

Visualizing Chrome Trace Events

You can use *full stack tracing* to trace from the top level QML or JavaScript down to the C++ and all the way to the kernel space. This enables you to measure the performance of an application and to check whether it is CPU or I/O bound or influenced by other applications running on the same system. Tracing provides insight into what a system is doing and why an application is performing in a particular way. It indicates how the hardware is utilized and what the kernel and application are doing.

Tracing information can also provide you additional insight into the data collected by [QML Profiler](#). For example, you could check why a trivial binding evaluation is taking so long. This might be caused by C++ being executed or the disk I/O being slow.

Several tracing tools (such as `chrome://about`) can generate information about Chrome trace events in Chrome Trace Format (CTF). You can open CTF files in Qt Creator for viewing. This is especially useful when viewing trace files larger than 100 MB, which are difficult to view with the built-in trace-viewer (`chrome://tracing`) due to its high memory usage.

The visualizer supports all event types used in data that the [LTTng](#) tracing framework generates, converted to CTF. However, some of the more advanced event types used, for example, in Android system traces, are not supported. The visualizer silently ignores unsupported event types.

The visualizer supports the following event types:

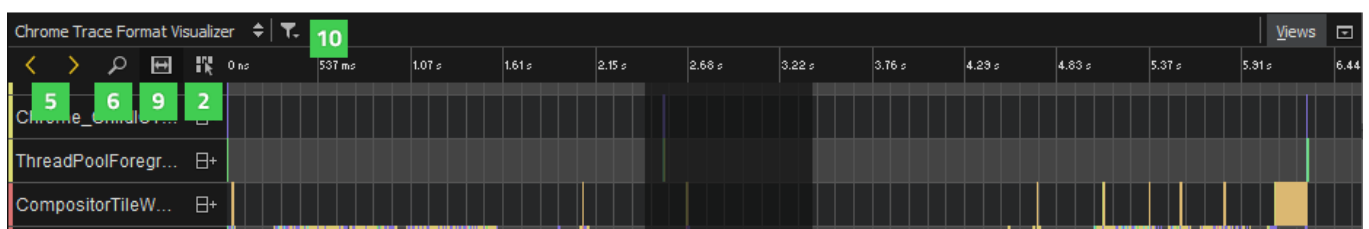
- › Begin, end, duration, and instant events
- › Counter events (graphs)
- › Metadata events (process and thread name)

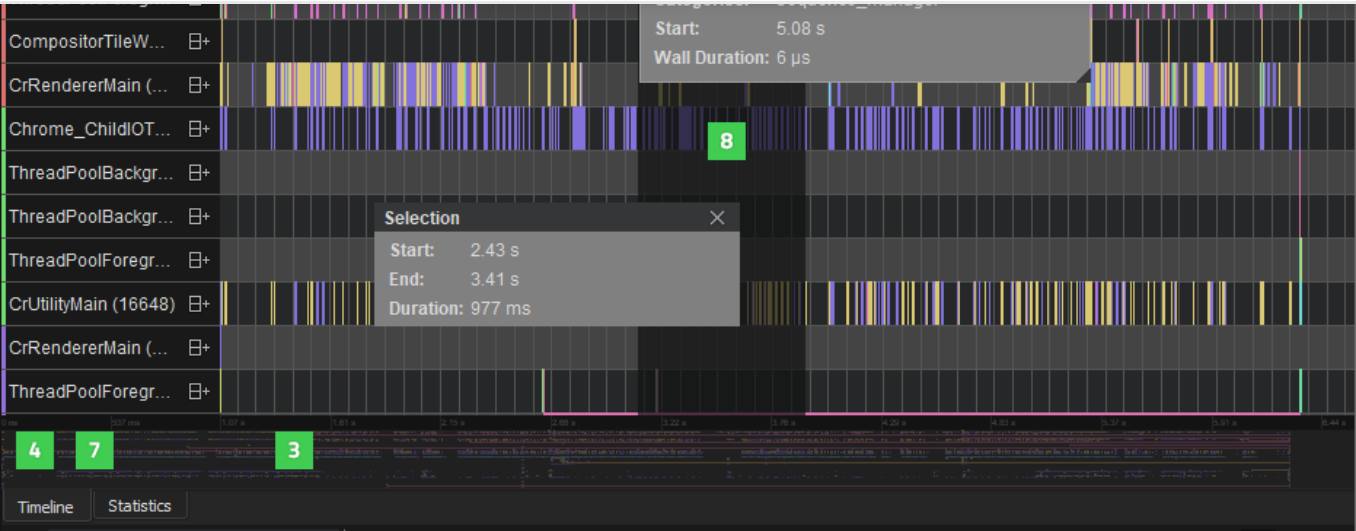
Opening JSON Files

To open JSON files for viewing, select **Analyze > Chrome Trace Format Viewer > Load JSON File**.

