UNSW AUSTRALIA

COMP9242 Advanced OS

S2/2016 W05: **Performance Evaluation**

@GernotHeiser

Never Stand Still | Engineering | Computer Science and Engineering

Copyright Notice

Overview

- **Performance**
- Benchmarking
- Profiling
- Performance analysis

Purpose of Performance Evaluation

Research:

- Establish performance advantages/drawbacks of an approach
  - may investigate performance limits
  - should investigate tradeoffs

Development:

- Ensure product meets performance objectives
  - new features must not unduly impact performance of existing features
  - quality assurance

Purchasing:

- Ensure proposed solution meets requirements
  - avoid buying snake oil
- Identify best of several competing products

Different objectives may require different approaches

- Unclear objectives will lead to unclear results

What Performance?

- Cold cache vs hot cache
  - hot-cache figures are easy to produce and reproduce
    - but are they meaningful?
- Best case vs average case vs worst case
  - best-case figures are nice — but are they useful?
  - average case — what defines the “average”?
  - expected case — what defines it?
  - worst case — is it really “worst” or just bad? Does it matter?
- What does “performance” mean?
  - is there an absolute measure?
  - can it be compared? With what?
  - Benchmarking

Note: Always analyse performance before optimising!

- Ensure that you focus on the bottlenecks, they may be non-obvious!

Overview

- Performance
- **Benchmarking**
- Profiling
- Performance analysis

Benchmarking in Research

- Generally one of two objectives:
  - Show new approach improves performance
    - Must satisfy progressive and conservative criteria:
      - *Progressive:* significant improvements of important aspect
      - *Conservative:* no significant degradation elsewhere
  - Show otherwise attractive approach does not undermine performance
- Requirement: objectivity/fairness
  - Selection of baseline
  - Inclusion of relevant alternatives
  - Fair evaluation of alternatives
- Requirement: analysis/explanation of results
  - Model of system, incorporating relevant parameters
  - Hypothesis of behaviour
  - Results must support hypothesis

Lies, Damned Lies, Benchmarks

- Micro- vs macro-benchmarks
- Synthetic vs “real-world”
- Benchmark suites, use of subsets
- Completeness of results
- Significance of results
- Baseline for comparison
- Benchmarking ethics
- What is good — analysing the results

Micro- vs Macro-Benchmarks

- Macro-benchmarks
  - Use realistic workloads
  - Measure real-life system performance (hopefully)
- Micro-benchmarks
  - Exercise particular operation, e.g. single system call
  - *Good for analysing performance / narrowing down down bottlenecks*
    - critical operation is slower than expected
    - critical operation performed more frequently than expected
    - operation is unexpectedly critical (because it's too slow)

Micro- vs Macro-Benchmarks

Benchmarking Crime: Micro-benchmarks only

- Pretend micro-benchmarks represent overall system performance

Real performance can generally not be assessed with micro-benchmarks

- Exceptions:
  - Focus is on improving particular operation known to be critical
  - There is an established base line

Note: My macro-benchmark is your micro-benchmark

- Depends on the level on which you are operating
- Eg: lmbench
  - … is a Linux micro-benchmark suite
  - … is a hypervsior macro-benchmark

Synthetic vs “Real-world” Benchmarks

- Real-world benchmarks:
  - real code taken from real problems
    - Livermore loops, SPEC, EEMBC, …
  - execution traces taken from real problems
  - distributions taken from real use
    - file sizes, network packet arrivals and sizes
  - Caution: representative for one scenario doesn't mean for *every* scenario!
    - may not provide complete coverage of relevant data space
    - may be biased
- Synthetic benchmarks
  - created to simulate certain scenarios
  - tend to use random data, or extreme data
  - may represent unrealistic workloads
  - may stress or omit pathological cases

Standard vs Ad-Hoc Benchmarks

Why use ad-hoc benchmarks?

- There may not be a suitable standard
  - Eg lack of standardised multi-tasking workloads
- Cannot run standard benchmarks
  - Limitations of experimental system
  - Resource-constrained embedded system

Why not use ad-hoc benchmarks?

- Not comparable to other work
- Poor reproducibility

Facit: Use ad-hoc BMs only if you have no choice!

- Justify your approach carefully
- Document your benchmarks well (for reproducibility!)

Benchmark Suites

- Widely used (and abused!)
- Collection of individual benchmarks, aiming to cover all of relevant data space
- Examples: SPEC CPU{92|95|2000|2006}
  - Originally aimed at evaluating processor performance
  - Heavily used by computer architects
  - Widely (ab)used for other purposes
  - Integer and floating-point suite
  - Some short, some long-running
  - Range of behaviours from memory-intensive to CPU-intensive
    - behaviour changes over time, as memory systems change
    - need to keep increasing working sets to ensure significant memory loads

Obtaining an Overall Score for a BM Suite

- How can we get a single figure of merit for the whole suite?
- Example: comparing 3 systems on suite of 2 BMs

Normalise to System X

Normalise to System Y

Geometric mean?

