

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

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 777?

Airplane	Capacity [persons]	Range [km]	Cruising speed [km/h]	Throughput [pers·km/h]
Boeing 737	240	4828	907	217680
BAC/Sud Concorde	132	6437	2172	286704
Boeing 777-200LR	301	15120	892	268492
Airbus A380-800	853	13642	944	805232



Program performance



HW or SW component	Impact on performance
Algorithm	Number of source-level statements and of I/O operations executed
Programming language, compiler, computer architecture	Number of instructions for each source-level statement
Processor, memory	How fast instructions can be executed
I/O system (hardware, operating system)	How fast I/O operations can be executed



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

$$Performance_X = \frac{1}{Execution\ time_X}$$

- **Now we can compare performance**

$$Performance_X > Performance_Y$$

$$\frac{1}{Execution\ time_X} > \frac{1}{Execution\ time_Y}$$

$$Execution\ time_X < Execution\ time_Y$$



Relative performance

- **Relating performance of two computers**

- X is n-times faster than Y (has higher performance)

$$\frac{Performance_X}{Performance_Y} = n$$

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

$$\frac{Performance_X}{Performance_Y} = \frac{Execution\ time\ Y}{Execution\ time\ X} = n$$



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

$$CPU\ execution\ time = \frac{CPU\ clock\ cycles}{CPU\ clock\ rate}$$

$$CPU\ execution\ time = CPU\ clock\ cycles \times CPU\ clock\ cycle\ time$$



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

CPU execution time = CPI \times Number of instructions \times CPU clock cycle time

$$CPU\ execution\ time = \frac{CPI \times \text{Number of instructions}}{CPU\ clock\ rate}$$



Program performance (2)



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



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?



Pitfall: Unrealistic expectations (2)

- Some „back of the envelope“ calculations

$$Execution_{fast} = \frac{Execution_{slow}}{5}$$

$$Execution_{slow} = 80 + 20$$

$$Execution_{fast} = \frac{80}{n} + 20$$

$$\frac{80}{n} + 20 = \frac{80 + 20}{5}$$

$$\frac{80}{n} + 20 = 20$$

$$\frac{80}{n} = 0$$

$$80 \neq 0$$



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)
 - CPI can vary significantly on the same processor

$$MIPS = \frac{InstructionCount}{10^6 \times ExecutionTime} = \frac{InstructionCount}{10^6 \times \frac{InstructionCount \times CPI}{ClockRate}} = \frac{ClockRate}{10^6 \times CPI}$$



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

$$\sqrt[n]{\prod_{i=1}^n SPECratio_i}$$



SPEC CINT2006 on AMD Opteron X4

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

Source: P&H



SPEC CINT2006 on Intel Core i7 920

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

Source: P&H



SPECspeed 2017 on Intel Xeon E5-2650L

Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Perl interpreter	perlbench	2684	0.42	0.556	627	1774	2.83
GNU C compiler	gcc	2322	0.67	0.556	863	3976	4.61
Route planning	mcf	1786	1.22	0.556	1215	4721	3.89
Discrete Event simulation - computer network	omnetpp	1107	0.82	0.556	507	1630	3.21
XML to HTML conversion via XSLT	xalancbmk	1314	0.75	0.556	549	1417	2.58
Video compression	x264	4488	0.32	0.556	813	1763	2.17
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	2216	0.57	0.556	698	1432	2.05
Artificial Intelligence: Monte Carlo tree search (Go)	leela	2236	0.79	0.556	987	1703	1.73
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	6683	0.46	0.556	1718	2939	1.71
General data compression	xz	8533	1.32	0.556	6290	6182	0.98
Geometric mean	–	–	–	–	–	–	2.36

