Measuring and Analysing Performance
@GernotHeiser
Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/4.0/legalcode
Today’s Lecture

- Principles of performance evaluation: why and how
- Benchmarking: assessing performance (how and how not)
- Profiling
- Performance analysis
- Understanding performance (establishing context)
Why Measure Performance?

• System performance is important in many cases
• Good performance is expected from systems
• **Important:** Don’t guess, measure!
  • don’t rely on models/assumptions/hearsay
  • validate your (performance) model of the system

Models are important, but you need to confirm that your system behaves according to the model!
Performance Considerations

What is performance?
• Is there an absolute measure
• Is there a baseline for relative comparison?

What are we comparing?
• Best case? Nice, but useful?
• Average case? What defines “average”?
• Expected case? What defines it?
• Worst case? Is it really “worst” or just “bad”?

Configuration matters:
• Hot cache – easy to do – or cold cache?
• What is most relevant for the purpose?
Benchmarking
Lies, Damned Lies, Benchmarks

Considerations:
- Micro- vs macro-benchmarks
- Benchmark suites, use of subsets
- Completeness of results
- Significance of results
- Baseline for comparison
- Benchmarking ethics
- What is good? — Analysing the results
Benchmarking in Research & Development

Must satisfy two criteria:

• Conservative: no significant degradation due to your work
• Progressive: actual & relevant performance improvement
  • only needed if your work is actually about improving performance

Must analyse and explain results!

• Discuss model of system
• Present hypothesis of behaviour
• Results must test and confirm hypothesis

Objectivity and fairness:
• Appropriate baseline
• Fairly evaluate alternatives
Micro- vs Macro-Benchmarks

Microbenchmark
• Exercise particular operation

Macrobenchmark
• Use realistic workload
• Aim to represent real-system perf

Micro-BMs are an analysis, not an assessment tool!
• drill down on performance

Benchmarking crime: Using micro-benchmarks only
Standard vs Ad-Hoc Benchmarks

• Standard benchmarks are designed by experts
  • Representative workloads, reproducible and comparable results
  • Use them whenever possible!
  • Examples: SPEC, EEMBC, YCSB, ...

• Only use ad-hoc benchmarks when you have no choice
  • no suitable standard
  • limitations of experimental system

Ad-hoc benchmarks reduce reproducibility and generality – need strong justification!
## Obtaining an Overall Score for a BM Suite

Arithmetic mean is meaningless for relative numbers

**Rule:** *arithmetic* mean for *raw* numbers, *geometric* mean for *normalised!* [Fleming & Wallace, ‘86]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>System X</th>
<th></th>
<th>System Y</th>
<th></th>
<th>System Z</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abs</td>
<td>Rel</td>
<td>Abs</td>
<td>Rel</td>
<td>Abs</td>
<td>Rel</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>1.00</td>
<td>10</td>
<td>0.50</td>
<td>40</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>1.00</td>
<td>80</td>
<td>2.00</td>
<td>20</td>
<td>0.50</td>
</tr>
<tr>
<td>Geom. mean</td>
<td></td>
<td>1.00</td>
<td></td>
<td>1.00</td>
<td></td>
<td>1.00</td>
</tr>
</tbody>
</table>

Normalise to System X

Normalise to System Y

Geometric mean?

Does the mean make sense?

Invariant under normalisation!
Benchmark Suite Abuse

“We evaluate performance using SPEC CPU2000. Fig 5 shows typical results.”

Benchmarking crime: Using a subset of a suite

Subsetting introduces bias, makes score meaningless!

Sometimes unavoidable (incomplete system) – treat with care, and justify well!

Results will have limited validity
Beware Partial Data

Frequently seen: Measurements show 10% throughput degradation. Authors conclude “10% overhead”.