Benchmark	System X		System Y		System Z	
	Abs	Rel	Abs	Rel	Abs	Rel
1	20	2.00	10	1.00	40	4.00
2	40	0.50	80	1.00	20	0.25
Geom. mean		**1.00**		**1.00**		**1.00**

Invariant under normalisation!

e sense?

Arithmetic mean is meaningless for relative numbers

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

Benchmark Suite Abuse

Benchmarking Crime: Select subset of suite

- Introduces bias
  - Point of suite is to cover a range of behaviour
  - Be wary of “typical results”, “representative subset”
- Sometimes unavoidable
  - some don't build on non-standard system or fail at run time
  - some may be too big for a particular system
    - eg, don't have file system and run from RAM disk...
- Treat with extreme care!
  - can only draw limited conclusion from results
  - cannot compare with (complete) published results
  - need to provide convincing explanation why only subset

Other SPEC crimes include use for multiprocessor scalability

- run multiple SPECs on different CPUs
- what does this prove?

Partial Data

- Frequently seen in I/O benchmarks:
  - Throughput is degraded by 10%
    - "Our super-reliable stack only adds 10% overhead"

    Almost certainly not true!
  - Why is throughput degraded?
    - latency too high
    - CPU saturated?
  - Also, changes to drivers or I/O subsystem may affect scheduling
    - interrupt coalescence: do more with fewer interrupts
  - *Throughput on its own is useless!*

Throughput Degradation

- Scenario: Network driver or protocol stack
  - New driver reduces throughput by 10% — why?
  - Compare:
    - 100 Mb/s, 100% CPU vs 90 Mb/s, 100% CPU
    - 100 Mb/s, 20% CPU vs 90 Mb/s, 40% CPU
  - Correct figure of merit is *processing cost per unit of data*
    - Proportional to *CPU load divided by throughput*
  - Correct overhead calculation:
    - 10 µs/kb vs 11 µs/kb: *10% overhead*
    - 2 µs/kb vs 4.4 µs/kb: *120% overhead*

CPU limited

Latency limited

Benchmarking crime: Show throughput degradation only

- … and pretend this represents total overhead

Overview

- Performance
- Benchmarking
- **Profiling**
- Performance analysis

Profiling

- Run-time collection of execution statistics
  - invasive (requires some degree of instrumentation)
    - unless use hardware debugging tools or cycle-accurate simulators
  - therefore affects the execution it's trying to analyse
  - good profiling approaches minimise this interference
- Identify parts of system where optimisation provides most benefit
- Complementary to microbenchmarks
- Example: gprof
  - compiles tracing into code, to record call graph
  - uses statistical sampling:
    - on each timer tick record program counter
    - post execution translate this into execution-time share

Gprof example output

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

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

Gprof example output (2)

```
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]
                 0.00    0.01       1/1           print [9]
                 0.00    0.01       9/9           fgets [12]
```

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

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
- Identify parts of system where optimisation provides most benefit
- Complementary to microbenchmarks
- Example: gprof
  - compiles tracing into code, to record call graph
  - uses statistical sampling:
    - on each timer tick record program counter
    - post execution translate this into execution-time share
- Example: oprof
  - collects hardware performance-counter readings
  - works for kernel and apps
  - minimal overhead

oprof example output

Count

Percentage

Performance counter used

```
$ 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
      832 0.1398 wine
…
```

