UNSW

Performance Evaluation

COMP9242
2009/S2 Week 8

Copyright Notice

UNSW

These slides are distributed under the Creative Commons Attribution 3.0 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"
- The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode

Overview

UNSW

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

Purpose of Performance Evaluation

UNSW

Research:

- → Establish performance advantages/disadvantages of 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

What Performance?

UNSW

- 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*

Overview

UNSW

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

Lies, Damned Lies, Benchmarks

UNSW

- Micro- vs macro-benchmarks
- Synthetic vs “real-world”
- Benchmark suites, use of subsets

Micro- vs Macrobenchmarks

UNSW

- Microbenchmarks are useful to
  - stress code
  - analyse performance
- Macrobenchmarks
  - measure real-life performance (hopefully)
- Real performance can in general not be assessed with microbenchmarks
  - may be good to narrow down performance bottlenecks
  - may be good to find out why performance sux, eg:
    - critical operation is slower than expected
    - critical operation performed more frequently than expected
    - operation is unexpectedly critical (because it's too slow)
- Exceptions:
  - know what critical operation is, and how it affects overall performance
  - there is an established target

Synthetic vs "Real-world" Benchmarks

UNSW

- 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

Benchmark Suites

UNSW

- 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
- Issue: How combine dozens of individual times into overall score?
  - sum/mean is biased in favour of long-running jobs
  - use normalised scores and *geometric mean* [Fleming & Wallace, 1986]

Benchmark Suite Abuse

UNSW

- Most frequent SPEC crime: select subset of suite
  - introduces bias
    - point of suite is to cover a range of behaviour
- 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?

Benchmarking Crime: Partial Data

UNSW

- Frequently seen in I/O benchmarks:
  - Throughput is degraded by 10%
    - "Our super-reliable stack only adds 10% overhead"
    - This is almost certainly not true. Why?
  - 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
  - Need to look at CPU load
    - throughput slightly down usually means CPU load is way up!
    - what is the overhead
  - Assume CPU utilisation is doubled, what is the overhead?
    - Relative cost per MiB: 2/0.9 = 2.2 — real overhead is 120%!
- Another bad one: CPU load increases from 31% to 36%
  - Is this a 5% or a 16% increase???

Overview

UNSW

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

Profiling

UNSW

- 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
- Use to 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

UNSW

```
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)

UNSW

```
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

UNSW

- 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
- Use to 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

UNSW


```
$ 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
…
```

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

oprof example output

UNSW

```
$ 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
               …
```

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

Overview

UNSW

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

How to Measure and Compare Performance

UNSW

General

- Many iterations, *collect statistics*
  - at minimum report the mean (μ) and standard deviation (σ)
  - don't believe anything 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 system benchmarking, reproducibility is often very good (i.e. σ is small)
    - *if they are not, make sure you understand why*
- Distrust standard deviations of small iteration counts
  - standard deviations are meaningless for small number of runs
  - … but ok if effect ≫ σ
- The proper way to check significance of differences is Student's t-test!

