Computer Architecture
Computer performance

http://d3s.mff.cuni.cz/teaching/computer_architecture/

Lubomír Bulej
bulej@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics
Great ideas in computer architecture

- Design for Moore’s law
- Use abstraction to simplify design
- Make the common cast fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy
Processor and memory technologies

- **Impact of technology**
  - What computers will be able to do
  - How fast will computers evolve

- **Race to design a better computer**
  - Embracing the latest in electronic technology

<table>
<thead>
<tr>
<th>Year</th>
<th>Technology</th>
<th>Relative performance / unit cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1951</td>
<td>Vacuum tube</td>
<td>1</td>
</tr>
<tr>
<td>1965</td>
<td><strong>Transistor</strong></td>
<td>35</td>
</tr>
<tr>
<td>1975</td>
<td>Integrated circuit (low integration)</td>
<td>900</td>
</tr>
<tr>
<td>1995</td>
<td>Integrated circuit (very large scale integration, VLSI)</td>
<td>2 400 000</td>
</tr>
<tr>
<td>2013</td>
<td>Integrated circuit (ultra large scale integration, ULSI)</td>
<td>250 000 000 000</td>
</tr>
</tbody>
</table>
Moore’s “law”

- **Gordon Moore** (*1929)
  - One of the founders of Intel
  - **Prediction:** The number of transistors integrated on a single chip will double every 18 – 24 months
    - 1960s
    - Smaller transistors allow higher speeds and capacities
    - Often applied to other domains
      - Storage capacity, network bandwidth
Moore’s “law” (2)

- Exponential growth in the last 40 years!
  - Keeping Moore’s “law” valid requires tremendous and continuous advances in technology
    - So far in a single domain (semiconductor transistors)
    - There are hard physical limits (quantum tunnel effect, waste heat, quantum noise)
  - Compromises needed
    - Number of transistors does not correspond to computational power for sequential algorithms
Growth of capacity per DRAM chip

Source: P&H
## Program performance

<table>
<thead>
<tr>
<th>HW or SW component</th>
<th>Impact on performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Number of source-level statements and of I/O operations executed</td>
</tr>
<tr>
<td>Programming language, compiler, computer architecture</td>
<td>Number of instructions for each source-level statement</td>
</tr>
<tr>
<td>Processor, memory</td>
<td>How fast instructions can be executed</td>
</tr>
<tr>
<td>I/O system (hardware, operating system)</td>
<td>How fast I/O operations can be executed</td>
</tr>
</tbody>
</table>
Why care about performance?

- **Comparing/ranking computers**
  - Cheaper and/or better product wins
    - Personal computers: fierce performance competition
    - Embedded computers: optimize price of final product
  - Important for buyers → important for designers and producers

- **Performance impact of architectural changes**
  - Systematic assessment is the only indication whether some progress is really a progress
How to define computer performance?

- **Computer A is “better” than computer B**
  - What does it mean? Better in what?
  - Is a truck “better” car than a sports car?
  - Is a Concorde “better” plane than a Boeing 747?

<table>
<thead>
<tr>
<th>Airplane</th>
<th>Capacity [persons]</th>
<th>Range [km]</th>
<th>Cruising speed [km/h]</th>
<th>Throughput [pers·km/h]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 777</td>
<td>375</td>
<td>9000</td>
<td>905</td>
<td>339375</td>
</tr>
<tr>
<td>Boeing 747</td>
<td>470</td>
<td>7700</td>
<td>905</td>
<td>425350</td>
</tr>
<tr>
<td>Concorde</td>
<td>132</td>
<td>7400</td>
<td>2158</td>
<td>284856</td>
</tr>
<tr>
<td>Douglas DC-8-50</td>
<td>146</td>
<td>16000</td>
<td>810</td>
<td>118260</td>
</tr>
</tbody>
</table>
How to define computer performance?

- Basic criteria
  - What do we need?
  - What do we compare?
- Basic metrics
  - Execution time (response time)
    - Time to complete a particular task
    - Important for users
  - Throughput
    - Amount of work completed in unit time
    - Important for server or data center operators
How to define computer performance?