Profiler

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

oprof example 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
            12988 7.9579 libfreetype.so.6.3.1
            10375 6.3569 vmlinux
               …
```

Drilldown of top consumers

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

Performance Monitoring Unit (PMU)

- Collects certain *events* at run time
- Typically 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
    - OS can sample counters
    - Counters can trigger exception on exceeding threshold

Event Examples (ARM11)

Ev #	Definition	Ev #	Definition	Ev #	Definition
0x00	I-cache miss	0x0b	D-cache miss	0x22	…
0x01	Instr. buffer stall	0x0c	D-cache writeback	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 mispredicted	0x12	Write-buffer drained	0x38	…
0x07	Instr executed	0x13	Cycles FIRQ disabled	0xff	Cycle counter
0x09	D-cache acc cachable	0x14	Cycles IRQ disabled		
0x0a	D-cache access any	0x20	…		

Overview

- Performance
- Benchmarking
- Profiling
- **Performance analysis**

Significance of Measurements

All measurements are subject to random errors

- Standard scientific approach: Many iterations, *collect statistics*
- Rarely done in systems work — why?
- Computer systems tend to be *highly deterministic*
  - Repeated measurements often give identical results
  - Main exception are experiments involving WANs
- However, it is dangerous to rely on this without checking!
  - Sometimes “random” fluctuations indicate *hidden parameters*

Benchmarking crime: results with no indication of significance

Non-criminal approach:

- Show at least standard deviation of your measurements
- … or state explicitly it was below a certain value throughout
- Admit results are insignificant unless well-separated std deviations

How to Measure and Compare Performance

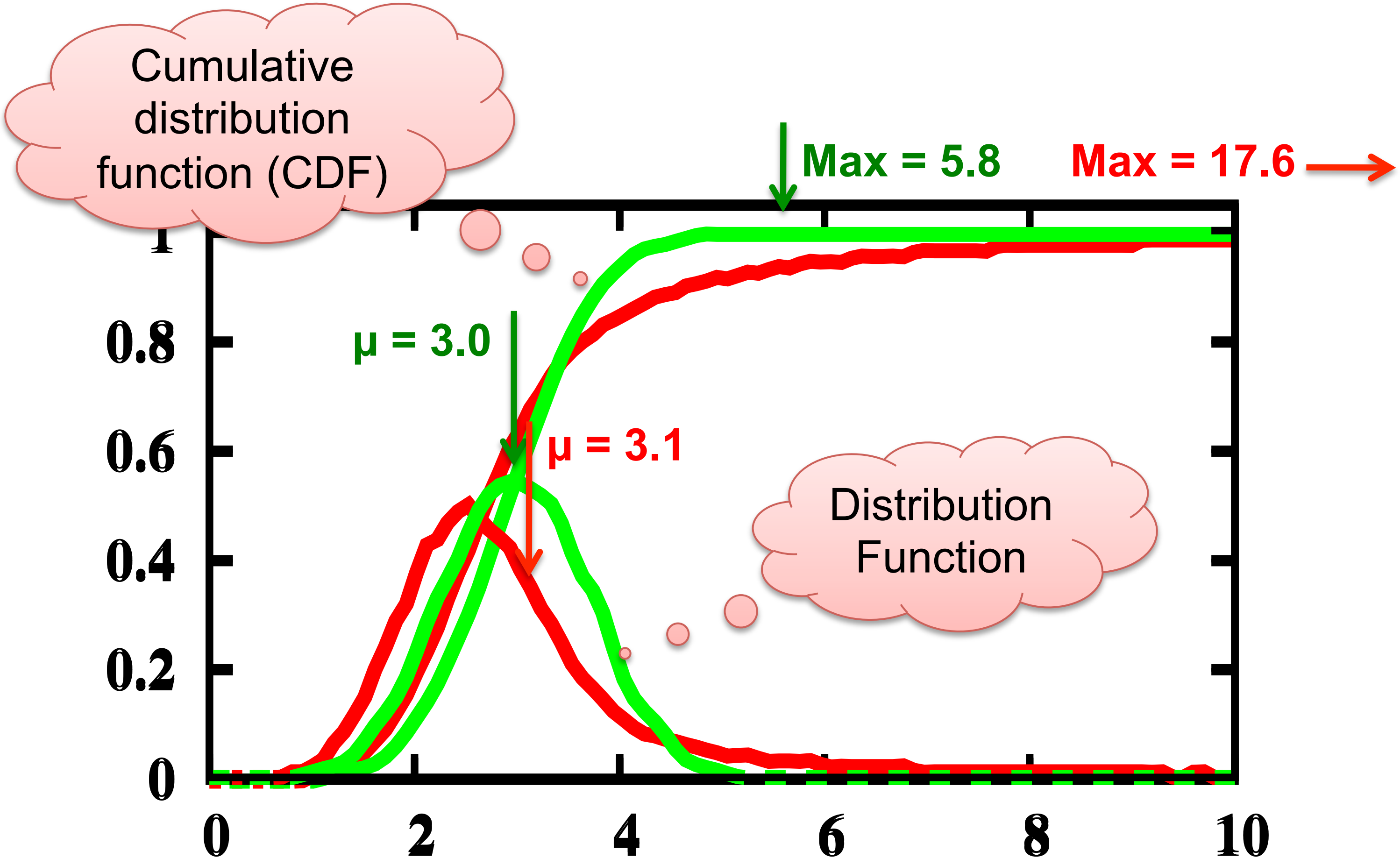
Bare-minimum statistics:

- At minimum 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
- Be *very suspicious* if reproducibility is poor (i.e. σ is *not* small)
  - Exception: non-local networks
- *Distrust* standard deviations of small iteration counts
  - standard deviations are meaningless for small number of runs
  - … but ok if effect $\gg$ σ
  - The proper way to check significance of differences is Student's t-test!

How to Measure and Compare Performance

