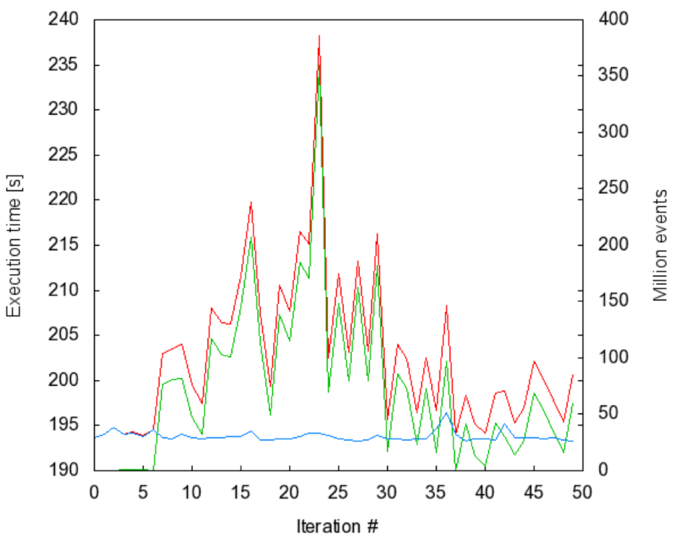
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2024 T3 Week 04 Part 1

Measuring and Analysing Performance

@GernotHeiser



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?

Engage brain first!

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

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

Macrobenchmark

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

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

Normalise to System X

Normalise to System Y

Benchmark	System X		System Y		System Z	
	Abs	Rel	Abs	Rel	Abs	Rel
1	20	1.00	10	0.50	40	2.00
2	40	1.00	80	2.00	20	0.50
Geom. mean		**1.00**		**1.00**		**1.00**

Does the mean make sense?

Geometric mean?

Invariant under normalisation!

Arithmetic mean is meaningless for relative numbers

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

Benchmark Suite Abuse

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

Subsetting introduces bias, makes score meaningless!

Benchmarking crime: Using a subset of a suite

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".

What degrades throughput?

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

CPU limited

Latency limited

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.

