Full-stack Tracing With LTTng

Combined Kernel and User Space Tracing

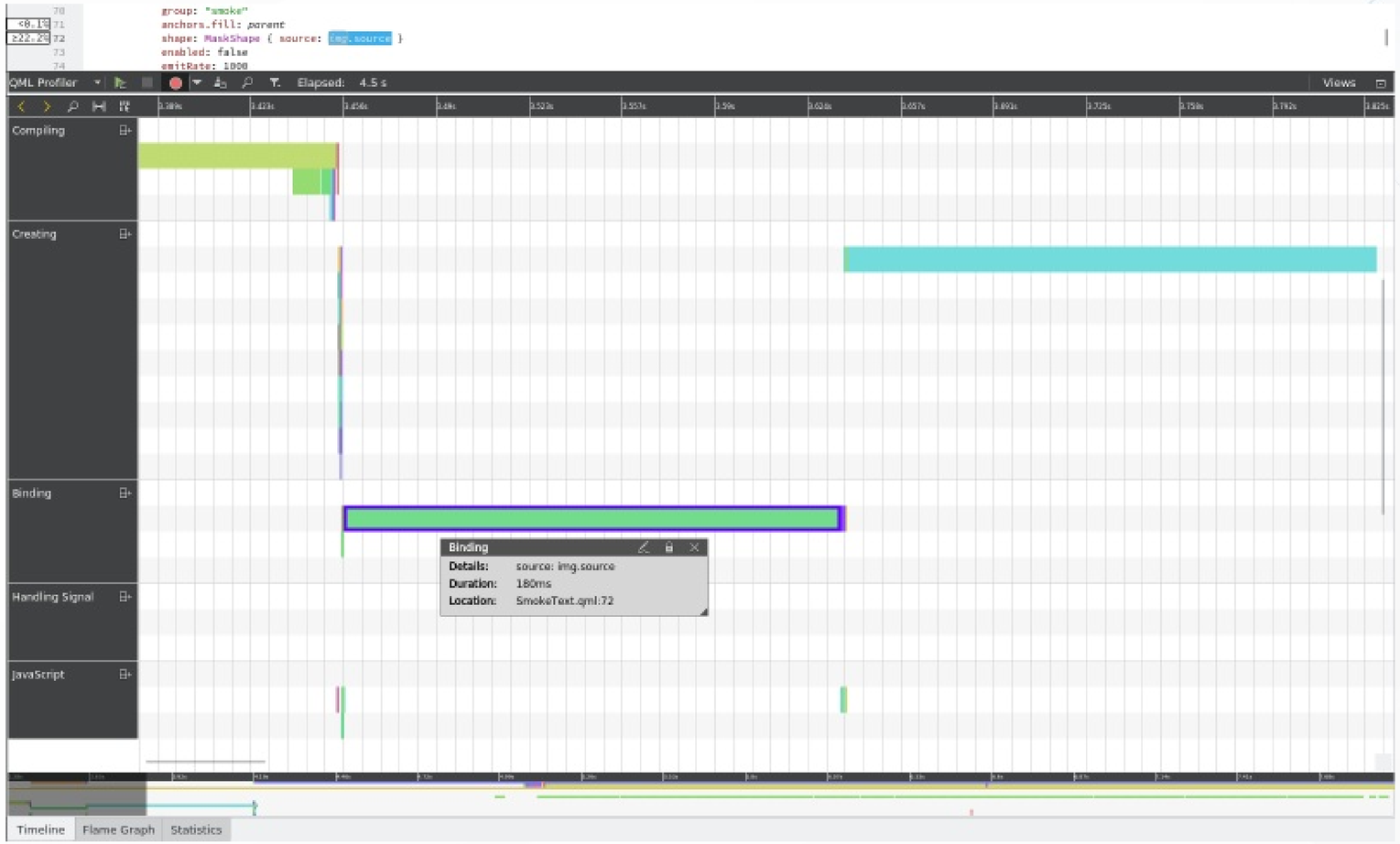
Qt World Summit 2019

presented by Milian Wolff

- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
- Future Outlook

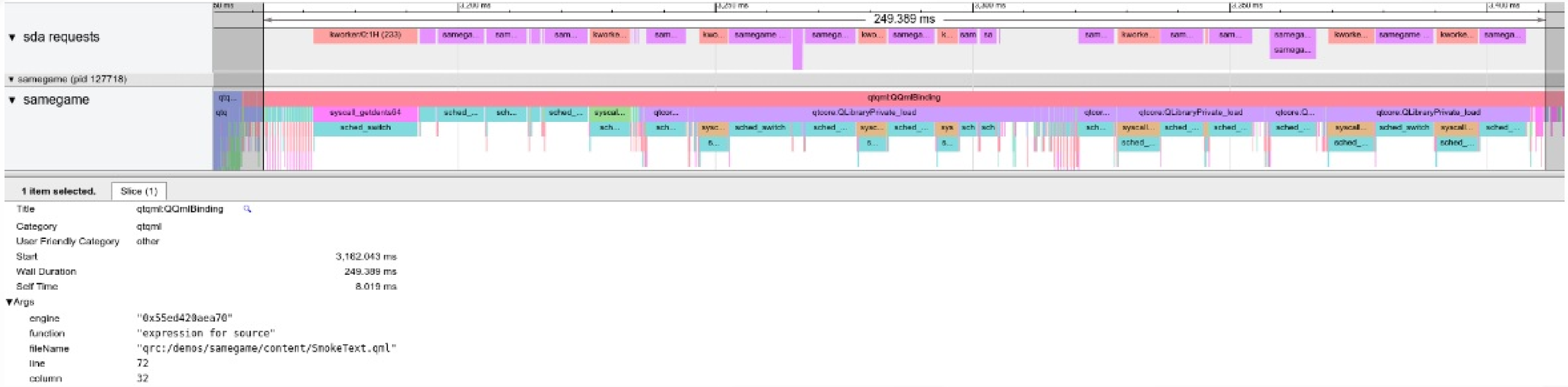
QML Profiler results are often hard to understand

- Slow C++ code?
- Preemption?
- Disk IO?
  - Page faults?
  - Memory swapping?



