APPS ON SPEED
IMPROVING THE PERFORMANCE OF C++ APPLICATIONS

Milian Wolff / milian.wolff@kdab.com
OUTLINE

• Motivation
• Preparation
• Tooling
• Qt Tips & Tricks
MOTIVATION
RUN EVERYWHERE

Shiny Qt apps on low-end embedded hardware? Sure!
FASTER IS (PERCEIVED) BETTER

Increase your users productivity.
DO MORE

... in less time.
ENERGY EFFICIENCY

Save costs, trees and battery time!
PREPARATIONS
LET OTHERS WORK FOR YOU

Enable optimizations and debug symbols for both applications and libraries!

```bash
g++  -02  -g  ...
cmake   -DCMAKE_BUILD_TYPE=RelWithDebInfo
qmake  CONFIG+=release  QMAKE_CXXFLAGS+=-g
```
Cover the code to be optimized by unit tests.

- Prevent regressions.
- Keep functionality - don't overoptimize.
- Don't be afraid of (extensive?) refactoring.
BENCHMARKING

Write a benchmark for every function you optimize.

- Always do a before/after comparison.
- Minimize code coverage per benchmark.
- Choose an adequate problem size.
- QTestLib has QBENCHMARK.
THE ROOT OF ALL EVIL

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

The more you know, the better you can optimize.

- Use existing solutions, don't reinvent the wheel.
- Know your <algorithm>'s.
- Beware: "Faster" code might be slower for less data.
TOOLING
LINUX PERF

Performance analysis tools for Linux

- Fast, sampling based wall time profiling.
- Versatile: hardware & software counters, tracepoints
- Cross platform: works wherever Linux runs
# LINUX PERF: TOP

Find system-wide hotspots.

Profile across process boundaries.

$ sudo perf top

<table>
<thead>
<tr>
<th>Samples: 10K of event 'cycles', Event count (approx.): 2036162123</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.93% quickopenbench</td>
</tr>
<tr>
<td>2.38% benchhashes</td>
</tr>
<tr>
<td>2.34% libc-2.18.so</td>
</tr>
<tr>
<td>2.12% benchhashes</td>
</tr>
<tr>
<td>2.17% libc-2.18.so</td>
</tr>
<tr>
<td>2.12% benchhashes</td>
</tr>
<tr>
<td>1.96% libc-2.18.so</td>
</tr>
<tr>
<td>1.87% libkdevplatformtests.so</td>
</tr>
<tr>
<td>1.59% libQtCore.so.4.8.5</td>
</tr>
<tr>
<td>1.53% benchhashes</td>
</tr>
<tr>
<td>1.48% quickopenbench</td>
</tr>
<tr>
<td>1.33% benchhashes</td>
</tr>
<tr>
<td>1.31% libkdevplatformproject.so</td>
</tr>
<tr>
<td>1.27% benchhashes</td>
</tr>
<tr>
<td>1.03% benchhashes</td>
</tr>
</tbody>
</table>

...
Linux Perf: Record/Report

Profile individual processes, see call graphs.

```
$ perf record -g dwarf <yourapp>
# or attach to a running app:
# perf record -g dwarf -p $(pidof <yourapp>)
$ perf report

- 14.57% quickopenbench libkdevplatformproject.so [.] QVector<QString>::at(int)
- QVector<QString>::at(int) const
- 98.94% KDevelop::Path::operator<(KDevelop::Path const&) const
  - bool QMapLessThanKey<KDevelop::Path>(KDevelop::Path const&, KDevelop::Path const&)
  + 65.32% QMap<KDevelop::Path, ProjectFile>::mutableFindNode(QMapData::Node**
  + 34.68% QMap<KDevelop::Path, ProjectFile>::remove(KDevelop::Path const&)
  + 0.99% generatePathOrUrl(bool, bool, QVector<QString> const&)
- 6.82% quickopenbench libkdevplatformproject.so [.] KDevelop::Path::operator-
- KDevelop::Path::operator<(KDevelop::Path const&)
  + 99.93% bool QMapLessThanKey<KDevelop::Path>(KDevelop::Path const&, KDevelop:
  + 4.60% quickopenbench libQtCore.so.4.8.5 [.] 0x0000000000b25e8
  + 3.29% quickopenbench libkdevplatformtests.so [.] QBasicAtomicInt::ref()
  + 3.15% quickopenbench quickopenbench [.] QBasicAtomicInt::dereff()
  + 3.14% quickopenbench libQtCore.so.4.8.5 [.] QString::compare(QString
  + 2.85% quickopenbench libkdevplatformproject.so [.] QVector<QString>::size()
  + 2.52% quickopenbench libkdevplatformtests.so [.] qt_noop()
...
```
LINUX PERF: STAT

Gather performance counter statistics.