Consider:
1. 100 Mb/s, 100% CPU → 90 Mb/s, 100% CPU
2. 100 Mb/s, 20% CPU → 90 MB/s, 40% CPU

Proper figure of merit is processing cost per unit data
1. 10 µs/kb → 11 µs/kb: 10% overhead
2. 2 µs/kb → 4.4 µs/kb: 120% overhead

Benchmarking crime: Throughput degradation = overhead!
Profiling
Profiling

Run-time collection of execution statistics
• invasive (requires some degree of instrumentation)
• therefore affects the execution it's trying to analyse
• good profiling approaches minimise this interference

Avoid with HW debuggers, cycle-accurate simulators

Identify targets for performance tuning – complementary to microbenchmarks

gprof:
• compiles tracing code into program
• uses statistical sampling with post-execution analysis
Example gprof output

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self seconds</th>
<th>total seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.34</td>
<td>0.02</td>
<td>0.02</td>
<td>7208</td>
<td>0.00</td>
<td>0.00</td>
<td>open</td>
</tr>
<tr>
<td>16.67</td>
<td>0.03</td>
<td>0.01</td>
<td>244</td>
<td>0.04</td>
<td>0.12</td>
<td>offtime</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.01</td>
<td>8</td>
<td>1.25</td>
<td>1.25</td>
<td>memccpy</td>
</tr>
<tr>
<td>16.67</td>
<td>0.05</td>
<td>0.01</td>
<td>7</td>
<td>1.43</td>
<td>1.43</td>
<td>write</td>
</tr>
<tr>
<td>16.67</td>
<td>0.06</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
<td>mcount</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>236</td>
<td>0.00</td>
<td>0.00</td>
<td>tzset</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>192</td>
<td>0.00</td>
<td>0.00</td>
<td>tolower</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>47</td>
<td>0.00</td>
<td>0.00</td>
<td>strlen</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>45</td>
<td>0.00</td>
<td>0.00</td>
<td>strchr</td>
</tr>
</tbody>
</table>

Source: http://sourceware.org/binutils/docs-2.19/gprof
Example gprof output

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td></td>
<td>start [1]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.05</td>
<td>1/1</td>
<td>main</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td>on_exit</td>
<td>[28]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td>exit</td>
<td>[59]</td>
</tr>
</tbody>
</table>

0.00    0.00       1/1           exit [59]

-----------------------------------------------
0.00    0.05       1/1           start [1]
[2]    100.0  0.00  0.05     1         main [2]
0.00   0.05      1/1           report [3]

-----------------------------------------------
0.00    0.05     1/1         main [2]
[3]    100.0  0.00  0.05     1         report [3]
0.00   0.03     8/8         timelocal [6]
```
Performance Monitoring Unit (PMU)

- Collects certain *events* at run time
- Supports many *events*, small number of *event counters*
  - Events refer to hardware (micro-architectural) features
    - Typically relating to instruction pipeline or memory hierarchy
    - Dozens or hundreds
- Counter can be bound to a particular event
  - via some configuration register, typically 2–4
- Counters can trigger exception on exceeding threshold
- OS can sample counters

Linux PMU interface: *oprof*
Can profile kernel and userland
Example oprof Output

$ opreport --exclude-dependent
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a ...

