OULUN AMMATTIKORKEAKOULU

Toni Saario

ENABLING SCCACHE IN CI SYSTEM

ENABLING SCCACHE IN CI SYSTEM

Toni Saario
Bachelor's Thesis
Autumn 2020
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Toni Saario
Title of thesis: Enabling Sccache in CI System
Supervisor: Teemu Korpela
Term and year of completion: Fall 2020 Pages: 35

This thesis work was made in order to improve the performance of The Qt Company's CI system named Coin. The objective in this thesis was to enable the use of shared compiler cache (sccache) in the Coin. The thesis was commissioned by The Qt Company.

In this thesis the use of sccache was enabled in the Coin and the infrastructure required by the sccache was set up. The effectiveness of the sccache in Coin was evaluated by comparing before and after build times.

Technologies used in the thesis were evaluated at the start of the work. Sccache with a MinIO storage via S3 API was selected as the caching solution.

The work started by setting up the required infrastructure components. After the infrastructure was set up, sccache was provisioned into the virtual machine images. Then the use of the sccache was enabled in the build scripts and the data collection began. Lastly the performance of the sccache was evaluated from the Grafana statistics and the build logs were inspected.

While collecting statistics from the build caching the sccache was enabled for building the tests. The effect of caching the test building was also evaluated via Grafana. There was also an optimization made to the build agent's archiving process during the thesis based on findings when evaluating the sccache statistics. In the end these optimizations showed good results and gave insights where the next optimizations should be aimed at.

Keywords: Sccache, Continuous Integration, Complier cache

PREFACE

After having worked with the Coin for over a year the objective of the thesis increasing the performance of the Coin, was certainly interesting.

The work on the thesis started in January 2020. The requirements and decision of which technologies to use in the work were decided with The Qt Company when the thesis was about to start.

The work progressed from setting up the infrastructure and provisioning the virtual machines to practically finalize the work in around a month. However, the process of deploying the system into production had some blocking items. This delayed the deployment of the system until August 2020. After the deployment to the production, the collection of the statistics could begin.

While analyzing the statistics, there was an interesting point regarding the artifact transfer between the agent and Coin being extremely slow. This was also optimized during the thesis and its effects were evaluated at the same time as the building of tests was cached.

In Oulu 11.7.2020

Toni Saario

CONTENTS

1 INTRODUCTION

The objective of this thesis was to improve Coin's performance by enabling the use of sccache in the Coin. The objective of this thesis work was commissioned by The Qt Company.

The load on the Coin has been increasing over the time by an increasing amount of contributions, branching of different Qt versions and by new platforms being introduced. While today's CI systems are a very powerful tool, their turnover times should also be reasonable. In the Qt's case, which is a large multi component product, the compilation process is long, up to 6 hours in worst cases. Qtbase, which is the main module of Qt, takes around 2 hours to integrate. This is what this thesis aims to improve.

Coin uses full sized virtual machines and they are deleted after running one job. This makes plain ccache, such as solutions, obsolete. There are multiple other CI systems, such as GitLab and Travis CI, using different kinds of caching solutions, which were evaluated during the thesis (1, 2.).

Sccache was selected for the Coin (3). This decision was based on the amount of branching, the number of similar platforms which produce same objects and the requirement for a dynamic environment where any number of any OS can be allocated at any time (4). Sccache also works on MSVC where ccache does not have support currently. This allows using the same caching tool on all platforms (5).

During the thesis, use of sccache was enabled in the Coin and the required infrastructure was set up. While collecting and analyzing the data from build times, the artifact transfer between the build agent and Coin was also optimized.

In the end the effect of sccache on build times and other optimizations were evaluated. All of them showed positive effects on build times.

2 CI ARCHITECHTURE

At the core of the Coin, a main service handles creating integrations and virtual machines, sending sources, dependencies and instructions to the virtual machines. The figure 1 illustrates the relationship between the infrastructure components (6).

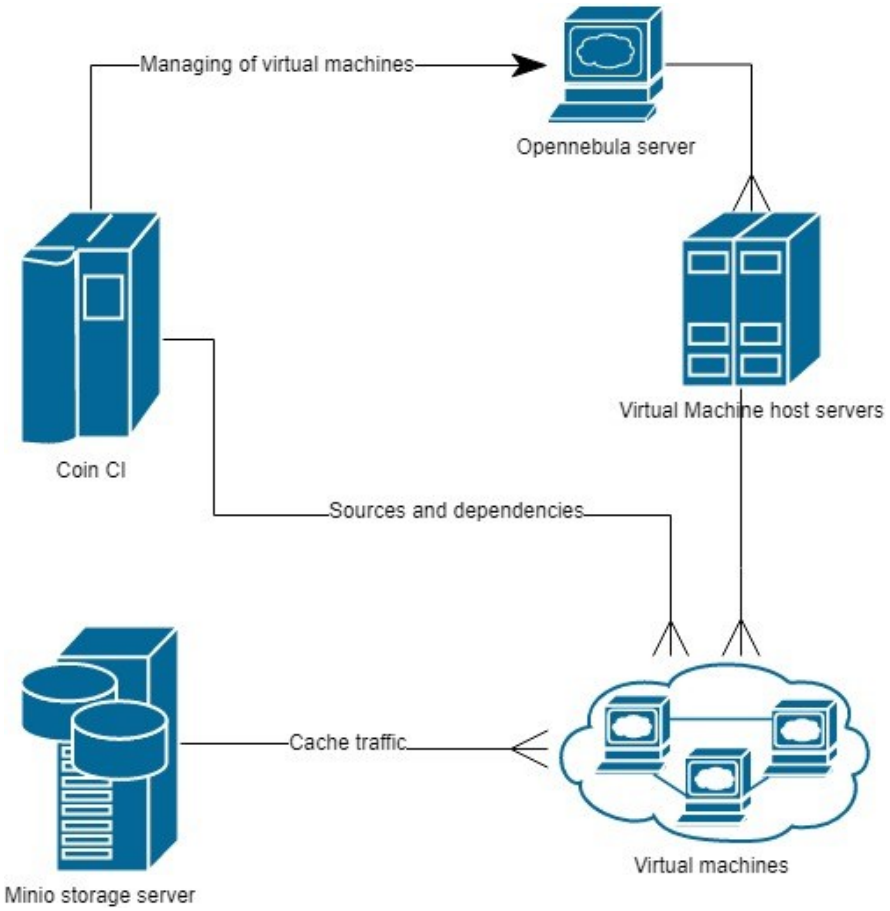

FIGURE 1. CI infrastructure and relationships (6)

An OpenNebula server is used between Coin and the virtual machine host servers to provide an interface to manage the lifecycle of the virtual machines (7).

A MinIO storage server was set up during the thesis to serve as a storage for the build objects stored by sccache. The MinIO storage is only accessed by the build machines (8).

2.1 OpenNebula

OpenNebula is a virtualization platform. It holds the virtual machine templates and provides an interface for managing the lifecycle of virtual machines. The communication between Coin and the OpenNebula server happens via XML-RPC. (7.)

