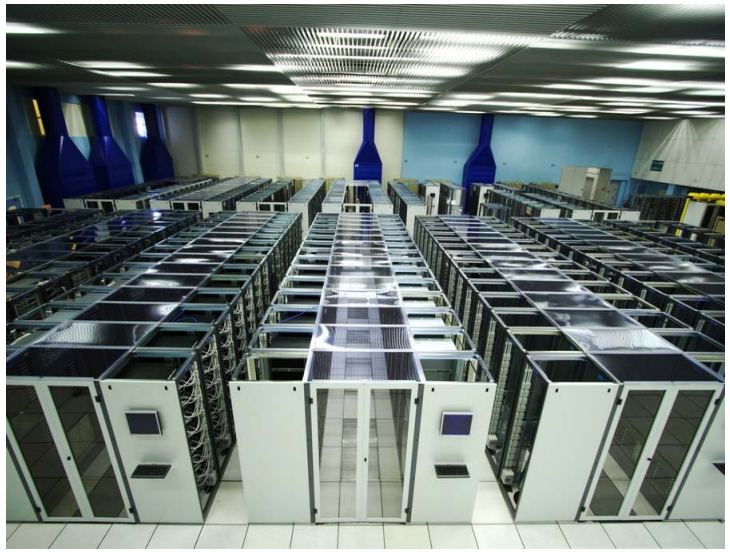
Computer Architecture and Performance Tuning

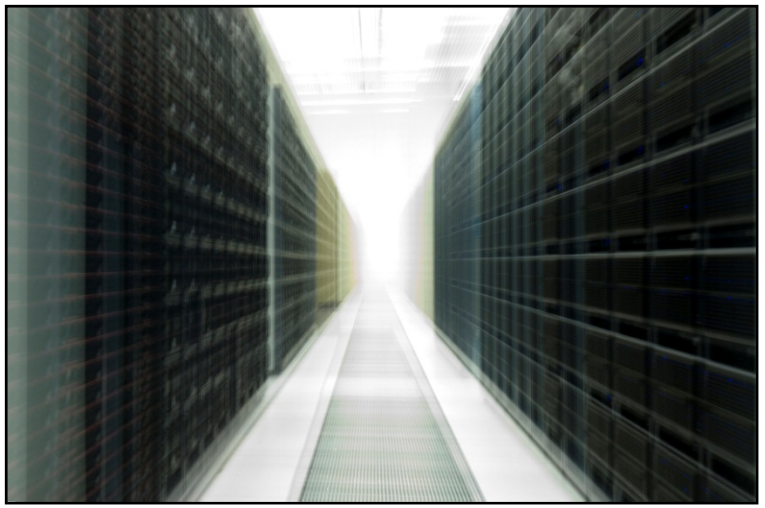
“Hunting for Performance in 7 dimensions”



Sverre Jarp
CERN
openlab
CTO

IT Dept.

CERN



Summer Student lecture - August 2008

Contents

- **Why worry about performance?**
- **Review of fundamental architectural principles**
- **Addressing performance "dimensions"**
- **Scaling within a core**
  - First 3 dimensions
  - Causes of execution delays
  - Performance metrics
- **Scaling within a node**
  - Next set of dimensions (without detailed discussion)
- **Conclusions**

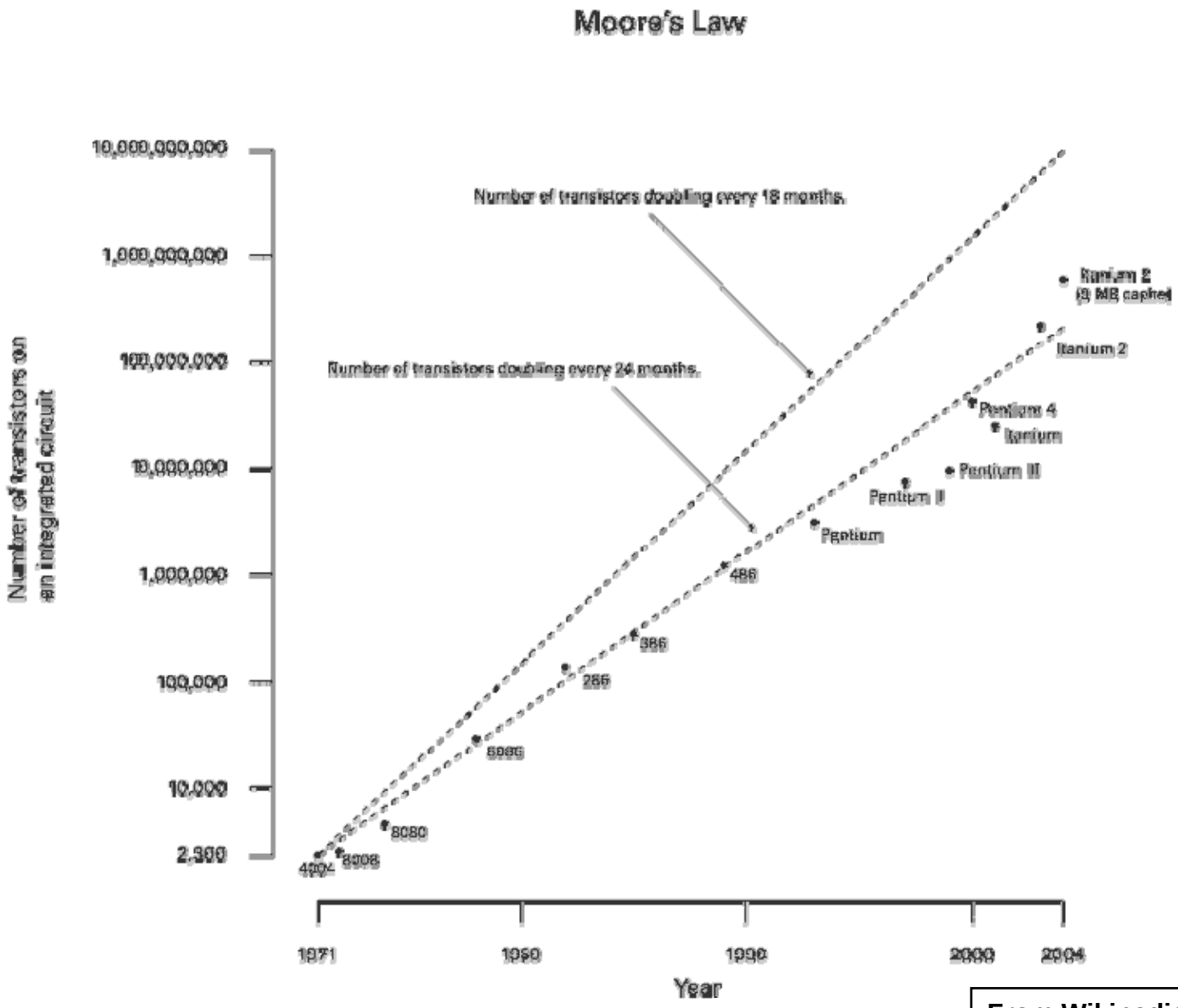
Why worry about performance?