$ perf stat <yourapp>

Performance counter stats for '<yourapp>':

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5606.202068 task-clock</td>
<td># 0.473</td>
<td>CPUs utilized</td>
</tr>
<tr>
<td>604,632 context-switches</td>
<td># 0.108</td>
<td>M/sec</td>
</tr>
<tr>
<td>146 cpu-migrations</td>
<td># 0.026</td>
<td>K/sec</td>
</tr>
<tr>
<td>1,740 page-faults</td>
<td># 0.310</td>
<td>K/sec</td>
</tr>
<tr>
<td>5,983,888,876 cycles</td>
<td># 1.067</td>
<td>GHz</td>
</tr>
<tr>
<td>3,373,405,595 stalled-cycles-frontend</td>
<td># 56.37%</td>
<td>frontend cycles idle</td>
</tr>
<tr>
<td>2,305,588,563 stalled-cycles-backend</td>
<td># 38.53%</td>
<td>backend cycles idle</td>
</tr>
<tr>
<td>5,558,174,058 instructions</td>
<td># 0.93</td>
<td>insns per cycle</td>
</tr>
<tr>
<td>1,224,492,195 branches</td>
<td># 218.417</td>
<td>M/sec</td>
</tr>
<tr>
<td>22,178,629 branch-misses</td>
<td># 1.81%</td>
<td>of all branches</td>
</tr>
</tbody>
</table>

11.845618581 seconds time elapsed
LINUX PERF: QTESTLIB IN QT5

Fast, reliable benchmark counting CPU cycles.

$ yourQTestLibBench -perf

RESULT : tst_QVector::calibration():
   6,121,714 CPU cycles per iteration (total: 6,121,714, iterations: 1)
LINUX PERF

Already good, but so much potential for more!

- Custom trace points enable custom tools to be build.
- Proper UI *desperately* needed!
"Commercial perf with a good UI."

- Fast, sampling based wall time profiling.
- Excellent visualizations, good workflow.
- Proprietary
- **Free license** for non-commercial Linux work.
- 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 wait on synchronizations. A significant amount of Wait time associated with a synchronization object reflects 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 0x6e1f8871</td>
<td>0.002s</td>
<td>257</td>
</tr>
<tr>
<td>Mutex 0xbf7f3edc</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 threads spent waiting in parallel. Threads are considered running if they are not actually running on a CPU or a thread. Thread Concurrency is a measurement of the number of threads that were not waiting. Threads are considered running if they are in the runnable state and not consuming CPU time.
**Detecting CPU hotspots**

*Hint: Don’t use QList by default. Prefer QVector with Q_DECLARE_TYPEINFO(Type, Q_MOVABLE_TYPE).*
**INTEL® VTUNE™ AMPLIFIER**

**Finding Locks and Waits**

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Wait Time by Utilization</th>
<th>Wait Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>KDevelop::DUC::CleanupThread::run</td>
<td>9.167s</td>
<td>10</td>
</tr>
<tr>
<td>KDevelop::DUC::waitForUpdate</td>
<td>7.336s</td>
<td>6,890</td>
</tr>
<tr>
<td>KDevelop::DUC::lockForWrite</td>
<td>4.600s</td>
<td>7,856</td>
</tr>
<tr>
<td>SimplePThreadMutex::tryLock</td>
<td>1.248s</td>
<td>125</td>
</tr>
<tr>
<td>KDevelop::URLParseLock::URLParseLock</td>
<td>0.997s</td>
<td>1</td>
</tr>
<tr>
<td>KDevelop::DUC::lockForRead</td>
<td>0.357s</td>
<td>610</td>
</tr>
<tr>
<td>QMutex::lockInLine</td>
<td>0.210s</td>
<td>26</td>
</tr>
<tr>
<td>KDevelop::ForegroundLock::relock</td>
<td>0.180s</td>
<td>123</td>
</tr>
<tr>
<td>KDevelop::SpinLock&lt;unsigned int, int&gt;::Spin</td>
<td>0.168s</td>
<td>142</td>
</tr>
<tr>
<td>SimplePThreadMutex::lock</td>
<td>0.019s</td>
<td>50</td>
</tr>
<tr>
<td>KateViewInternal::mouseMoveEvent</td>
<td>0.008s</td>
<td>71</td>
</tr>
</tbody>
</table>