Full-stack tracing can be used to find an explanation

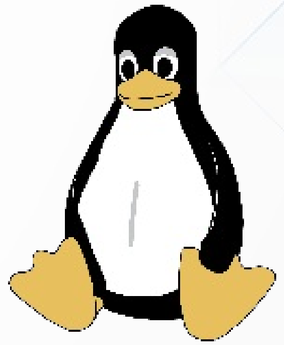
- Binding runs C++ code which is loading libraries (image formats)
- Lots of disk I/O moves the application off the CPU (context switches)



- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
- Future Outlook

Enabling Tracing in the Linux Kernel

- Required kernel config options:
  - `CONFIG_MODULES`
  - `CONFIG_KALLSYMS`
  - `CONFIG_HIGH_RES_TIMERS`
  - `CONFIG_TRACEPOINTS`
- Recommended additional kernel config options:
  - `CONFIG_HAVE_SYSCALL_TRACEPOINTS`
  - `CONFIG_EVENT_TRACING`
  - `CONFIG_KALLSYMS_ALL`
- Cf. Kernel Hacking > Tracers in `menuconfig`



Installing LTTng

- Packages to install:
  - `lttng-tools` to control the tracing session
  - `lttng-modules` for kernel trace points
  - `lttng-ust` for user space trace points
- Yocto enables these by default with
  - `EXTRA_IMAGE_FEATURES += "tools-profile"`

Enabling Tracing in Qt

- Use latest sources from git (5.13 or newer)
  - For additional trace points in `qtdeclarative` apply:
    - https://codereview.qt-project.org/c/qt/qtdeclarative/+/277210
    - https://codereview.qt-project.org/c/qt/qtdeclarative/+/277666
- Then build Qt with:
  - `configure -trace lttng ...`
- For yocto, apply:
  - https://github.com/meta-qt5/meta-qt5/pull/240

- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
- Future Outlook

Execute the following script to start full-stack tracing:

trace_start.sh

```
#!/bin/sh
if [ ! -d /tmp/lttng ]; then
    mkdir /tmp/lttng
fi
lttng create -o /tmp/lttng/$(date -Iseconds)

# enable most important kernel trace points
lttng enable-channel kernel -k
kernel_events=(
    "sched_switch,sched_process_*" "lttng_statedump_*"
    "irq_*" "signal_*" "workqueue_*" "power_cpu_frequency"
    "kmem_"{mm_page,cache}_{alloc,free} "block_rq_"{issue,complete,requeue}
    # "x86_exceptions_page_fault_"{user,kernel}
)
for event in "${kernel_events[@]}"; do
    lttng enable-event -c kernel -k "$event"
done
lttng enable-event -c kernel -k --syscall -a

# enable all user space tracepoints
lttng enable-channel ust -u
lttng enable-event -c ust -u -a

# actually start tracing
lttng start
```

Execute the following script to stop tracing:

lttng_stop.sh

```
#!/bin/sh

if [ ! -z "$1" ]; then
    # delay stopping
    sleep "$1"
fi

lttng stop
lttng destroy
```

Putting it all together to trace startup of an application:

```
lttng_start.sh
lttng_stop.sh 20 &
./start_long_running_process
```

/etc/systemd/system/lttng_start.service

```
[Unit]
Description=start lttng tracing
After=sysinit.target
Before=multi-user.target

[Service]
Type=oneshot
ExecStart=/usr/bin/lttng_start.sh
RemainAfterExit=true

[Install]
WantedBy=sysinit.target
```

/etc/systemd/system/lttng_stop.service

```
[Unit]
Description=stop lttng tracing
After=multi-user.target

[Service]
Type=oneshot
ExecStart=/usr/bin/lttng_stop.sh 20

[Install]
WantedBy=multi-user.target
```

```
systemctl enable lttng_start.service
systemctl enable lttng_stop.service
```

- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
- Future Outlook

Babeltrace is the basic library for Common Trace Format parsing

- CLI utility to convert CTF data to text
  - Tedious to grasp what's going on
- C API to parse CTF data for custom analyses

```
[16:21:53.959371489] (+0.001033737) irq_handler_entry: { cpu_id = 4 }, { irq = 26, name = "ahci[0000:00:1f.2]" }
[16:21:53.959376537] (+0.000005048) block_rq_complete: { cpu_id = 4 }, { dev = 8388640, sector = 412553920, nr_sector = 192, error = 0, rwbs = 12 }
[16:21:53.959383667] (+0.000007130) kmem_cache_free: { cpu_id = 4 }, { call_site = 0xFFFFFFFF997A02D6, ptr = 0xFFFF8E8D764FB000 }
[16:21:53.959384148] (+0.000000481) kmem_cache_free: { cpu_id = 4 }, { call_site = 0xFFFFFFFF997A9023, ptr = 0xFFFF8E8D78778D00 }
[16:21:53.959384386] (+0.000000238) sched_switch: { cpu_id = 6 }, { prev_comm = "swapper/6", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "samegame", next_tid = 12 7718, next_prio = 20 }
[16:21:53.959385565] (+0.000001179) irq_handler_exit: { cpu_id = 4 }, { irq = 26, ret = 1 }
[16:21:53.959413189] (+0.000027624) qtcore:QObject_ctor: { cpu_id = 6 }, { object = 0x7FFD1B520660 }
[16:21:53.959419236] (+0.000006047) qtgui:QGuiApplicationPrivate_init_entry: { cpu_id = 6 }, { }
[16:21:53.959423690] (+0.000004454) qtcore:QCoreApplicationPrivate_init_entry: { cpu_id = 6 }, { }
...
[16:21:53.962944916] (+0.000023918) sched_switch: { cpu_id = 6 }, { prev_comm = "swapper/6", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "samegame", next_tid = 127718, next_prio = 20 }
[16:21:53.962968468] (+0.000023552) syscall_entry_openat: { cpu_id = 6 }, { dfd = -100, filename = "/usr/lib/locale/locale-archive", flags = 524288, mode = 0 }
[16:21:53.962971371] (+0.000002903) kmem_cache_alloc: { cpu_id = 6 }, { call_site = 0xFFFFFFFF9967EC4B, ptr = 0xFFFF8E87FDFBA000, bytes_req = 4096, bytes_alloc = 4096, gfp_flags = 6291648 }
[16:21:53.962976139] (+0.000004768) kmem_cache_alloc: { cpu_id = 6 }, { call_site = 0xFFFFFFFF9966FA13, ptr = 0xFFFF8E8874118D00, bytes_req = 256, bytes_alloc = 256, gfp_flags = 6324416 }
[16:21:53.962986535] (+0.000010396) kmem_cache_free: { cpu_id = 6 }, { call_site = 0xFFFFFFFF9966B8EB, ptr = 0xFFFF8E87FDFBA000 }
[16:21:53.962988213] (+0.000001678) syscall_exit_openat: { cpu_id = 6 }, { ret = 41 }
[16:21:53.962990485] (+0.000002272) syscall_entry_newfstat: { cpu_id = 6 }, { fd = 41 }
[16:21:53.962993755] (+0.000003270) syscall_exit_newfstat: { cpu_id = 6 }, { ret = 0, statbuf = 140443163546912 }
[16:21:53.962995661] (+0.000001906) syscall_entry_mmap: { cpu_id = 6 }, { addr = 0x0, len = 3035216, prot = 1, flags = 2, fd = 41, offset = 0 }
[16:21:53.963002217] (+0.000006556) kmem_cache_alloc: { cpu_id = 6 }, { call_site = 0xFFFFFFFF99480ADA, ptr = 0xFFFF8E88743F8480, bytes_req = 192, bytes_alloc = 192, gfp_flags = 6291648 }
[16:21:53.963007890] (+0.000005673) syscall_exit_mmap: { cpu_id = 6 }, { ret = 0x7FBB6E688000 }
```

Trace Compass is a very good UI for analysis of kernel and syscall trace events

- Sadly, user space events aren't visible directly in the time line
  - Scripting features got added recently, may improve this situation in the future

KDAB RND project: https://github.com/KDAB/ctf2ctf

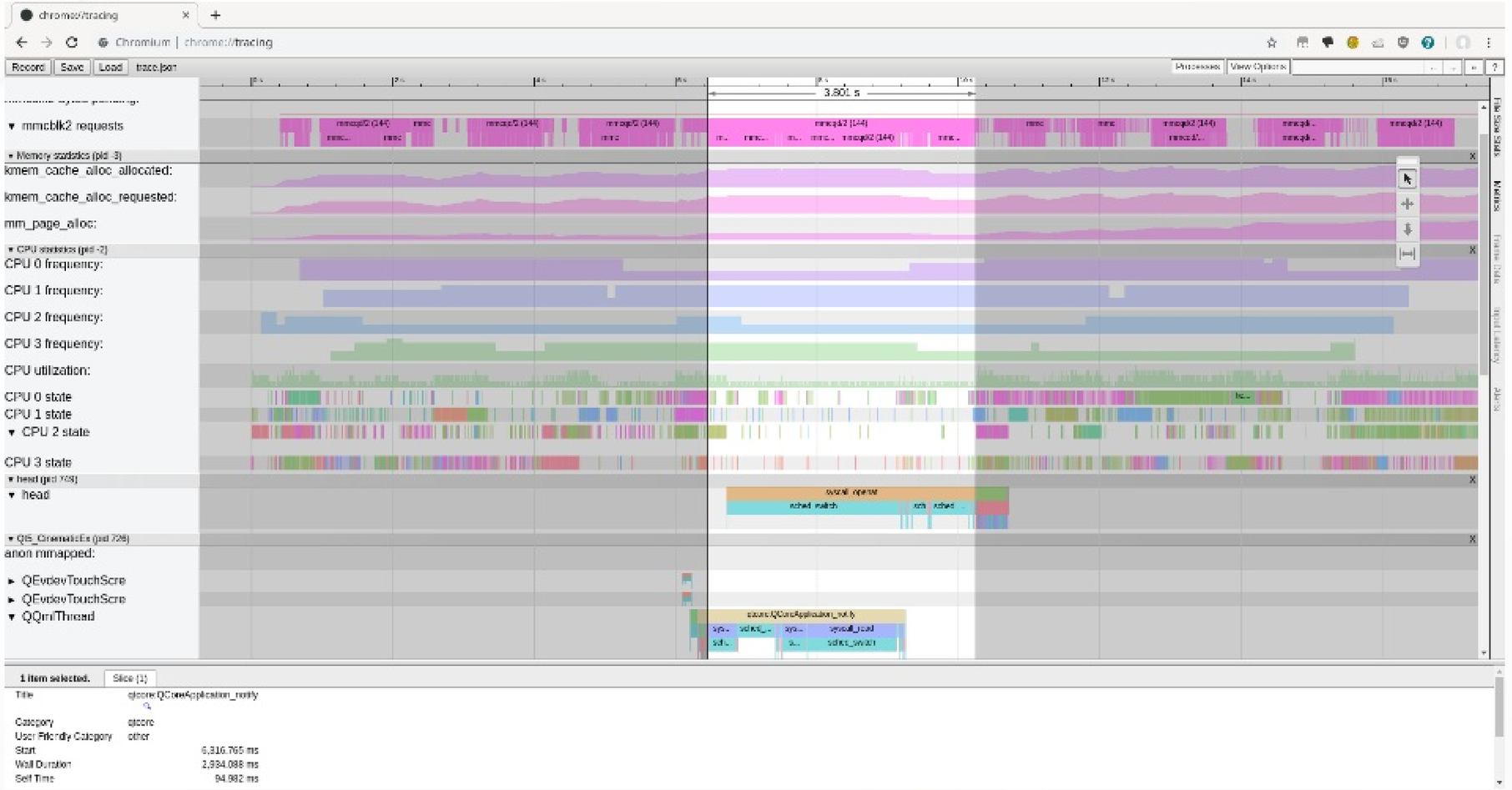
- Parse binary Common Trace Format via babeltrace
- Custom analyses steps
  - map file descriptor to file name
  - map page faults to file name
  - annotate interrupts and block devices with names
  - convert UTF-16 QString data to UTF-8 strings
  - count memory page allocations
- Generates Chrome Trace Format (JSON)
  - This is already easier to understand than raw babeltrace text output
  - Result can be visualized in Chrome or Qt Creator

