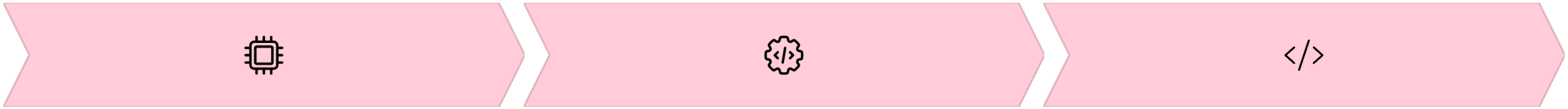
Processor Performance Equation

CPU Time = Clock Cycle Time x Cycles Per Instruction x Instruction Count



Clock Cycle Time

Determined by hardware technology and organization

CPI (Cycles Per Instruction)

Determined by organization and instruction set architecture

Instruction Count

Determined by instruction set architecture and compiler technology

Taking Advantage of Parallelism

1

System-Level Parallelism

Multiple processors and disks improve throughput on server benchmarks like SPECWeb or TPC-C by spreading workload across resources.

This **scalability** is valuable for servers, enabling data-level parallelism across disks and request/thread-level parallelism across processors.

2

Instruction-Level Parallelism

Pipelining overlaps instruction execution to reduce total time for instruction sequences.

This works because not every instruction depends on its immediate predecessor, allowing partial or complete parallel execution.

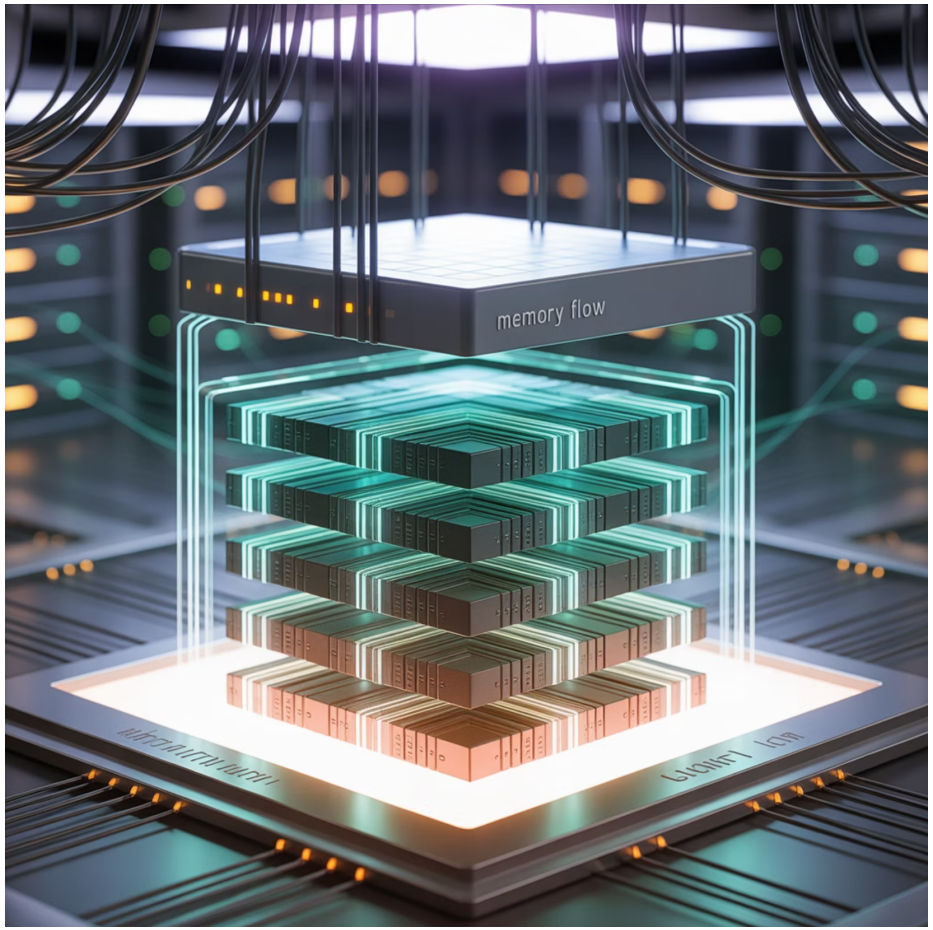
3

Digital Design Parallelism

Set-associative caches use multiple memory banks searched in parallel to find items.

Modern ALUs use carry-lookahead to speed computation from linear to logarithmic in the number of bits per operand.

Principle of Locality



> Programs tend to reuse data and instructions they have used recently.

A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.

Temporal Locality

Recently accessed items are likely to be accessed again in the near future.

Spatial Locality

Items whose addresses are near one another tend to be referenced close together in time.

Focus on the Common Case

Perhaps the most important and pervasive principle of computer design is to **favor the frequent case over the infrequent case** when making design trade-offs.

Resource Allocation

Spend resources where they will have the highest impact on frequent operations.

Power Optimization

Optimize frequently used units like instruction fetch before less used ones like multipliers.

Dependability

Focus on components that dominate system reliability (e.g., storage in database servers).

Performance

The frequent case is often simpler and can be done faster.

Amdahl's Law provides a quantitative framework for applying this principle.

Amdahl's Law

$$Speedup_{Overall} = \frac{1}{(1 - Fraction_{Enhanced}) + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}}}$$

Amdahl's law quantifies the performance improvement gained by enhancing a portion of a system.

1

Key Factors

- Fraction of computation time that can use the enhancement
- Improvement gained by the enhanced execution mode

2

Example: Web Server

New processor is 10× faster on computation, which takes 40% of time (60% is I/O waiting)

Speedup = 1/(0.6 + 0.4/10) = 1/0.64 ≈ 1.56

3

Diminishing Returns

The incremental improvement in speedup diminishes as enhancements are added

Maximum speedup limited to 1/(1-f) where f is the fraction that can be enhanced

Amdahl's Law Examples

Graphics Processor Example

FP square root (FPSQR) is 20% of execution time in a critical graphics benchmark.

Option 1: Speed up FPSQR by 10×

Speedup = 1/(0.8 + 0.2/10) = 1/0.82 = 1.22

Option 2: Speed up all FP instructions (50% of time) by 1.6×

Speedup = 1/(0.5 + 0.5/1.6) = 1/0.8125 = 1.23

Option 2 is slightly better due to higher frequency of use.

Reliability Example

System failure rate = 23/1,000,000 hours

Power supply failure rate = 5/1,000,000 hours (22% of total)

Improving power supply reliability by 4150× results in:

Improvement = 1/(0.78 + 0.22/4150) = 1/0.78 = 1.28

Despite 4150× component improvement, system benefit is only 1.28×