Bare-minimum stats are sometimes insufficient

- Eg: Old: μ = 3.1 sec, New: μ = 3 sec



How to Measure and Compare Performance

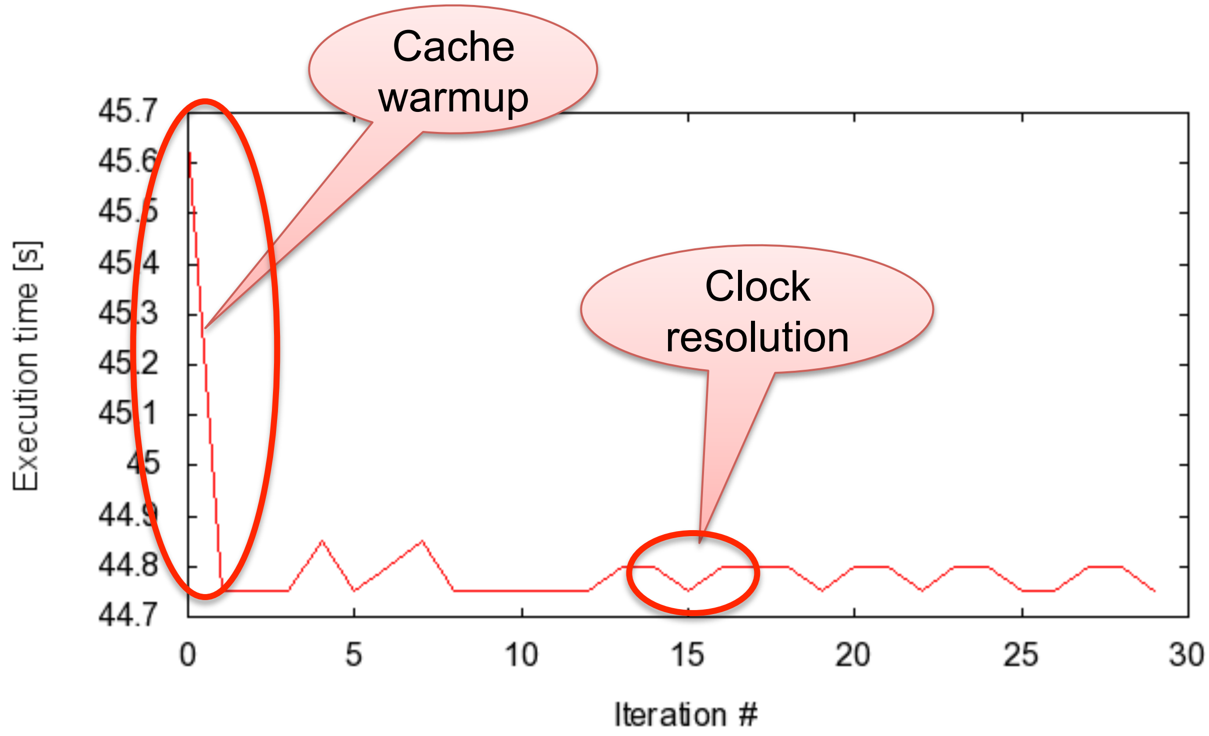
Obtaining meaningful execution times:

- Make sure execution times are long enough
  - What is the granularity of your time measurements?
  - make sure the effect you're looking for is much bigger
  - many repetitions won't help if your effect is dominated by clock resolution
  - do many repetitions in a tight loop if necessary

Example: gzip from SPEC CPU2000

Observations?

- First iteration is special
- 20 Hz clock
  - will not be able to observe any effects that account for less than 0.1 sec



Lesson?

- Need a mental model of the system
  - Here: repeated runs should give the same result
- Find reason (hidden parameters) if results do not comply!

How to Measure and Compare Performance

Noisy data:

- Sometimes it isn't feasible to get a “clean” system
  - e.g. running apps on a “standard configuration”
  - this can lead to very noisy results, large standard deviations

Possible ways out:

- Ignoring lowest and highest result
- Taking the floor of results
  - makes only sense if you're looking for minimum
    - but beware of difference-taking!

Both of these are dangerous, use with great care!

- Only if you know what you are doing
  - need to give a convincing explanation of why this is justified
- Only if you explicitly state what you've done in your paper/report

How to Measure and Compare Performance

Check outputs!

- Benchmarks must check results are correct!
  - Sometimes things are very fast because no work is done!
  - Beware of compiler optimisations, implementation bugs
- Sometimes checking all results is infeasible
  - eg takes too long, checking dominates effect you're looking for
  - check at least *some* runs
  - run same setup with checks en/disabled

How to Measure and Compare Performance

Vary inputs!

- Easy to produce low standard deviations by using identical runs
  - but this is often not representative
  - can lead to unrealistic caching effects
    - especially in benchmarks involving I/O
    - *disks are notorious for this*
      - controllers do caching, pre-fetching etc out of control of OS
- Good ways to achieve variations:
  - time stamps for randomising inputs (but see below!)
  - varying order:
    - forward vs backward