Usage to convert LTTng data to CTF:

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

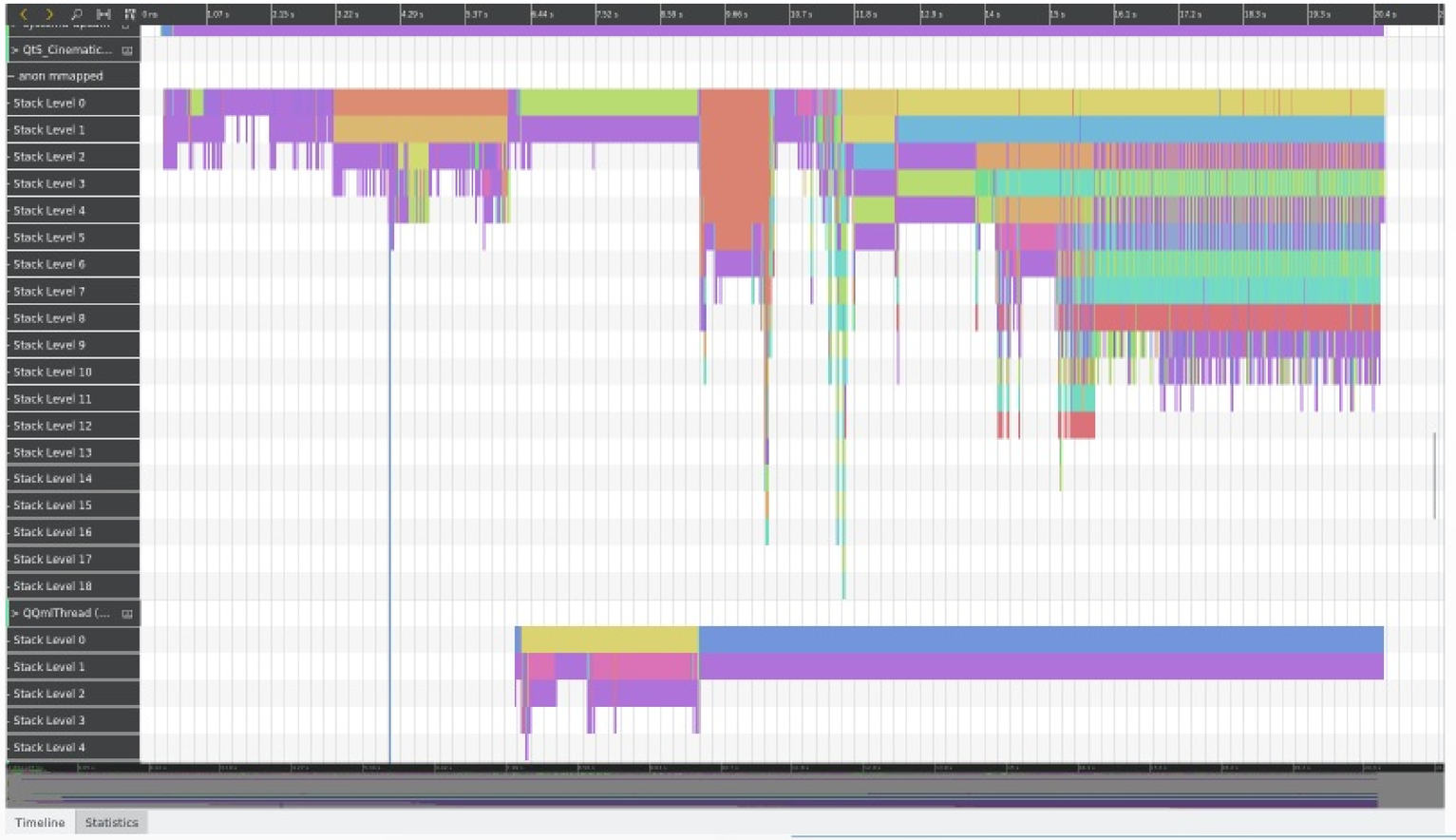
Chrome can load and visualize CTF JSON files via chrome://tracing