- **Performance based on execution time**
  - We desire: higher number = higher performance
  - Execution time is the opposite → needs fixing

\[
\text{Performance}_X = \frac{1}{\text{Execution time}_X}
\]

- **Now we can compare performance**

\[
\text{Performance}_X > \text{Performance}_Y \\
\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}
\]

\[
\text{Execution time}_Y > \text{Execution time}_X
\]
Relative performance

Relating performance of two computers

- X is n-times faster than Y

\[
\frac{\text{Performance}_X}{\text{Performance}_Y} = n
\]

- If X is n-times faster than Y, then execution time on Y is n-times as long as on X

\[
\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n
\]
Performance: user perspective

- **Total execution time**
  - *Wall-clock time, response time, elapsed time*
  - Includes waiting for I/O operations, OS overhead, etc.
    - Including sharing resources (CPU) with other users
  - Reflects whole-system performance

- **Processor time**
  - *CPU execution time, CPU time*
  - Time when the program was actually executing
    - Does not include waiting for I/O operations
    - Does not include time when the program was not running
    - Includes OS overhead (*user* vs *system* CPU time)
  - Reflects processor performance
Performance: CPU designer perspective

- **Speed for executing instructions**
  - Clock rate
  - Clock cycle length

\[
\text{CPU execution time} = \frac{\text{CPU clock cycles}}{\text{CPU clock rate}}
\]

\[
\text{CPU execution time} = \text{CPU clock cycles} \times \text{CPU clock cycle time}
\]
Performance: compiler perspective

- **Average number of cycles per instruction**
  - *Clock cycles per instruction* (CPI)
  - Specific to a particular program or its part
  - Allows comparing different implementations of the same architecture
    - Given a fixed number of instructions

\[
\text{CPU clock cycles} = \text{CPI} \times \text{Number of instructions}
\]
Classic processor performance equation

- Relates number of instructions, CPI and clock cycle length
  - 3 different factors influencing performance
    - Allows comparing different implementations
    - Allows assessing alternative architectures

\[
\text{CPU execution time} = \text{CPI} \times \text{Number of instructions} \times \text{CPU clock cycle time}
\]

\[
\text{CPU execution time} = \frac{\text{CPI} \times \text{Number of instructions}}{\text{CPU clock rate}}
\]
## Alternative view of program performance

<table>
<thead>
<tr>
<th>Component</th>
<th>Affects what?</th>
<th>Affects how?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong></td>
<td>Instruction count</td>
<td>Number and kind of source program statements and operations, data types</td>
</tr>
<tr>
<td></td>
<td>CPI</td>
<td>(integer vs. floating point)</td>
</tr>
<tr>
<td><strong>Programming Language</strong></td>
<td>Instruction count</td>
<td>Kind of source program statements, abstractions used to express the algorithm.</td>
</tr>
<tr>
<td></td>
<td>CPI</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>Instruction count</td>
<td>How program statements are translated to machine code, choice and layout of</td>
</tr>
<tr>
<td></td>
<td>CPI</td>
<td>instructions.</td>
</tr>
<tr>
<td><strong>Instruction set</strong></td>
<td>Instruction count</td>
<td>Instructions available to compiler, cost in cycles for each instruction,</td>
</tr>
<tr>
<td><strong>architecture</strong></td>
<td>CPI, Clock rate</td>
<td>overall clock rate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pitfall: Unrealistic expectations

- Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of the improvement.

- Total execution time: 100 s
  - Out of which multiplication operations: 80 s
- How much do we need to improve multiplication to make the program run 5× faster?
Some „back of the envelope“ calculations

\[
\text{Execution}_{\text{fast}} = \frac{\text{Execution}_{\text{slow}}}{5}
\]

\[
\text{Execution}_{\text{slow}} = 80 + 20
\]

\[
\text{Execution}_{\text{fast}} = \frac{80}{n} + 20
\]

\[
\frac{80}{n} + 20 = \frac{80 + 20}{5}
\]

\[
\frac{80}{n} + 20 = 20
\]

\[
\frac{80}{n} = 0
\]

\[
80 \neq 0
\]
Pitfall: Wrong performance metrics

- Using a subset of the performance equation as a performance metric
  - Using a single factor is almost always wrong
  - Using two factors may be valid in limited context
    - Easily misused: dependencies between factors
  - Other metrics dressing up other known factors
Pitfall: Wrong performance metrics (2)