How to Measure and Compare Performance

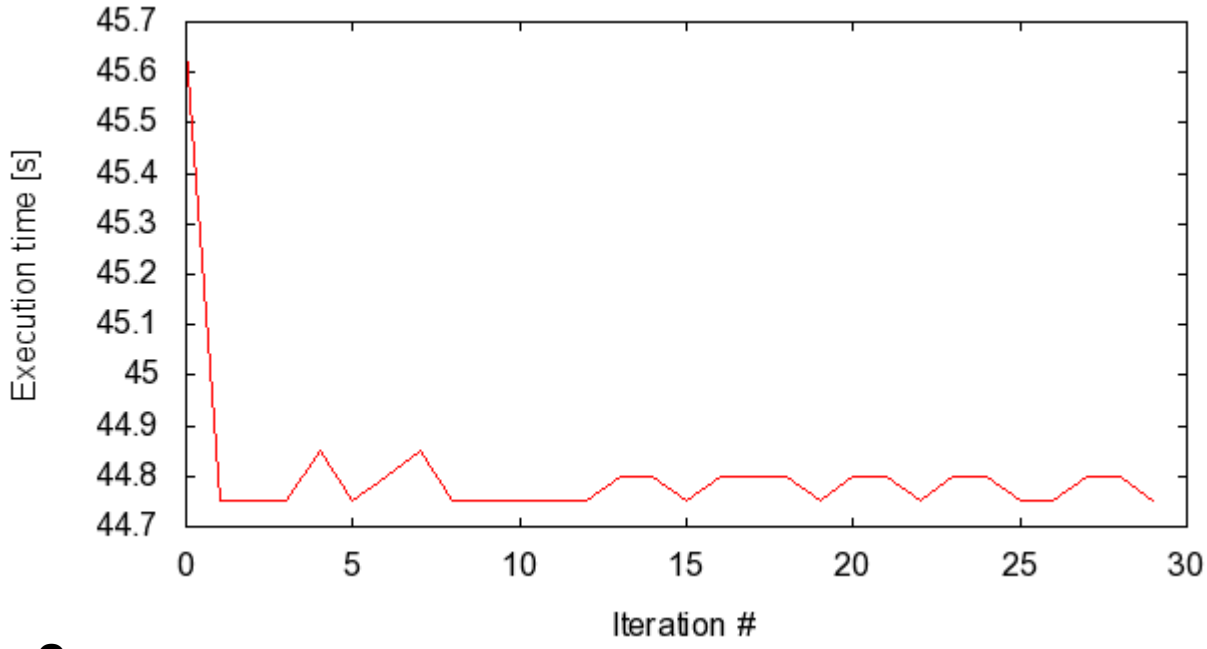
UNSW

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

UNSW



Observations?

- → First iteration is special
  - need to do warm-up runs to get reliable data
- → Clock has a 50ms resolution
  - will not be able to observe any effects that account for less than 0.1 sec

How to Measure and Compare Performance

UNSW

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!

Use these 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

UNSW

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

UNSW

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

UNSW

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 from different implementations!
  - exceptions exist
    - crypto algorithms are designed to have execution path independent of inputs
- → 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

UNSW

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...
- → Ensure compared runs start from the same system state (as far as possible)
  - back-to-back processes may *not* find the system in the same state

Real-World Example

UNSW

Benchmark:

- `300.twolf` from SPEC CPU2000 suite

Platform:

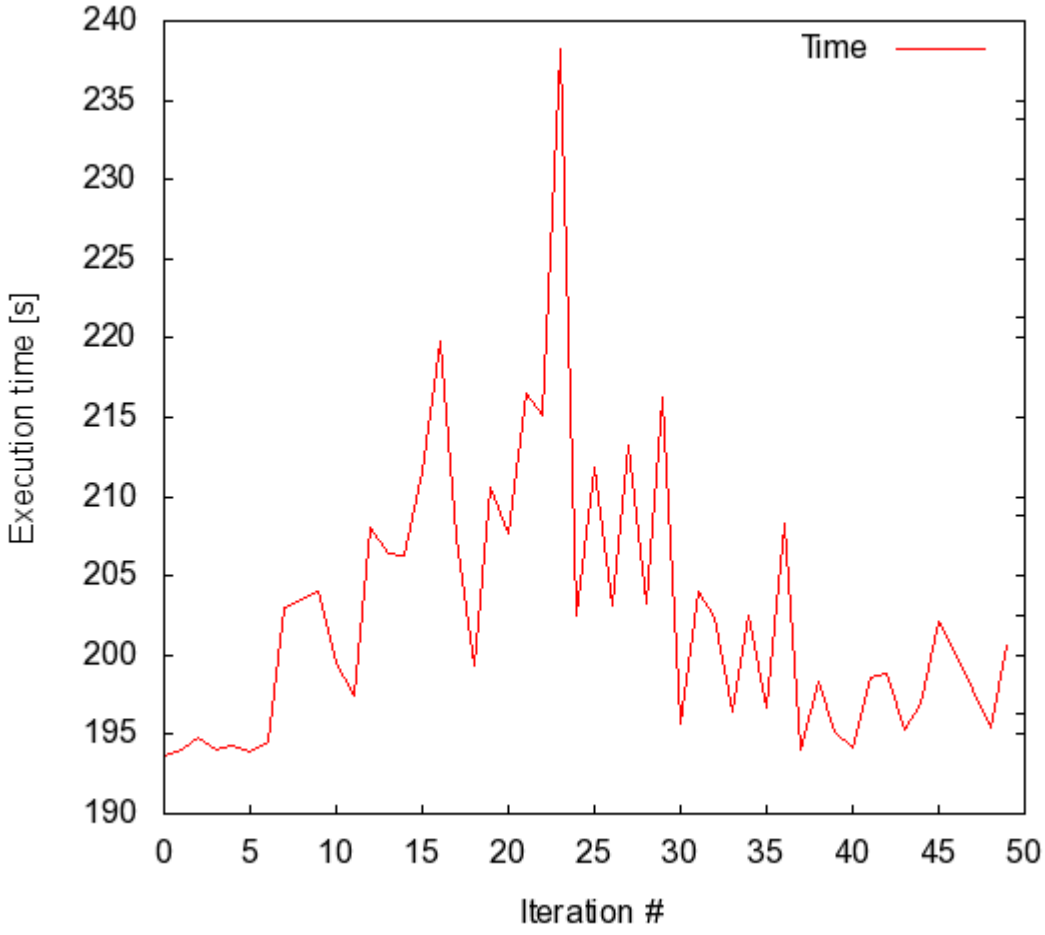
- Dell Latitude D600
  - Pentium M @ 1.8GHz
  - 32KiB L1 cache, 8-way
  - 1MiB L2 cache, 8-way
  - DDR memory @ effective 266MHz
- Linux kernel version 2.6.24

Methodology:

- Multiple identical runs for statistics...