- Don't try to load really large traces, JavaScript is limited to 4GB of memory
- Advanced analyses are undocumented, require Android system traces, don't work with CTF



Qt Creator 4.11 supports visualizing Chrome Trace Format data

- KDAB RND project
- Most important functionality already in place
- More features planned for the future



- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
  - Kernel Tracepoints
  - Qt Tracepoints
- Future Outlook

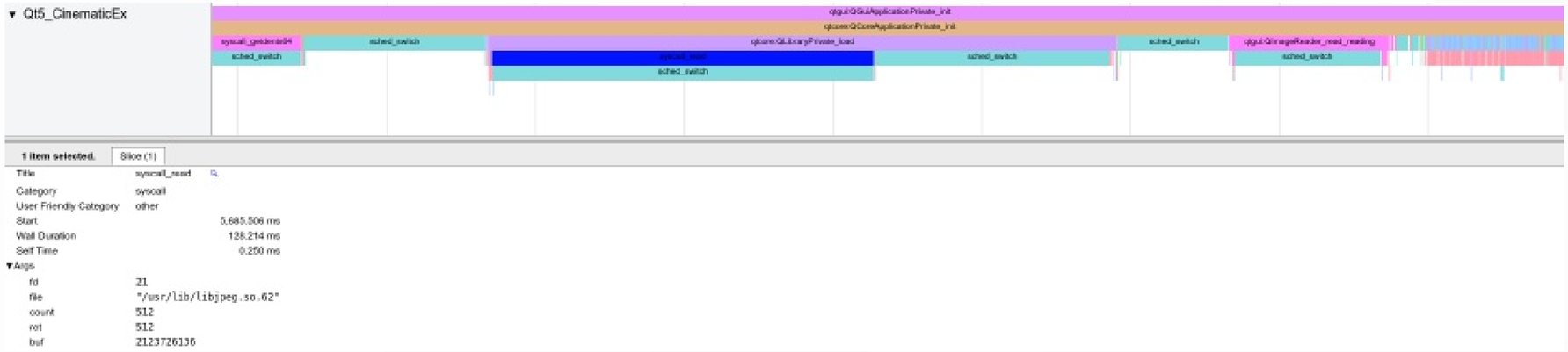
- Kernel Tracepoints
- Qt Tracepoints

Scheduler trace points are crucial to answer:

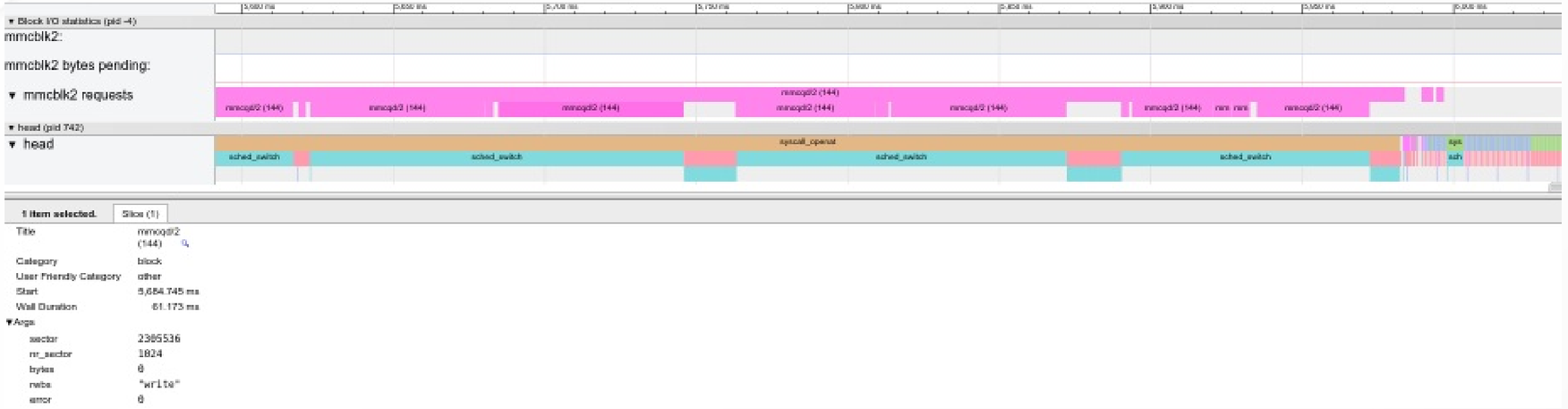
- Which thread is currently running?
- Which process does it belong to?
- When is a process started?
- When does it stop?

```
[09:36:22.257803501] sched_process_fork: { cpu_id = 3 },
    { parent_comm = "systemd", parent_tid = 1, parent_pid = 1, parent_ns_inum = 4026531836,
      child_comm = "systemd", child_tid = 720, _vtids_length = 1, vtids = [ [0] = 720 ],
      child_pid = 720, child_ns_inum = 4026531836 }
[09:36:22.257864167] sched_switch: { cpu_id = 0 },
    { prev_comm = "swapper/0", prev_tid = 0, prev_prio = 20, prev_state = 0,
      next_comm = "systemd", next_tid = 720, next_prio = 20 }
[09:36:22.291247834] sched_process_exec: { cpu_id = 1 },
    { filename = "/usr/share/cinematicexperience-1.0/Qt5_CinematicExperience", tid = 720, old_tid = 720 }
[09:36:22.293127834] sched_switch: { cpu_id = 1 },
    { prev_comm = "Qt5_CinematicEx", prev_tid = 720, prev_prio = 20, prev_state = 130,
      next_comm = "swapper/1", next_tid = 0, next_prio = 20 }
[09:36:32.294152834] sched_process_free: { cpu_id = 3 },
    { comm = "Qt5_CinematicEx", tid = 720, prio = 20 }
```