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

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
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                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 cc1plus
    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
```

Performance counter used

Count

Percentage

Profiler

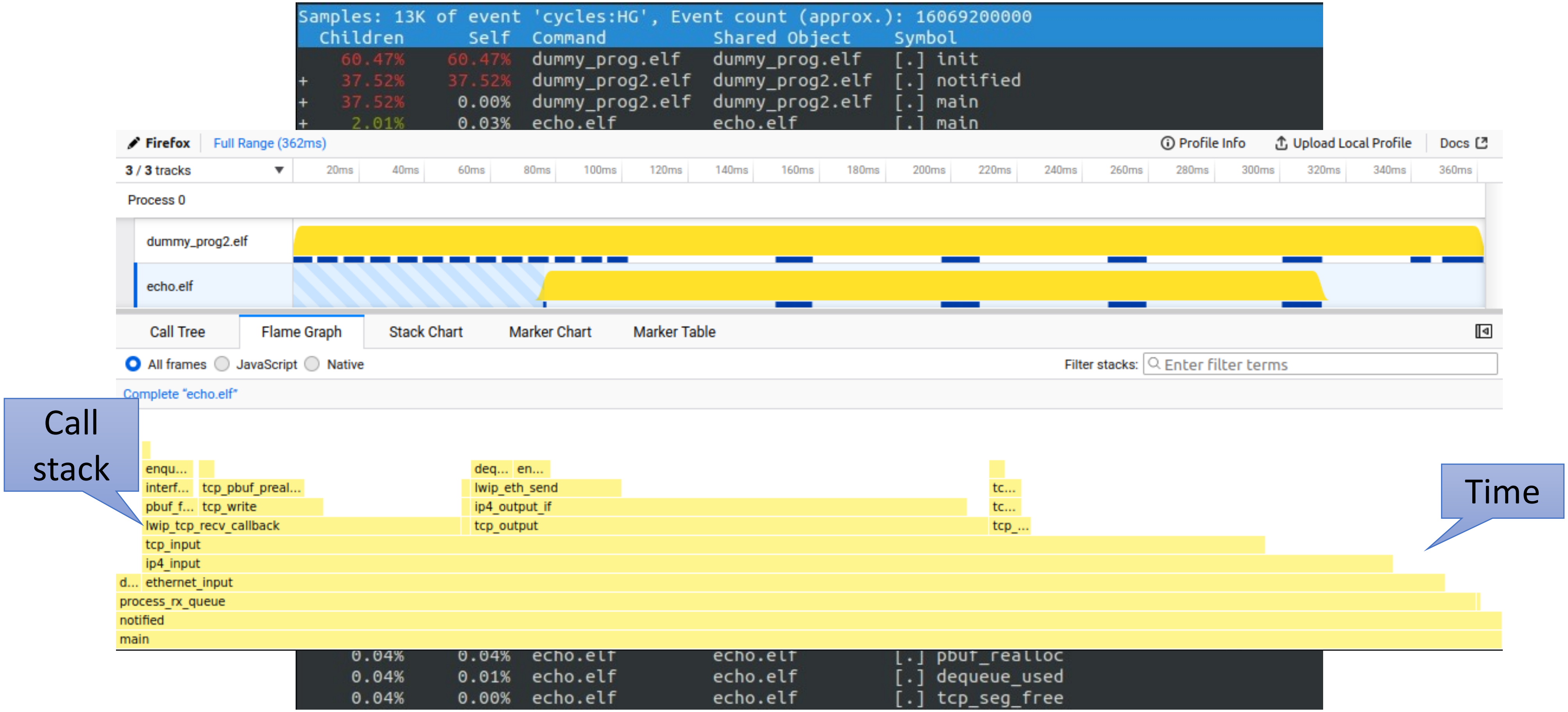
Source: http://oprofile.sourceforge.net/examples/

Example oprof Output