Coin creates unique virtual machine templates for every project and branch combination out of provisioning scripts in the repositories. Provisioning of the sccache was added into these provisioning scripts. (9.)

A new template is created by booting a clean operating system on a virtual machine and running the provisioning scripts on the virtual machine. After all the scripts are run the virtual machine is shut down and it is saved as a virtual machine template.

2.2 Gerrit monitor

Coin uses a special component called a Gerrit monitor to monitor a specialized instance of Gerrit for staging events. This system enables the use of Coin from the Gerrit via a stage button. The Gerrit integration with Coin is visualized in the figure 2. (10.)

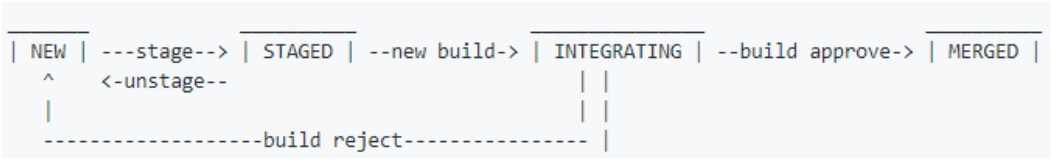

FIGURE 2: Gerrit plugin workflow (10.)

When a user presses the stage button in Gerrit, the change moves to a staged state and it is cherry-picked to a staging branch. Coin receives information from this and knows that there is something staged for a certain branch in a repository. After a minute from the last change in the staging branch, Coin calls Gerrit to move changes in the staging branch into an integrating state. The move to integrating state also causes Gerrit to create a build branch by taking a clone of the staging branch at that moment. This build branch is then run through Coin. If the integration passes, a build branch is merged to the main branch. If the integration fails, changes are moved back to the new state to wait for fixing or re-staging. (10.)

2.3 Coin

Coin's main process handles creation of the integration tasks, spawning of virtual machines via OpenNebula and supplying sources and dependencies to the build agents.

After spawning a virtual machine, Coin waits for a contact from the agent that auto starts on the virtual machines. When receiving a contact from the agent, Coin sends instructions to the agent to run the specific job in question. In the end of the job the agent reports back to Coin of the result of the job and uploads artifacts if any were produced.

Coin can create an integration which can have any number of patches in one run. This means that users can stage multiple commits and all of them get integrated at once. This allows multiple commits to be tested in a batch instead of testing each commit individually. This mimics what it would be to try a branch/pull request merge in a GitHub style workflow where branches and pull requests are used instead of single commits.

Normally, an integration tests around 1 to 10 commits at a time but sometimes there can be up to 50 commits. The amount of commits in an integration depends on how many commits were staged when the last integration was ongoing, as there can be only one integration ongoing per branch. This obviously has the downside that if one of the patches in the integration does not work, all of them will be rejected, the upside being that it has much less load on the CI.

Currently, the dev branch of qtbase integrates in around 2 hours and it has an integration ongoing almost 24/7. This is a place where sccache could help to speed up the builds. To put it in perspective, if the integration were 20% faster, it means 20% more integrations to the repository. The more integrations the repository can do the less commits need to be tested at once. This reduces the amount of false negative results for patches which are alright but were integrated along with a patch which does not work. It is also a quality of life improvement for the developers when they get the feedback faster from the CI.

There was discussion ongoing during the thesis to enable some level of parallelization on the integrations, which will be implemented later. With the parallelization the build time

reduction means that less integrations are running parallel at a time or that more parallelization can be achieved. With a shorter integration time the head of the branch is more up to date. This reduces the risk that parallel integrations break the head of the branch when both integrations pass separately but would fail together. Also, the risk of a merge conflict when a build branch is merged to the main branch is reduced.

Coin is special in the way it handles dependencies compared to other CI systems. Coin allows defining binary/cross-module dependencies by defining the repository and a ref for each dependency. These dependencies are built during the integration and artifacts from them are cached. (4.)

With this it is possible to build the Qt5 super-module only up to the module being integrated with artifacts from the dependencies always available. This enables fast integrations to any module in the qt 6 dependency tree in figure 3. Installing binaries from the dependency jobs also creates a highly modular environment where any module can easily be tested against different revisions of the dependencies.

The Qt 6 alpha dependency graph in figure 3 illustrates the modularity of Qt. Each of the modules can be integrated on its own and the dependencies update at will. For example, integrating qt3d also requires building qtshadertools, qtbase, qtdeclarative and qtsvg. These dependencies are cached if the integration is not a dependency update and a qt3d build can start immediately.

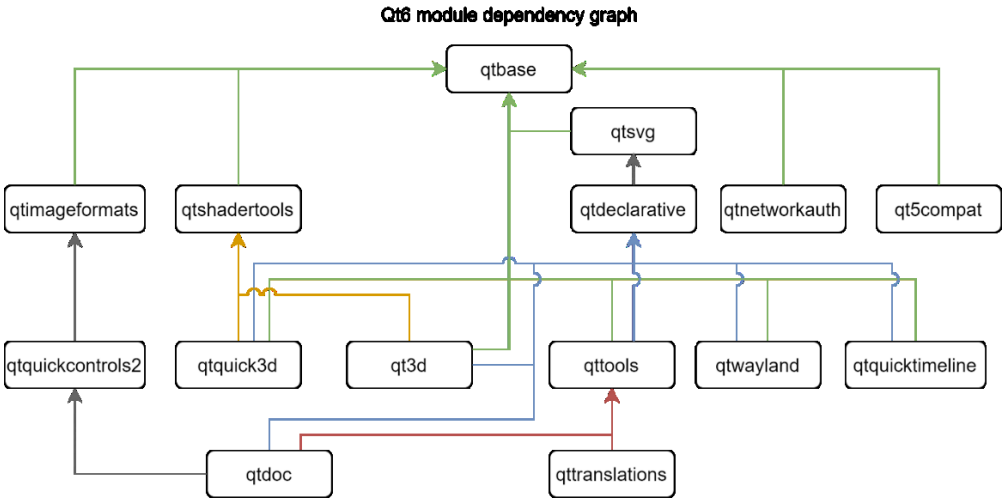

FIGURE 3. Dependency graph of Qt 6 alpha.

Figure 3 only has modules being built in the Qt 6 alpha at the time of the thesis. A similar graph for the Qt 5.15 would have over twice as much modules.

2.4 Storage

A storage server was required for storing objects cached by the sccache. A MinIO storage service was set up on a separate server to act as the storage for objects. Configuring the MinIO server required initializing a storage bucket for the cache, which is technically only a folder on the disk. This was done with a MinIO make bucket command (8.).

The bucket also required configuring read write permissions before usage. Read/write access was given to all users with authentication. The permissions were configured using the “policy” command.

The storage access was configured with a key and password configuration. This might not be necessary in this case as the storage is in a private network and the stored information is not sensitive. It could just as well have read access to everyone to let developers use the cached objects produced by Coin.