    - sequential with increasing strides
    - random access
  - best is to use combinations of the above, to ensure that results are sane

How to Measure and Compare Performance

Ensure runs are comparable and reproducible:

- Avoid true randomness!
  - tends to lead to different execution paths or data access patterns
  - makes results non-reproducible
  - makes impossible to fairly compare results across implementations!
  - exceptions exist
    - crypto algorithms are designed for input-independent execution paths
- Pseudo-random is good for benchmarking
  - reproducible sequence of “random” inputs
    - capture sequence and replay for each run
    - use pseudo-random generator with same seed

How to Measure and Compare Performance

Environment

- Ensure system is quiescent
  - to the degree possible, turn off any unneeded functionality
    - run Unix systems in single-user mode
    - turn off wireless, disconnect networks, put disk to sleep, etc
  - Be aware of self-interference
    - eg logging benchmark results may wake up disk...
- Start different runs from the same system state (where possible)
  - back-to-back processes may *not* find the system in the same state

Real-World Example

Benchmark:

- `300.twolf` from SPEC CPU2000 suite

Platform:

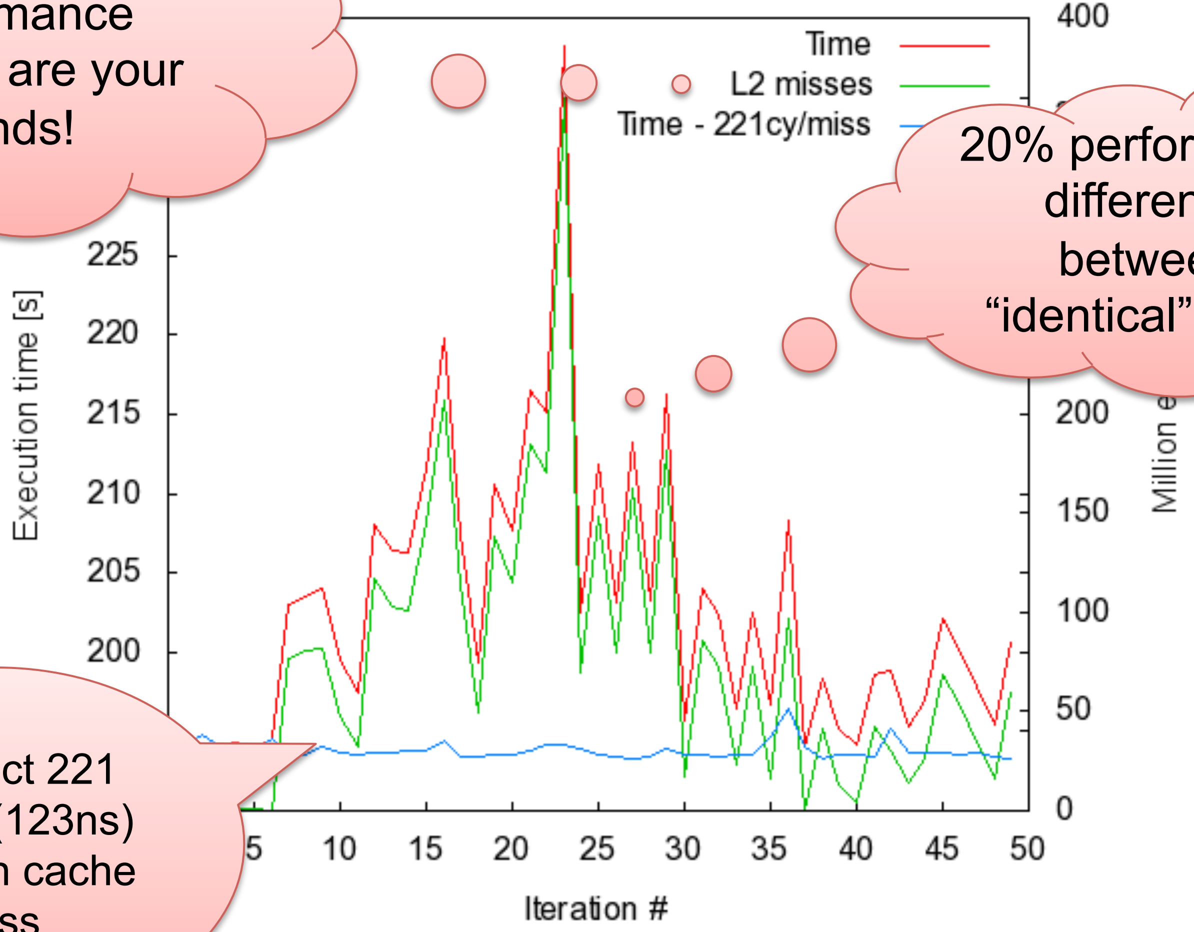
- Dell Latitude D600
  - Pentium M @ 1.8GHz
  - 32KiB L1 cache, 8-way
  - 1MiB L2 cache, 8-way
  - DDR memory @ effective 266MHz
- Linux kernel version 2.6.24

Methodology:

- Multiple identical runs for statistics...

`twolf` on Linux: What's going on?

Performance counters are your friends!



20% performance difference between "identical" runs!

Subtract 221 cycles (123ns) for each cache miss

`twolf` on Linux: Lessons?

- Pointer to problem was standard deviation
  - σ for "twolf" was much higher than normal for SPEC programs
- Standard deviation did not conform to mental model
  - Shows the value of verifying that model holds
  - Correcting model improved results dramatically
- Shows danger of assuming reproducibility without checking!

Conclusion: *Always* collect and analyse standard deviations!

How to Measure and Compare Performance

Vary only one thing at a time!

- Typical example: used a combination of techniques to improve system
  - what can you learn from a 20% overall improvement?
- Need to run sequence of evaluations, looking at individual changes
