C++ UG 06/2015

PROFILING C++ CODE ON LINUX

Milian Wolff / milianw.de / www.kdab.com

WHY DO WE CARE?

- makes our users happy
- saves energy, money, trees, battery
- room for more features
- why use C++ otherwise?

GENERAL REMARKS

- write tests
- avoid premature pessimization
- avoid premature optimization
- measure, measure, measure
- write benchmarks

LET OTHERS WORK FOR YOU

Enable optimizations *and* debug symbols
for both applications and libraries!

```
g++ -O2 -g ...
```

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

```
qmake CONFIG+=release QMAKE_CXXFLAGS+=-g
```

LATENCY NUMBERS EVERY PROGRAMMER SHOULD KNOW

```
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
Compress 1K bytes with Zippy             3,000   ns
Send 1K bytes over 1 Gbps network       10,000   ns    0.01 ms
Read 4K randomly from SSD*             150,000   ns    0.15 ms
Read 1 MB sequentially from memory     250,000   ns    0.25 ms
Round trip within same datacenter      500,000   ns    0.5  ms
Read 1 MB sequentially from SSD*     1,000,000   ns    1    ms
Disk seek                           10,000,000   ns   10    ms
Read 1 MB sequentially from disk    20,000,000   ns   20    ms
Send packet CA->Netherlands->CA    150,000,000   ns  150    ms

* Assuming ~1GB/sec SSD
```

source: Jonas Bonér, gist.github.com/jboner/2841832

TYPES OF PROFILERS

- stopwatch: time, perf
- emulation: valgrind
- sampling: VTune, perf, gdb
- tracing: perf, heaptrack

LINUX PERF

Performance analysis tools for Linux

- fast, sampling based profiling
- hardware & software counters, trace points
- works wherever Linux runs

PERF TOP

- live view of profiling data
- find bottlenecks system-wide

```
sudo perf top
```

demo time

PERF STAT

- better `time` replacement
- ideal for before/after verification

```
perf stat -r N -- DEBUGGEE ARGS
```

demo time

PERF RECORD/REPORT

- finding hot spots
- ugly ASCII UI
- can attach to running process
- has system-wide mode

```
perf record --call-graph dwarf -- DEBUGGEE ARGS
# "visualize" generated perf.data file
perf report --no-children -g graph
```

demo time

PERF TRACE

- a faster `strace` replacement
- not yet as functional

```
perf trace -S -- DEBUGGEE ARGS
```

demo time

ADVANCED PERF

- `perf list`
- `perf probe`
- `perf script`
- `perf mem`
- `perf lock`

PERF RESOURCES

- ML: linux-perf-users@vger.kernel.org
- Wiki: https://perf.wiki.kernel.org/
- Brendan Gregg's Perf Examples: http://www.brendangregg.com/perf.html

INTEL® VTUNE™

commercial perf with a good UI

- fast, sampling based wall time profiling
- Excellent visualizations, good workflow
- proprietary, costly
- **most features require Intel CPUs!**

INTEL® VTUNE™ AMPLIFIER

Profile Overview

Top Waiting Objects

This section lists the objects that spent the most time waiting in your application. Objects can wa
synchronizations. A significant amount of Wait time associated with a synchronization object refl
parallelism.

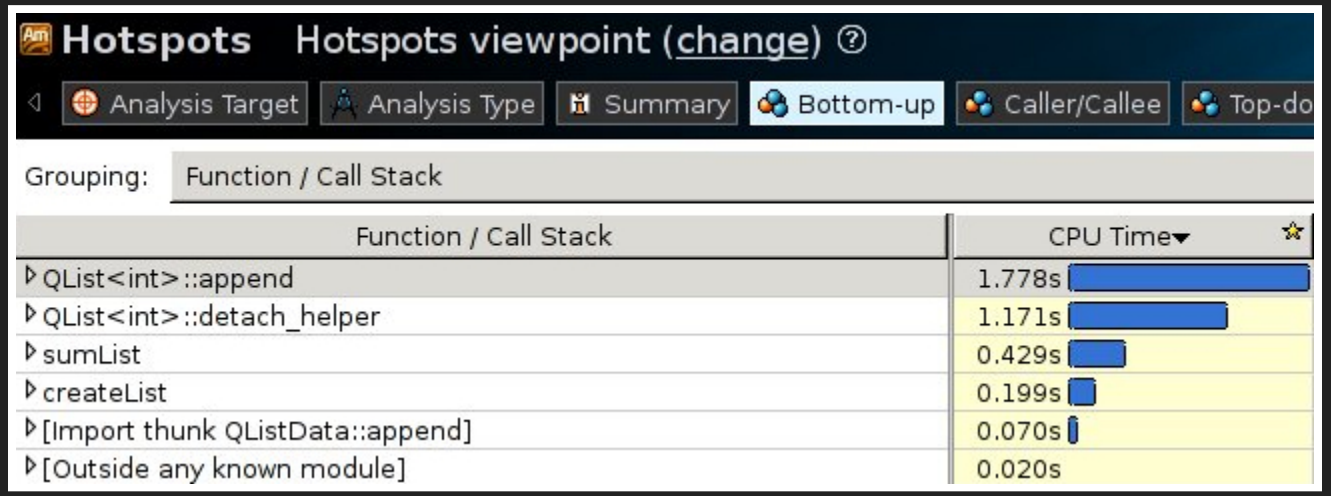
Sync Object	Wait Time	Wait Count
Sleep	13.190s	15,072
Futex 0xe9ec8d73	0.009s	2,426
poll	0.091s	260
Futex 0x6e1f8871	0.002s	257
Mutex 0xbf7f3edc	1.267s	175
[Others]	9.761s	729

Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the
simultaneously. Threads are considered running if they are either actually running on a CPU or a
Thread Concurrency is a measurement of the number of threads that were not waiting. Thread C
in the runnable state and not consuming CPU time.

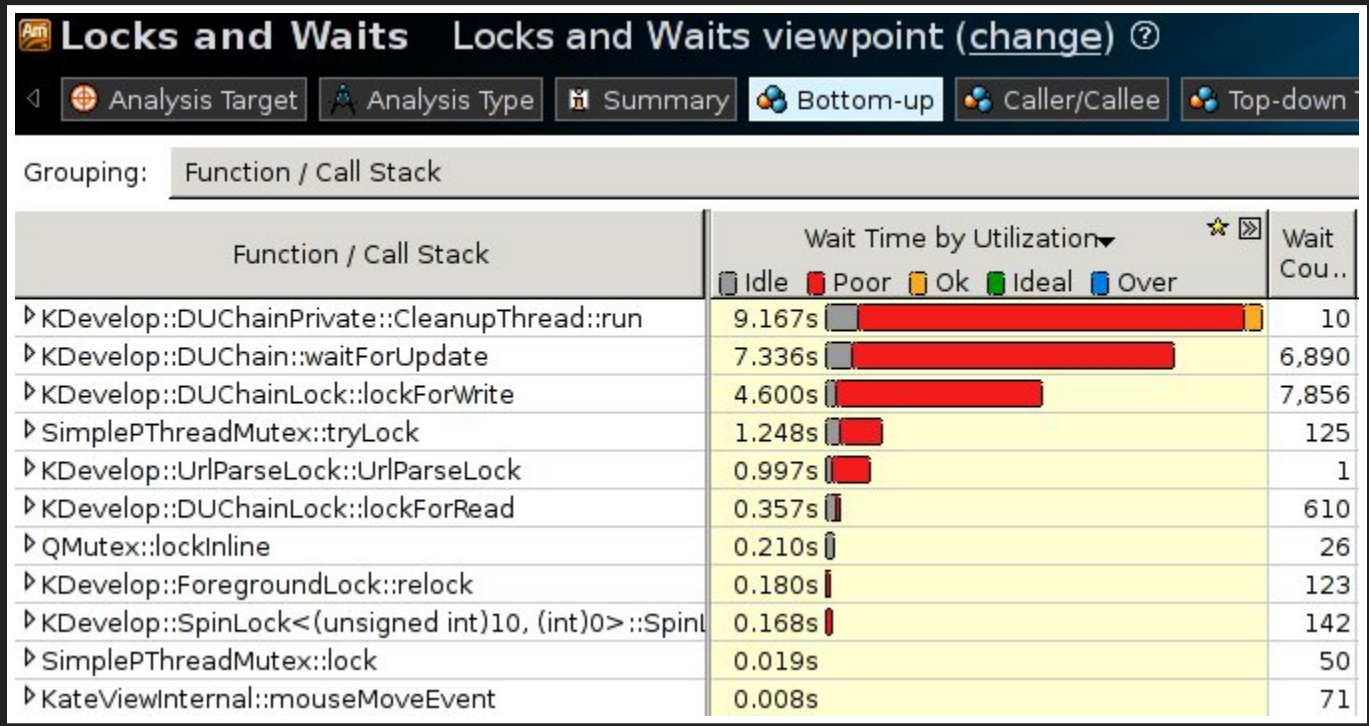
INTEL® VTUNE™ AMPLIFIER

Detecting CPU hotspots



INTEL® VTUNE™ AMPLIFIER

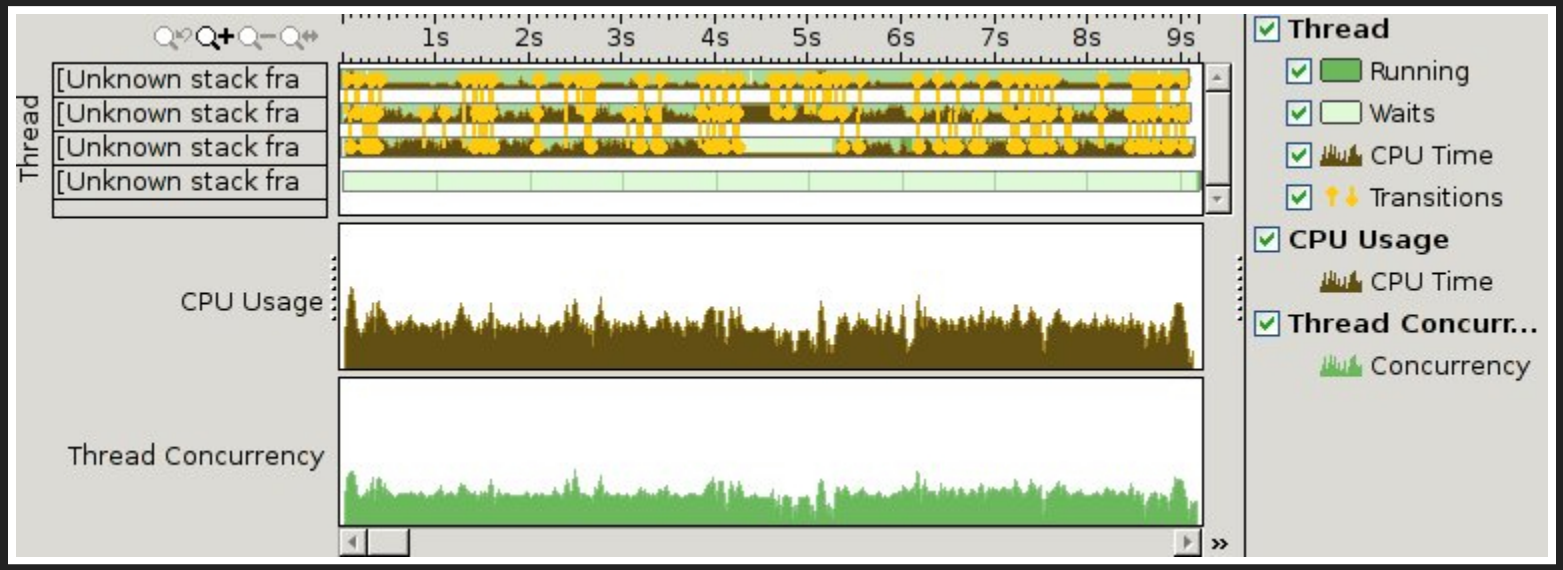
Finding Locks and Waits



Note: Not all waits are bad - an idle `QThread` will wait in the eventloop e.g.

INTEL® VTUNE™ AMPLIFIER

Per-Thread CPU Usage, Context Switches, Waits



HEAPTRACK

a heap memory profiler for Linux

```
git clone git://anongit.kde.org/heaptrack.git
```

read announcement

HEAPTRACK

- fast malloc tracer
- (peak) memory consumption
- number of allocations
- sizes of allocations
- memory leaks
- supports runtime attaching
- very hackable

RECORD HEAP USAGE

```
# start new process
heaptrack -- DEBUGGEE ARGS
# or attach to existing process
heaptrack -p $(pidof ...)
```

demo time

INTERPRET DATA