```
450385  75.6634  cclplus
  60213  10.1156  lyx
  29313  4.9245  XFree86
   11633  1.9543  as
  10204  1.7142  oprofiled
   7289  1.2245  vmlinux
   7066  1.1871  bash
  6417  1.0780  oprofile
   6397  1.0747  vim
  3027  0.5085  wineserver
  1165  0.1957  kdeinit
```

Profiler

Source: http://oprofile.sourceforge.net/examples/
Example ofprof Output

$ oreport
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a ...
   506605 54.0125 cc1plus
       450385 88.9026 cc1plus
       28201 5.5667 libc-2.3.2.so
   27194 5.3679 vmlinux
       677 0.1336 uhci_hcd
...
   163209 17.4008 lyx
       60213 36.8932 lyx
   23881 14.6322 libc-2.3.2.so
  21968 13.4600 libstdc++.so.5.0.1
   13676 8.3794 libpthread-0.10.so
## PMU Event Examples: ARM11 (Armv6)

<table>
<thead>
<tr>
<th>Ev #</th>
<th>Definition</th>
<th>Ev #</th>
<th>Definition</th>
<th>Ev #</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>I-cache miss</td>
<td>0x0b</td>
<td>D-cache miss</td>
<td>0x22</td>
<td>...</td>
</tr>
<tr>
<td>0x01</td>
<td>Instr. buffer stall</td>
<td>0x0c</td>
<td>D-cache write-back</td>
<td>0x23</td>
<td>Funct. call</td>
</tr>
<tr>
<td>0x02</td>
<td>Data depend. stall</td>
<td>0x0d</td>
<td>PC changed by SW</td>
<td>0x24</td>
<td>Funct. return</td>
</tr>
<tr>
<td>0x03</td>
<td>Instr. micro-TLB miss</td>
<td>0x0f</td>
<td>Main TLB miss</td>
<td>0x25</td>
<td>Funct. ret. predict</td>
</tr>
<tr>
<td>0x04</td>
<td>Data micro-TLB miss</td>
<td>0x10</td>
<td>Ext data access</td>
<td>0x26</td>
<td>Funct. ret. mispred.</td>
</tr>
<tr>
<td>0x05</td>
<td>Branch executed</td>
<td>0x11</td>
<td>Load-store unit stall</td>
<td>0x30</td>
<td>...</td>
</tr>
<tr>
<td>0x06</td>
<td>Branch mis-predicted</td>
<td>0x12</td>
<td>Write-buffer drained</td>
<td>0x38</td>
<td>...</td>
</tr>
<tr>
<td>0x07</td>
<td>Instr. executed</td>
<td>0x13</td>
<td>Cycles FIRQ disabled</td>
<td>0xff</td>
<td>Cycle counter</td>
</tr>
<tr>
<td>0x09</td>
<td>D-cache acc. cacheable</td>
<td>0x14</td>
<td>Cycles IRQ disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0a</td>
<td>D-cache access any</td>
<td>0x20</td>
<td>...</td>
<td></td>
<td>Developer’s best friend!</td>
</tr>
</tbody>
</table>

**seL4 profiler in the works!**
Performance Analysis
Significance of Measurements

• Standard approach: repeat & collect stats
• Computer systems are high deterministic
  • Usually variances are tiny, except across WAN

All measurements are subject to random errors

Watch for divergence from this hypothesis, could indicate hidden parameters!

Benchmarking crime: No indication of significance of data!

Always show standard deviations, or clearly state they are tiny!
How to Measure and Compare Performance

Bare-minimum statistics:

• At least report the mean (µ) and standard deviation (σ)
  • Don't believe any effect that is less than a standard deviation
    • 10.2±1.5 is not significantly different from 11.5
    • Be highly suspicious if it is less than two standard deviations
      • 10.2±0.8 may not be different from 11.5

For systems work, must be very suspicious if σ is not small!

Standard deviation is meaning-less for small samples!
  • Ok if effect ≫ σ
  • use t-test if in doubt!
Example from SPEC CPU2000

Observations:
• First iteration is special
• 20 Hz timer: accuracy 0.1 s!

Lesson: Need mental model of system, look for hidden parameters if model fails!
How To Measure and Compare Performance

Noisy data:

• Eliminate sources of noise, re-run from same initial state
  • single-user mode
  • dedicated network

• Possible ways out:
  • ignore highest & lowest values
  • ignore above threshold in bi-modal distribution resulting from interference
  • take floor of data
    • maybe minimum is what matters

Not always possible!

Proceed with extreme care!
• Document and justify!
Real-World Example: seL4 Syscall Latency

Interference from test rig

<table>
<thead>
<tr>
<th>Syscall (cy)</th>
<th>Min</th>
<th>Mean</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>120</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>IPC Call</td>
<td>313</td>
<td>314</td>
<td>1</td>
</tr>
<tr>
<td>Signal→low</td>
<td>139</td>
<td>139</td>
<td>0</td>
</tr>
<tr>
<td>Signal→high</td>
<td>377</td>
<td>486</td>
<td>298</td>
</tr>
</tbody>
</table>

Platform: Sabre (Armv7-a Cortex-A9)

Real syscall cost: 377 cy

Signalling a Notification

Courtesy Shane Kadish
Problem: Benchmarking Methodology

```c
// Write stalls on platform with low memory bandwidth!

t0 = time();
for (i=0; i<n; i++) {
    syscall(...)
    t1 = time();
    buffer[i] = t1 - t0;
    t0 = t1;
}
/* now compute mean, std deviation ... */
...

// All data in registers!

t0 = time();
for (i=0; i<n; i++) {
    syscall(...)
    t1 = time();
t = t1 - t0;
sum_t += t;
sum_sq += t * t;
t0 = t1;
}
/* now compute mean, std deviation ... */
mean = sum_t / n;
stddev = sqrt((n * sum_sq - sum_s) / (n * (n-1)));
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>709</td>
<td>1770</td>
<td>933</td>
<td>195</td>
</tr>
<tr>
<td>Sum in loop</td>
<td>695</td>
<td>770</td>
<td>730</td>
<td>15</td>
</tr>
</tbody>
</table>

Platform: Sabre
different syscall!