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

UNSW

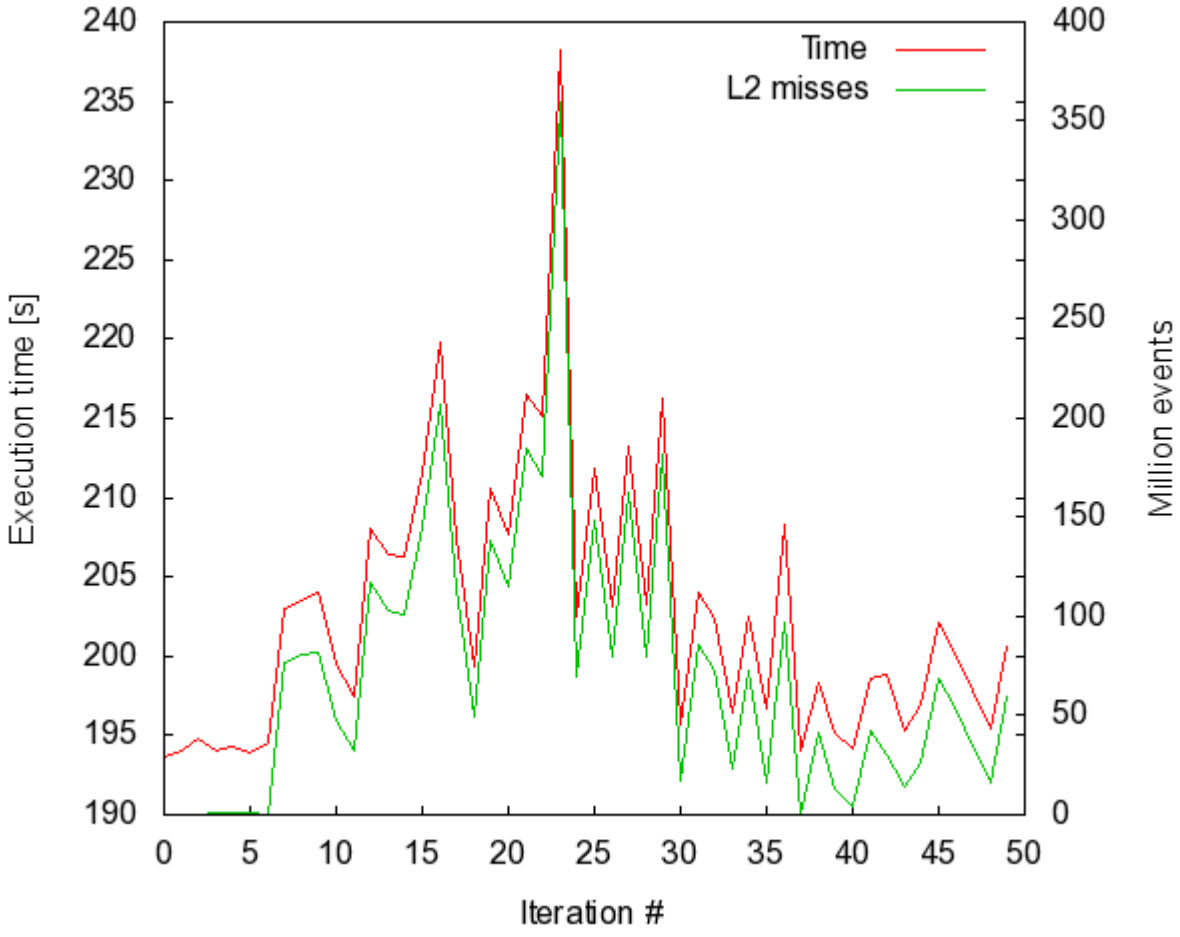


20% variation in execution time between “identical” runs!

`twolf` on Linux:

Performance counters are your friends!