  - identify contribution and relevance
  - understand how they combine to an overall effect
    - they may enhance or counter-balance each other
  - *make sure you understand what's going on!!!!*

Record all configurations and data!

- May have overlooked something at first
- May develop better model later
  - could be much faster to re-analyse existing data than re-run all benchmarks

How to Measure and Compare Performance

Measure as directly as possible:

- Eg, when looking at effects of pinning TLB entries
  - don't just look at overall execution time (combination of many things)
  - use performance counter to compare
    - TLB misses
    - cache misses (from page table reloads)
    - ...
- Cannot always measure directly
  - eg, actual TLB-miss cost not known
    - extrapolate by artificially reducing TLB size
    - eg by pinning useless entries

How to Measure and Compare Performance

Avoid incorrect conclusions from pathological cases

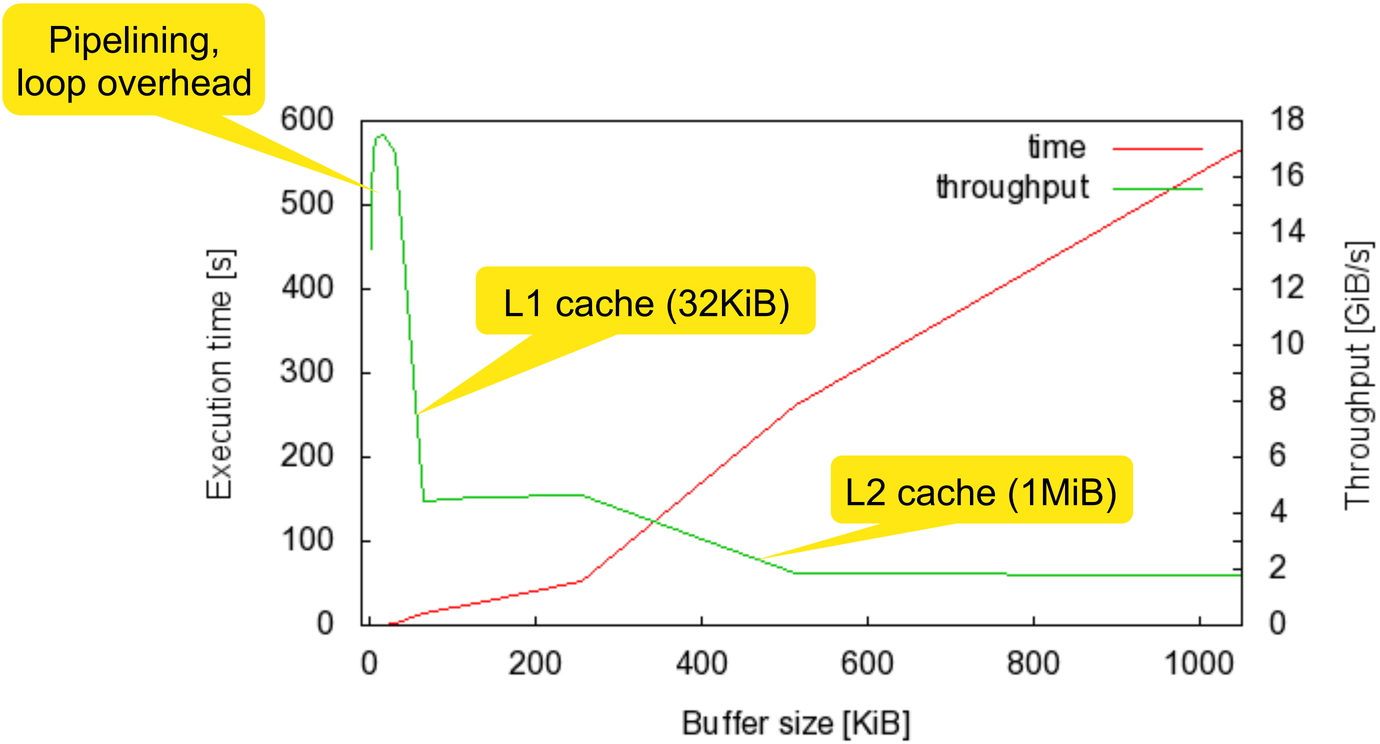
- Typical cases:
  - sequential access optimised by underlying hardware/disk controller...
  - potentially massive differences between sequentially up/down
    - pre-fetching by processor, disk cache
  - random access may be an unrealistic scenario that destroys performance
    - for file systems
  - powers of two may be particularly good or particularly bad for strides
    - often good for cache utilisation
      - minimise number of cache lines used
    - often bad for cache utilisation
      - maximise cache conflicts
  - similarly just-off powers (2^n-1, 2^n+1)
- What is “pathological” depends a **lot** on what you're measuring
  - e.g. caching in underlying hardware

How to Measure and Compare Performance

Use a model

- *You need a (mental or explicit) model of the behaviour of your system*
  - benchmarking should aim to support or disprove that model
  - need to think about this in selecting data, evaluating results
  - eg: I/O performance dependent on FS layout, caching in controller...
  - cache sizes (HW & SW caches)
  - buffer sizes vs cache size
- Model should tell you roughly what to expect
  - you should understand that a 2ns cache miss penalty can't be right

Example: Memory Copy



How to Measure and Compare Performance

Understand your results!

- Results you don't understand will almost certainly hide a problem
