IMPROVING THE PERFORMANCE OF C++ APPLICATIONS

Milian Wolff / mail@milianw.de
OUTLINE

• Motivation
• Preparations
• Tooling
  ▪ Linux Perf
  ▪ Intel® VTune™ Amplifier
  ▪ Valgrind: Callgrind & Massif
  ▪ pmap
  ▪ mallocinfo
MOTIVATION

• Faster is (perceived) better
• Run Everywhere
• Do More
PREPARATIONS
LET OTHERS WORK FOR YOUR

Enable optimizations and debug symbols.

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
# i.e. let it generate Makefiles containing:
g++ -02 -g ...
```
• Prevent regressions, keep functionality - don't overoptimize.
• Don't be afraid of (extensive?) refactoring.
• Write standalone benchmarks.
THE ROOT OF ALL EVIL

• Don't microoptimize, know the 90/10 rule.
• Beware of premature optimizations!
• Yet also keep premature pessimizations in mind.
KNOWLEDGE IS KING

- Knowing the ins and outs of your codebase and libraries allows for true optimizations.
- A better algorithm often yields more performance, than optimizing a bad alternative.
- Be aware: "Faster" code might be slower for smaller datasets.
LINUX PERF

- Fast, sampling based -Wall time profiling.
- Versatile: hardware & software counters, tracepoints
- Cross platform: works wherever Linux runs
$ sudo perf top

Samples: 10K of event 'cycles', Event count (approx.): 2036162123
10.67% libQtGui.so.4.8.5          [.] 0x00000000000230f07
10.25% libc-2.17.so               [.] __memcpy_sse3_back
2.63% libQtCore.so.4.8.5          [.] 0x00000000001a7d17
2.42% Xorg                        [.] 0x0000000003ff6f
2.05% perf                       [.] symbol_filter
1.87% libc-2.17.so                [.] __int_malloc
1.72% perf                       [.] d_print_comp.part.8
1.66% libc-2.17.so                [.] __strcmp_sse42
1.57% perf                       [.] symbols__insert
1.38% [kernel]                    [k] unix_poll
1.37% [kernel]                    [k] fget_light
1.18% libc-2.17.so                [.] __strstr_sse42
1.13% libglib-2.0.so.0.3600.3     [.] 0x000000000086e9d
...

LINUX PERF

Find system-wide hotspots.
Profile across process boundaries.
Profile individual processes, see callgraphs.

```
$ perf record -g dwarf <yourapp>
# or attach to a running app:
# perf record -g dwarf -p $(pidof <yourapp>)
$ perf report
+ 7.00%  kdevelop  libQtCore.so.4.8.4  [.] 0x0000000000000b71e5
+ 4.52%  kdevelop  libsqlite3.so.0.8.6  [.] 0x000000000000279f4
+ 2.45%  kdevelop  libkdecore.so.5.10.3  [.] 0x000000000000ad310
- 2.27%  kdevelop  libc-2.17.so  [.] _int_malloc
- _int_malloc
  - 94.29% malloc
    - 30.10% operator new(unsigned long)
    - 8.50% KDevelop::DUContext::findDeclarations(KDevelop::Identifier...
      - 72.20% CMakeProjectVisitor::createUses(CMakeFunctionDesc const&)
        - CMakeProjectVisitor::walk(QLi
          - 70.44% CMakeProjectVisitor::visit(IfAst const*)
            IfAst::accept(CMakeAstVisitor*) const
```

LINUX PERF

Gather performance counter statistics.

$ perf stat <yourapp>

Performance counter stats for '<yourapp>':

- 5606.202068 task-clock # 0.473 CPUs utilized
- 604,632 context-switches # 0.108 M/sec
- 146 cpu-migrations # 0.026 K/sec
- 1,740 page-faults # 0.310 K/sec

- 5,983,888,876 cycles # 1.067 GHz
- 3,373,405,595 stalled-cycles-frontend # 56.37% frontend cycles idle
- 2,305,588,563 stalled-cycles-backend # 38.53% backend cycles idle
- 5,558,174,058 instructions # 0.93 insns per cycle
- 22,178,629 branch-misses # 0.61 stalled cycles per insn

- 1,224,492,195 branches # 218.417 M/sec
- 11.845618581 seconds time elapsed
LINUX PERF

• Great potential to become the profiling tool on Linux.
• Custom trace points enable custom tools to be built.
• Proper UI desperately needed!
INTEL® VTUNE™ AMPLIFIER

- Fast, sampling based -Wall time profiling.
- Excellent visualizations, good workflow.
- Proprietary, but available free of charge for non-commercial Linux work.
- Most features require Intel CPUs!
Profile Overview

Top Waiting Objects
This section lists the objects that spent the most time waiting in your application. Objects can wait on synchronizations. A significant amount of wait time associated with a synchronization object reflects poor parallelism.

<table>
<thead>
<tr>
<th>Sync Object</th>
<th>Wait Time</th>
<th>Wait Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleep</td>
<td>13.190s</td>
<td>15,072</td>
</tr>
<tr>
<td>Futex 0xe9ec8d73</td>
<td>0.009s</td>
<td>2,426</td>
</tr>
<tr>
<td>poll</td>
<td>0.091s</td>
<td>260</td>
</tr>
<tr>
<td>Futex 0x6e18871</td>
<td>0.002s</td>
<td>257</td>
</tr>
<tr>
<td>Mutex 0xb7f3edc</td>
<td>1.267s</td>
<td>175</td>
</tr>
<tr>
<td>[Others]</td>
<td>9.761s</td>
<td>729</td>
</tr>
</tbody>
</table>

Thread Concurrency Histogram
This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the time that threads are consuming simultaneously. Threads are considered running if they are either actually running on a CPU or a thread context switch occurs that makes the thread runnable. Thread Concurrency is a measure of the number of threads that were not waiting. Thread Concurrency reflects the number of threads that were consuming CPU time.

- Elapsed Time:
  - 0s: Idle
  - 1s: Poor
  - 2s: Ok
  - 3s: Ideal
  - 4+ s: Over

- Simultaneously Running Threads:
  - 0: Idle
  - 1: Poor
  - 2: Ok
  - 3: Ideal
  - 4+: Over
INTEL® VTUNE™ AMPLIFIER

Detecting CPU hotspots

Hint: Don't use `QList` by default, prefer `QVector` with proper `Q_DECLARE_METATYPE` hints.
Note: Not all waits are bad - an idle QThread will wait in the eventloop e.g.

Hint: Avoid synchronous API; prefer message passing over locking
INTEL® VTUNE™ AMPLIFIER

Per-Thread CPU Usage, Context Switches, Waits

Note: Fixing KDevelop requires extensive refactoring, proper API design with threading in mind.
• **Callgrind:** deterministic instruction profiler
• **Massif:** heap profiler

Sadly large overhead, very slow

$ valgrind --tool={massif,callgrind} <yourapp>

*Hint:* For apps using a JIT compiler (i.e. via QtScript, QtWebKit, QML, QRegularExpression), add the following argument to valgrind: `--smc-check=all-non-file.`
Todo: support for perf.data files.
MASSIF VISUALIZER
PMAP

Lightweight memory tracking

$ track_memory.sh <yourapp>
$ show_memory.sh mem.log.PID
MALLOCINFO

Track malloc statistics, memory fragmentation

$ run_mallocinfo <yourapp>
# requires mallocinfo branch in massif-visualizer
$ massif-visualizer mem.log.PID
THE END

QUESTIONS? FEEDBACK?

MILIAN WOLFF / HTTP://MILIANW.DE