Source: P&H

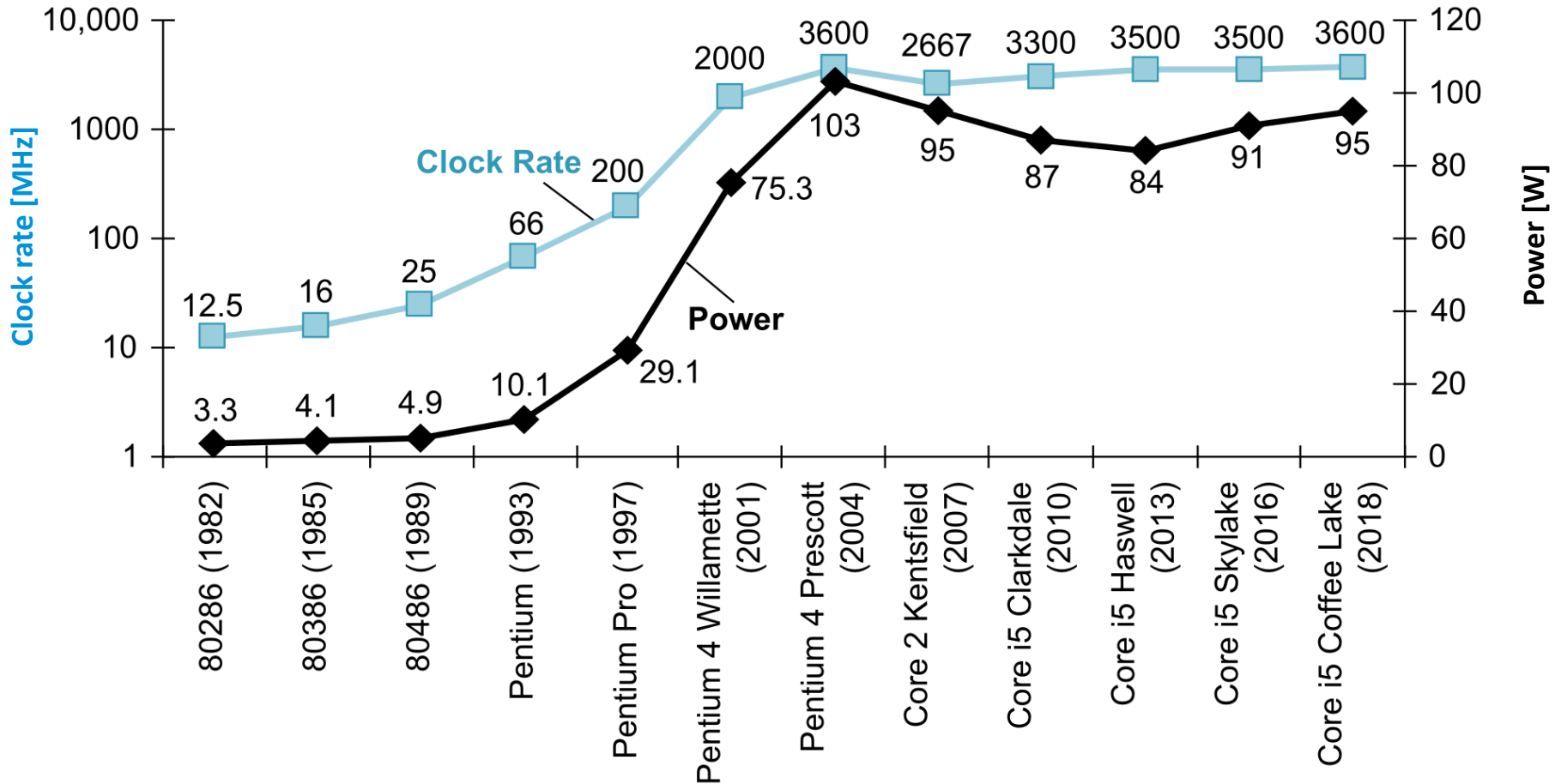




End of the golden era



The Power Wall



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)

$$Power \approx \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$



The Power Wall (3)

- **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.

$$\frac{Power_{new}}{Power_{old}} = \frac{(CapacitiveLoad \times 0.85) \times (Voltage \times 0.85) \times (FrequencySwitched \times 0.85)}{CapacitiveLoad \times Voltage \times FrequencySwitched} = 0.85^4 = 0.52$$



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**





Computer Architecture,, Computer Performance, summer 2022/2023

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 difficulty 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**

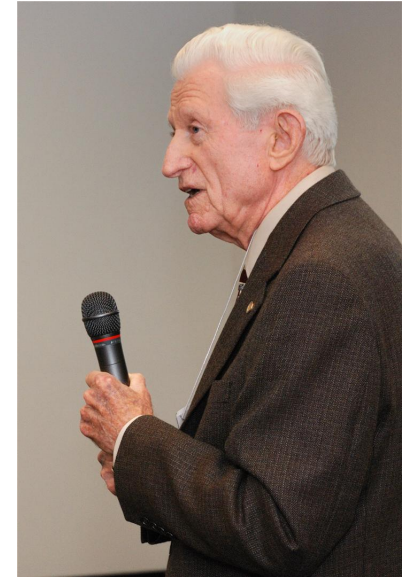
- 1 reporter writes 1 article in 2 hours
 - Can we get 8 reporters to write 1 article in 15 minutes?
- Actual problems
 - Scheduling
 - Who writes what?
 - Load balancing
 - No reporter is idle
 - Communication and synchronization overhead
 - How to put the final article together?



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 versions 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.***



[1]

$$\text{Speedup}(n) = \frac{1}{B + \frac{1}{n}(1 - B)} \quad \begin{array}{l} n \in \mathbb{N} \\ B \in \langle 0, 1 \rangle \end{array}$$



Amdahl's law (2)

- **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