- **My arguments:**
  - The frequency scaling we enjoyed in the past does not exist any longer
  - There are important thermal issues associated with large scale computing
    - Even when 1W processors exist!
  - There are important cost issues associated with large scale computing
    - Even when using “commodity equipment”

Moore's law

- **We continue to double the number of transistors**
  - Latest consequence
    - Single core
      → Multicore
      → Manycore

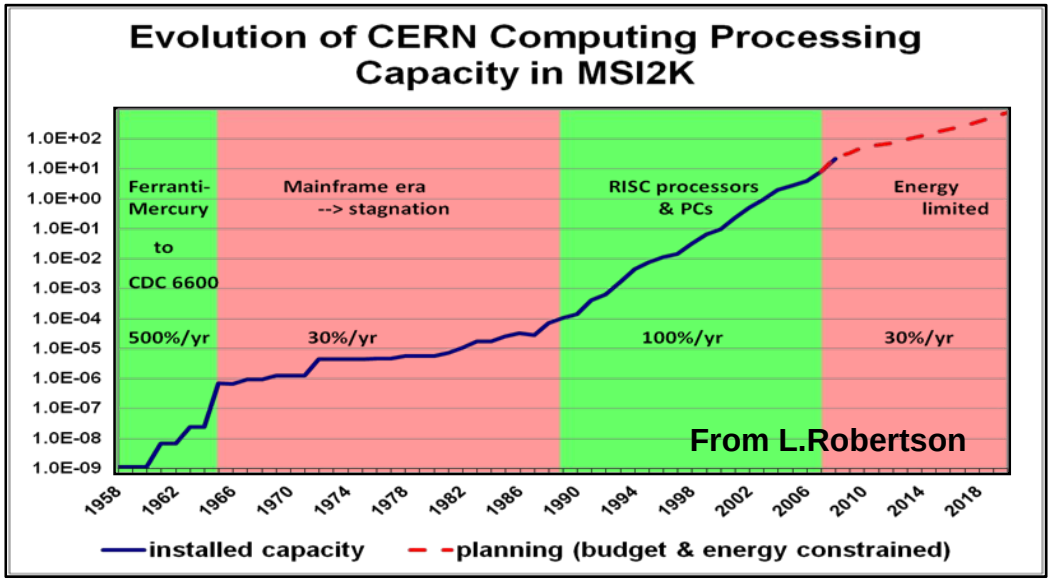
The derivative "law" which stated that the frequency would also double is no longer true!

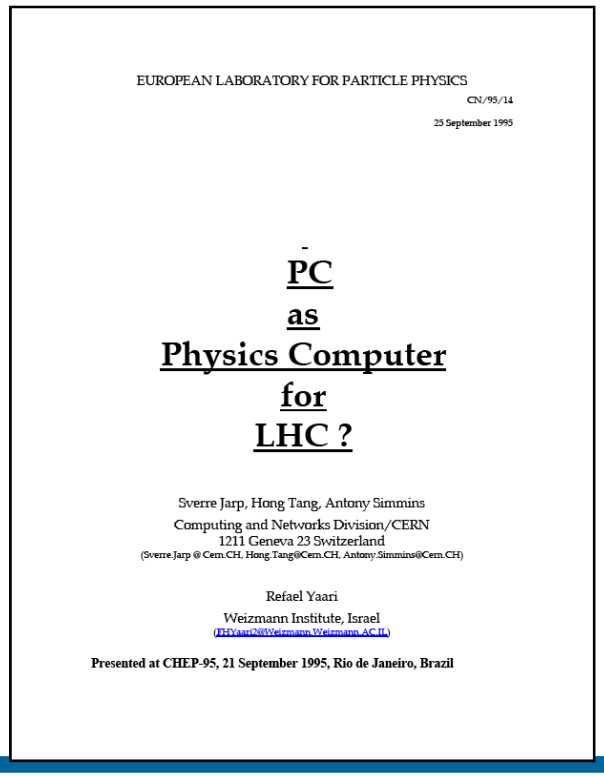


From Wikipedia

Evolution of CERN's computing capacity

- **During the LEP era (1989 – 2000):**
  - Doubling of compute power every year
  - Initiated with the move from mainframes to RISC systems
- **At CHEP-95:**
  - I made the first recommendation to move to PCs
    - After a set of encouraging benchmark results





EUROPEAN LABORATORY FOR PARTICLE PHYSICS

CN/95/14

25 September 1995

PC as Physics Computer for LHC ?

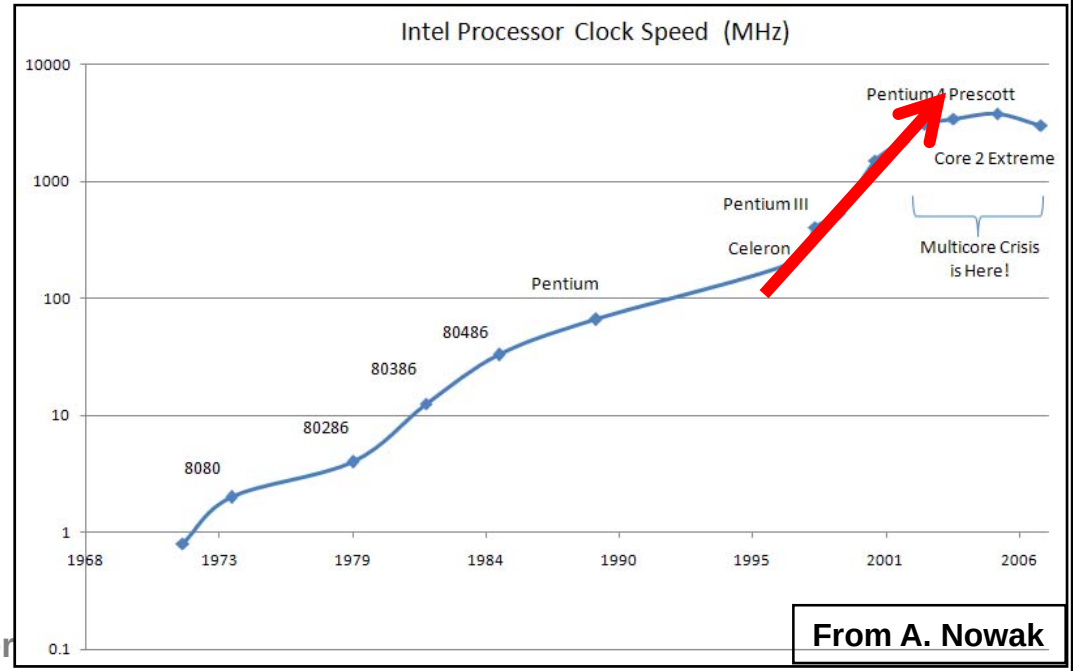
Sverre Jarp, Hong Tang, Antony Simmins
Computing and Networks Division/CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH, Hong.Tang@Cern.CH, Antony.Simmins@Cern.CH)

Refael Yaari
Weizmann Institute, Israel
(FHYaari2@Weizmann.Weizmann.AC.IL)

Presented at CHEP-95, 21 September 1995, Rio de Janeiro, Brazil

Frequency scaling

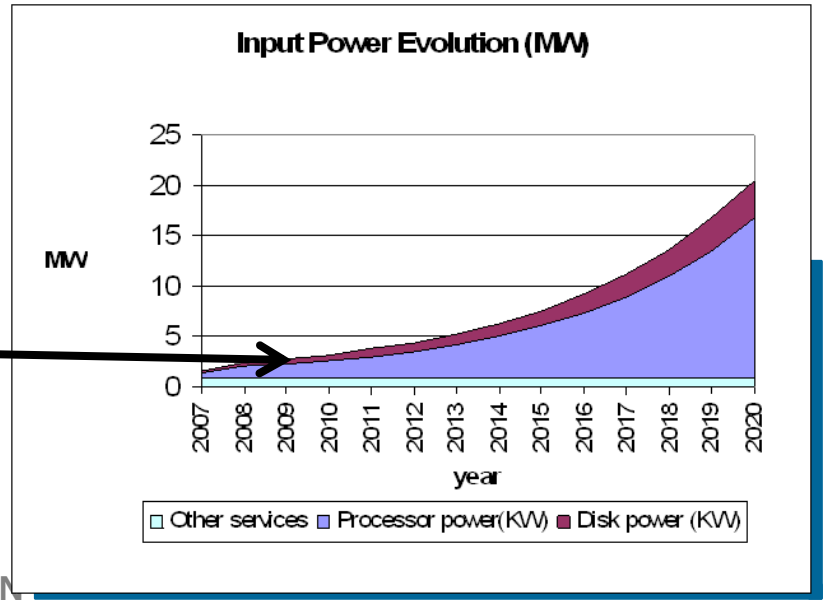
- **The 7 “fat” years of frequency scaling in HEP**
  - From the Pentium Pro in 1996: 150 MHz
  - To the Pentium 4 in 2003: 3.8 GHz (~25x)
- **Since then**
  - Core 2 systems:
    - ~3 GHz
    - Multi-core



