Performance Considerations

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

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

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 performance improvement in important cases
  • only needed if you 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
Does the mean geometric make sense?

Proper figure of merit is processing cost per unit data

Beware Partial Data

Obtaining an Overall Score for a BM Suite

Benchmark Suite Abuse

Beware Partial Data

Profile
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

Example gprof 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>33.34</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>16.67</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>16.67</td>
<td>0.05</td>
<td>0.01</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
</tr>
</tbody>
</table>

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

Example oprof Output

```
$ oreport --exclude-dependent
CPU: PIII, speed 863.195 MHz (estimated)

Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a ...

%  Count            Percentage  CPU: PIII, speed 863.195 MHz (estimated)

350385   39.3546  cc1plus
450385   49.8046  cc1plus
28201    2.9837  libc-2.3.2.so
27194    2.9557  vmlinux
677  0.0736   uhci_hcd
163209   17.4008  lyx
```

Drilldown of top consumers

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

Example oprof Output

```
$ oreport
CPU: PIII, speed 863.195 MHz (estimated)

Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a ...

%  Count            Percentage  CPU: PIII, speed 863.195 MHz (estimated)

465385   49.5026  cc1plus
38201    3.5667  libo-2.3.2.so
27194    2.9557  vmlinux
677  0.0736   uhci_hcd
163209   17.4008  lyx
```

Drilldown of top consumers

Source: http://oprofile.sourceforge.net/examples/
### PMU Event Examples: ARM11 (Armv6)

<table>
<thead>
<tr>
<th>Ev #</th>
<th>Definition</th>
<th>Ev #</th>
<th>Definition</th>
<th>Ev #</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>I-cache miss</td>
<td>0x0b</td>
<td>D-cache miss</td>
<td>0x23</td>
<td>Funcs. call</td>
</tr>
<tr>
<td>0x01</td>
<td>Inst. buffer stall</td>
<td>0x0d</td>
<td>D-cache writeback</td>
<td>0x24</td>
<td>Funcs. return</td>
</tr>
<tr>
<td>0x02</td>
<td>Data depend. stall</td>
<td>0x0e</td>
<td>PC changed by SW</td>
<td>0x25</td>
<td>Funcs. ret. predict</td>
</tr>
<tr>
<td>0x03</td>
<td>Inst. micro-TLB miss</td>
<td>0x0f</td>
<td>Main TLB miss</td>
<td>0x26</td>
<td>Funcs. ret. mispred</td>
</tr>
<tr>
<td>0x04</td>
<td>Data micro-TLB miss</td>
<td>0x10</td>
<td>Ext data access</td>
<td>0x27</td>
<td>...</td>
</tr>
<tr>
<td>0x05</td>
<td>Branch executed</td>
<td>0x11</td>
<td>Load-store unit stall</td>
<td>0x30</td>
<td>...</td>
</tr>
<tr>
<td>0x06</td>
<td>Branch mispredicted</td>
<td>0x12</td>
<td>White buffer drained</td>
<td>0x38</td>
<td>...</td>
</tr>
<tr>
<td>0x07</td>
<td>Inst executed</td>
<td>0x13</td>
<td>Cycles FIRQ disabled</td>
<td>0x39</td>
<td>...</td>
</tr>
<tr>
<td>0x08</td>
<td>D-cache acc cachable</td>
<td>0x14</td>
<td>Cycles IRQ disabled</td>
<td>0x3a</td>
<td>...</td>
</tr>
<tr>
<td>0x09</td>
<td>D-cache access any</td>
<td>0x20</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Performance Analysis

#### Significance of Measurements
- Standard approach: repeat & collect stats
- Computer systems are high deterministic
  - Typically variances are tiny, except across WAN
- Watch for divergence from this hypothesis, could indicate
- **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 ($\mu$) and standard deviation ($\sigma$)
- Don’t believe any effect that is less than a standard deviation
  - $10.2 \pm 1.5$ is not significantly different from $11.5$
  - Be highly suspicious if it is less than two standard deviations
  - $10.2 \pm 0.8$ may not be different from $11.5$

**For systems work, must be very suspicious if $\sigma$ is not small!**

**Standard deviation is meaningless for small samples!**
- Ok if effect $\gg \sigma$
- Use t-test if in doubt!

#### Example from SPEC CPU2000

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

**Cache warmup**
**Clock resolution**

**Lesson:** Need mental model of system, look for hidden parameters if model fails!

#### How To Measure and Compare Performance

**Noisy data:**
- Eliminate sources of noise, re-run from same initial state
  - single-user mode
  - dedicated network
- Not always possible, may have to live with noise
- Possible ways out:
  - Ignore highest & lowest values
  - Take floor of data
  - Maybe minimum is what matters

**Proceed with extreme care!**
**Document and justify!**
How To Measure and Compare Performance

Vary inputs, check outputs!
- Vary data and addresses!
- e.g. 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 pseudo-random with same seed

Real-World Example

Benchmark:
- 3D0.twolf from SPEC CPU2000 suite

Platform:
- Dell Latitude D600
- Pentium III @ 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...

Real-World Example

Benchmark:
- 3D0.twolf from SPEC CPU2000 suite

Platform:
- Dell Latitude D600
- Pentium III @ 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...

A Few More Performance Evaluation Rules

- Vary one parameter at a time
- Record 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

Most Important: Use a Model/Hypothesis

Model of the system that says how the system should behave
- Benchmarking should aim to support or disprove that model
- Need to think about this 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!
Loop and Timing Overhead

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

```c
#include <stdio.h>

int main()
{
    int i, t0, t1, t2;
    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 \$\$\n", (t2-2*t1+t0)*1000000/MAX);
}
```

Beware compiler optimisations!

Relative vs Absolute Data

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

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

Benchmarking crime: Relative numbers only!

Data Range

Example: Scaling database load

![Graph showing data range](image)

seems to scale well?

Looking a bit further

Benchmarking crime: Selective data set hiding deficiencies!

Benchmarking Ethics

Comparisons with prior work

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

Benchmarking crime: Unfair benchmarking of competitor!

Other Ways of Cheating with Benchmarks

- Benchmark-specific optimisations
  - Recognise particular benchmark, insert BM-specific optimised code
  - Popular with compiler-writers
  - Pioneered for smartphone performance by Samsung
  - http://bgr.com/2014/03/05/samsung-benchmark-cheating-ends
- Benchmarking simulated system
  - … with simulation simplifications matching model assumptions
- Uniprocessor benchmarks to “measure” multicore scalability
  - … by running multiple copies of benchmark on different cores
- CPU-intensive benchmark to “measure” networking performance

These are simply lies, and I’ve seen them all!

Understanding Performance
What is “Good” Performance?

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

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

Remember: progressive and conservative criterion!

Remember: BM ethics!

Most elegant, but hardest!

Performance Counters Are Your Friends!

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

Good or bad?

Real-World Example: Virtualisation Overhead

Symbian null-syscall microbenchmark:

- Native: 0.24µs, virtualized (on OKL4): 0.79µs. 
  - 230% overhead
- ARM11 processor runs at 368 MHz:
  - Native: 0.24µs = 93 cy
  - Virtualized: 0.79µs = 292 cy
  - Overhead: 0.55µs = 199 cy
  - Cache-miss penalty = 20 cy

Model:
- native: 2 mode switches, 0 context switches, 1 × save+restore state
- virt.: 4 mode switches, 2 context switches, 3 × save+restore state

And Another One…

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:
http://gernot-heiser.org/benchmarking/crimes.html