Visualizing Events

The **Timeline** view displays a graphical representation of trace events and a condensed view of all recorded events.





Each category in the timeline describes a thread in the application. Move the cursor on an event (1) on a row to view its duration and event category. To display the information only when an event is selected, disable the **View Event Information on Mouseover** button (2).

The outline (3) summarizes the period for which data was collected. Drag the zoom range (4) or click the outline to move on the outline. To move between events, select the **Jump to Previous Event** and **Jump to Next Event** buttons (5).

Select the **Show Zoom Slider** button (6) to open a slider that you can use to set the zoom level. You can also drag the zoom handles (7). To reset the default zoom level, right-click the timeline to open the context menu, and select **Reset Zoom**.

Select the  (**Restrict to Threads**) button (10) to select the threads to show.

Selecting Event Ranges



You can select an event range (8) to view the time it represents or to zoom into a specific region of the trace. Select the **Select Range** button (9) to activate the selection tool. Then click in the timeline to specify the beginning of the event range. Drag the selection handle to define the end of the range.

You can use event ranges also to measure delays between two subsequent events. Place a range between the end of the first event and the beginning of the second event. The **Duration** field displays the delay between the events in milliseconds.

To zoom into an event range, double-click it.

To remove an event range, close the **Selection** dialog.

Viewing Statistics

Chrome Trace Format Visualizer 							Views 
Title	Count	Total Time	Percentage	Minimum Time	Average Time	Maximum Time	
XHRRReadyStateChange	39	4.49 ms	0.07 %	7 µs	115 µs	307 µs	
XHRLoad	10	174 µs	0.00 %	6 µs	17.4 µs	30 µs	
WorkerThreadThread born	1	-	-	0 ns	-	-	
WorkerThreadThread active	502	23.1 s	368.26 %	29 µs	46.1 ms	5.37 s	
WindowTreeHost::OnHostMovedInPixels	1	126 µs	0.00 %	126 µs	126 µs	126 µs	
WidgetMsg_WasShown	2	278 µs	0.00 %	119 µs	139 µs	159 µs	
WidgetMsg_WasHidden	2	383 µs	0.01 %	67 µs	192 µs	316 µs	
WidgetMsg_UpdateVisualProperties	1	56 µs	0.00 %	56 µs	56 µs	56 µs	
WidgetMsg_UpdateScreenRects	4	114 µs	0.00 %	13 µs	28.5 µs	44 µs	
WidgetMsg_SetActive	4	239 µs	0.00 %	4 µs	59.8 µs	221 µs	
WidgetInputHandlerManager::HandledInputEvent	212	4.32 ms	0.07 %	3 µs	20.4 µs	152 µs	

WidgetHostMsg_SelectionBoundsChanged	1	26 µs	0.00 %	26 µs	26 µs	26 µs
WidgetHostMsg_FrameSwapMessages	1	7 µs	0.00 %	7 µs	7 µs	7 µs
WidgetHostMsg_DidFirstVisuallyNonEmptyPaint	1	15 µs	0.00 %	15 µs	15 µs	15 µs
Widget::Show	2	5.78 ms	0.09 %	1.75 ms	2.89 ms	4.03 ms
WebViewImpl::updateAllLifecyclePhases	126	42.1 ms	0.67 %	66 µs	335 µs	9.1 ms
WebViewImpl::handleInputEvent	122	47.7 ms	0.76 %	44 µs	391 µs	2.27 ms
WebViewImpl::beginFrame	126	7.15 ms	0.11 %	15 µs	56.7 µs	511 µs
WebView::SetWebContents	6	14 ms	0.22 %	3 µs	2.33 ms	4.83 ms
WebView::DetachWebContentsNativeView	4	7.85 ms	0.12 %	2 µs	1.96 ms	4.77 ms
WebView::AttachWebContentsNativeView	4	6.05 ms	0.10 %	2 µs	1.51 ms	4.8 ms
WebURLLoaderImpl::loadAsynchronously	10	4.69 ms	0.07 %	163 µs	469 µs	1.28 ms
WebURLLoaderImpl::Context::Start	10	4.22 ms	0.07 %	140 µs	422 µs	1.18 ms
WebURLLoaderImpl::Context::OnReceivedResponse	10	1.22 ms	0.02 %	66 µs	122 µs	268 µs
WebURLLoaderImpl::Context::OnCompletedRequest	10	4.68 ms	0.07 %	218 µs	468 µs	989 µs
WebURLLoaderImpl::Context::Cancel	10	1.22 ms	0.02 %	67 µs	122 µs	201 µs
Waiting for next BeginFrame	164	-	-	0 ns	-	-
viz.mojom.GpuService	24	611 µs	0.01 %	8 µs	25.5 µs	68 µs
viz.mojom.GpuHost	8	223 µs	0.00 %	6 µs	27.9 µs	58 µs