From A. Nowak

The Power Wall

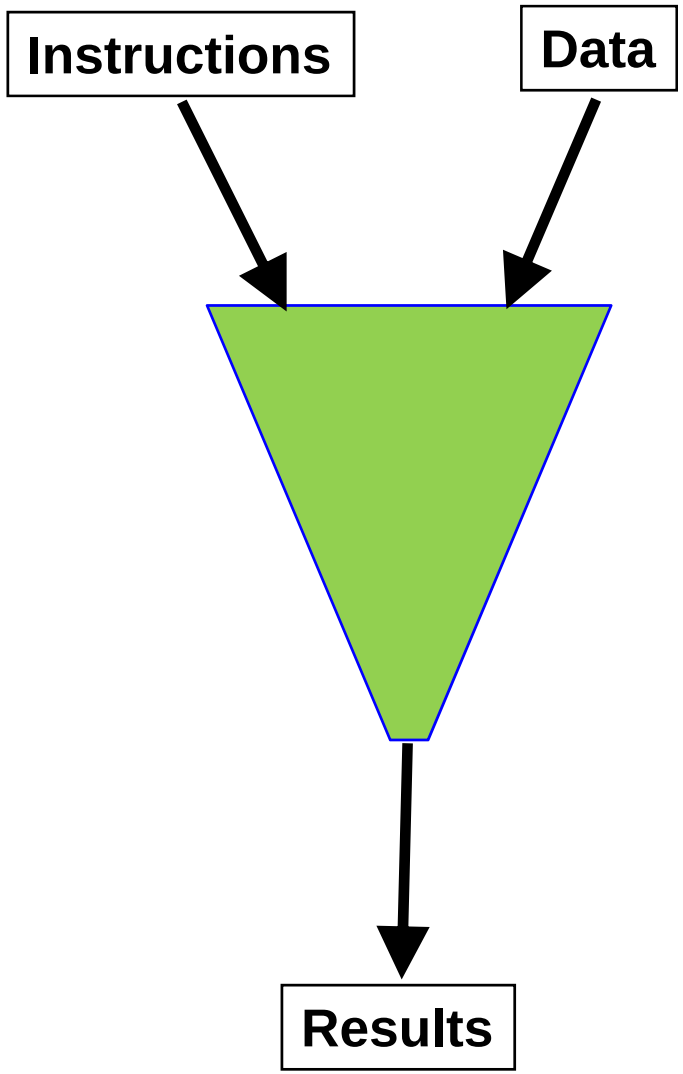
- **For example, the CERN Computer Centre can supply 2.5MW of electric power**
  - Plus 2MW to remove the corresponding heat!
- **Spread over a complex infrastructure:**
  - CPU servers; Disk servers
  - Tape servers + robotic equipment
  - Database servers
  - Infrastructure servers.
  - Network switches and routers
- **This limit will be reached in 2009!**



Let’s look at some processor details!

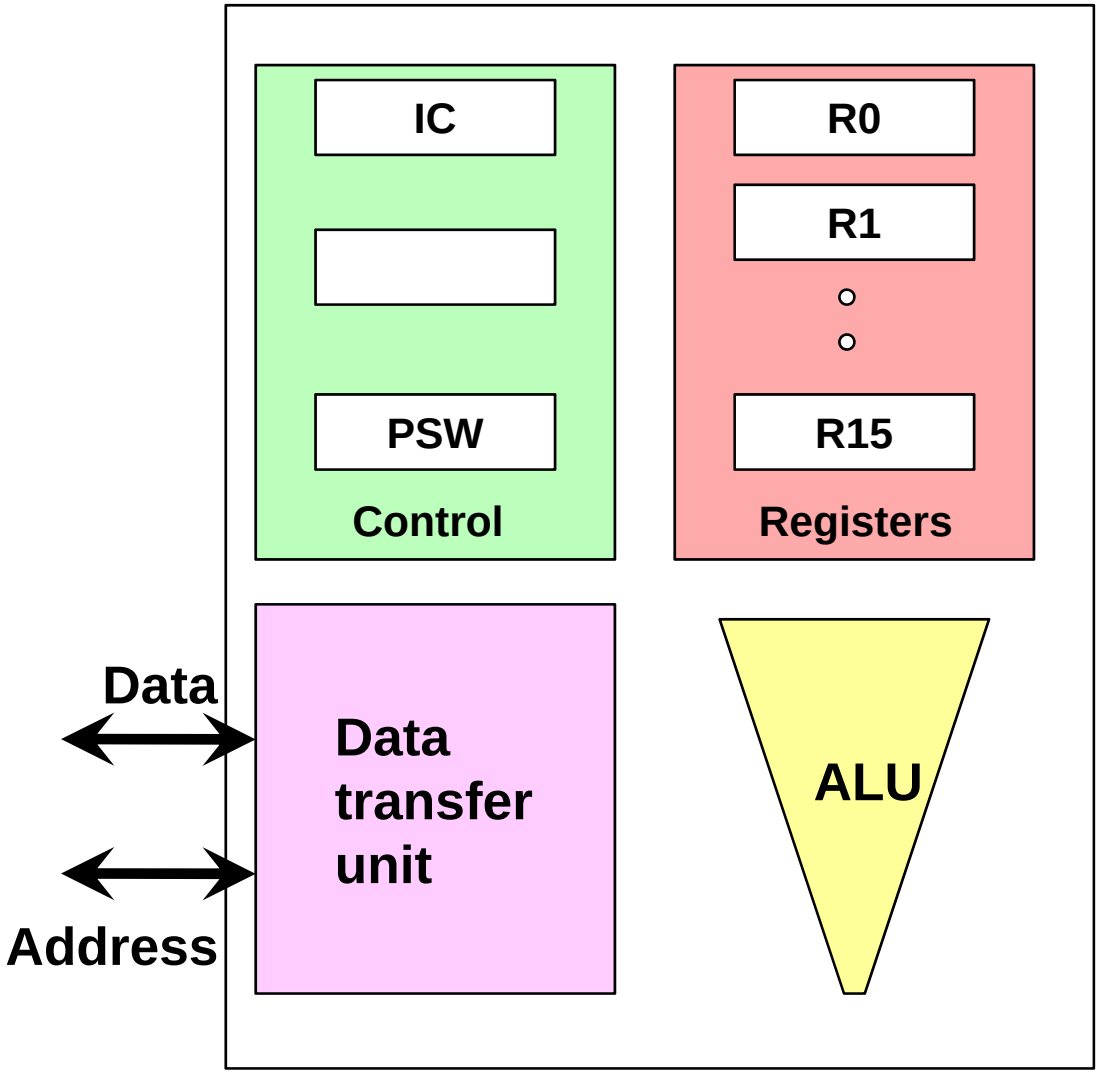
Von Neumann architecture

- **From Wikipedia:**
  - The von Neumann architecture is a computer design model that uses a processing unit and a single separate storage structure to hold both instructions and data.
- **It can be viewed as an entity into which one streams instructions and data in order to produce results**
- **Our goal is to produce results as fast as possible**