Courtesy Nataliya Korovkina
How To Measure and Compare Performance

Vary inputs, check outputs!

- Vary data *and* addresses!
  - eg time-stamp or randomise inputs
  - be careful with sequential patterns!

- Check outputs are correct
  - read back after writing and compare

- Complete checking infeasible?
  - do spot checks
  - run with checking on/off

Beware optimisations!

- compilers eliminating code
- disks pre-fetching, de-duplicating

• True randomness may affect reproducibility
  • Use speudo-random with same seed
Real-World Example: SPEC on Linux

Benchmark:
• *300.twolf* from SPEC CPU2000 suite

Platform:
• Dell Latitude D600
  • Pentium M @ 1.8GHz
  • 32KiB L1 cache, 8-way
  • 1MiB L2 cache, 8-way
  • DDR memory @ effective 266MHz
• Linux kernel version 2.6.24

Methodology:
• Multiple identical runs for statistics...
20% performance difference between “identical” runs!

Subtract 221 cycles (123ns) for each L2-cache miss

Lesson: Check system behaves according to your model – large $\sigma$ was the giveaway!
A Few More Performance Evaluation Rules

• Vary one parameter at a time
• Record & date all configurations!
• Measure as directly as possible
• Avoid incorrect conclusions from pathological data
  • sequential vs random access may mess with prefetching
  • $2^n$ vs $2^{n-1}$, $2^{n+1}$ sizes may mess with caching

What is pathological depends a lot on circumstances!
Most Important: Use a Model/Hypothesis

Model of the system that predicts system behaviour

• Benchmarking should aim to support or disprove that model
• Need to consider in selecting data, evaluating results, e.g:
  • I/O performance dependent on FS layout, caching in controller...
  • Cache sizes (HW & SW caches)
  • Buffer sizes vs cache size

Always check your system behaves according to the model!
Example: Memory Copy

Pipelining, loop overhead

Hypothesis: Execution time vs buffer size?

L1 cache (32KiB)

L2 cache (1MiB)

Make sure you understand all results!
Loop and Timing Overhead

• Ensure measurement overhead does not affect results!
• Eliminate by measuring in tight loop, subtract timer cost

```c
int MAX = 1000000;
t0 = time();
for (i=0; i<MAX; i++) {
    asm(nop); 
}
t1 = time();
for (i=0; i<MAX; i++) {
    asm(syscall);
}
t2 = time();
printf("Cost is %dus\n", (t2-2*t1+t0)*1000000/MAX);
```

Beware compiler optimisations!
Relative vs Absolute Data

From a real paper [Armand&Gien, IEEE CCNC’09]:

• No data other than this figure
• No figure caption
• Only explanation in text:
  “The L4 overhead compared to VLX ranges from a 2x to 20x factor depending on the Linux system call benchmark”
• No definition of “overhead factor”
• No native Linux data

Benchmarking crime: Relative numbers only!
Data Range

Example: Scaling database load

Seems to scale well?

Looking a bit further

Benchmarking crime: Selective data set hiding deficiencies!
Benchmarking Ethics

Comparisons with prior work

• Sensible and necessary, but must be fair!
  • Comparable setup/equipment
  • Prior work might have different focus, must understand & acknowledge
    • eg they optimised for multicore scalability, you for mobile-system energy
  • Ensure you choose appropriate configuration
  • Make sure you understand what’s going on!

Benchmarking crime: Unfair benchmarking of competitor!
Other Ways of Cheating with Benchmarks

- Benchmark-specific optimisations
  - Recognise particular benchmark, insert BM-specific optimised code
  - Popular with compiler writers
  - Pioneered for smartphone performance by Samsung
    [http://bgr.com/2014/03/05/samsung-benchmark-cheating-ends](http://bgr.com/2014/03/05/samsung-benchmark-cheating-ends)

- Benchmarking simulated system
  - … with simulation simplifications matching model assumptions

- Uniprocessor benchmarks to “measure” multicore scalability
  - … by running multiple copies of benchmark on different cores

- CPU-intensive benchmark to “measure” networking performance
  These are simply lies, and I’ve seen them all!
Understanding Performance
What is “Good” Performance?