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

**Hint:** Prefer asynchronous, task-based code over locking.
**INTEL® VTUNE™ AMPLIFIER**

Per-Thread CPU Usage, Context Switches, Waits

**Hint:** Prefer asynchronous, task-based code over locking.

**Hint:** Use value semantics, don't share data between multiple threads.
VALGRIND

The developer's swiss army knife.
Available on Linux, Mac; x86 and ARM supported.

- **Callgrind**: deterministic instruction profiler
- **Massif**: heap profiler

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

Sadly large overhead, very slow

*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: Wouldn't it be nice to view perf.data files like this?

Hint: Cleanup Qt results: --fn-skip="QMetaObject::activate*" --fn-skip="QMetaObject::metacall*" --fn-skip="*:qt_metacall*" --fn-skip="*:qt_static_metacall*"
Note: Every allocation you save improves runtime performance!
PMAP

Track per-process memory consumption over time.

- Works nicely on embedded Linux systems.
- Could be ported to OS X, Windows
- Download: github.com/milianw/shell-helpers
PMAP: SINGLE PROCESS

Record and show RSS, Heap and Stack over time.

```
$ track_memory.sh <yourapp>
# wait until finished, then
$ show_memory.sh mem.log.PID
```
PMAP: SYSTEM WIDE

Record and show RSS and total memory over time.

$ system_track_memory.sh [filter apps]
$ system_show_memory.sh mem.log.system.PID
QSTRING ALLOCATIONS

```cpp
bool containsAt(const QString& source, const QString& needle, int pos) {
    return needle == source.mid(pos, needle.size());
}

// common usage example:
containsAt(longString, "myNeedle", 42);
```

Two QString allocations per call!
```
source.mid(pos, needle.size())
QString("myNeedle")
```
QSTRING ALLOCATIONS: IMPROVED

```cpp
bool containsAt(const QString& source, const QString& needle, int pos)
{
    return needle == source.midRef(pos, needle.size());
}

// common usage example:
containsAt(longString, QStringLiteral("myNeedle"), 42);
```

No runtime allocations!

Slow variant: 54.7 msecs / 153000985.4 CPU cycles
Optimized: 6.7 msecs / 19049591.6 CPU cycles
Improvement: ~87%
QSTRING PARSING

Tokenize strings containing repeated keywords.

```cpp
// common input: foo bar foo bar foo bar ...
QVector<QString> tokenize(const QString& input) {
    QVector<QString> ret;
    Tokenizer tokenizer;
    Token token;
    while(tokenizer.nextToken(&token)) {
        ret.append(input.mid(token.start, token.end));
    }
}
```

100% memory overhead.
QSTRING PARSING

Interning: Leverage implicit sharing.

```cpp
QString tokenText(const QStringRef& token)
{
    static const QVector<QString> keywords = {
        QStringLiteral("foo"),
        QStringLiteral("bar"),
        ...
    };
    auto it = std::find(keywords.begin(), keywords.end(), token);
    if (it != keywords.end()) {
        // hot path: no string allocation, and we share memory
        return *it;
    } else {
        // slow path
        return token.toString();
    }
}
```

Slow variant: 198.3 msecs
Optimized: 153.9 msecs
Improvement: ~22%, no memory overhead (keywords are shared)
THREAD-SAFE MESSAGE PASSING

Use Qt's Signal/Slots for efficient message passing.

class Worker : public QObject {
    Q_OBJECT
public:
    Worker(QThread *thread) {
        moveToThread(thread);
        connect(thread, &QThread::finished, this, &QObject::deleteLater);
    }
    void work(const QString& input) { // thread safe dispatching
        QMetaObject::invokeMethod(this, "workInternal", Q_ARG(QString, input));
    }
private slots:
    // only called from background thread
    void workInternal(const QString& input) {
        const QString result = doExpensiveStuff(input);
        emit finished(result); // will be queued and sent to listeners
    }
signals:
    void finished(const QString& result);
};
THE END

QUESTIONS? FEEDBACK?

MILIAN WOLFF / WWW.KDAB.COM