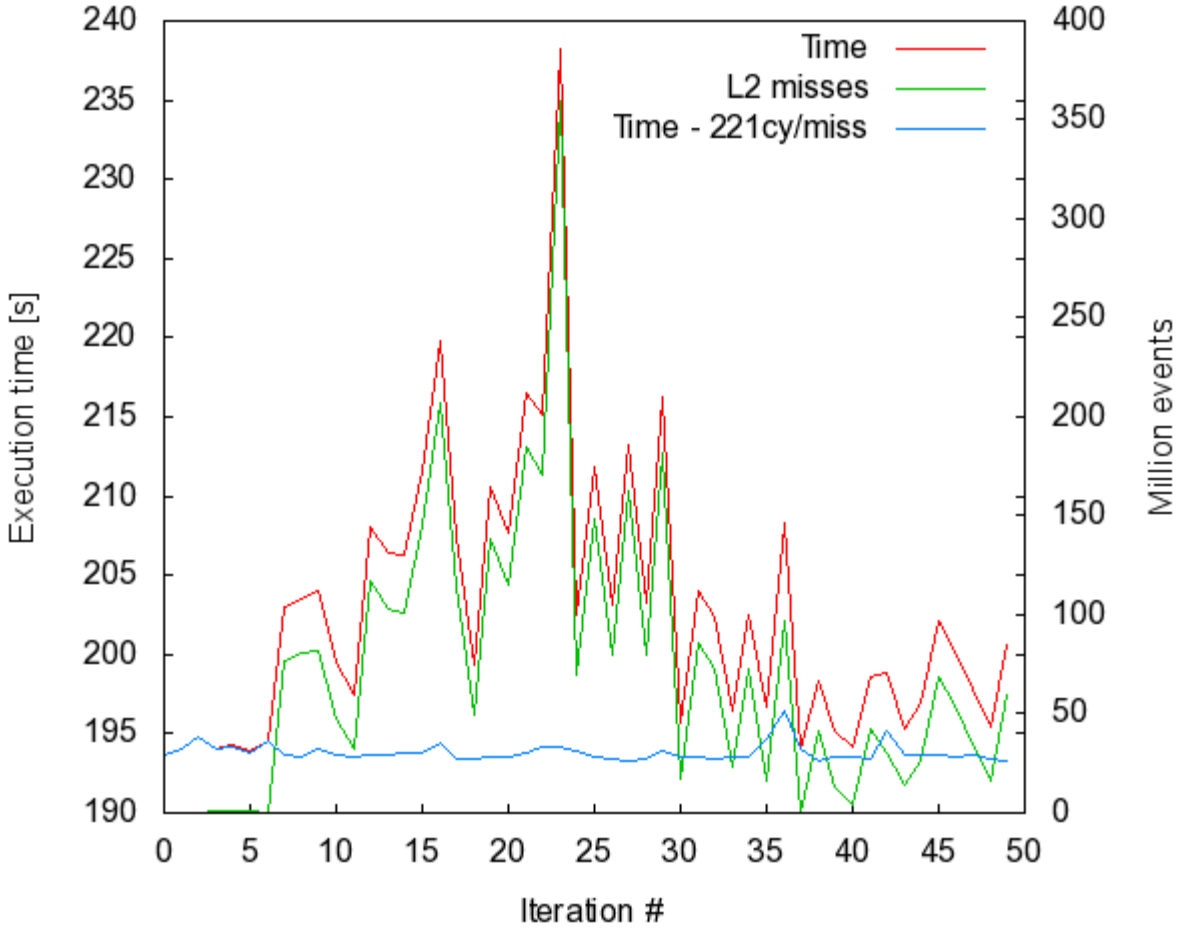
UNSW



`twolf` on Linux:

Subtract 221 cy (123ns) for each cache miss

UNSW



What's going on???

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

UNSW

Observations:

- “Identical” runs ain't
- >90% of variation explained by L2 misses
- Obviously, environment changes between executions — how?
- Benchmarks themselves should be deterministic
  - fixed input data
  - each run should have same data accesses, same cache footprint
- L2 is physically-addressed
  - differences must stem from physical memory layout, resulting in conflict misses
- Linux randomises physical memory allocation

How to Measure and Compare Performance

UNSW

Vary only one thing at a time!

- → Typical example: your 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

UNSW

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

UNSW

Avoid incorrect conclusions from pathological cases

- Typical cases:
  - sequential access may be optimised by underlying hardware/disk controller...
  - there may or may not be 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