- **MIPS (Million Instructions Per Second)**
  - Instruction execution rate
  - Intuitive (higher number → faster computer)
  - **Problems**
    - Ignores instruction capabilities, execution time of individual instructions, different number of instructions for different ISAs
      - Impossible to compare computers with different ISA
    - Depends on the instruction mix of a particular program (a single value to not represent the performance of a computer)

\[
MIPS = \frac{\text{Instruction count}}{10^6 \times \text{Execution time}}
\]
Processor performance

- **Performance while executing a particular program**
  - Depends on the number of instructions, average number of cycles per instructions (CPI), clock cycle length (or clock rate)
  - **No single factor can completely express performance**
    - Reducing number of instructions → architecture with lower clock frequency or higher CPI
    - CPI depends on the **instruction mix** (frequency and type of executed instructions) of a given program
      - Code with the lowest number of instructions is not necessarily the fastest
**Processor performance (2)**

- **Performance while executing a particular program**
  - The only complete and reliable metrics is processor time
    - Does not tell anything about processor time for other programs
Performance evaluation

- Comparing performance of different computers
  - Easy for one specific program (processor execution time)
  - Comparing isolated components (clock rate, CPI, number of instructions) not indicative for other programs
  - How to approximate performance with respect to a set of programs?
Performance evaluation (2)

- **Workload**
  - A set of programs and tasks capturing a user’s workload
  - Compare execution time of the workload on different computers
  - Difficult to define (domain specific)
  - Difficult to automate (repeated execution)

- **Benchmark**
  - Program specifically made to measure performance
  - Set of benchmarks
    - Statistically relevant representative of a typical workload
    - Hoping that benchmark results will reflect how well a computer will perform with the user’s workload
Performance evaluation (3)

- **SPEC (Standard Performance Evaluation Corporation)**
  - Funded by commercial and non-commercial entities
    - Manufacturers of processors and computers
    - Producers of compilers, operating systems
    - Research institutes
  - **Goal**: Define a standard set of benchmarks to enable comparison of computer systems’ performance
    - Different benchmarks for different workloads
    - Primarily focusing on CPU performance
    - Now CPU power, GPU performance & power, compilers, databases, e-mail systems, transaction processing, etc.
**Processor performance**

- **CINT2006** (integer computation)
  - 12 benchmarks (C compiler, chess algorithm, quantum computer simulation, etc.)

- **CFP2006** (floating point computation)
  - 17 benchmarks (finite elements, molecular dynamics, etc.)

- **SPECratio**
  - Ratio of reference vs. measured benchmark execution time
  - Summary score (single number): geometric mean

\[
\sqrt[n]{\prod_{i=1}^{n} \text{SPECratio}_i}
\]
## SPEC CINT2006 on AMD Opteron X4