The MinIO server and the sccache client communicate though an S3 API. The access to the storage on build machines was configured via an S3 credentials file. Credentials were written into the AWS default credentials file “~/.aws/credentials” on Unix and “%UserProfile%/.aws/credentials” on Windows (11.)

Initially, a 500GB of storage was allocated for the MinIO and a retention policy of 1 week was configured to the bucket. The retention policy works via a separate script timed with systemd timer to periodically sweep the bucket and delete objects older than a week.

A future implementation detail will be to create a more intelligent garbage collector based on how much the objects are requested and to keep the objects that are still used even if they are old.

3 SCCACHE

Sccache is a compilation caching tool developed by Mozilla. Sccache itself is only a compiler wrapper which uses a remote or local storage to cache the build objects. Sccache works so that it is prefixed into the complier call command. Sccache uses a preprocessor of the complier in use to generate a hash for the object files. This hash is then used to store the object file generated by the compile in the storage. (3.)

On some compilers this causes more overhead than on others. For example, Windows compilers MinGW and MSVC have a relatively slow preprocessor, where GCC's preprocessor work faster. This becomes visible later while analyzing build times.

Prebuilt sccache binaries are available, but for this thesis binaries were compiled from the latest sccache sources due to bugs in the latest release at that time and Windows release being unavailable.

Compiling a portable sccache binary required static OpenSSL binaries. Linux also required using musl libc to generate binaries which work across multiple Linux distributions (12). These sccache binaries are then installed when running the provisioning scripts and creating the virtual machine templates. (3.)

The provisioning script extracts the prebuilt sccache binary on the virtual machine and extends a PATH environment variable to include path to it. Provisioning also partly configures the sccache. A SCCACHE_IDLE_TIMEOUT environment variable was set to value "0" during provisioning to prevent the sccache service shutting down automatically. (9.)

As a part of the agent initialization on the virtual machines, sccache is started as a background service. On the agent side sccache requires configuring a storage endpoint along with possible authentication to the endpoint. Configuring the endpoint was done in the connection handshake between the agent and Coin. Credentials for the MinIO storage were also transported to the agent in this handshake. At the end of each job sccache statistics are printed by calling sccache with a parameter "-s".

One interesting use case that the sccache enables is to allow developers to access the stored objects (13). This in theory allows developers to reproduce a build failure or similar issue quickly, which happens in the CI as all objects until that point are cached. This is inconvenient with sccache since it requires that the absolute paths are same on the build machines to produce a cache hit (3). However, currently Coin gives developers access to the virtual machine at the failed state immediately after a failure, reducing the need for such implementation.

4 BUILD SYSTEMS

Two build systems were in use during the thesis. These were qmake and CMake. During the thesis, a default build system was being changed from qmake to CMake in Qt 6. Both build systems are open source. However, CMake is more popular among the community.

CMake is maintained by the community with some contribution from Qt. It is in general more powerful and flexible and can build a much wider variety of projects than the qmake. (14.)

Qmake comes with Qt and is pretty much used only to build Qt projects. Qmake makes developing cross-platform applications with Qt easy. It has limited use outside of Qt projects. (15.)

Due to the community support and contribution and the popularity of the CMake, it was chosen to succeed qmake as a default build system in Qt 6.

The implementation of a caching solution to any build system is similar in effort if the build system supports explicitly setting a compiler prefix by any means. If the build system were to change from CMake to some other system later, the framework created for sccache in this thesis does not need any changes.

If the build system does not support setting the compiler prefix by any means, sccache can still be used. This happens by creating a symbolic link from the sccache binary with the name of the used compiler, for example, “ln -s sccache g++” on Unix. Sccache realizes that it was called via a symbolic link and acts like it were called like “sccache g++”. The same is functionality is supported by ccache. (3, 16.)

4.1.1 Qmake

Enabling sccache on qmake targets can be done via a qmake pro file configuration which prepends the sccache into the complier commands. However, sccache was not enabled on qmake targets during the thesis, it and serves a as future target after evalu-

ating if it makes sense to enable them as qmake is being replaced by CMake. Configurations to enable sccache on qmake are
QMAKE_CC=’sccache <compiler>’

QMAKE_CXX=’sccache <compiler>’

This implementation on qmake is suboptimal with Qt as it does not work on MacOS due to qmake not respecting the configuration on MacOS. MacOS targets require creating a configuration argument to enable the use of sccache in qtbase. (17.)

4.1.2 CMake

Sccache on CMake was enabled via command line arguments to the CMake command. The required command line arguments for CMake were
-DCMAKE_C_COMPILER_LAUNCHER=sccache

-DCMAKE_CXX_COMPILER_LAUNCHER=sccache

These arguments were added to the platform configuration files in the Qt5 super repository, where the platform configurations are read from for all Qt modules. With these arguments CMake prefixes compile commands with “sccache”. This is equivalent to the sccache being called with the original compile command as an argument. (3.)

Same configuration arguments are used when building the module and the tests. Passing the arguments for tests, required configuring the build scripts to append new arguments to the tests CMake call. (18.)

5 RESULTS

A method of measuring the effectiveness of the sccache was in place. This was done by reporting build, test and integration times to an InfluxDB database and using Grafana to visualize the data. (19.)

Coin can reduce the allocated core count and RAM of the virtual machine to half if the CI is under a high load. This behavior explains the individual virtual machines which can take nearly double the time to complete compared to the average. Targets which run tests build them in the build phase. This lengthens the build time and shortens the test phase. Building the tests in the build phase is done to allow rerunning the tests without rebuilding them in case the test run fails to a flaky test.

5.1 Sccache vs ccache

The work in this thesis does not touch ccache and the results provided from Coin are between no cache at all and sccache. At the time one experiment was available as a reference for a comparison to ccache.

An experiment done by Mike Hommey compares ccache to sccache while building Mozilla Firefox. In the experiment Mike had static hosts. This means that the build machine stays same and different revisions of the project are being built on it. In the experiment build times on sccache were 1-10 minutes or 10-50% slower than static ccache, illustrated by figure 4. (20.)

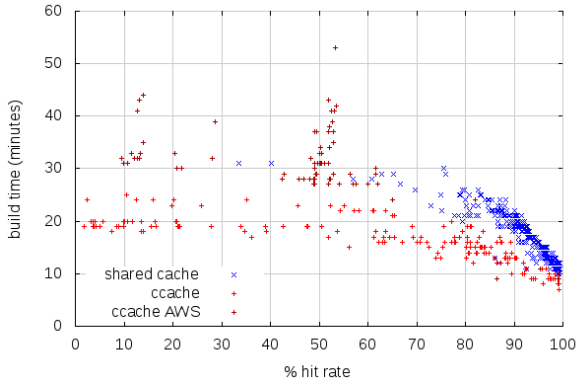

FIGURE 4. Ccache vs Sccache build time to hit ratio. (20.)