```
$ opreport
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
```

Drill down of top consumers

Now Available on seL4 Microkit/LionsOS!



PMU Event Examples: ARM11 (Armv6)

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

Developer's best friend!

Performance Analysis

Significance of Measurements

All measurements are subject to random errors

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

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
    - often don't have a Gaussian distribution
    - 10.2±0.8 may not be significantly different from 11.5

Standard deviation is meaning-less for small samples!

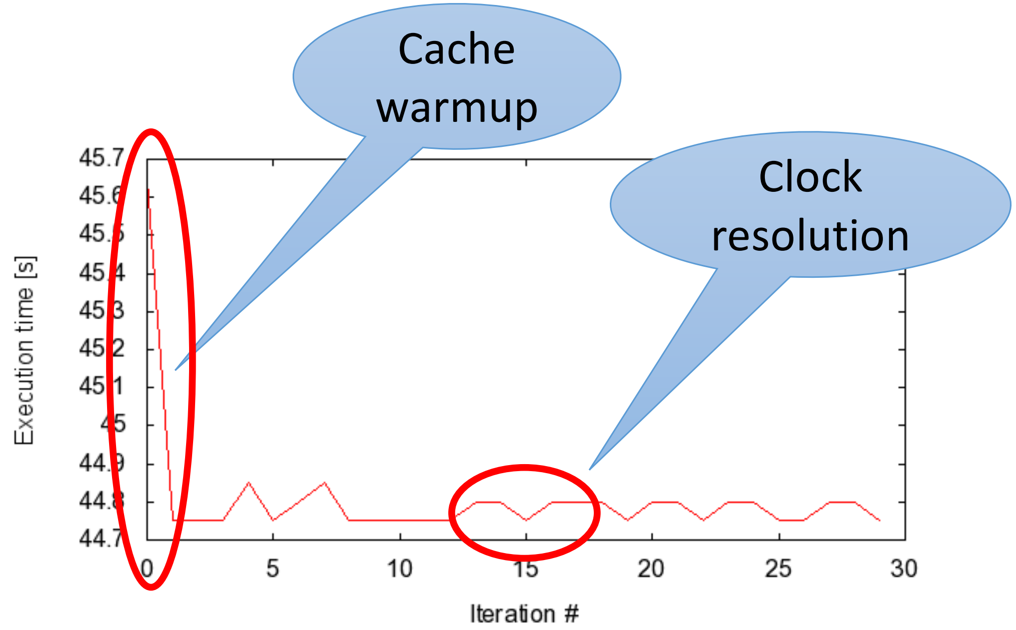
- Ok if effect ≫ σ
- use t-test if in doubt!

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