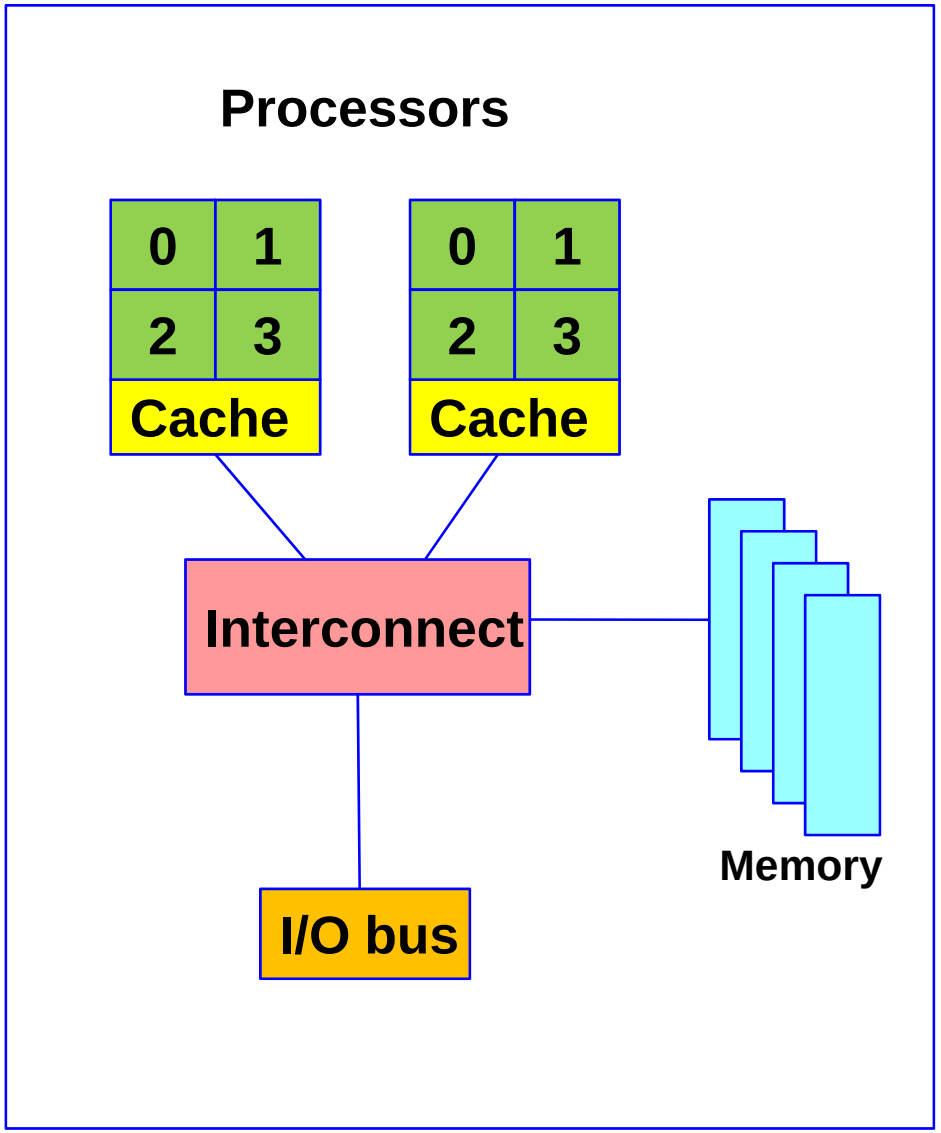
Simple processor layout

- **A simple processor with four key components:**
  - Control Logic
    - Instruction Counter
    - Program Status Word
  - Register File
  - Data Transfer Unit
    - Data bus
    - Address bus
  - Arithmetic Logic Unit



Simple server diagram

- **Multiple components which interact during the execution of a program:**
  - Processors/cores
  - Cache
    - Instructions (I-cache)
    - Data (D-cache)
  - Memory bus
  - Memory
  - I/O subsystem
    - Network attachment
    - Disk subsystem



Initial premise

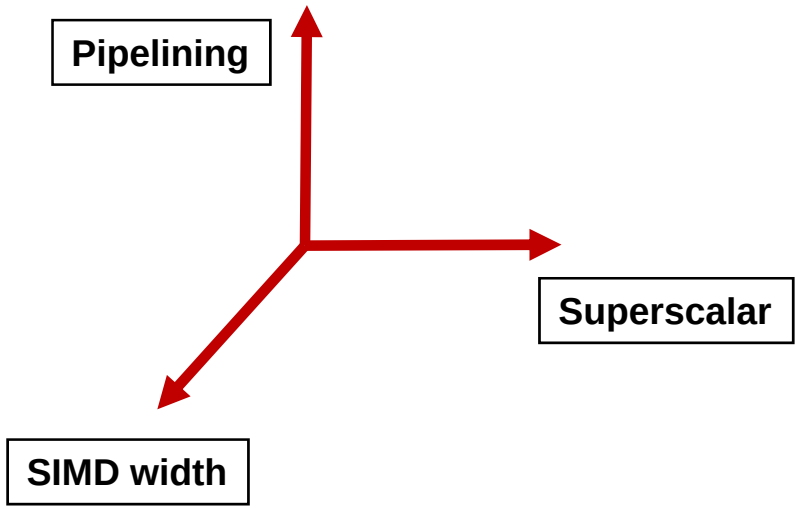
- **To reach completion, a compute job (a process) requires the execution of a given number of (machine-level) instructions**
- **We typically want the process to complete in the shortest possible time**
  - This time corresponds to a given number of machine cycles
- **Simple example:**
  - A program consists of (the execution of) 10^{10} instructions
  - We measure an execution time of 6 seconds on a processor running at 2.0 GHz
  - We can now compute a key value:
    - Cycles per Instruction (CPI)
    - Our result: $(6 * 2 * 10^9) / 10^{10} = 1.2$

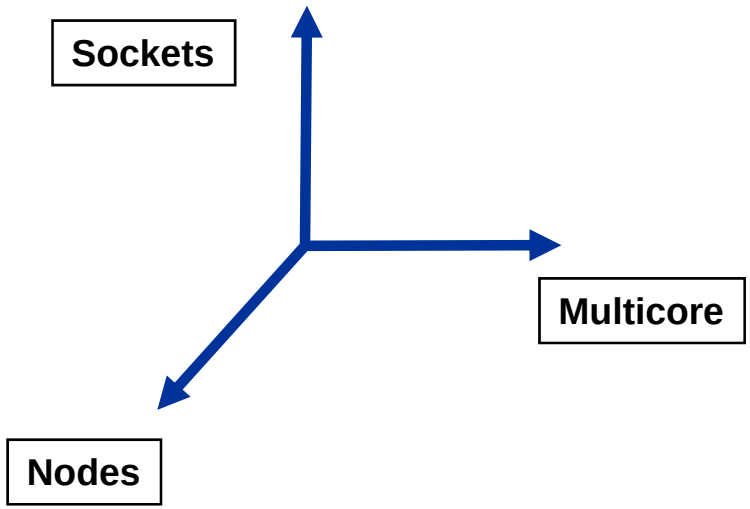
A complicated story!

- **We start with a concrete, real-life problem to solve**
  - For instance, simulate the passage of elementary particles through matter
- **We write programs in high level languages**
  - C++, JAVA, Python, etc.
- **A compiler (or an interpreter) transforms the high-level code to machine-level code**
- **We link in external libraries**
- **A sophisticated processor with a complex architecture and even more complex micro-architecture executes the code**
- **In most cases, we have little clue as to the efficiency of this transformation process**

Seven dimensions of performance

- **First three dimensions:**
  - Superscalar
  - Pipelining
  - Computational width/SIMD
- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes

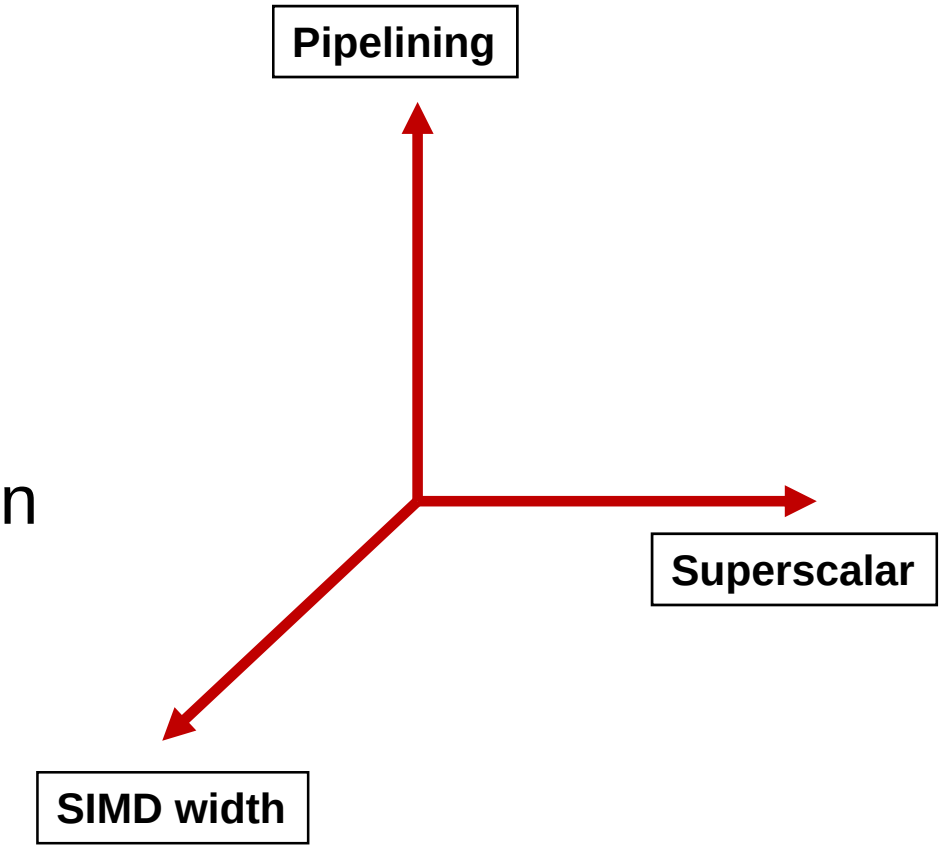




SIMD = Single Instruction Multiple Data

Part 1: Opportunities for scaling performance inside a core

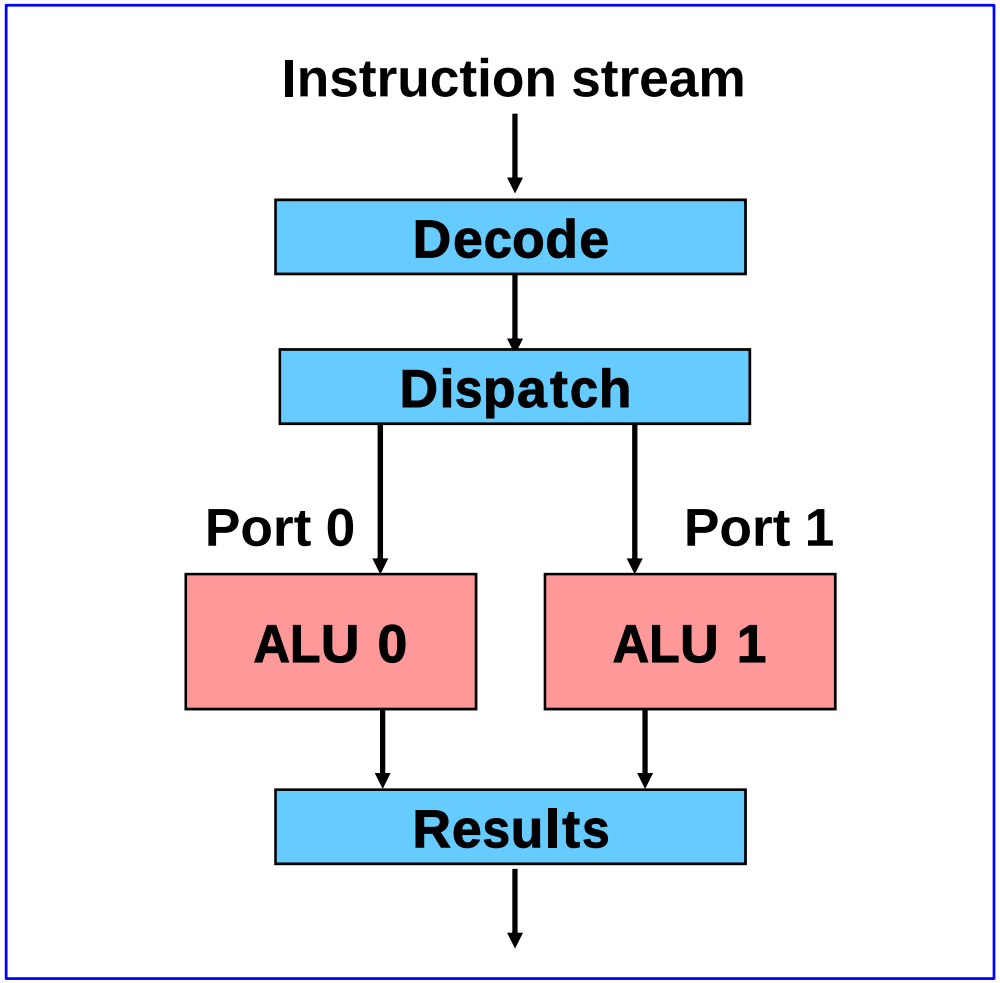
- **Let's look at the first three dimensions**



- **Data parallelism via**
  - Loop/straight-line vectorization
- **The resources:**
  - Superscalar: Fill the ports
  - Pipelined: Fill the stages
  - SIMD: Fill the computational width

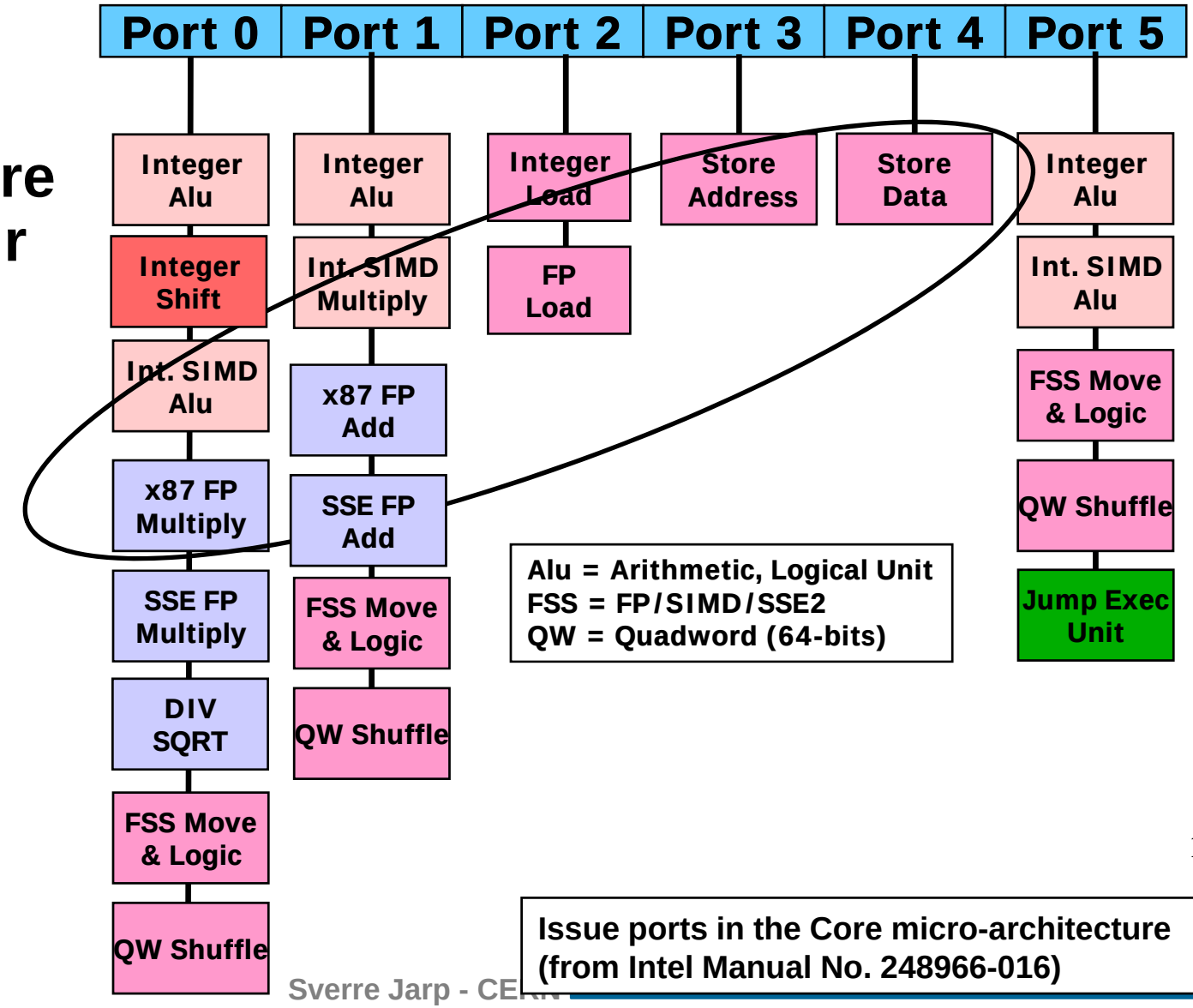
Superscalar architecture

- **In this simplified design, instructions are decoded serially, but dispatched to two ALUs.**
  - The decoder and dispatcher ought to be able to handle two instructions per cycle
  - The ALUs can have identical or different execution capabilities