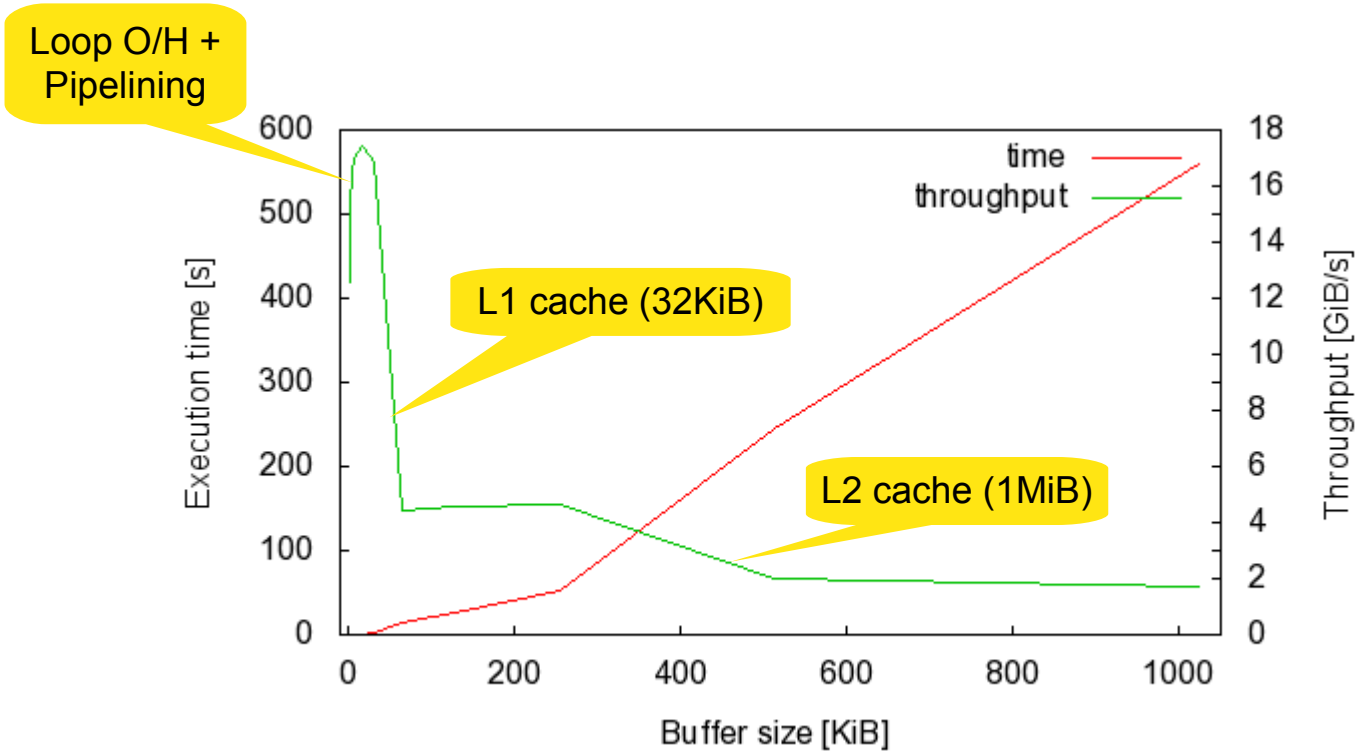
UNSW

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
- → Should tell you the size of what to expect
  - you should understand that a 2ns cache miss penalty can't be right

Example: Memory Copy

UNSW



How to Measure and Compare Performance

UNSW

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

UNSW

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

UNSW

```
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!

Benchmarking Against Competitors

UNSW

One thing is to compare against published data

→ Need to be really careful to ensure comparable setup
- same hardware, or *really* convincing argument why differences don't matter
- you may be comparing a performance aspect the competitor didn't focus on
- eg tradeoffs: they designed for large NUMA, you optimise for embedded

Other thing is to benchmark competitor system yourself

→ Are you sure you're running the competitor system optimally?
- you could have the system mis-configured (eg debugging enabled)

It is really important to be ultra-fair!

→ Making competitors unfairly look bad may constitute misconduct!
- at best it's incompetence

→ Make sure you understand *exactly* what's going on!
- run additional traces/profiling/microbenchmark to explain performance difference!
- explain this in your paper/report!

What Is “Good”?

UNSW

- 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
  - but 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!
- Hard if nothing directly to compare to
  - Frequent scenario with microbenchmarks
    - they may be specifically designed to analyse your setup
    - when are they good enough?
    - if your overall performance isn't good, which microbenchmarks should you focus on?

Another Real-World Example

UNSW

- Null-syscall microbenchmark: native: 0.24µs, virtualized: 0.79µs
  - good or bad?
- Model:
  - native does 2 mode switches, 0 context switches, 1 save+restore state
  - virtualized does 4 mode switches, 2 context switches, 3 save+restore state
  - expected overhead?
- ARM11 processor runs at 368 MHz: 0.24µs = 93 cy, 0.79µs = 292 cy

Performance Counters are Your Friends!

UNSW

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...

UNSW

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

First step: improve representation!

Benchmark	Native	Virtualized	Difference	Overhead
Context switch [µs]	1.63	2.25	0.62	39%
Create/close [µs]	11	15	4	36%
Suspend [µs]	0.81	1.54	0.73	90%

More of the Same...

UNSW

Then represent the overheads in the right 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

Further analysing the create/close benchmark:

→ guest dis/enables interrupts 22 times

- extra instructions required to manipulate virtual interrupt flag
- account for most of extra overhead

Yet Another One...

UNSW

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

Note: these are purely user-level operations!

- What's going on?

Yet Another One...

UNSW

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

Note: these are purely user-level operations!

- What's going on?
- Timer tick is 1000 Hz
- Overhead from virtualizing timer interrupt!
- Good or bad?

Lessons Learned

UNSW

- 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