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:

Not always possible!

- 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

- Proceed with extreme care!
- Document and justify!

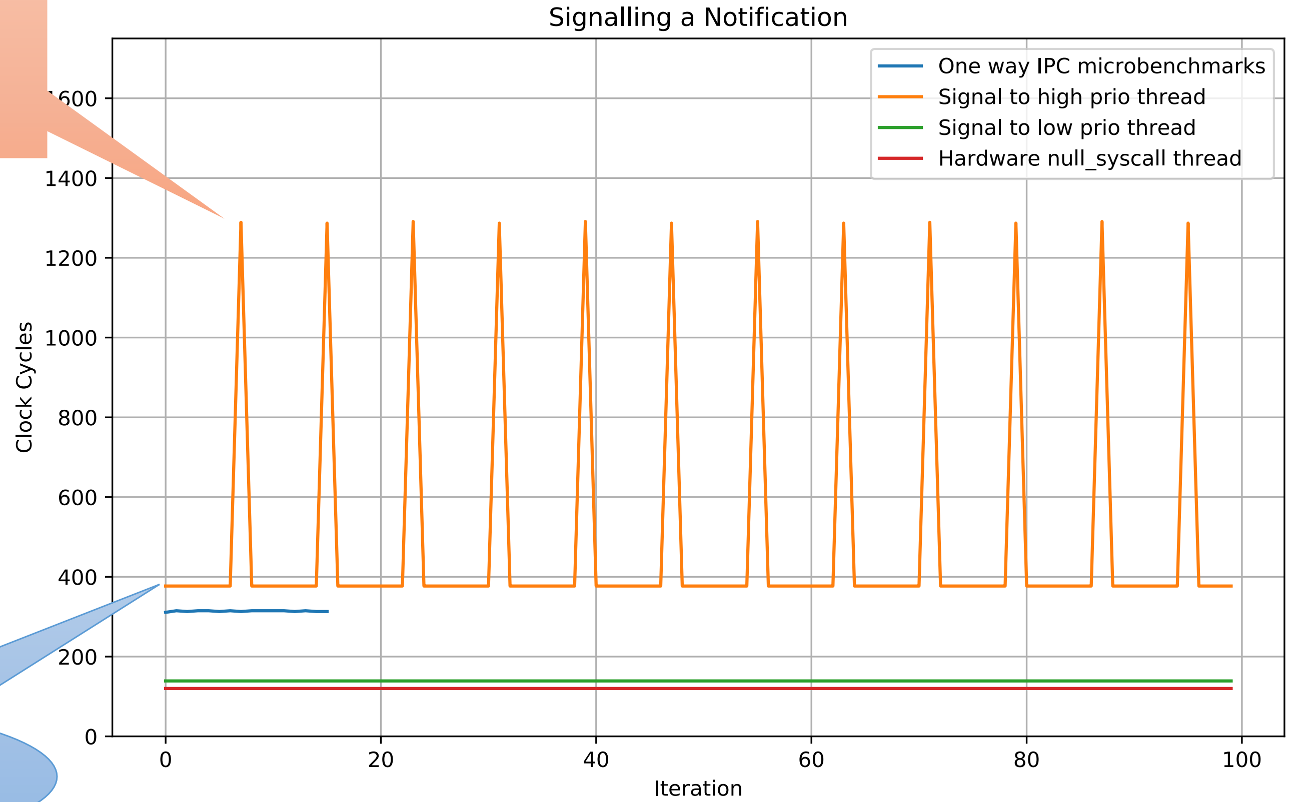
Real-World Example: seL4 Syscall Latency

Interference from test rig

Syscall (cy)	Min	Mean	σ
Null	120	120	0
IPC Call	313	314	1
Signal→low	139	139	0
Signal→high	377	486	298

Platform: Sabre (Armv7-a Cortex-A9)

Real syscall cost: 377 cy



Courtesy Shane Kadish

Problem: Benchmarking Methodology

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

Write stalls on platform with low memory bandwidth!

Method.	Min	Max	Mean	σ
Buffer	709	1770	933	195
Sum in loop	695	770	730	15

Courtesy Nataliya Korovkina

Platform: Sabre
different syscall!