```
# plain ASCII GUI
heaptrack_print heaptrack.DEBUGGEE.PID.gz | less
# KF5 / Qt 5 GUI
heaptrack_gui heaptrack.DEBUGGEE.PID.gz
```

demo time

HEAPTRACK DEPENDENCIES

- `libunwind`: fast stack unwinding
- `3rdparty/libbacktrace`: translate instruction pointer to debug information
- `link.h, dlfcn.h`: access to ELF linker info
- `boost`: gzip compression, program options
- `C++11`: threading, hash maps, ...
- `Qt5/KF5`: GUI

HEAPTRACK INTERNALS

- `libheaptrack.a`: common tracing API
- `libheaptrack_preload.so`: trace overloaded symbols via `LD_PRELOAD`
- `libheaptrack_inject.so`: trace manually overloaded symbols after runtime injection
- `heaptrack_interpret`: out-of-process IP/DWARF translation
- `heaptrack`: shell script for connecting everything
- `heaptrack_print`: ASCII GUI
- `heaptrack_gui`: Qt5/KF5 GUI

BEYOND HEAP PROFILING

- I/O tracing
- locks & waits
- any other ideas?

QUESTIONS?

milian.wolff@kdab.com
http://milianw.de
http://www.kdab.com

What's new in C++11 / C++14?
3 day training beginning 2015-08-18
KDAB GmbH & Co. KG
Reuchlinstr. 10-11 in 10553 Berlin