This overhead is caused by sccache calculating the dependency hash for the objects and uploading and downloading the objects instead of using a local disk as a storage like ccache does. In the experiment the latency to the storage server was 3-80ms. This explains the variation in the amount of overhead. (21.)

Despite the substantial overhead introduced by the sccache, the average build times in the experiment were 14:20 for sccache and 15:27 for ccache. In other word the sccache showed 7.8% faster builds over ccache on average. This difference comes from the fact that the sccache uses the same storage on all build hosts whereas each ccache build hosts have their own storage. Ccache hosts can have cold cache if the previous build was on another host or the host has not done any building previously. On sccache all built objects are on the same storage and no such issues are present. (20.)

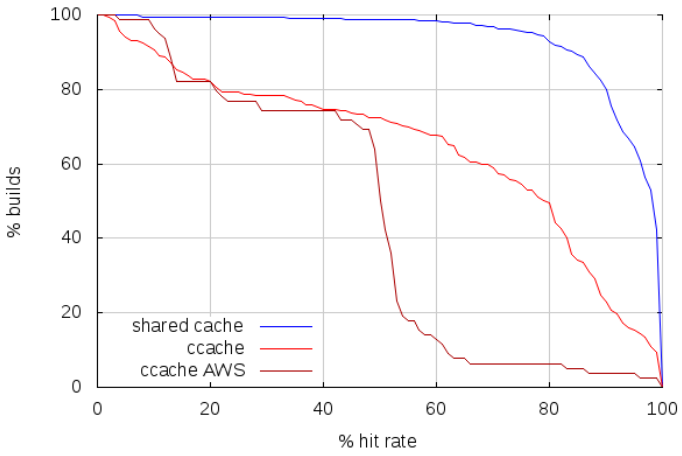

FIGURE 5. Cache hit rate comparison between sccache and ccache (20.)

Figure 5 shows that in a static host environment the sccache hit rate is far superior to the ccache. This is very much what can be expected out of a scenario where ccache is used and the cache is stored as artifact, only that transferring the artifact causes some additional overhead.

One advantage of the ccache is that it can have less overhead with the direct mode which does not invoke a pre-processor, but instead calculates the hash of the object itself. This manifests as a benefit mostly on compilers with slow preprocessors, e.g. MinGW and

MSVC. However, in case of building Qt, the “-MD” flag is used which is not supported by direct mode. (22.)

Ccache has another mode which relies on the “-MD” flag for dependencies. With this it should be possible to avoid the preprocessor calls also in Qt’s case. The depend mode is relatively simple in the hash calculation and results in a cache miss in some insignificant changes. In depend mode the “-MD” will include system header files in the hash calculation slowing down the hash calculation in theory when compared to the direct mode (21). This is not necessarily always the case. Rosen Matev e.g. has tested that the depend mode performed equally to the direct mode even in a larger C++ project (23.).

5.2 Infrastructure statistics

The infrastructure used by Coin and the build agents are physically closely packed. This results in a high bandwidth and a low latency across the components. The latency between build agent and the storage was 200-400µs. This significantly reduces the network overhead compared to what the Mike Hommey’s experiment had with building Mozilla Firefox.

The network overhead could possibly be considered irrelevant as the latency to the virtual machine’s disk is in the same range.

One consideration in the planning phase was that if one MinIO server can handle the I/O from the agents. The current infrastructure capacity allows around 250 parallel build virtual machines. The MinIO storage server was set up with 40 cores, 400GB of RAM and 10Gbps network link, which turned out as an overkill. The CPU usage on the MinIO storage server was extremely low. It was not possible to conclude if the CPU usage ever raised even to 1%. The maximum traffic of around 1Gbps was observed when around 100 connections were open to the storage, meaning that 100 build machines were using the storage at that time. Calculating from this around 1,000 agents would saturate the network connection, which is not possible with the current amount of hardware.

The one MinIO server now can easily handle the current traffic from all the virtual machines. If the storage I/O or the single 10Gbps network link were to become the bottleneck, it would be easy to implement multiple MinIO storage servers and allocate certain repositories to each of them.

5.3 Builds

In Coin the sccache was enabled for most of the CMake targets. Data from these targets is evaluated for a period of a week for both before and after enabling the sccache. During the collection of data for average and median times, the test caching was not enabled.

5.3.1 Overall statistics

The sccache was enabled on 8 out of 9 CMake targets. These include Ubuntu, OpenSUSE and RHEL Linux distributions compiled with GCC, an RHEL cross compile with a clang to android-arm64, an MacOS 10.14 host build and cross compile to IOS with a clang and a Windows compiled with MinGW.

Only the Windows compiled with MSVC was not enabled due to separate debug files produced in the compilation. Sccache currently is unable to cache these. Most likely fixing this is changing Qt's MSVC build configuration to use the "-Z7" debug flag instead of "-Zi" currently used. This includes the debug info into the objects produced instead of being produced as separate files. Some research was in progress during the thesis to allow the use of "-Z7" flag instead of "-Zi". Tests of the caching of the build and test building on the MSVC showed that the build time reduced from 45 minutes to 20 minutes. After the change to allow the use of the "-Z7" debug flag is merged, all CMake targets can be cached. (24.)

Looking at overall build times of the qtbase in the figure 6, a minimal reduction in build times can be seen when the sccache was enabled on 08/19. Figure 6 does not tell much due to the number of different platforms stacked on top of each other. One reason why we do not see a greater impact here is that the integrations have the exact same target ran on qmake which is included in this chart. Ubuntu 20.04, RHEL 7.6 and MacOS 10.14

have these qmake targets included in their data. With this we are most interested in the data from the other targets which contain only relevant data of the effect of the sccache.