Core 2 execution ports

- **Intel's Core microarchitecture can execute four instructions in parallel:**

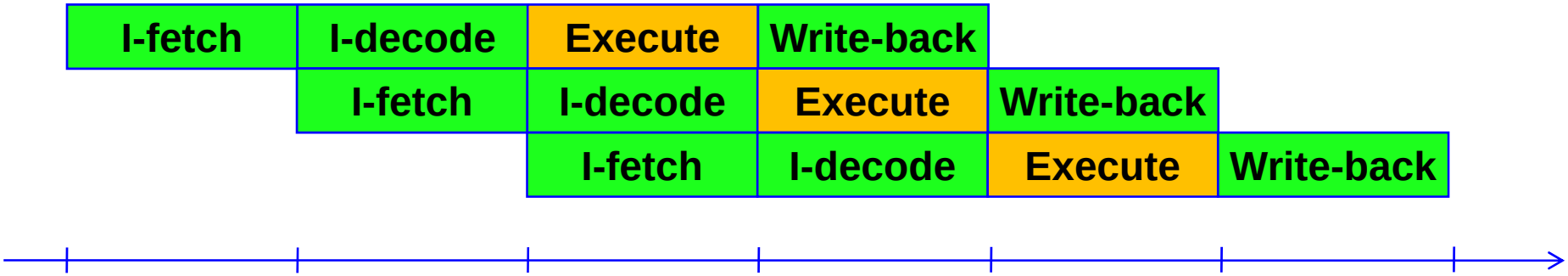


Issue ports in the Core micro-architecture (from Intel Manual No. 248966-016)

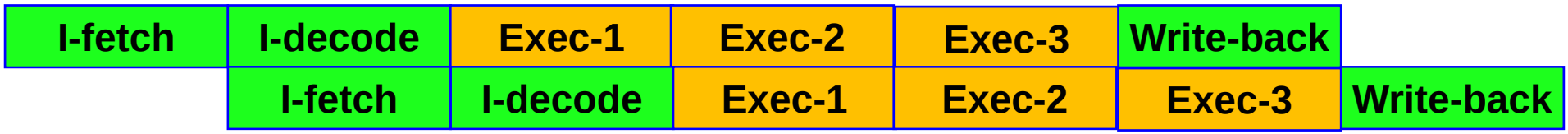
17

Next topic: Instruction pipelining

- **Instructions are broken up into stages.**
  - With a one-cycle execution latency (simplified):



  - With a three-cycle execution latency:



Real-life latencies

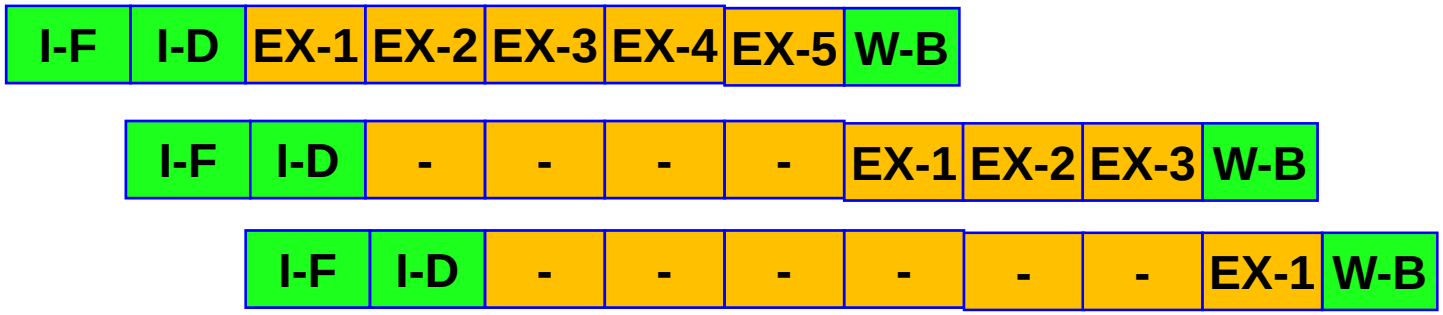
- **Most integer/logic instructions have a one-cycle execution latency:**
  - ADD, AND, SHL (shift left), ROR (rotate right)
  - Some exceptions:
    - IMUL (integer multiply: 3)
    - IDIV (integer divide: 13 – 23)
- **Floating-point latencies are typically multi-cycle**
  - FADD (3), FMUL (5)
    - Same for both x87 and SIMD variants
  - Exception: FABS (absolute value: 1)

Latencies in the Core micro-architecture (from Intel Manual No. 248966-016)

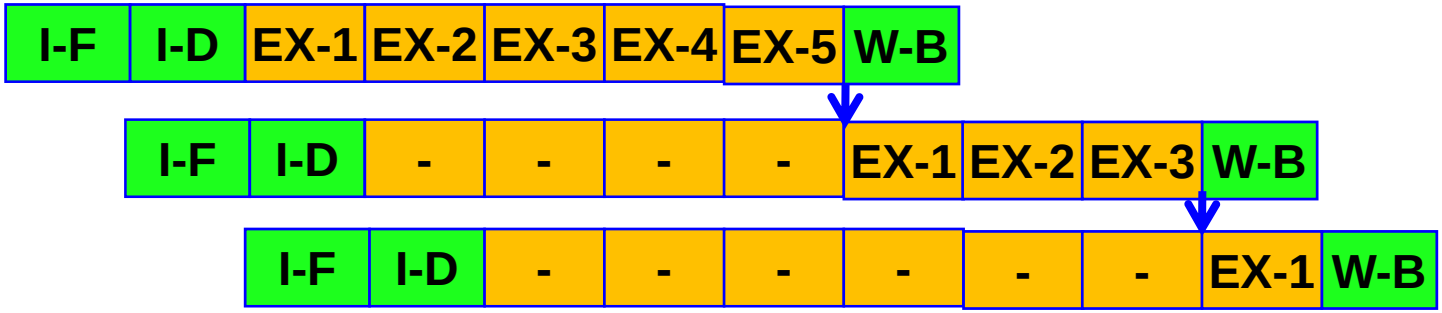
Latencies and serial code (1)

- **In serial programs, we typically pay the penalty of a multi-cycle latency during execution:**
  - In this example:
    - Statement 2 cannot be started before statement 1 has finished
    - Statement 3 cannot be started before statement 2 has finished

```
double a, b, c, d, e, f;

b = 2.0; c = 3.0; e = 4.0;

a = b * c;  // Statement 1

d = a + e;  // Statement 2

f = fabs(d);   // Statement 3
```



Latencies and serial code (2)



- **Observations:**
  - Even if the processor can fetch and decode a new instruction every cycle, it must wait for the previous result to be made available
    - Fortunately, the result takes a ‘bypass’, so that the write-back stage does not cause even further delays
  - The result here:
    - 9 execution cycles are needed for three instructions!
      - CPI is equal to 3

Example of real-life serial code

- **Suffers long latencies:**

High level C++ code →

```
if (abs(point[0] - origin[0]) > xhalfsz) return FALSE;
```

Machine instructions →

```
movsd 16(%rsi), %xmm0
subsd 48(%rdi), %xmm0   // load & subtract
andpd _2il0floatpacket.1(%rip), %xmm0 // and with a mask
comisd 24(%rdi), %xmm0 // load and compare
jbe ..B5.3      # Prob 43%  // jump if FALSE
```