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

All data in registers!

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

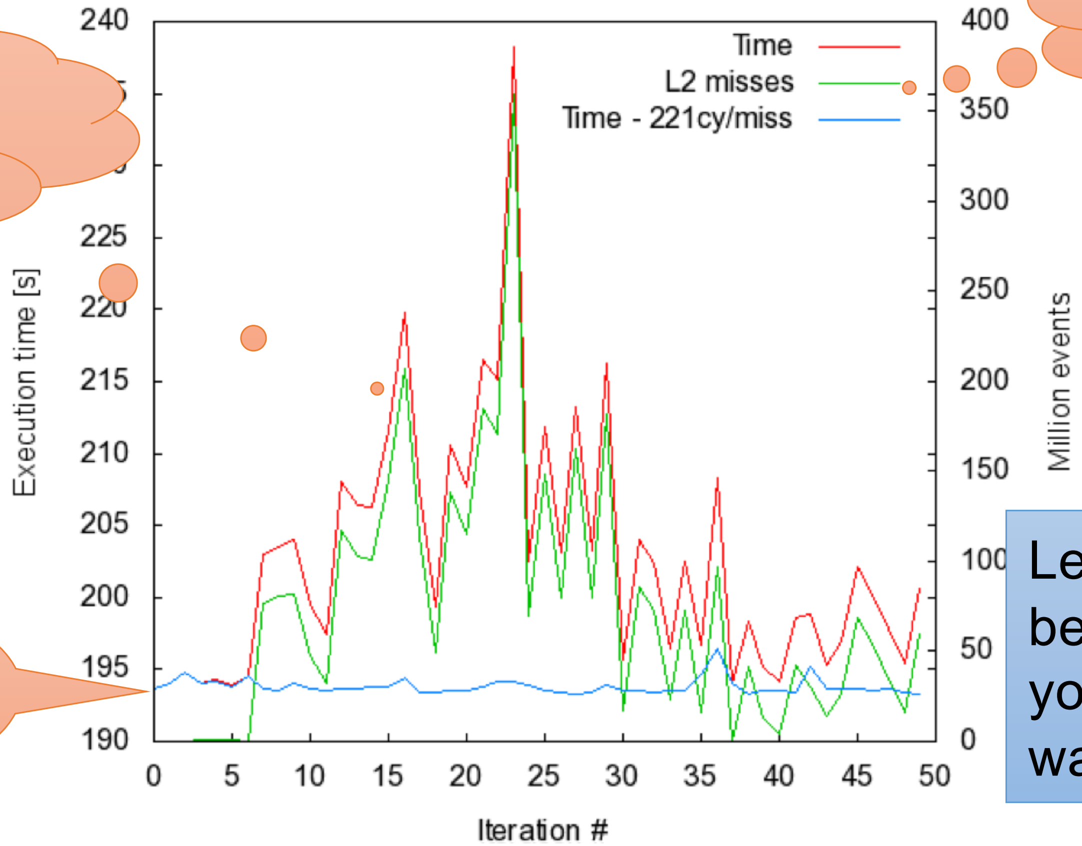
Methodology:

- Multiple identical runs for statistics...

twolf on Linux – What's Going On?

20% performance difference between "identical" runs!

Performance counters are your best friends!



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

Lesson: Check system behaves according to your model – large σ 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!

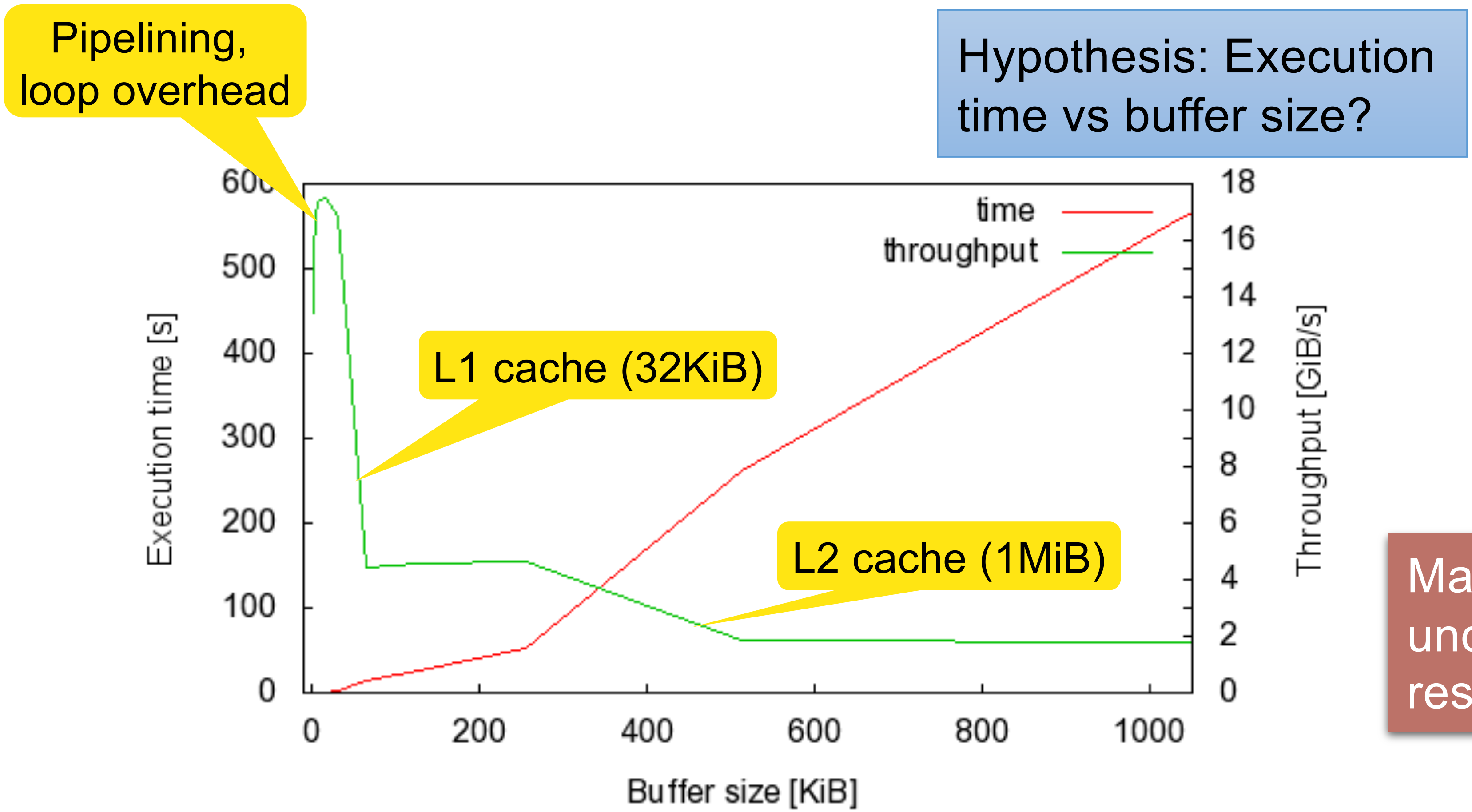
Most Important: Use a Model/Hypothesis

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



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
- Eliminate cache effects by warm-up loops