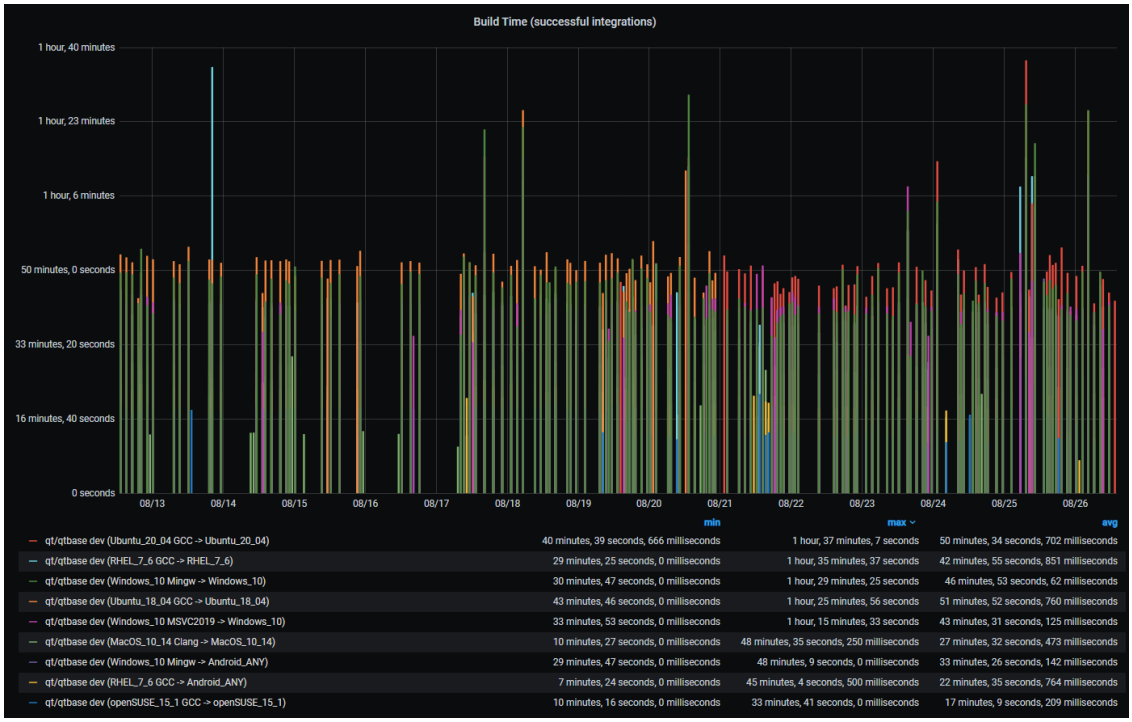

FIGURE 6. Overall view of integration times

Figure 7 shows the same time as in the previous figure 6 but only includes OpenSUSE. Now, the time when sccache was enabled is clearly visible. The OpenSUSE target does not build tests during the build and it does not have a qmake counterpart. This makes it excellent for comparison because it has least other variables than the build.

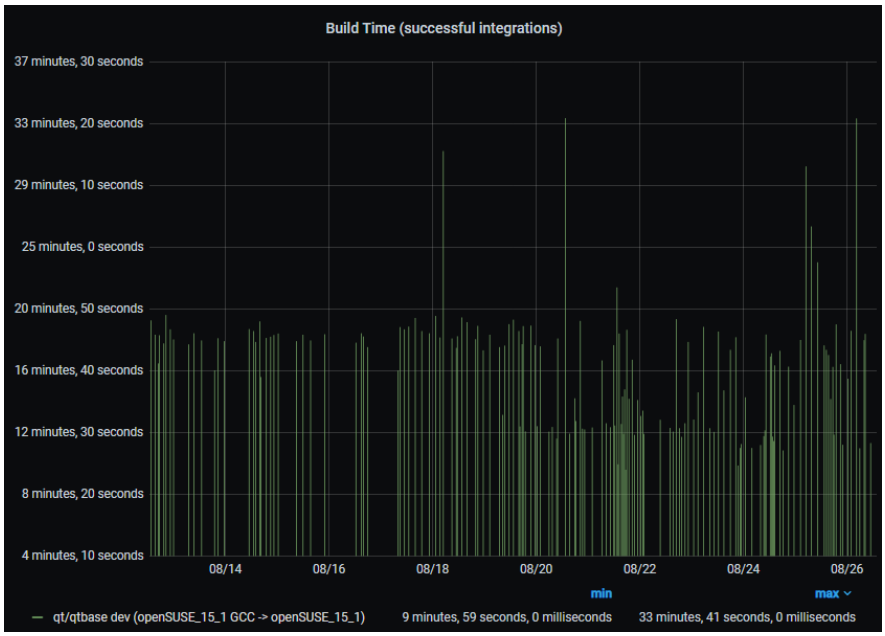

FIGURE 7. OpenSUSE build time statistics

Builds were few minutes slower built with sccache and a 0% cache hit rate than not using sccache at all, in worst cases the overhead from sccache was 10%. Most of the overhead is caused by sccache invoking compiler's preprocessor for a hash calculation. In these charts this is not visible as it is very rare that a 0% cache hit rate build happens.

In the preprocessor overhead can also be seen in that the SLES on GCC builds in ~35 minutes without sccache and when cached in 2.5 minutes. On Windows with MinGW and without sccache the build time is around the same ~35 minutes. When cached, it only reduces to 12 minutes due to the preprocessor overhead being larger on MinGW.

The rest of the overhead comes from the fact that sccache needs to communicate over a network to store the objects.

5.3.2 Average build times

Plotting same data from the figure 7 on top of each other in figure 8 illustrates the difference more clearly. The average build time was reduced from 18:52 to 15:05, an improvement of 20%.

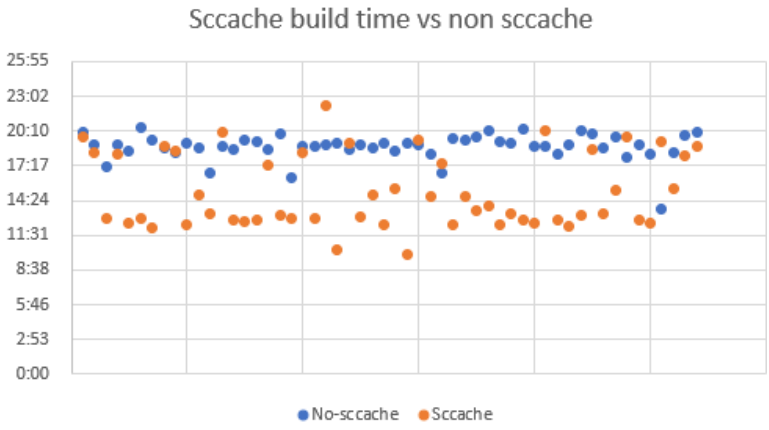

FIGURE 8. OpenSUSE sccache vs non-sccache build times.

From the figure 8 it is possible to conclude that sccache can reduce the build time from around 19 minutes down to around 12 minutes with a 100% cache hit ratio. This would mean that in the best case sccache can have an effect of 37% on the total build job's time.

Looking next at the overview of most built targets and modules in figure 9 it is visible that the overall effect across the board is positive when comparing average build times 2 weeks before and after enabling the sccache. It is also visible in this chart that RHEL, MacOS and Ubuntu targets contain data from the qmake builds and have much less effect on either direction.