- Usually a good idea to trace all syscalls
  - Gives you a rough idea of what's going on in your code
- Some important syscalls to trace:
  - `openat, close`: map file descriptors to file names
  - `mmap`: map page faults to file
  - `read, write`: synchronous I/O
  - `nanosleep, futex, poll`: explanation for scheduler switches
  - `ioctl`: e.g. for GPU/display control
  - ... many more that can be useful to know for debugging and profiling

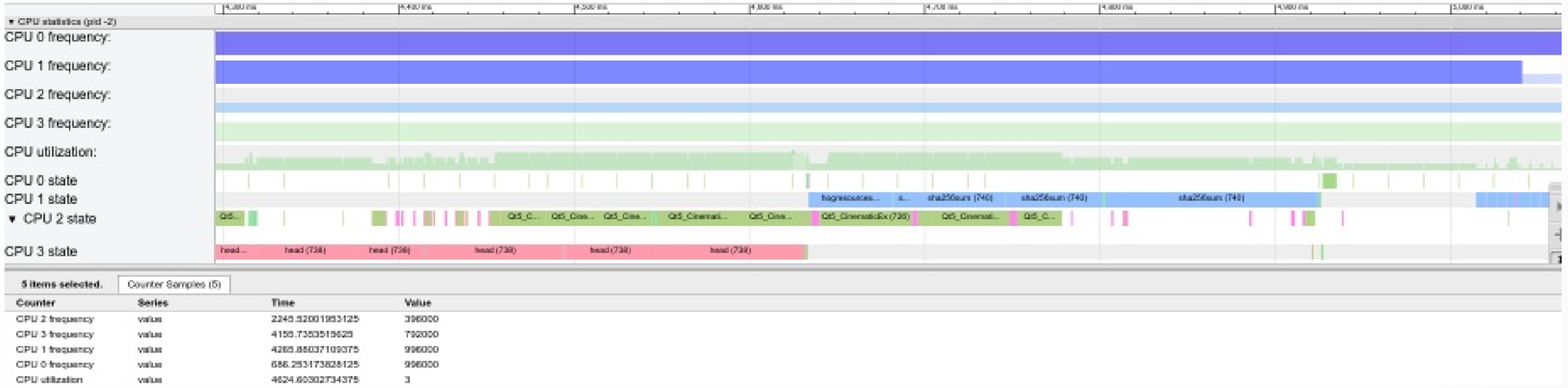


- Tracing on the block-device layer:

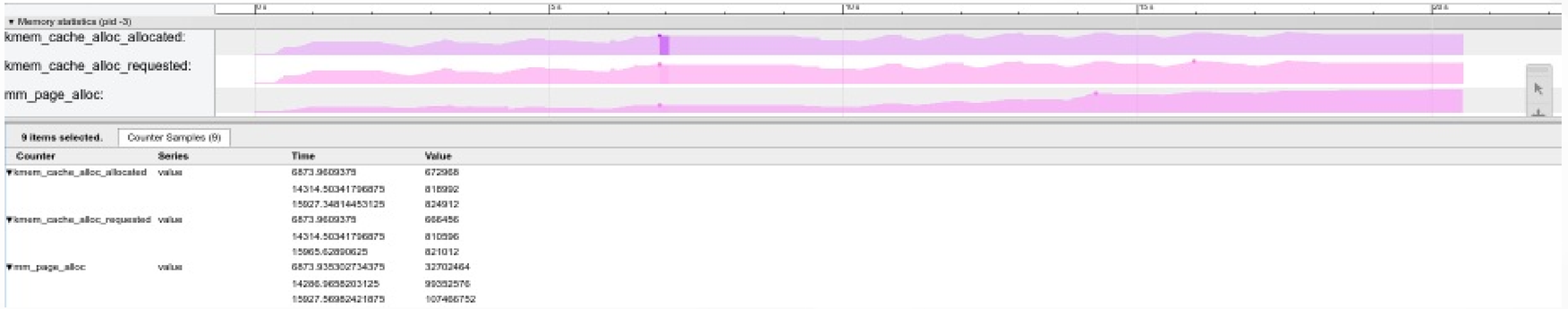


- Page faults (x86 only)
  - Also trace `mmap`, `openat` to know where the address belongs to
- Interrupts

- Deduce number of concurrently running threads/processes from `sched_switch`
- Trace CPU frequency changes



- System overall system memory consumption:



- Kernel Tracepoints
- Qt Tracepoints

- Tracing subsystem in Qt is platform agnostic
  - Currently ETW and LTTng are supported
- Relies on a new code generator in Qt: `tracegen`

qtbase/src/corelib/qtcore.tracepoints

```
1 QCoreApplicationPrivate_init_entry()
2 QCoreApplicationPrivate_init_exit()
```

qtbase/src/corelib.pro

```
1 TRACEPOINT_PROVIDER = $$PWD/qtcore.tracepoints
2 CONFIG += qt_tracepoints
```

qtbase/src/corelib/kernel/qcoreapplication.cpp

```
1  #include <qtcore_tracepoints_p.h>
2  void QCoreApplicationPrivate::init()
3  {
4      // either call entry/exit automatically:
5      Q_TRACE_SCOPE(QCoreApplicationPrivate_init);
6
7      // or manually:
8      Q_TRACE(QCoreApplicationPrivate_init_entry);
9      Q_TRACE(QCoreApplicationPrivate_init_exit);
10 }
```

Useful tracepoints in QtCore:

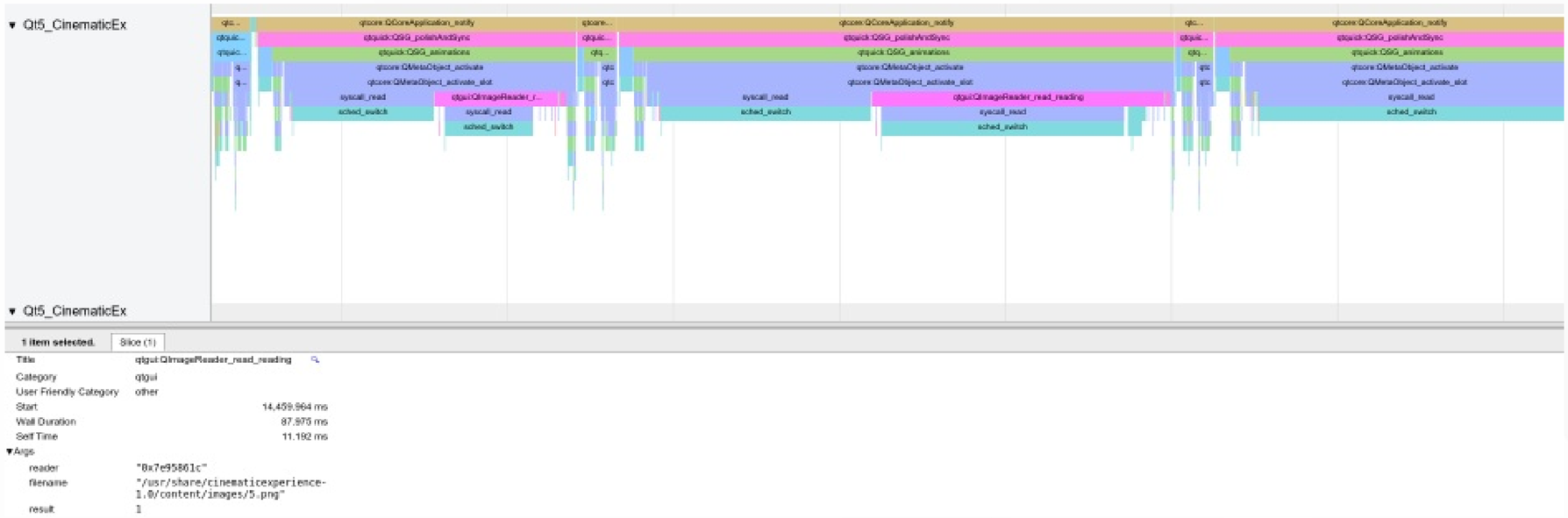
- QCoreApplication initialization
- QLibrary loading
- QDebug output
- Event handling
- Signal/Slot handling is already traced, but needs more work
  - Offline symbolication of PMF signals and PMF/lambda slots
  - Metatype tracing

Note: QString data is UTF-16 encoded

Contents of `QString` tracepoint arguments are passed as-is in UTF-16 encoding.
This data is not directly readable by babeltrace or tracecompass.