  - Never publish results you don't understand
    - chances are the reviewers understand them, and will reject the paper
    - maybe worse: someone at the conference does it
      - this will make you look like an idiot

Of course, if this happens you ***are*** an idiot!

Loop and Timing Overhead

Ensure that measuring overhead does not affect results:

- Cost of accessing clock may be significant
- Loop overhead may be significant
- Stub overhead may be significant

Approaches:

- May iterations in tight loop
- Measure and eliminate timer overhead
- Measure and eliminate loop overhead
- Eliminate effect of any instrumentation code

Eliminating Overhead

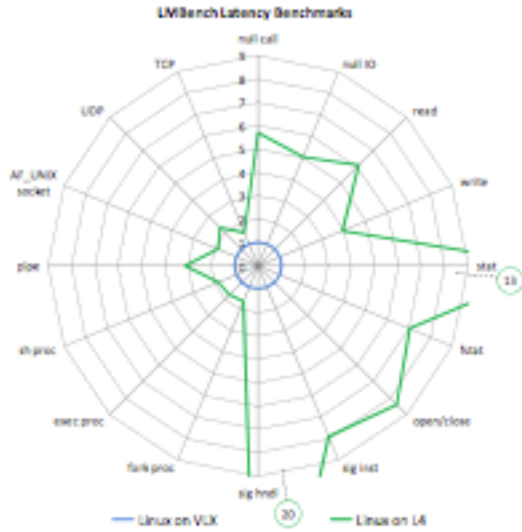
```
t0 = time();
for (i=0; i<MAX; i++) {
    asm(nop);
}
t1 = time();
for (i=0; i<MAX; i++) {
    asm(syscall);
}
t2 = time();
printf("Cost is %dus\n", (t2-2*t1+t0)*1000000/MAX);
```

Beware of compiler optimizations!

Relative vs Absolute Data

From a real paper (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

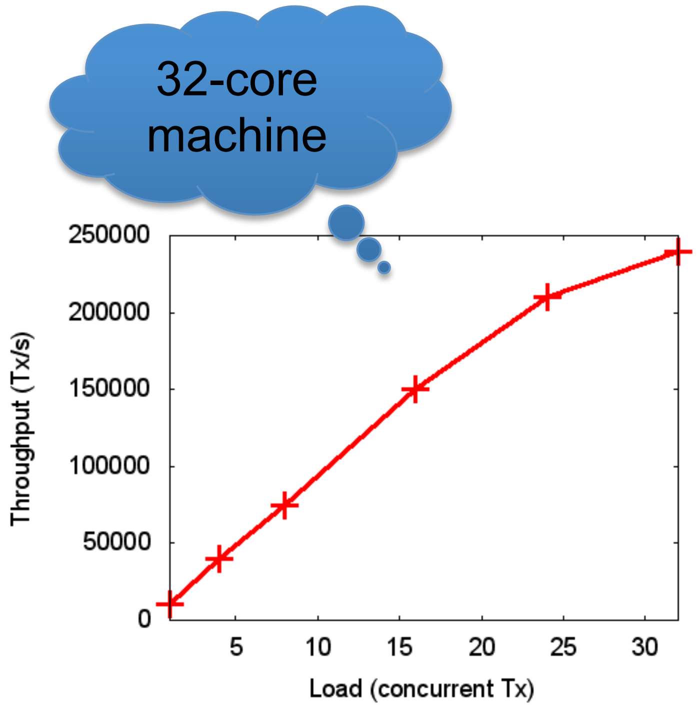


Benchmarking crime: Relative numbers only

- Makes it impossible to check whether results make sense
- How hard did they try to get the competitor system to perform?
  - Eg, did they run it with default build parameters (debugging enabled)?

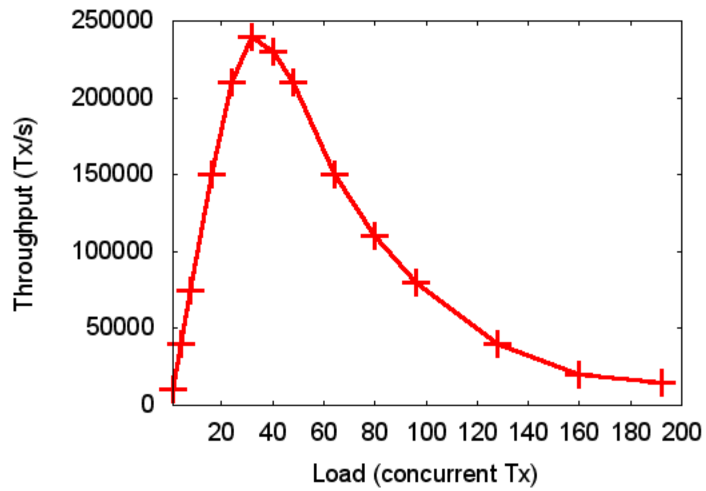
Data Range

Example: Scaling database load



Scales well, right?

Looking a bit further:



Benchmarking crime: Selective data set hiding deficiencies

Benchmarking Ethics

- Do compare with published competitor data, but…
  - Ensure comparable setup
    - Same hardware (or *convincing* argument why it doesn't matter)
  - You may be looking at an aspect the competitor didn't focus on
    - eg: they designed for large NUMA, you optimise for embedded
- Be ultra-careful when benchmarking competitor's system yourself
  - Are you sure you're running the competitor system optimally?
    - you could have the system mis-configured (eg debugging enabled)
    - Do your results match their (published or else) data?
  - Make sure you understand exactly what is going on!
    - Eg use profiling/tracing to understand source of difference
    - Explain it!

Benchmarking crime: Unethical benchmarking of competitor

- Lack of care is unethical too!

Other Ways to Cheat With Benchmarks

