Performance Metrics

Why study performance metrics?
• determine the benefit/lack of benefit of designs
• computer design is too complex to intuit performance & performance bottlenecks
• have to be careful about what you mean to measure & how you measure it

What you should get out of this discussion
• good metrics for measuring computer performance
• what they should be used for
• what metrics you shouldn’t use & how metrics are misused

Performance of Computer Systems

Many different factors to take into account when determining performance:
• Technology
  • circuit speed (clock, MHz)
  • processor technology (how many transistors on a chip)
• Architecture & microarchitecture
  • type of processor (RISC or CISC)
  • configuration of the memory hierarchy
  • type of I/O devices
  • number of processors in the system
• Software
  • quality of the compilers
  • organization & quality of OS, databases, etc.
“Principles” of Experimentation

Meaningful metrics
execution time & component metrics that explain it

Reproducibility
machine configuration, compiler & optimization level, OS, input

Real programs
no toys, kernels, synthetic programs
SPEC is the norm (integer, floating point, graphics, webserver)
TPC-B, TPC-C & TPC-D for database transactions

Simulation
long executions, warm start to mimic steady-state behavior
usually applications only; some OS simulation
simulator “validation” & internal checks for accuracy

Metrics that Measure Performance

Raw speed: peak performance (never attained)

Execution time: time to execute one program from beginning to end
• the “performance bottom line”
• wall clock time, response time
• Unix time function: 13.7u 23.6s 18:27 3%

Throughput: total amount of work completed in a given time
• instructions / cycle
• transactions (database) or packets (web servers) / second
• an indication of how well hardware resources are being used
• good metrics for chip designers or managers of computer systems

(Often improving execution time will improve throughput & vice versa.)

Component metrics: subsystem performance, e.g., memory behavior
• help explain how execution time was obtained
• pinpoints performance bottlenecks
Execution Time

\[
\text{Performance}_A = \frac{1}{\text{ExecutionTime}_A}
\]

Processor A is faster than processor B, i.e.,
\[
\text{ExecutionTime}_A < \text{ExecutionTime}_B
\]
\[
\text{Performance}_A > \text{Performance}_B
\]

Relative Performance

\[
\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{ExecutionTime}_B}{\text{ExecutionTime}_A} = n
\]

performance of A is \(n\) times greater than B
execution time of B is \(n\) times longer than A

CPU Execution Time

The time the CPU spends executing an application
- no memory effects
- no I/O
- no effects of multiprogramming

\[
\text{CPUExecutionTime} = \text{CPUclockCycles} \times \text{clockCycleTime}
\]

Cycle time (clock period) is measured in time or rate
- clock cycle time = 1/clock cycle rate

\[
\text{CPUExecutionTime} = \frac{\text{CPUclockCycles}}{\text{clockCycleRate}}
\]

- clock cycle rate of 1 MHz ⇒ cycle time of 1 \(\mu\)s
- clock cycle rate of 1 GHz ⇒ cycle time of 1 ns
CPI

\[ \text{CPU Clock Cycles} = \text{NumberOfInstructions} \times \text{CPI} \]

Average number of clock cycles per instruction
• throughput metric
• component metric, not a measure of performance
• used for processor organization studies, given a fixed compiler & ISA

Can have different CPIs for classes of instructions
e.g., floating point instructions take longer than integer instructions

\[ \text{CPU Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i) \]
where \( \text{CPI}_i = \text{CPI} \) for a particular class of instructions
where \( C_i = \) the number of instructions of the \( i^{th} \) class that have been executed

Improving part of the architecture can improve a \( \text{CPI}_i \)
• Talk about the contribution to CPI of a class of instructions

CPU Execution Time

\[ \text{CPU Execution Time} = \text{NumberOfInstructions} \times \text{CPI} \times \text{Clock Cycle Time} \]

To measure:
• execution time: depends on all 3 factors
  • time the program
  • number of instructions: determined by the ISA
    • programmable hardware counters
    • profiling
      • count number of times each basic block is executed & multiply by the number of instructions in each basic block
      • instruction sampling
  • CPI: determined by the ISA & implementation
    • simulator: interpret (in software) every instruction & calculate the number of cycles it takes to simulate it
  • clock cycle time: determined by the implementation & process technology

Factors are interdependent:
• RISC: increases instructions/program, but decreases CPI & clock cycle time because the instructions are simple
• CISC: decreases instructions/program, but increases CPI & clock cycle time because many instructions are more complex
Metrics Not to Use

MIPS (millions of instructions per second)
\[
\frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}
\]
- instruction set-dependent (even true for similar architectures)
- implementation-dependent
- compiler technology-dependent
- program- & input-dependent
+ intuitive: the higher, the better

MFLOPS (millions of floating point operations per second)
\[
\frac{\text{floating point operations}}{\text{execution time} \times 10^6}
\]
+ FP operations are independent of FP instruction implementation
- different machines implement different FP operations
- different FP operations take different amounts of time
- only measures FP code

static metrics (code size)

Means

Measuring the performance of a workload
• arithmetic: used for averaging execution times
\[
\left( \sum_{i=1}^{n} \text{time}_i \right) \times \frac{1}{n}
\]
• harmonic: used for averaging rates
\[
\frac{n}{\sum_{i=1}^{n} \frac{1}{\text{rate}_i}} = \frac{1}{\text{arithmetic mean}}
\]
• weighted means: the programs are executed with different frequencies, for example:
\[
\left( \sum_{i=1}^{n} \text{time}_i \times \text{weight}_i \right) \times \frac{1}{n}
\]
Means

<table>
<thead>
<tr>
<th></th>
<th>FP Ops</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Computer A</td>
</tr>
<tr>
<td>program 1</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>program 2</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>total</td>
<td>1001</td>
<td>110</td>
</tr>
<tr>
<td>arith mean</td>
<td>500.5</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FP Ops</th>
<th>Rate (FLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Computer A</td>
</tr>
<tr>
<td>program 1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>program 2</td>
<td>100</td>
<td>.1</td>
</tr>
<tr>
<td>harm mean</td>
<td></td>
<td>.2</td>
</tr>
<tr>
<td>arith mean</td>
<td></td>
<td>50.1</td>
</tr>
</tbody>
</table>

Computer C is ~25 times faster than A when measuring execution time.
Still true when measuring MFLOPS (a rate) with the harmonic mean.

Speedup

speedup = \( \frac{\text{execution time}_{\text{before Improvement}}}{\text{execution time}_{\text{after Improvement}}} \)

Amdahl’s Law:
Performance improvement from speeding up a part of a computer system is limited by the proportion of time the enhancement is used.