Performance Evaluation

Overview

- Performance
- Benchmarking
- Profiling
- Performance analysis

Purpose of Performance Evaluation

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?

- 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
Lies, Damned Lies, Benchmarks

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

Micro-vs Macrobenchmarks

- Microbenchmarks are useful to:
  - stress code
  - analyze performance
- Macrobenchmarks:
  - measure real-life performance (hopeful)
- 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 sucks, e.g.,
    - 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

- Real-world benchmarks:
  - real code taken from real problems
  - Livermore loops, SPEC, EEMBC, ...
  - execution traces 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
- 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

- 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
    - Some short, some long-running
    - Range of behaviors 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 favor of long-running jobs
    - use normalised scores and geometric mean [Fleming & Wallace, 1998]

Benchmark Suite Abuse

- Most frequent SPEC crime: select subset of suite
  - introduces bias
  - point of suite is to cover a range of behavior
  - Sometimes unavoidable
  - some don’t build on non-standard systems 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

- 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 MB: 20.9 * 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

- Performance
- Benchmarking
- Profiling
- Performance analysis

Profiling

- Run-time collection of execution statistics
  - invasive (requires some degree of instrumentation)
  - 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

Gprof example output

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>33.34</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.04</td>
</tr>
</tbody>
</table>

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 (2)

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>1</td>
<td>start [1]</td>
</tr>
<tr>
<td>[2]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1</td>
<td>main [2]</td>
</tr>
<tr>
<td>[3]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1</td>
<td>report [3]</td>
</tr>
</tbody>
</table>

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

oprof example 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
...

Source: http://oprofile.sourceforge.net/examples/
oprof example output

```bash
$ opreport
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a ...
504834 5.6827 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
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/
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 enabled/disabled

Benchmarks must check results are correct!

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
  - data are not controllable for I/O
    - 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

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

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

Benchmark:
- 300.twolf from SPEC CPU2000 suite

Platform:
- Dell Latitude D600
  - Pentium M @ 1.6GHz
  - 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% variation in execution time between "identical" runs!

twolf on Linux: What's going on?
How to Measure and Compare Performance

Measure as directly as possible:
- 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

Avoid incorrect conclusions from pathological cases
- Typical cases:
  - sequential access may be optimised by underlying hardware/disk controller...
  - there may or may not be large differences between sequentially upward...
  - 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
    - eg, 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
  - 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**

<table>
<thead>
<tr>
<th>Loop O/H</th>
<th>Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache (2KB)</td>
<td></td>
</tr>
<tr>
<td>L2 cache (16KB)</td>
<td></td>
</tr>
</tbody>
</table>

---

**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**

```c
double t0 = time();
for (i=0; i<NMAX; i++) { // and stuff }
double t1 = time();
for (i=0; i<NMAX; i++) { // and stuff }
double t2 = time();
printf("Cost is %.6f\n", (t2-2*t1+t0)*1000000/NMAX);
```

Beware of compiler optimizations!

---

**Benchmarking Against Competitors**

**One thing is to compare against published data**
- Need to be really careful to ensure comparable setup
  - same hardware, or *very* 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 least it's incompetence
  - Make sure you understand exactly what's going on!
    - run additional traces/profiles/microbenchmarks to explain performance difference
    - explain this in your paper/report!
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 affected, 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  

- 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!  

 Benchmark | Native | Virtualized | Difference |
------------|--------|-------------|------------|
 Branch mis-pred | 1 | 1 | 0 |
 D-cache miss | 0 | 0 | 0 |
 Instruction | 0 | 0 | 0 |
 D-pTLB miss | 0 | 0 | 0 |
 I-pTLB miss | 0 | 0 | 0 |
 Macro TLB miss | 30 | 125 | 95 |
 D-wall cycles | 0 | 27 | 27 |
 I-wall cycles | 0 | 45 | 45 |
 Total Cycles | 93 | 292 | 199 |

Good or bad?

More of the Same...  

- First step: improve representation!  

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td>1.63</td>
<td>2.25</td>
<td>230</td>
</tr>
<tr>
<td>Create/close</td>
<td>11</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>suspend</td>
<td>0.81</td>
<td>1.54</td>
<td>736</td>
</tr>
</tbody>
</table>

First step: improve representation!  

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td>1.63</td>
<td>2.25</td>
<td>230</td>
</tr>
<tr>
<td>Create/close</td>
<td>11</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>suspend</td>
<td>0.81</td>
<td>1.54</td>
<td>736</td>
</tr>
</tbody>
</table>

More of the Same...  

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDes16_Num0</td>
<td>1.2900</td>
<td>1.2936</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDes16_RadixHex1</td>
<td>0.7110</td>
<td>0.7129</td>
<td>0.27%</td>
</tr>
<tr>
<td>TDes16_RadixDecimal2</td>
<td>1.2338</td>
<td>1.2373</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDes16_Num_RadixOctal3</td>
<td>0.6306</td>
<td>0.6324</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDes16_Compare5</td>
<td>0.9621</td>
<td>0.9647</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDesC16_MatchF9</td>
<td>1.1060</td>
<td>1.1090</td>
<td>0.28%</td>
</tr>
</tbody>
</table>

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...  

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDes16_Name0</td>
<td>1.2900</td>
<td>1.2936</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDesC16_Name1</td>
<td>0.7110</td>
<td>0.7129</td>
<td>0.27%</td>
</tr>
<tr>
<td>TDesC16_Name2</td>
<td>1.2338</td>
<td>1.2373</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDesC16_Name3</td>
<td>0.6306</td>
<td>0.6324</td>
<td>0.28%</td>
</tr>
<tr>
<td>TDesC16_Name4</td>
<td>1.1060</td>
<td>1.1090</td>
<td>0.28%</td>
</tr>
</tbody>
</table>

Note: these are purely user-level operations!  
- What’s going on?
### Yet 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 us</td>
</tr>
<tr>
<td>TDes16_RadixHex1</td>
<td>0.7110</td>
<td>0.7129</td>
<td>0.27%</td>
<td>2.7 us</td>
</tr>
<tr>
<td>TDes16_RadixDecimal2</td>
<td>0.7154</td>
<td>0.7135</td>
<td>0.38%</td>
<td>2.6 us</td>
</tr>
<tr>
<td>TDes16_RadixOctal3</td>
<td>0.6506</td>
<td>0.6524</td>
<td>0.28%</td>
<td>2.6 us</td>
</tr>
<tr>
<td>TDes16_RadixBinary4</td>
<td>1.0088</td>
<td>1.0110</td>
<td>0.27%</td>
<td>2.7 us</td>
</tr>
<tr>
<td>TDes16_COMPARE0</td>
<td>0.9621</td>
<td>0.9647</td>
<td>0.27%</td>
<td>2.7 us</td>
</tr>
<tr>
<td>TDes16_COMPARE1</td>
<td>1.0053</td>
<td>1.0084</td>
<td>0.32%</td>
<td>2.7 us</td>
</tr>
<tr>
<td>TDesC16_COMPARE0</td>
<td>1.0060</td>
<td>1.0090</td>
<td>0.27%</td>
<td>2.7 us</td>
</tr>
<tr>
<td>TDesC16_COMPARE1</td>
<td>1.0054</td>
<td>1.0090</td>
<td>0.27%</td>
<td>2.7 us</td>
</tr>
</tbody>
</table>

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

- **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