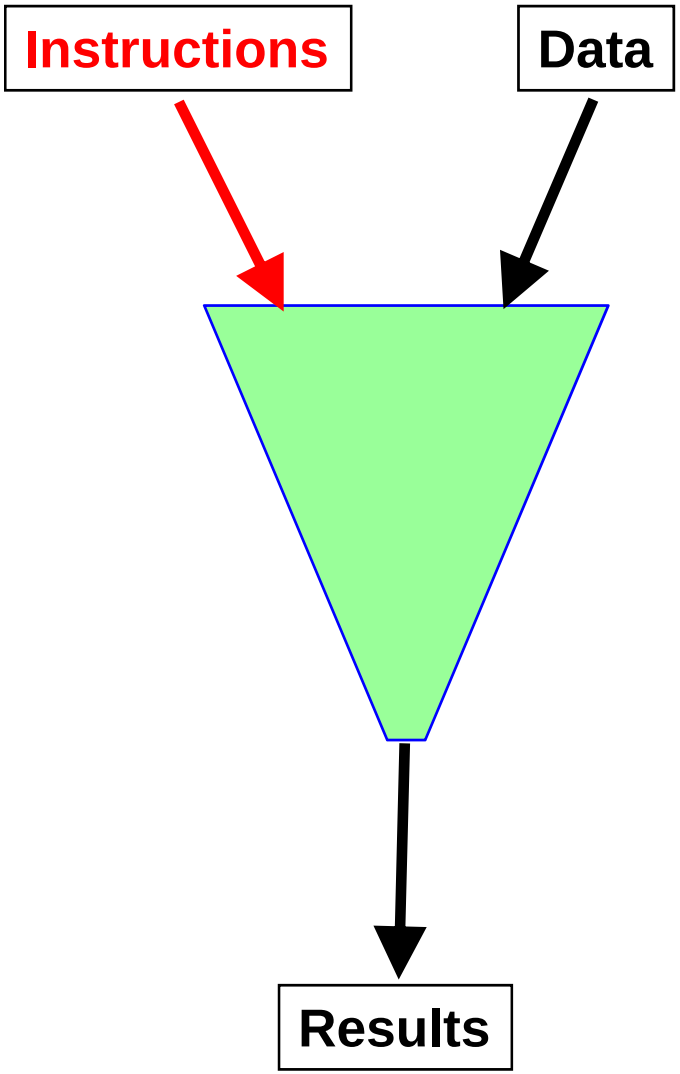
Same instructions laid out according to latencies on the Core 2 processor →

NB: Out-of-order scheduling not taken into account.

Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

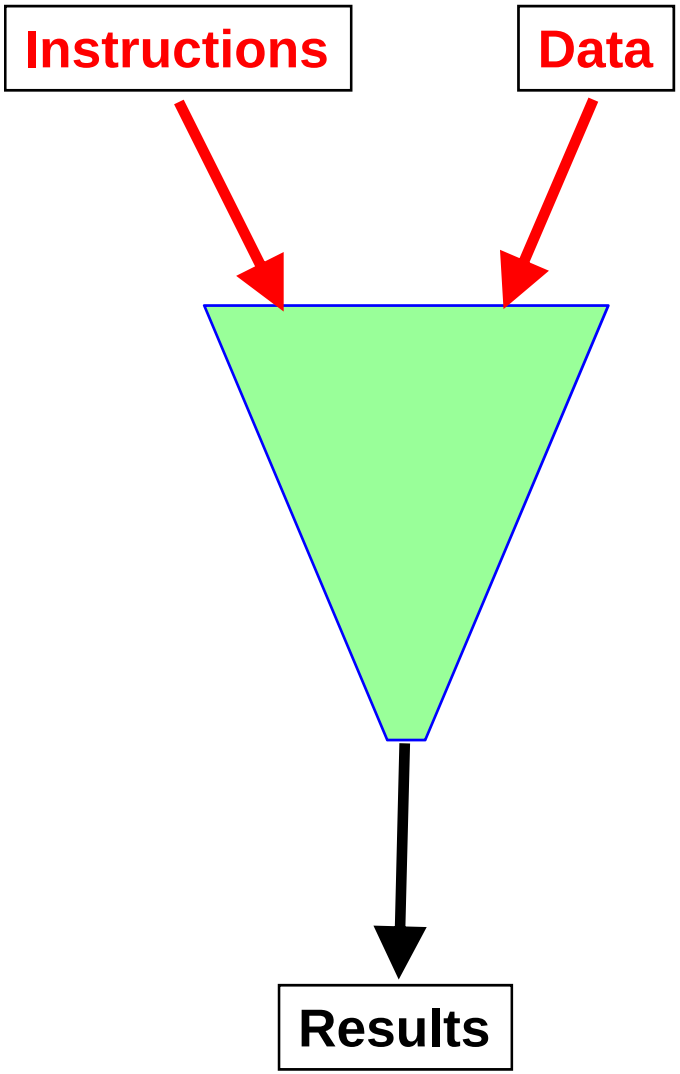
Other causes of execution delays (1)

- **We already stated that the aim is to keep instructions and data flowing, so that results are generated optimally**
- **First issue:**
  - Instructions stop flowing
    - Typically caused by branching
    - There may be a branch instruction in every 10 machine instructions!
      - Or even less
    - If the branch is mispredicted, we suffer a stall (cycles clock up, but no work gets done)



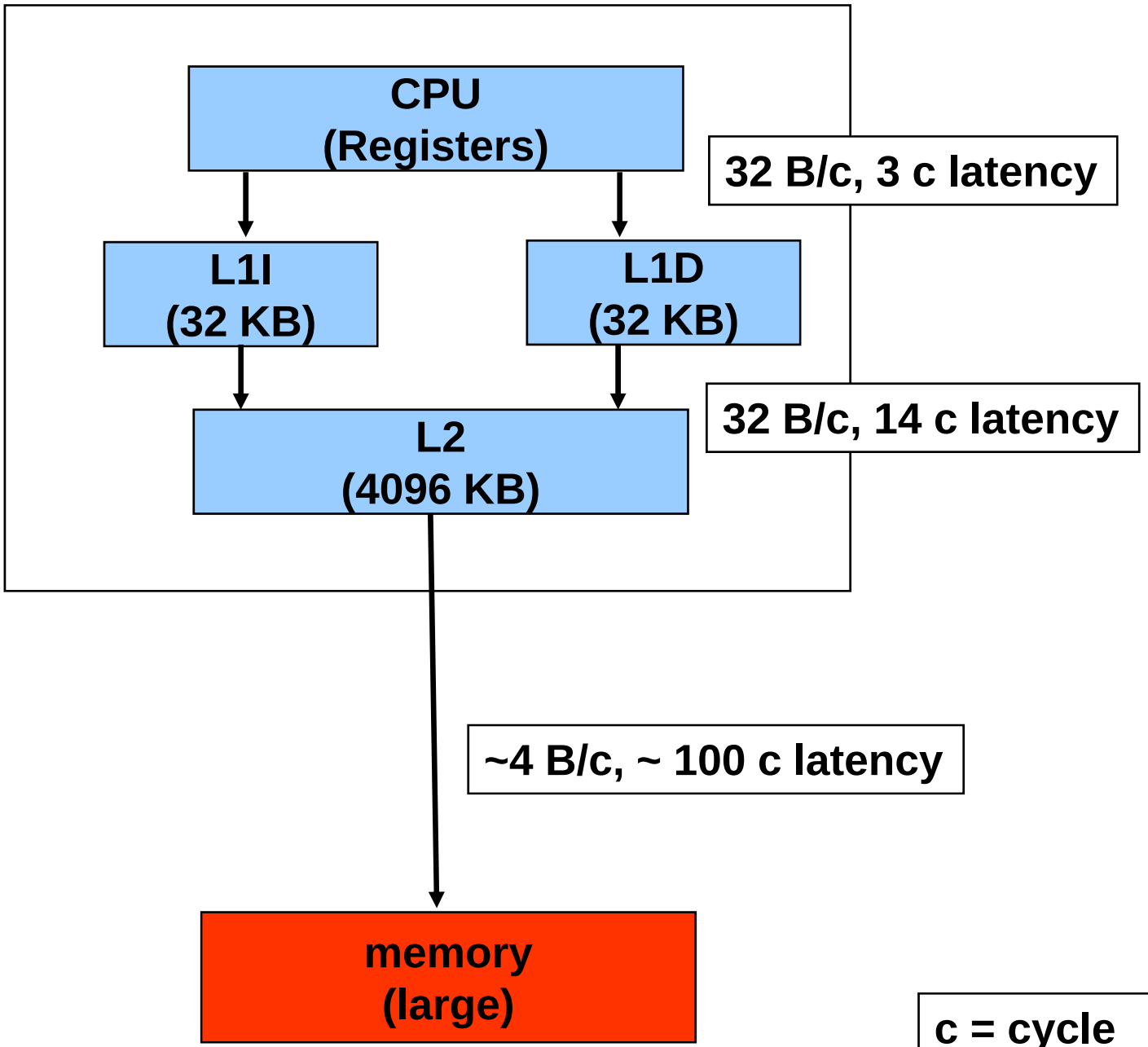
Other causes of execution delays (2)