```
t0 = time();
for (i=0; i<MAX; i++) {asm(nop);} /* overhead*/
t1 = time();

for (i=0; i<10; i++) {asm(syscall);} /* warmup

t2 = time();
for (i=0; i<MAX; i++) {asm(syscall);} /* measure */
t3 = time();
printf("Cost is %dus\n", (t3-t2-(t1-t0))*1000000/MAX);
```

Beware compiler optimisations!

See "Methodology" slide re stats

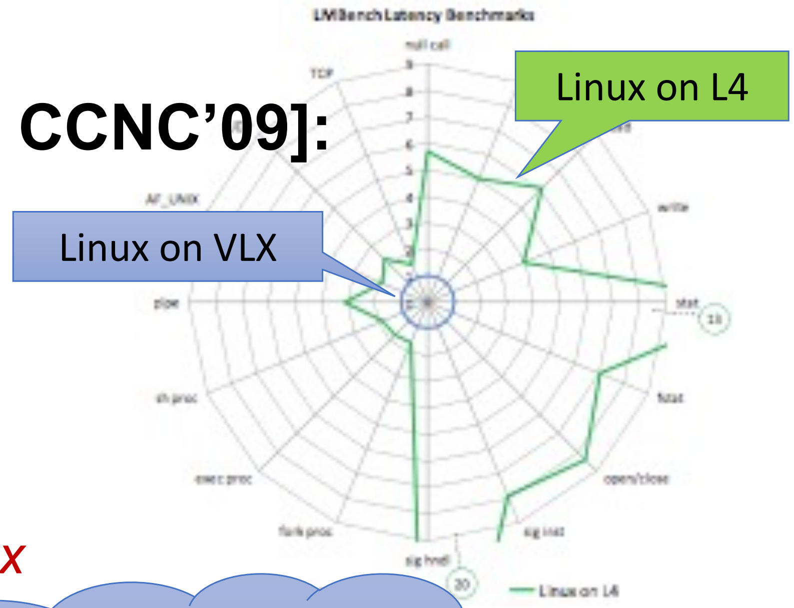
Relative vs Absolute Data

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



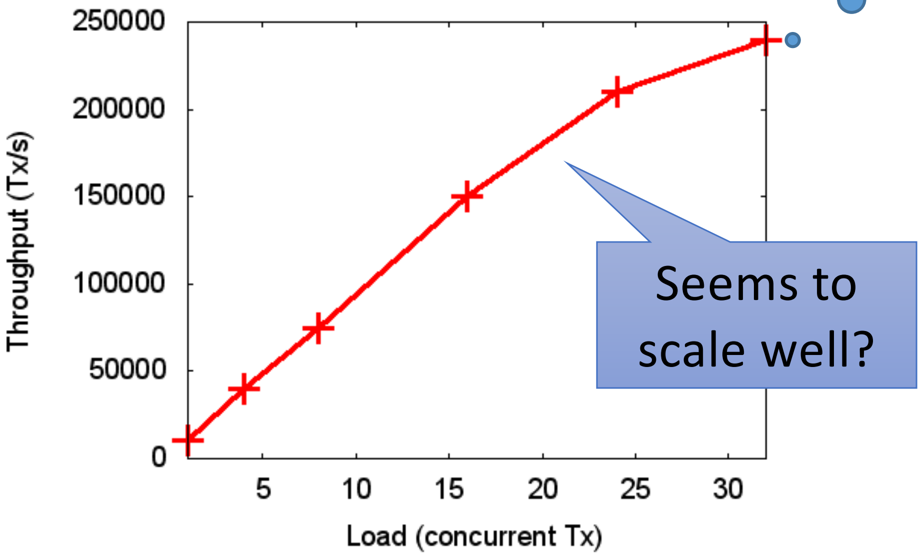
Probably used default L4 config, enables debugging, profiling!

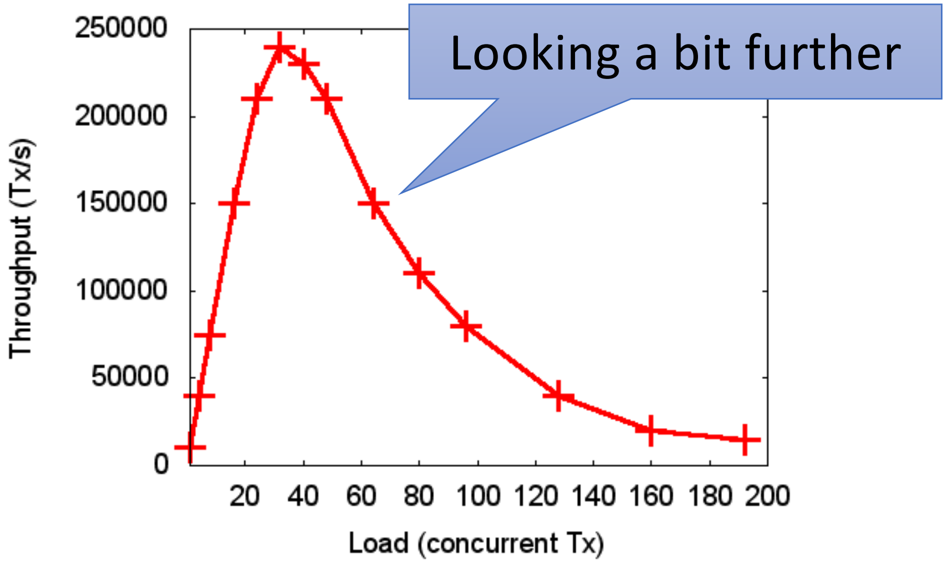
Benchmarking crime: Relative numbers only!

Data Range

Example: Scaling database load

32-core machine





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
- 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)

Remember: progressive and conservative criteria!

- 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: 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

Good or bad?

- 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

Expected overhead?

- **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!

Counter	Native	Virtualized	Difference