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Instruction Count $\times 10^9$</th>
<th>CPI</th>
<th>Clock cycle time (seconds $\times 10^9$)</th>
<th>Execution Time (seconds)</th>
<th>Reference Time (seconds)</th>
<th>SPECratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>perl</td>
<td>2,118</td>
<td>0.75</td>
<td>0.4</td>
<td>637</td>
<td>9,770</td>
<td>15.3</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>bzip2</td>
<td>2,389</td>
<td>0.85</td>
<td>0.4</td>
<td>817</td>
<td>9,650</td>
<td>11.8</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>gcc</td>
<td>1,050</td>
<td>1.72</td>
<td>0.4</td>
<td>724</td>
<td>8,050</td>
<td>11.1</td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>mcf</td>
<td>336</td>
<td>10.00</td>
<td>0.4</td>
<td>1,345</td>
<td>9,120</td>
<td>6.8</td>
</tr>
<tr>
<td>Go game (AI)</td>
<td>go</td>
<td>1,658</td>
<td>1.09</td>
<td>0.4</td>
<td>721</td>
<td>10,490</td>
<td>14.6</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>hmmmer</td>
<td>2,783</td>
<td>0.80</td>
<td>0.4</td>
<td>890</td>
<td>9,330</td>
<td>10.5</td>
</tr>
<tr>
<td>Chess game (AI)</td>
<td>sjeng</td>
<td>2,176</td>
<td>0.96</td>
<td>0.4</td>
<td>837</td>
<td>12,100</td>
<td>14.5</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>libquantum</td>
<td>1,623</td>
<td>1.61</td>
<td>0.4</td>
<td>1,047</td>
<td>20,720</td>
<td>19.8</td>
</tr>
<tr>
<td>Video compression</td>
<td>h264avc</td>
<td>3,102</td>
<td>0.80</td>
<td>0.4</td>
<td>993</td>
<td>22,130</td>
<td>22.3</td>
</tr>
<tr>
<td>Discrete event simulation</td>
<td>omnetpp</td>
<td>587</td>
<td>2.94</td>
<td>0.4</td>
<td>690</td>
<td>6,250</td>
<td>9.1</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>astar</td>
<td>1,082</td>
<td>1.79</td>
<td>0.4</td>
<td>773</td>
<td>7,020</td>
<td>9.1</td>
</tr>
<tr>
<td>XML parsing</td>
<td>xalancbmk</td>
<td>1,058</td>
<td>2.70</td>
<td>0.4</td>
<td>1,143</td>
<td>6,900</td>
<td>6.0</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11.7</td>
</tr>
</tbody>
</table>

*Source: P&H*
## SPEC CINT2006 on Intel Core i7 920

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Instruction Count x $10^9$</th>
<th>CPI</th>
<th>Clock cycle time (seconds x $10^{-9}$)</th>
<th>Execution Time (seconds)</th>
<th>Reference Time (seconds)</th>
<th>SPECratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>perl</td>
<td>2252</td>
<td>0.60</td>
<td>0.376</td>
<td>508</td>
<td>9770</td>
<td>19.2</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>bzip2</td>
<td>2390</td>
<td>0.70</td>
<td>0.376</td>
<td>629</td>
<td>9650</td>
<td>15.4</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>gcc</td>
<td>794</td>
<td>1.20</td>
<td>0.376</td>
<td>358</td>
<td>8050</td>
<td>22.5</td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>mcf</td>
<td>221</td>
<td>2.66</td>
<td>0.376</td>
<td>221</td>
<td>9120</td>
<td>41.2</td>
</tr>
<tr>
<td>Go game (AI)</td>
<td>go</td>
<td>1274</td>
<td>1.10</td>
<td>0.376</td>
<td>527</td>
<td>10490</td>
<td>19.9</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>hhammer</td>
<td>2616</td>
<td>0.60</td>
<td>0.376</td>
<td>590</td>
<td>9330</td>
<td>15.8</td>
</tr>
<tr>
<td>Chess game (AI)</td>
<td>sjeng</td>
<td>1948</td>
<td>0.80</td>
<td>0.376</td>
<td>586</td>
<td>12100</td>
<td>20.7</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>libquantum</td>
<td>659</td>
<td>0.44</td>
<td>0.376</td>
<td>109</td>
<td>20720</td>
<td>190.0</td>
</tr>
<tr>
<td>Video compression</td>
<td>h264avc</td>
<td>3793</td>
<td>0.50</td>
<td>0.376</td>
<td>713</td>
<td>22130</td>
<td>31.0</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>omnetpp</td>
<td>367</td>
<td>2.10</td>
<td>0.376</td>
<td>290</td>
<td>6250</td>
<td>21.5</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>astart</td>
<td>1250</td>
<td>1.00</td>
<td>0.376</td>
<td>470</td>
<td>7020</td>
<td>14.9</td>
</tr>
<tr>
<td>XML parsing</td>
<td>xalanbmk</td>
<td>1045</td>
<td>0.70</td>
<td>0.376</td>
<td>275</td>
<td>6900</td>
<td>25.1</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>25.7</td>
</tr>
</tbody>
</table>

Source: P&H
End of the golden era
The Power Wall

Clock rate [MHz]

Power [W]

Source: P&H
The Power Wall (2)

