally compress all files it puts into the cache using the compression library zlib. While this may involve a tiny performance slowdown, it increases the number of files that fit in the cache. You can turn on compression with the **compression** configuration setting and you can also tweak the compression level with **compression_level**.

CACHE STATISTICS

ccache −s/−−show−stats can show the following statistics:

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 MD4, a very fast cryptographic hash algorithm, for the hashing. (MD4 is nowadays too weak to be useful in cryptographic contexts, but it should be safe enough to be used to identify recompilations.) On a cache hit, ccache is able to supply all of the correct compiler outputs (including all warnings, dependency file, etc) from the cache.

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

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

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 the **compiler_check** setting)
- the name of the compiler
- the current directory (if the **hash_dir** setting is enabled)
- contents of files specified by the **extra_files_to_hash** setting (if any)

The direct mode

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

- the input source file
- the command line 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:

- the configuration setting **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,***X* 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 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 options that affect include files (**−I**, **−include**, **−D**, etc; the theory is that these 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 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:

- the configuration setting **depend_mode** is false
- the configuration setting **run_second_cpp** is false
- the compiler is not generating dependencies using **−MD** or **−MMD**

CACHE DEBUGGING

To find out what information ccache actually is hashing, you can enable the debug mode via the configuration setting **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.

In the direct mode, ccache uses the MD4 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 **−fdebug−prefix−map=***old***=***new* option 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 to 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__** or **__DATE__** 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 must either:
  - use the **−include** compiler option 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 **−include−pch** compiler option to include the PCH file generated from the precompiled header; or
  - (for the GCC compiler) add the **−fpch−preprocess** compiler option 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 "preprocessor error" (if the non−precompiled header file is not available).

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 setting **hard_link** is false (which is the default).
- Make sure that all users are in the same group.
- Set the configuration setting **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.

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 the configuration setting **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 the **prefix_command_cpp** configuration setting 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.
- When run via ccache, warning messages produced by GCC 4.9 and newer will only be colored when the environment variable **GCC_COLORS** is set. An alternative to setting **GCC_COLORS** is to pass **−fdiagnostics−color** explicitly when compiling (but then color codes will also be present when redirecting stderr to a file).
- If ccache guesses that the compiler may emit colored warnings, then a compilation with stderr referring to a TTY will be considered different from a compilation with a redirected stderr, thus not sharing cache entries. This happens for clang by default and for GCC when **GCC_COLORS** is set as mentioned above. If you want to share cache hits, you can pass **−f[no−]diagnostics−color** (GCC) or **−f[no−]color−diagnostics** (clang) explicitly when compiling (but then color codes will be either on or off for both the TTY and the redirected case).

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 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 "cache hit (preprocessed)" has been incremented instead of "cache hit (direct)", 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,***X* (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 setting.
  - A modification time of one of the include files is too new (created the same second as the compilation is being done). This check is made to avoid a race condition. To fix this, create the include file earlier in the build process, if possible, or set **sloppiness** to **include_file_ctime, include_file_mtime** if you are willing to take the risk. (The race condition consists of these events: the preprocessor is run; an include file is modified by someone; the new include file is hashed by ccache; the real compiler is run on the preprocessor's output, which contains data from the old header file; the wrong object file is stored in the cache.)
  - 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 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 "cache miss" has been 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__** or **__DATE__** 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 option was rejected.

- If "preprocessor error" has been incremented, one possible reason is that precompiled headers are being used. See PRECOMPILED HEADERS for how to remedy this.
- If "can't use precompiled header" has been incremented, see PRECOMPILED HEADERS.

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.