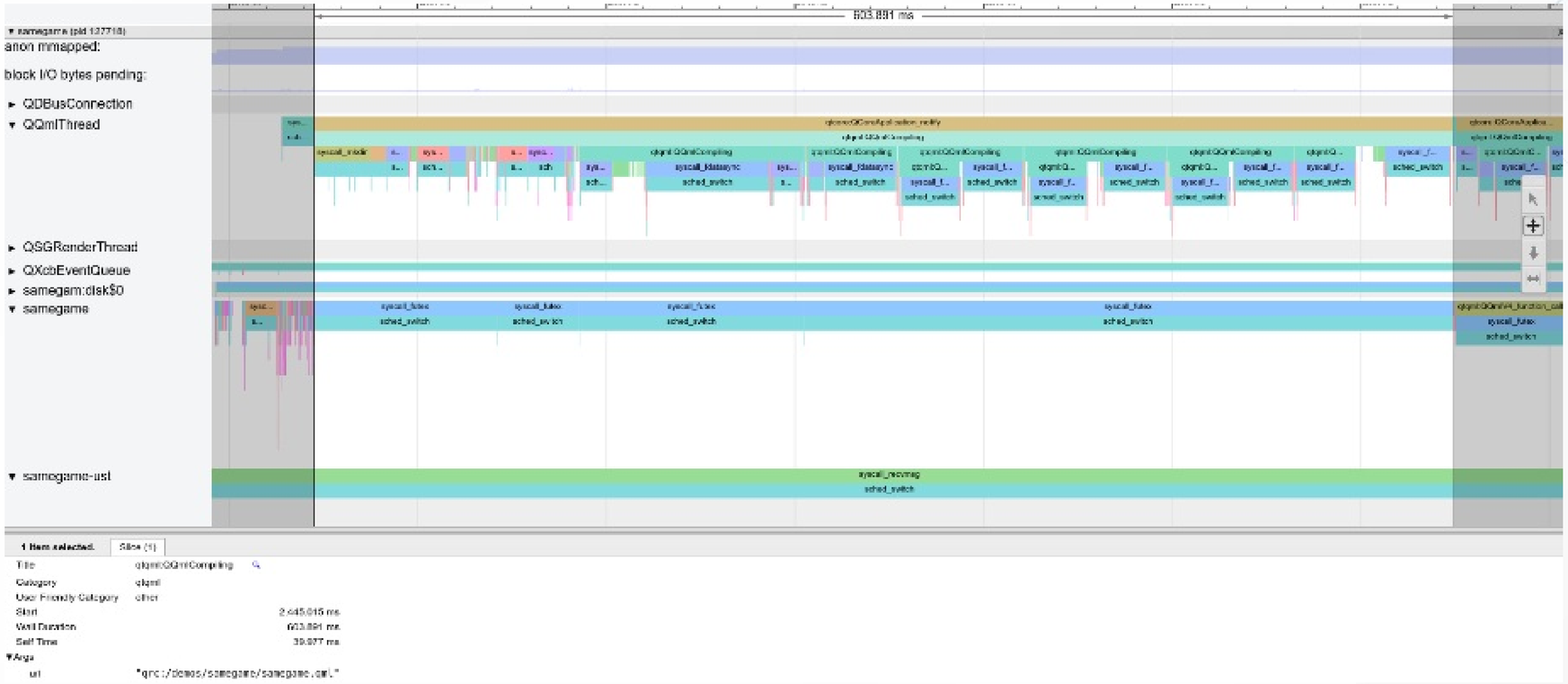
Useful tracepoints in QtGui:

- QImageReader reading
- QFontDatabase loading

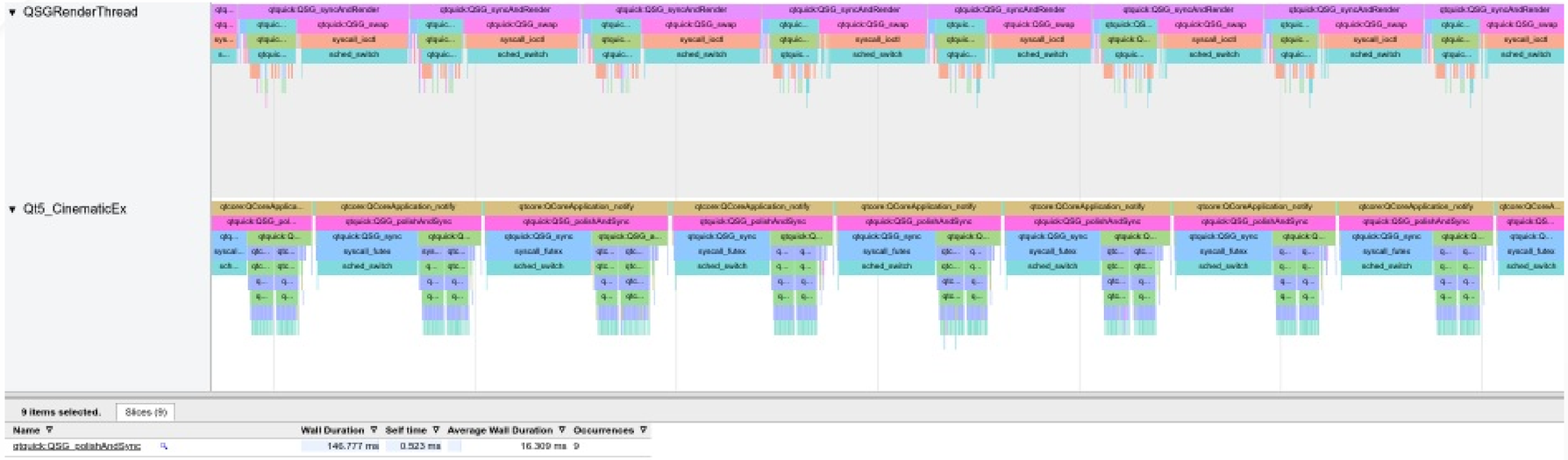


Useful tracepoints in QtQml:

- Creating QML object instances
- QML V4 functions calls
- Binding execution
- QML Compilation
- Signal Handling



QtQuick Scene Graph (https://codereview.qt-project.org/c/qt/qtdeclarative/+/277210)



Also available:

- Window and scene rendering
- Animations
- Texture uploads / atlas creation
- Distance field glyph caching

- Introduction
- Setup
- Recording
- Analysis
- Tracepoints
- **Future Outlook**

- Signal/slot symbolication
- Add more tracepoints throughout Qt
- Make `tracegen` and `Q_TRACE` API public
  - Greatly simplifies adding tracepoints for users of Qt
- Improve trace analysis support in Qt Creator
- Bring Windows/ETW support out of POC status
- Investigate tracing support for Android, macOS, iOS, ...
- Convert LTTng trace data to perfetto.dev protobuf format

Help Wanted!

Questions?

Milian Wolff

www.kdab.com

milian.wolff@kdab.com

github.com/KDAB/ctf2ctf