2 week average build time	qtbase	declarative	svg	quickcontrols2	qt3d	qtdoc	qtimageformats	qtnetworkauth	qtwayland
OpenSUSE	-2m52s (-15.1%)	-4m11s (-16.4%)	-20s (-13.5%)	-2m42s (-22.5%)	-7m55s (-24.1%)	+37s (+11%)	No Data	No Data	+31s (+19.1%)
RHEL	+22s (+ 0.8%)	-1m18s (-6.3%)	-8s (-8.8%)	+19s (+2.7%)	-2m36s (-11.2%)	No Data	-12s (-12.8%)	-11s (-12.2%)	-13s (-1.4%)
RHEL-Android	-4m38s (-19.7%)	-2m9s (-9.8%)	-37s (-29.8%)	+11s (+1.5%)	-5m40s (-15.1%)	-20s (-24.4%)	-30s (-29.7%)	No Data	-1m15s (-42.6%)
MacOS	+18s (+1.1%)	-13s (-1.4%)	-11s (-11%)	+38s (+9%)	-1m13s (-7.7%)	-13s (-18.8%)	-8s (-8.7%)	-5s (-6.7%)	No Data
MacOS-IOS	-3m3s (-12.2%)	-1m31s (-6%)	-21s (-14%)	+1m2s (+7.7%)	+11s (+0.6%)	-24s (-16.4%)	-24s (-17.5%)	-41s (-32.5%)	No Data
Windows-minGW	-1m36s (-3.3%)	-6m46s (-34.2%)	-33s (-20.1%)	-1m (-6%)	-3m49s (-20.8%)	-3s (-2.3%)	-25s (-16.2%)	-39s (-30.1%)	No Data
Ubuntu	-2m5s (-4%)	-1m8s (-3.1%)	-34s (-19.8%)	+55s (+6.5%)	-50s (-2.1%)	-5s (-3.9%)	-15s (-11.1%)	No Data	+46s (+4.2%)

FIGURE 9. Relative overview of 2 weeks before and after enabling sccache

Some targets which have longer a build time in figure 9 have such short build time that the difference in the actual build time fades under other variables. For example, qtsvg,

qtdoc, qtimageformats, qtnetworkauth and qtwayland all build in less than 1 minute on average.

Figure 9 should be taken with a note of caution as Coin can generate virtual machines which have half the cores and memory if the system is under a high load. This behavior increases average build times. These lower capacity virtual machines are the spikes seen mostly after sccache was enabled in the figure 7. Some of the modules in figure 9 also have a short virtual machine lifespan and a low number of datapoints, which creates somewhat unreliable data.

5.3.3 Median build times

Median build times in the figure 10 show different perspective from the average values.

2 week median	qtbase	declarative	quickcontrols2	qt3d
OpenSUSE	-4m15s (-22.2%)	-5m3s (-20.1%)	-2m58s (-26.2%)	-9m57s (-35.2%)
Windows-minGW	-4m44s (-9.8%)	-6m45s (-24.3%)	-3m18s (-19.5%)	-2m38s (-10.1%)

FIGURE 10. Median build times 2 weeks before and after enabling sccache

Figure 10 only contains modules which have a longer build time and enough datapoints to produce a reliable median value. These targets have at least 50 datapoints across the total of 4 weeks and over 10-minute build time. The median value is not affected by the few virtual machines which have been created with a lower capacity and take nearly twice as long to build. This mitigates the variance caused by different sized virtual machines and shows a more positive effect.

5.3.4 Build job overview

Taking a closer look at what happens during a build job on the agent in the figure 11, the effect of sccache is much more noticeable.

OpenSUSE build Overview		
	No-sccache	Sccache
Agent initialization	7s	7s
Configure	39s	40s
Cmake build	10m 45s	3m 32s
Install	11s	16s
Export artifacts	7m 29s	9m 32s
total:	19min 4s	14m 36s

FIGURE 11. OpenSUSE build job overview.

Figure 11 consists of two builds one with sccache and another one without it. The cache hit rate on the sccache build was 99%. In this case sccache reduced the actual build time by 68%. This is around twice what can be seen from outside of the job.

Exporting the artifacts should have no effect from the sccache as it is a single upload to a remote storage after the build is done. The difference seen in the upload time is a normal variance based on the load on the CI. From this it can be concluded that the sccache can reduce the actual time build spent building down to one third of the original on the OpenSUSE target.

Other good information that this chart provides is that the artifact upload can take up to 75% of the total build job's time. This suggests that the upload process could most likely be optimized. Currently, the upload process happens so that the build agent opens an HTTP connection to the storage server and archives the artifacts on the fly while uploading via the HTTP stream.

5.4 Cache hit rates

Comparing the cache hit rates was somewhat complicated. Failed integrations also generated objects to the storage but did not produce any numbers what their hit rate was or what they produced. With this it is possible to have successful integrations with nearly a hit rate of 100% without knowing where the objects came from. For example, 2 successful integrations back to back can have a cache hit rate of 90% even if none of the objects are same between them, if a failed integration generated the objects in between.

However, there are almost always different changes being tested in bunch and exact same objects are rarely used in the builds. Coin also caches the artifacts if a build was done in such case.

Cache hit rate values were collected from the 50 last successful integrations into the dev branch of the qtbase. This is not the same data that the previous average and median values were calculated from. Figure 12 shows the scaling between the cache hit rate and the build time, both charts have 50 datapoints.

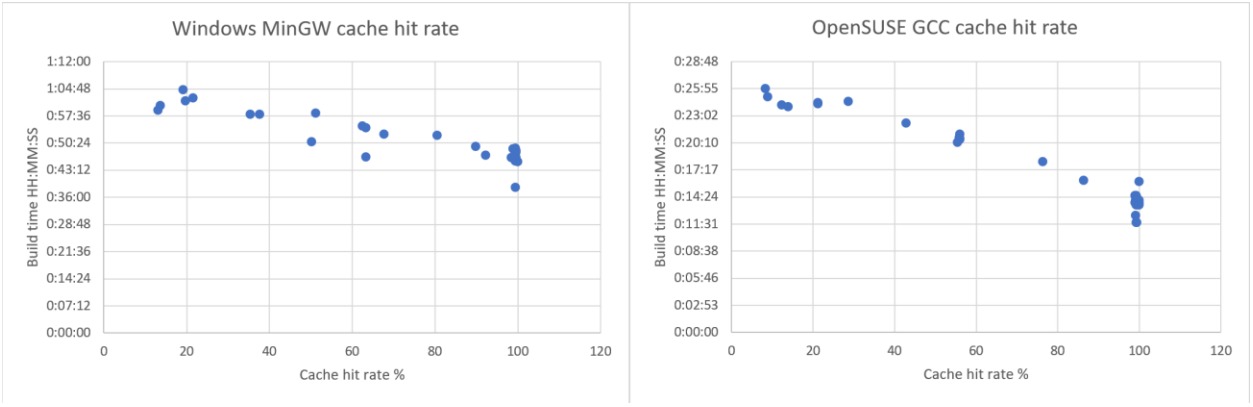

FIGURE 12. Build time scaling compared to the cache hit rate.

Average cache hit rates were 78.6% for Windows and 76.5% for OpenSUSE. The build time averages were 50:22 for Windows and 16:59 for OpenSUSE. The average cache hit rate of around 75% tells that sccache benefits quite well in most of the builds relative to what it can do. The cache hit rate could be possibly increased with a more intelligent garbage collector.

On the OpenSUSE the average build time for a cache hit rate of >99% builds fall around the 14-minute mark. As earlier noticed, this leaves the marginal time for anything else than the artifact upload which takes on average 7-9 minutes.