• Easy if improving recognised state of the art
  • E.g. improving best Linux performance (where optimised)

• Harder if no established best-of-class baseline:
  • Evaluate best-of-breed system yourself
  • Establish performance limits
    • Theoretical optimal scenario
    • Hardware-imposed performance limits

Remember: progressive and conservative criteria!
Remember: BM ethics!
Most elegant, but hardest!
Real-World Example: Virtualisation Overhead

Symbian null-syscall microbenchmark:
• Native: 0.24µs, virtualized (on OKL4): 0.79µs
  • 230% overhead

• ARM11 processor runs at 368 MHz:
  • Native: 0.24µs = 93 cy
  • Virtualized: 0.79µs = 292 cy
  • Overhead: 0.55µs = 199 cy
  • Cache-miss penalty ≈ 20 cy

• Model:
  • native: 2 mode switches, 0 context switches, 1 × save+restore state
  • virt.: 4 mode switches, 2 context switches, 3 × save+restore state
Performance Counters Are Your Friends!

<table>
<thead>
<tr>
<th>Counter</th>
<th>Native</th>
<th>Virtualized</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch miss-pred</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D-cache miss</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I-cache miss</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D-μTLB miss</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I-μTLB miss</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Main-TLB miss</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Instructions</td>
<td>30</td>
<td>125</td>
<td>95</td>
</tr>
<tr>
<td>D-stall cycles</td>
<td>0</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>I-stall cycles</td>
<td>0</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Total Cycles</td>
<td>93</td>
<td>292</td>
<td>199</td>
</tr>
</tbody>
</table>
More of the Same

First step: improve representation!

Second step: overheads in appropriate units!

Further Analysis shows guest dis- & enables IRQs 22 times!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td>1.63</td>
<td>2.25</td>
<td>0.62</td>
<td>230</td>
<td>1</td>
<td>230</td>
</tr>
<tr>
<td>[1/s]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create/close</td>
<td>11</td>
<td>15</td>
<td>4</td>
<td>1472</td>
<td>2</td>
<td>736</td>
</tr>
<tr>
<td>[µs]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suspend</td>
<td>0.81</td>
<td>1.54</td>
<td>0.73</td>
<td>269</td>
<td>1</td>
<td>269</td>
</tr>
<tr>
<td>[10ns]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And Another One…

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native [µs]</th>
<th>Virt. [µs]</th>
<th>Overhead</th>
<th>Per tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDes16_Num0</td>
<td>1.2900</td>
<td>1.2936</td>
<td>0.28%</td>
<td>2.8 µs</td>
</tr>
<tr>
<td>TDes16_RadixHex1</td>
<td>0.7110</td>
<td>0.7129</td>
<td>0.27%</td>
<td>2.7 µs</td>
</tr>
<tr>
<td>TDes16_RadixDecimal2</td>
<td>1.2338</td>
<td>1.2373</td>
<td>0.28%</td>
<td>2.8 µs</td>
</tr>
<tr>
<td>TDes16_Num_RadixOctal3</td>
<td>0.6306</td>
<td>0.6324</td>
<td>0.28%</td>
<td>2.8 µs</td>
</tr>
<tr>
<td>TDes16_Num_RadixBinary4</td>
<td>1.0088</td>
<td>1.0116</td>
<td>0.27%</td>
<td>2.7 µs</td>
</tr>
<tr>
<td>TDesC16_Compare5</td>
<td>0.9621</td>
<td>0.9647</td>
<td>0.27%</td>
<td>2.7 µs</td>
</tr>
<tr>
<td>TDesC16_CompareF7</td>
<td>1.9392</td>
<td>1.9444</td>
<td>0.27%</td>
<td>2.7 µs</td>
</tr>
<tr>
<td>TdesC16_MatchF9</td>
<td>1.1060</td>
<td>1.1090</td>
<td>0.27%</td>
<td>2.7 µs</td>
</tr>
</tbody>
</table>

Good or bad?

Timer interrupt virtualization overhead!
Lessons Learned

• Ensure stable results
  • Get small variances, investigate if they are not

• Have a model of what to expect
  • Investigate if behaviour is different
  • Unexplained effects are likely to indications of problems – don't ignore!

• Tools are your friends
  • Performance counters
  • Simulators
  • Traces
  • Spreadsheets

Annotated list of benchmarking crimes: https://gernot-heiser.org/benchmarking-crimes.html