Branch miss-pred	1	1	0
D-cache miss	0	0	0
I-cache miss	0	1	1
D-µTLB miss	0	0	0
I-µTLB miss	0	0	0
Main-TLB miss	0	0	0
Instructions	30	125	95
D-stall cycles	0	27	27
I-stall cycles	0	45	45
Total Cycles	93	292	199

Good or bad?

More of the Same

First step: improve representation!

Benchmark	Native	Virtualized
Context switch [1/s]	615,046	444,504
Create/close [µs]	11	15
Suspend [10ns]	81	154

Second step: overheads in appropriate units!

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

Benchmark	Native	Virt.	Diff [µs]	Diff [cy]	# sysc	Cy/sysc
Context switch [µs]	1.63	2.25	0.62	230	1	230
Create/close [µs]	11	15	4	1472	2	736
Suspend [µs]	0.81	1.54	0.73	269	1	269

And Another One…

Good or bad?

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

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

Reminders

- Arista Advanced Operating Systems Prize for top performer in this course
- OS Hall of Fame for straight HDs in OS, AOS, (Dist Syst,) OS Thesis
- Taste-of-Research opportunities at Trustworthy Systems
  https://trustworthy.systems/students/internships
  Deadline: 27 October!
- Honours theses at Trustworthy Systems
  https://trustworthy.systems/students/theses