The scaling from a 0 to 100% cache hit rate looks quite linear in the figure 12, however there are some cases where the MinGW built with a hit rate of 50% builds as fast as with a hit rate of 100%. This variance in the build time with the same hit rate is caused by the load on the host which is running the virtual machines. These hosts might run only one

virtual machine or over a dozen parallel virtual machines. Running multiple virtual machines on a single host reduces the performance of a single virtual machine marginally. Another factor is that some objects are more expensive to compile than others, and the same hit rate can have in fact a different load in such case.

The Windows target builds tests during the build and the OpenSUSE target does not. This explains the longer build on the Windows in figure 12. The preprocessor overhead is also larger on Windows, but it is not comparable in the figure 12 when tests are built only on Windows.

5.5 Caching test building

Changes to cache the test building process were done when collecting the data for the first charts. The caching of the test building was enabled after the first set of data was collected for the average and median values.

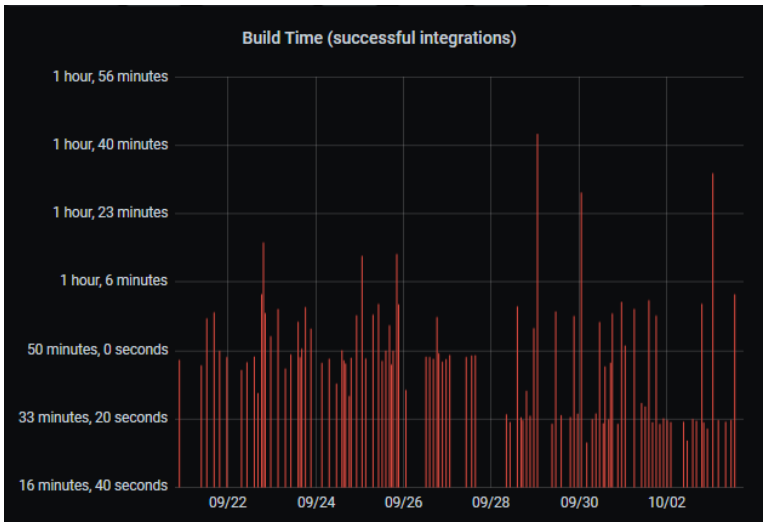

FIGURE 13. Effect of test caching on Windows MinGW.

Figure 13 shows the effect when the test caching was enabled on the Windows with MinGW. Building the tests with caching shows slightly better gains compared to the caching of the module build itself on Windows with MinGW. The first half of figure 13 shows most builds around the 50-minute mark which matches to the cached result in figure 12. The latter half has the tests cached and the cached results are around the 33-minute mark.

The full cached result was reduced by around 16 minutes only by caching the building of the tests. By caching the module build, the build time reduced by 12 minutes, as seen in the earlier chapter on the hit rate comparison. Figure 14 shows the test caching effect on the RHEL.

The reason why the test caching can be more effective than the build itself comes down to the single tests being small in overall size. Another thing is the relatively simple dependency hash calculation for the tests, which reduces the overhead caused by the preprocessor calls done by sccache. Also, there is no complex linking happening while building the tests. On the MinGW building the tests take as long as the module build making the comparison easy, this is not the case on all targets. Across all targets the percentual reduction of time spent building tests stayed around 80-90% with the cache hit rate of >99%.

5.6 Optimizing artifact transfer

The artifact transfer process between the agent and Coin was optimized late during the thesis This optimization is not included in any of the previous data in the thesis. The bottleneck in the transfer was the agent archiving in a single serial thread. The use of gzip library in Golang's standard library was replaced with a 3rd party pgzip library which allows parallelization of the archiving process. (25.)

The figure 14 shows a comparison of the build overview before and after the build and tests were cached and the artifact transfer optimized.

RHEL build Overview			
	No-sccache	Sccache 100%	Difference
Agent initialization	8s	8s	0%
Configure	51s	48s	0%
Cmake build	25m 47s	3m 25s	-87%
Install	13s	13s	0%
Export artifacts	7m 29s	26s	-95%
Cmake build tests	14m 45s	2m 11s	-85%
Export test artifact	12m 4s	1m 5s	-91%
Total:	60min 17s	8m 49s	-85%

FIGURE 14. Overview of effects of the optimizations on RHEL.

Figure 14 shows the best-case scenario with a cache hit rate of 100%. Obviously, this is not always the case. In the previous chapter the average cache hit rate turned out to be

around 75%. From this it can be calculated that when the scaling is linear between the build time and the cache hit rate, the sccache reduces the build time on average by 26 minutes on the RHEL. The optimized artifact transfer reduces the build time by a fixed amount of time, around 18 minutes in this case.

Other platforms showed a similar uplift in the artifact export process. The slowest of the targets after all optimizations was Windows with MinGW with a reduction from around 1 hour to around 20 minutes comparing to the build without caching at all.

6 SHARING THE CACHE

There are many caching solutions for CI systems. These usually work in one of the following ways; running jobs on a static build machine, saving local cache for the next job as an artifact on a remote storage, using a remote storage directly to store the objects. While evaluating which system to choose, the use of static build machines was dismissed instantly as Qt requires a highly dynamic environment for testing against different dependencies. After evaluating all the factors, the decision eventually leaned to selecting the sccache, which stores objects directly to a remote storage.

6.1 Static build machines

Using a static build machine to build different revisions of a project solves the caching problem as the cache's storage also remains static. However, this introduces loads of other problems. The main issue in using this approach in the Qt's case is that provisioning the build machine with different versions of build tools and dependencies becomes very difficult. It is also difficult to have a dynamic environment where the number of build machines of each OS running could be changed quickly.

This kind of system was used in the very early Coin without any cache and it suffered also from random issues caused by the dirty starting point to subsequent integrations, even when the whole workspace was wiped for each integration.

6.2 Storing the cache as artifact

A solution used by many CI systems, such as Travis CI and GitLab is to store the whole cache as a single artifact instead of storing each object separately. (1, 2.)

Storing the whole cache as a single artifact is better than storing single objects for large builds, as the build machine can read from the local disk instead of communicating over the network. However, small builds potentially suffer due to them having to download the whole cache instead of few objects which are needed.

In the Qt's case, it was calculated that if a separate cache was created for each branch, project and target combination, it would result in 22,275 cache artifacts as there are around 45 modules, 55 platforms and 9 active branches. If one cache artifact were 50 Megabytes on average, it would result in total size of 1.1 Terabytes. Having many cache artifacts potentially makes also the garbage collection more complicated.

6.3 Storing objects remotely

The approach that the sccache takes is to store each object remotely. Many CI systems can use sccache as it does not necessarily need changes to the underlying CI system, only the remote storage server and the build configuration changes are needed to enable the use of sccache. Sccache stores each object separately in one shared remote storage available to all build machines. The advantage of this approach is the availability of the objects, which is its key point.

Some of the main benefits of this approach in the Qt's case are

- Branching causes there to be many branches which in state are close to each other, sometimes even identical when a release branch is separated.
- In some cases, the targets differ only by a configure argument or an environment variable, resulting in a nearly identical build. In these cases, the cached objects will be available even for parallel builds in the same integration.
- Sccache implementing this strategy can be used on all platforms and compilers currently used in the Coin.

