Extending the Performance Analysis Toolset

Christoph Sterz, KDAB
Embedded Performance

This system is different than your desktop. Time in this system is spent differently.

be scientific and measure!

2. Analyze the info to change your code with Hotspot.

3. Produce additional, different information with LTTNG Userspace Tracing in Qt.
Linux Perf

Don’t forget perf in your kernel config!

$ zgrep -E "PERF|TRACE" /proc/config.gz

[...]

# can we do perf at all?
CONFIG_HAVE_PERF_EVENTS=y
CONFIG_PERF_EVENTS=y

# can we do dwarf unwinding?
CONFIG_HAVE_PERF_REGS=y
CONFIG_HAVE_PERF_USER_STACK_DUMP=y

# do we have tracepoints?
CONFIG_FTRACE=y
CONFIG_TRACEPOINTS=y
CONFIG_SCHED_TRACER=y
CONFIG_HAVE_SYSCALL_TRACEPOINTS=y
Debug Symbols

For profiling:
Always **build Release** with Debug info!

```c
int intSum(int to)
{
    int sum = 0;
    for (int i = 0; i < to; ++i) {
        sum += i;
    }
    return sum;
}
```

# Compiled with and without `-O2`
Debug Symbols
How does the (embedded) world look like?

We don’t have space

libQt5Core.so
Debug Symbols
How does the (embedded) world look like?

We don’t have space

<table>
<thead>
<tr>
<th>libQt5Core.so</th>
<th>90M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5M</td>
</tr>
</tbody>
</table>

Separate dwarf debug Info

/debug on host
(somewhere in SDK)
Debug Symbols

How does the (embedded) world look like?

Architecture mismatch

target
ARM/MIPS

↑

x86

↓

host
Debug Symbols
How does the (embedded) world look like?

Architecture mismatch

target
ARM/MIPS

x86
host

unwind embedded architecture
SP info on the host.
Linux Perf

Does the CPU support low-overhead with PMU?

Look for “PMU” in dmesg.

→ On some ARM CPUs, only core0 is counted.

Let’s get our debug info.

target ~$ perf record --callgraph dwarf -- myapp
Now that we have our profiling info, we need to process it.
On the device, unwind result looks like this

<table>
<thead>
<tr>
<th>Samples: 1K of event 'cycles:uppp', Event count (approx.): 1524000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>36.61%</td>
</tr>
<tr>
<td>29.33%</td>
</tr>
<tr>
<td>28.41%</td>
</tr>
<tr>
<td>27.76%</td>
</tr>
<tr>
<td>23.16%</td>
</tr>
<tr>
<td>20.34%</td>
</tr>
<tr>
<td>20.14%</td>
</tr>
<tr>
<td>19.29%</td>
</tr>
<tr>
<td>15.81%</td>
</tr>
<tr>
<td>14.11%</td>
</tr>
<tr>
<td>14.11%</td>
</tr>
<tr>
<td>13.85%</td>
</tr>
<tr>
<td>13.65%</td>
</tr>
<tr>
<td>12.86%</td>
</tr>
<tr>
<td>12.20%</td>
</tr>
<tr>
<td>12.07%</td>
</tr>
<tr>
<td>12.07%</td>
</tr>
<tr>
<td>12.01%</td>
</tr>
<tr>
<td>12.01%</td>
</tr>
<tr>
<td>11.35%</td>
</tr>
<tr>
<td>11.09%</td>
</tr>
<tr>
<td>8.27%</td>
</tr>
</tbody>
</table>
On the device, unwind result looks like this

<table>
<thead>
<tr>
<th>Children</th>
<th>Self</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.00%</td>
<td>0.00%</td>
<td>settings</td>
<td>libc-2.26.so</td>
<td>[.] __libc_start_main</td>
</tr>
<tr>
<td>0.24%</td>
<td>0.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.00%</td>
<td>0.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- __libc_start_main
  - 20.54% main
    - 20.08% QCoresApplication::exec
      - QEventLoop::exec
        - 19.88% 0x5151
          QEventDispatcherGlib::processEvents
      + g_main_context_iteration
      2.30% __libc_csu_init
    - 22.77% settings
      settings

- __libc_start_main
  - 20.47% main
    - 20.01% QCoresApplication::exec
      - QEventLoop::exec
        - 19.82% 0x5151
          QEventDispatcherGlib::processEvents
      - g_main_context_iteration
        - 19.49% 0x6cf68
          g_main_context_dispatch
        - 16.93% 0x2f82d1
          QTimerInfoList::activateTimers
Hotspot

Hotspot is a FOSS Linux UI for visualizing perf results written in Qt and C++

... it uses perfparser from QtCreator

... which uses libunwind/libdw for unwinding even off-architecture!

<Demo>

99.5 % of the courtesy go to @milianw
Here is what I like most

# settings to help you unwind off-target

--debugPaths <paths>
--extraLibPaths <paths>
--appPath <paths>
--sysroot <path>
--kallsyms <path>
LTTNG: We want it inside Qt

Great source of information

Existing visualization tools, but traces are mostly kernel :(
LTNG: How is it optional?

Tracepoint providers can be combined with your app differently:

- Compile time:
  - `your app`
  - `tp provider`

- `LD_PRELOAD`
  - `your app`
  - `tp provider`
  - `dlopen()`
LTTNG in Userspace Qt

We added a tool called tracegen, which creates the tracepoints for qtbase. This is not just for LTTNG, but also for ETW.

QGuiApplicationPrivate_init_entry()
QGuiApplicationPrivate_init_exit()

QGuiApplicationPrivate_processWindowSystemEvent_entry(int type)
QGuiApplicationPrivate_processWindowSystemEvent_exit(int type)

QFontDatabase_addApplicationFont(const QString &filename)
QFontDatabase_load(const QString &family, int pointSize)
QFontDatabase_loadEngine(const QString &family, int pointSize)
QFontDatabasePrivate_addAppFont(const QString &fileName)

QImageReader_read_before_reading(QImageReader *reader, const QString &filename)
QImageReader_read_after_reading(QImageReader *reader, bool result)
#ifndef TP_QTGUI_QIMAGEREADER_READ_AFTER_READING
#define TP_QTGUI_QIMAGEREADER_READ_AFTER_READING
namespace QtPrivate {
inline void trace_QImageReader_read_after_reading(QImageReader * reader, bool result) {
    tracepoint(qtgui, QImageReader_read_after_reading, reader, result);
}
inline void do_trace_QImageReader_read_after_reading(QImageReader * reader, bool result) {
    do_tracepoint(qtgui, QImageReader_read_after_reading, reader, result);
}
inline bool trace_QImageReader_read_after_reading_enabled() {
    return tracepoint_enabled(qtgui, QImageReader_read_after_reading);
}
} // namespace QtPrivate

#ifdef TP_QTGUI_QIMAGEREADER_READ_AFTER_READING
#endif // TP_QTGUI_QIMAGEREADER_READ_AFTER_READING
LTNG in Userspace Qt

And are used like this:

```cpp
// read the image
if (Q_TRACE_ENABLED(QImageReader_read_before_reading)) {
    QString fileName = QStringLiteral("unknown");
    if (QFile *file = qobject_cast<QFile *>(d->device))
        fileName = file->fileName();
    Q_TRACE(QImageReader_read_before_reading, this, fileName);
}
const bool result = d->handler->read(image);
Q_TRACE(QImageReader_read_after_reading, this, result);
```

[...]
Tracing is now available as a preview starting from Qt 5.12.
How is LTTNG different from Perf?
Thanks!
How can I help?

christoph.sterz@kdab.com

Pictures: Wikimedia