Timeline

Statistics

The **Statistics** view displays the number of samples each function in the timeline was contained in, in total and when on the top of the stack (called `self`). This allows you to examine which functions you need to optimize. A high number of occurrences might indicate that a function is triggered unnecessarily or takes very long to execute.

Collecting LTTng Data

LTTng is a tracing toolkit for Linux that you can apply on embedded Linux systems to find out how to optimize the startup time of an application.

Since Qt 5.13, Qt provides a set of kernel trace points and a tracing subsystem for custom user space trace points.

Configuring the Kernel

To use LTTng, you have to set the following configuration options for the kernel before building it:

- › CONFIG_HIGH_RES_TIMERS
- › CONFIG_KALLSYMS
- › CONFIG_MODULES
- › CONFIG_TRACEPOINTS

We recommend that you set the following additional options:

- › CONFIG_EVENT_TRACING
- › CONFIG_HAVE_SYSCALL_TRACEPOINTS
- › CONFIG_KALLSYMS_ALL

In Yocto, you can activate the above options in **Menu > Config > Kernel Hacking > Tracers**.

Installing LTTng

After you build the kernel and deploy it on your device, you'll need to install the following LTTng packages on your device:

- › `lttng-tools` to control the tracing session
- › `lttng-modules` for kernel trace points
- › `lttng-ust` for user space trace points

Building Qt with tracepoints

Trace points are continuously being added to Qt versions. To use them, you need to build Qt yourself with the `configure -trace lttng` option.

Recording Events

To create a session, you call the `lttng create` command. Then you call `lttng enable-channel kernel -k` to enable the kernel channel. Within the kernel channel, you specify the appropriate trace points as `kernel_events` and call `lttng enable-event` to enable them. Finally, you call `lttng start` to start tracing.

You call `lttng stop` to stop tracing. You can use `sleep` to set the length of the session. After stopping, you can call `lttng destroy` to destroy the session.

You can write and run scripts that contain the above commands to start and stop full-stack tracing. You can use `systemd` to execute the scripts.

Enabling Trace Points

Data is recorded according to the trace points that you enable in the LTTng session. Usually, it is useful to enable scheduler switch, syscall, and Qt trace points.

Scheduler Switch Trace Points

Scheduler switch trace points are reached when an application is switched out due to preemption, for example, when another process gets the chance to run on the CPU core. Enable scheduler switch trace points to record the thread that is currently running and the process it belongs to, as well as the time when the process started and stopped.

Syscall Trace Points

Syscall trace points help you to understand why a scheduler switch happened. The following are examples of syscalls to trace:

- › `openat` and `close` map file descriptors to file names
- › `mmap` maps page faults to files
- › `read` and `write` are triggered by I/O operations
- › `nanosleep`, `futex`, and `poll` explain scheduler switches
- › `ioctl` controls the GPU and display

Converting LTTng Data to CTF

The `ctf2ctf` tool uses `babeltrace` to parse binary Common Trace Format (CTF) and converts it to Chrome Trace Format (CTF). It performs the following custom tasks to make the recording more human-readable:

- › Map file descriptors to file names
- › Map page faults to file names
- › Annotate interrupts and block devices with names
- › Convert UTF-16 `QString` data to UTF-8 strings
- › Count memory page allocations

```
ctf2ctf -o trace.json path/to/lttng trace/
```

< Analyzing Code with Cppcheck

Running Autotests >

© 2022 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.



Contact Us

Company

- About Us
- Investors
- Newsroom
- Careers
- Office Locations

Licensing

- Terms & Conditions
- Open Source
- FAQ

Support

- Support Services
- Professional Services
- Partners
- Training

For Customers

- Support Center
- Downloads
- Qt Login
- Contact Us
- Customer Success

Community

- Contribute to Qt
- Forum
- Wiki
- Downloads
- Marketplace