- Benchmark-specific optimisations
  - Recognise particular benchmark, insert BM-specific hand-optimised code
  - Popular with compiler-writers, rarely an issue in OS area
  - 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
  - GIGO

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

- CPU-intensive benchmark to “measure” networking performance

I’ve seen all of these BM crimes!

What Is "Good"?

- Easy if there are established and published benchmarks
  - Eg your improved algorithm beats best published Linux data by *x*%
  - But are you sure that it doesn't lead to worse performance elsewhere?
    - important to run complete benchmark suites
    - think of everything that could be adversely effected, and *measure*!
- Tricky if no published standard
  - Can run competitor/incumbent
    - eg run lmbench, kernel compile etc on your modified Linux and standard Linux
    - but be *very careful* to avoid running the competitor sub-optimally!
  - Establish performance limits
    - ie compare against optimal scenario
    - establish hardware limits on performance
    - micro-benchmarks or profiling can be highly valuable here!

Real-World Example: Virtualization 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 x save+restore state
  - virtualized: 4 mode switches, 2 context switches, 3 x save+restore state

Good or bad?

Expected overhead?

Performance Counters are Your Friends!

Good or bad?

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**

More of the Same...

First step: improve representation!

Benchmark	Native	Virtualized
Context switch [1/s]	615046	444504
Create/close [μs]	11	15
Suspend [10ns]	81	154

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

Second step: overheads in appropriate units!

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

Yet 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!

Note: these are purely user-level operations!

- What's going on?

Lessons Learned

- Ensure stable results
  - repeat for good statistics
  - investigate source of apparent randomness
- Have a model of what you expect
  - investigate if behaviour is different
  - unexplained effects are likely to indicate problems — don't ignore them!
- Tools are your friends
  - performance counters
  - simulators
  - traces
  - spreadsheets

Annotated list of benchmarking crimes: http://www.gernot-heiser.org/