Due to the branching of Qt and the number of similar platforms being run parallel, this approach has more advantages over the other solutions. Compared to the previous solutions this solution eliminates the need to build an object again across all platforms, projects and branches while the others only for one combination.

One of the largest downsides in the sccache which implements this solution is that the preprocessor is always run to calculate the dependency hash, which causes the overhead depending on the preprocessor used. There are plans to implement such system on

sccache which mimics the direct mode of the ccache. This would reduce the overhead of the sccache a lot. (26.)

Another way to share the cache remotely is via a filesystem mount or by an NFS share (27.) This is supported by ccache, too. This requires setting up the NFS share or a filesystem mount to be available for each build slave and ensuring safe access to the storage simultaneously by all the build machines. In principle this is close to how the sccache works. Sccache only uses some storage backend instead of direct access.

Looking purely at the build performance this approach with ccache possibly has the fastest builds of a non-static slave implementation, however, no data is available of this approach.

Compared to the sccache which has a storage backend to handle the I/O to the disk, this approach lets ccache to do the writing possibly causing a more I/O load.

What eventually tipped the selection to sccache in the Qt's case is that the sccache supports also the MSVC compiler. In cases where the MSVC is not part of the system where the cache is being implemented perhaps ccache via a shared NFS drive is worth the shot.

7 CONCLUSION

This thesis aimed to enable the use of sccache in Coin. Sccache was enabled for most of the CMake platforms in use and it was evaluated that it has a positive effect. Only one MSVC target was not enabled during the thesis of the CMake targets, but the changes to enable it are already in progress. This can be considered a success, and it is easier to start extending the coverage from this.

Based on the success in this thesis, the sccache seems to fit well in a distributed compiling environment like Coin. The MinIO storage seems to scale well enough as it is more than capable of handling the load from sccache in the current infrastructure.

Despite the promising results in build times, there was no observable difference in overall integration times. This is due to the qmake targets without any cache still in the same integrations being the limiting factor. After the qmake targets deprecate in the very near future there will certainly be an effect on the integration time. Currently, the longest cached CMake target runs for 1h10min and longest qmake target brings this up to 2h5min.

For some further development on the sccache topic, the garbage collector logic could be made more intelligent with the statistics from the storage to show which objects are being used frequently. With the current logic, the last week worth of objects are cached. For example, there could be multiple rules, e.g. older than week and not used in last 48 hours. This would avoid deleting some objects which stay the same for a long time. Also, the qmake targets could be evaluated if it makes sense to put the effort in to enable the sccache on them.

After sccache is enabled for all targets, the build caching topic does not offer much for future development other than optimizing a cacheable call ratio and a garbage collector. One thing to consider is the direct mode equivalent support in the upstream of sccache.

Now, that the build time has been optimized, and the testing is now the longest part of the integration, there could be efforts to see if the test execution can be parallelized to speed up the test phase.

REFERENCES

1. GitLab CI. GitLab Inc. Date of retrieval 3.10.2020
   https://gitlab.com/

2. Travis CI. Travis CI. Date of retrieval 3.10.2020
   https://travis-ci.org/

3. Sccache. Mozilla. Date of retrieval 22.3.2020
   https://github.com/mozilla/sccache

4. Gladhorn Frederik, 2016. Coin - Continuous Integration for Qt. Date of retrieval 3.10.2020 https://www.qt.io/blog/2016/08/08/coin-continuous-integration-for-qt

5. Christian Adam, 2020. Visual C/C++ compiler support. Date of retrieval 3.10.2020
   https://github.com/ccache/ccache/pull/506

6. Coin - Qt Continuous Integration. The Qt Company. Date of retrieval 3.10.2020
   https://testresults.qt.io/coin/doc/overview.html

7. OpenNebula. OpenNebula Systems. Date of retrieval 3.10.2020
   https://opennebula.io/

8. MinIO. Minio, Inc. Date of retrieval 3.10.2020
   https://min.io/

9. Saario Toni – Keskimölö Aapo - Jędrzej Nowacki. Enable use of sccache in the CI Date of retrieval 3.10.2020 https://codereview.qt-project.org/c/qt/qt5/+/232218

10. Qtqa Gerrit plugin. The Qt Company. Date of retrieval 3.10.2020
    https://github.com/qtproject/qtqa-gerrit-plugin-qt-workflow

11. Amazon AWS. Configuration and credential file settings. Date of retrieval 3.10.2020
    https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html

12. Musl libc. Musl. Date of retrieval 3.10.2020
https://musl.libc.org/

13. Maudoux, G., & Mens, K. (2018). Correct, Efficient, and Tailored: The Future of Build Systems. IEEE Software, p. 6.

14. CMake. CMake. Date of retrieval 3.10.2020
https://cmake.org/

15. Qmake Manual. The Qt Company. Date of retrieval 3.10.2020
https://doc.qt.io/qt-5/qmake-manual.html

16. Ccache. Run Modes. Date of retrieval 3.10.2020
https://ccache.dev/manual/3.7.11.html#_run_modes

17. Toni Saario. Add qmake feature and configure option to use sccache. Date of retrieval 3.10.2020 https://codereview.qt-project.org/c/qt/qtbase/+/291129

18. Toni Saario. Read test specific configure arguments from environment. Date of retrieval 3.10.2020 https://codereview.qt-project.org/c/qt/qtbase/+/311366

19. Qt Test Results. The Qt Company. Date of retrieval 3.10.2020
https://testresults.qt.io/grafana

20. Mike Hommey 2014. Shared compilation cache experiment. Date of retrieval 28.3.2020 https://glandium.org/blog/?p=3054

21. Mike Hommey 2014. Testing shared cache on try. Date of retrieval 28.3.2020
https://glandium.org/blog/?p=3188

22. Ccache. Modes. Date of retrieval 3.10.2020
https://ccache.dev/manual/3.7.11.html#_the_direct_mode.

23. Matev Rosen 2019. Fast distributed compilation and testing of large C++ projects. Date of retrieval 3.10.2020 https://cds.cern.ch/record/2699544/files/Matev_distributed_compilation%2005.11.pdf

24. Microsoft 2020. /Z7, /Zi, /ZI (Debug Information Format). Date of retrieval 3.10.2020 https://docs.microsoft.com/en-us/cpp/build/reference/z7-zi-zi-debug-information-format?view=vs-2019

25. Pgzip. Post Klaus. Date of retrieval 3.10.2020 https://github.com/klauspost/pgzip

26. Mielczarek Ted 2018. Add support for an equivalent to ccache direct mode. Date of retrieval 3.10.2020 https://github.com/mozilla/sccache/issues/219

27. Ccache. Sharing a cache. Date of retrieval 3.10.2020 https://ccache.dev/manual/3.7.11.html#_sharing_a_cache