- **Complementary Metal Oxide Semiconductor (CMOS)**
  - Dominant technology for integrated circuits
  - Very low static consumption
  - Dynamic power consumption
    - Capacitive load (conductors, transistors, output load)
    - Operating voltage (affects switching speed)
    - Switching frequency (function of clock rate)

\[
\text{Power} \approx \frac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}
\]
Real-world impact

- In the last 20 years
  - Clock rate growth by factor of 1000
  - Power growth (only) by factor of 30
  - How: voltage dropped from 5 V to 1 V
    - 15% reduction with each generation

Example

- New technology results in 85% capacitive load of old technology. Also, the operating voltage and switching frequency can be reduced by 15% to save power.
  - Compared to a processor based on the previous technology, a new processor would only consume 52% of the power.
The Power Wall (4)

- **Further lowering of voltage difficult/impossible**
  - Makes transistors too leaky
  - 40% of power consumption in server chips is due to leakage
  - Low signal/noise ratio
    - Difficult to tell ones from zeroes reliably
- **Cooling cannot be easily improved**
  - Power dissipated from a rather small area of the chip
  - Parts of chip not used in a clock cycle can be turned off
  - Water (and other) cooling techniques too complex/expensive
    - Not even an option for personal mobile devices
The Power Wall (5)

- New way to improve performance needed
  - Dramatic change in microprocessor design

  The switch from Uniprocessors to Multiprocessors
Growth in processor performance

Source: P&H
Multiprocessor systems

- **Then**
  - Multiple physical processors (*multiprocessor*)
  - Where: Supercomputers, high-end servers
  - Rare in personal and embedded computers
- **Now**
  - Multiple processor *cores* in a single microprocessor package
    - Post-Moore’s „law“ world, shrinking transistors difficult/expensive,
      but we can still put more of them on a single (bigger) chip
  - Where: everywhere
Multicore systems

- **Impact on performance**
  - Increased throughput
    - Processing more requests in parallel
  - Clock rate and CPI remain the same
    - Performance of sequential algorithms stays the same

- **Impact on programmers**
  - Technology does not make programs faster (anymore)
  - Programs need to take advantage of multiple cores
    - Better APIs needed (executor frameworks, parallel collections, ...)
  - Programs need to be improved as number of cores increases
    - Increasing number of cores from 4 to 32 will not make a parallel program 8 times faster
Why is this such a big deal?

- **Fundamental change in HW/SW interface**
  - Parallelism was always important, but used to be hidden
    - Instruction-level parallelism, pipelining, and other techniques
    - Programmer and compiler alike produced sequential code
  - Now parallelism needs to be explicit!

- **Parallel architectures known for 40 years...**
  - ... but whoever relied on explicit parallelism failed!
    - Programmers never accepted the new paradigm
  - Now the whole IT industry bets on programmers to switch to explicit parallelism
Why is parallel programming difficult?

- **Programming focused on performance**
  - Increases difficult of programming
    - Not only does the program need to be correct, it also needs to be fast
    - If you don’t need performance, just write a sequential program.
  - People think “sequentially” in a “single thread”

- **Problem: split work equally between processors**
  - Ensure that the overhead of planning and coordinating the work does not take away the performance benefit
Why is parallel programming difficult? (2)

- **Real-world analogy**
  - 1 reporter writes 1 article in 2 hours
    - Can we get 8 reporters to write 1 article in 15 minutes?

- **Actual problems**
  - Scheduling
    - Who writes what?
  - Load balancing
    - No reporter is idle
  - Communication and synchronization overhead
    - How to put the final article together?
Amdahl’s law

- Gene Amdahl (* 1922)
  - Multiple variants
  - Most general for theoretical speed-up of a sequential algorithm using multiple threads (formulated in 1967)
  - A quantitative version of the law of diminishing returns

- The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

\[
\text{Speedup}(n) = \frac{1}{B + \frac{1}{n}(1 - B)} \quad n \in \mathbb{N} \\
B \in \langle 0, 1 \rangle
\]
Practical impact

- Make the common case fast
  Optimize for the common case

- Optimization impacts the common case the most
  - The common case is often much simpler than the special cases, and therefore easier to optimize

- Even massive optimization of special cases often provide only very little benefit compared to modest optimization of the common cases