- **Second issue:**
  - Instructions and/or data stop flowing
    - Instructions are not found in the I-cache
    - Data is not found in the D-cache
  - Before execution can continue, instructions and data must be fetched from a lower level



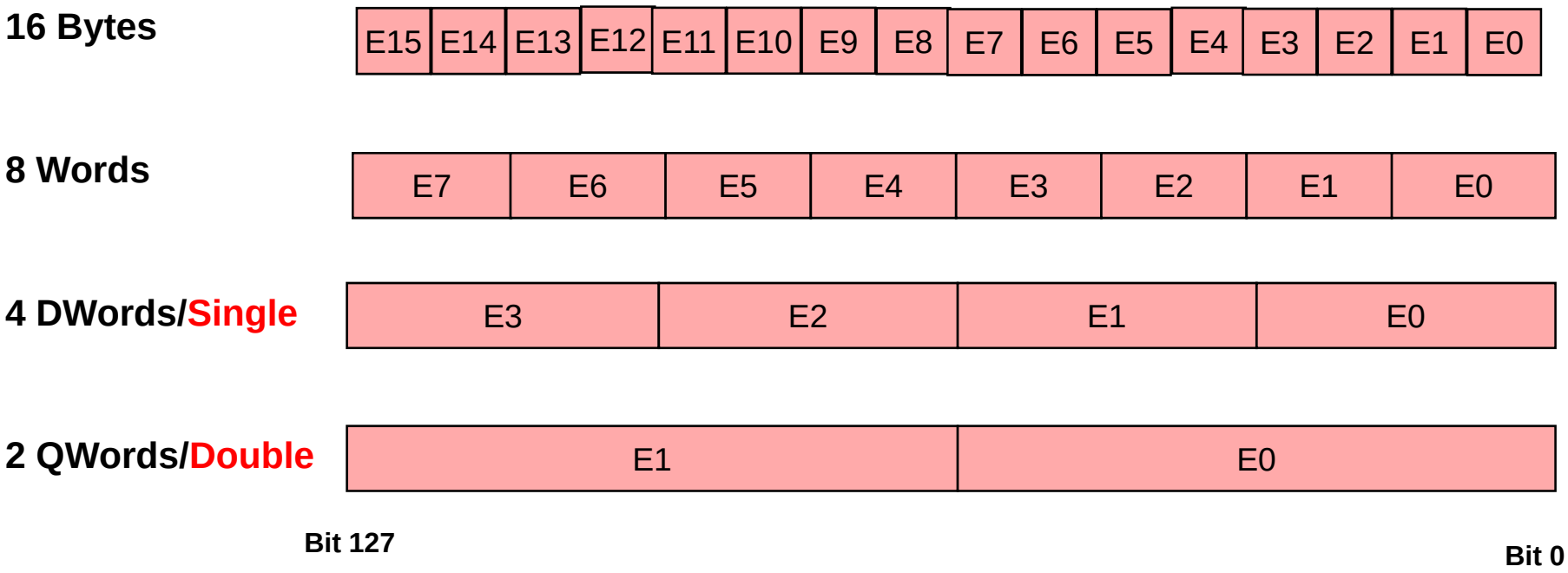
Memory Hierarchy

- **From CPU to main memory on a Core 2 uni-processor**
  - With multicore, memory bandwidth is shared between cores on the same bus



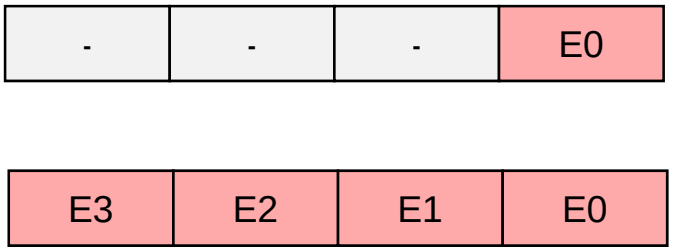
XMM registers for SSE

- **16 registers with 128 bits each in 64-bit mode (x86-64)**

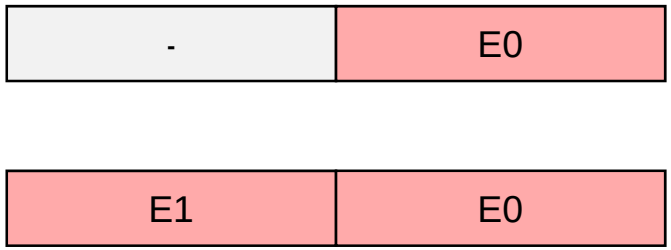


Four FP data flavours

- **Single precision**
  - Scalar single (SS)
  - Packed single (PS)



- **Double precision**
  - Scalar Double (SD)
  - Packed Double (PD)



- **Note that the scalar variants replace x87 in x86-64**
  - Possibly impacting precision!

Summary of important items to measure

- **Review of what we have discussed:**
  - The total cycle count (C)
  - The total instruction count (I)
  - Derived value: CPI
  - Bubble count: Cycles when no (new instruction) execution occurred
  - Total number of executed branch instructions
  - Total number of mispredicted branches

- **Plus:**
  - Total number of (last-level) cache misses
  - Total number of cache accesses
  - Bus occupancy
  - The total number of SSE instructions
  - The total number (and the type) of computational SSE instructions

Scalable programming for a single core

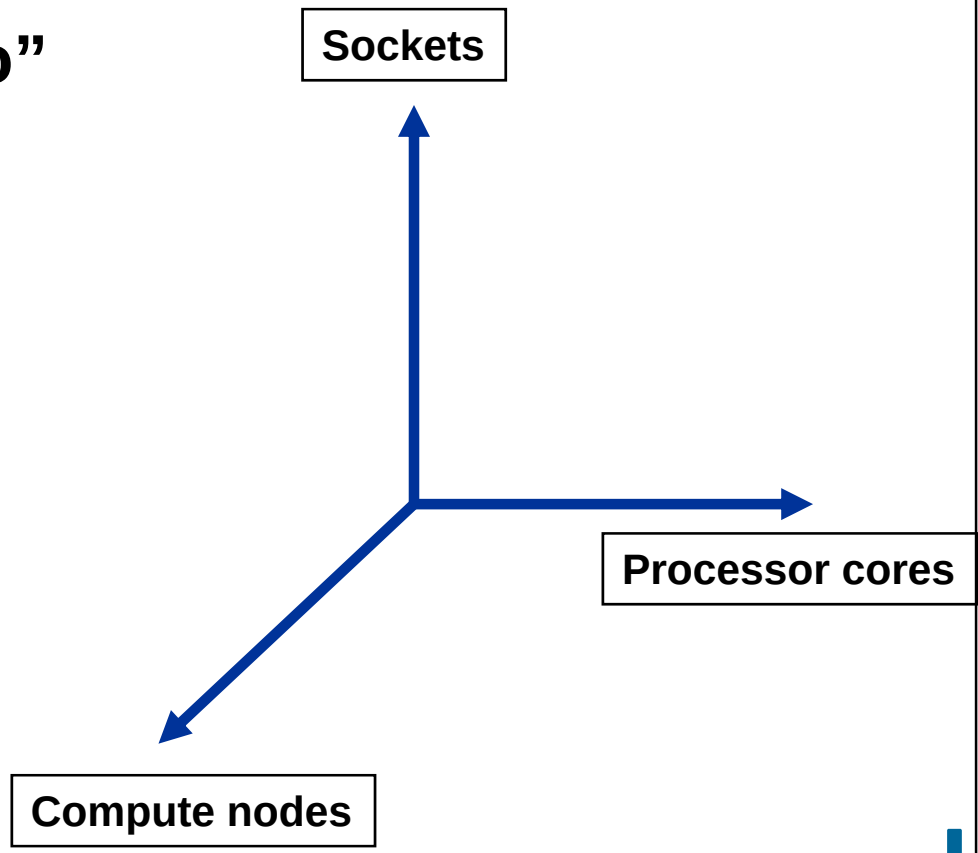
- **Easiest way to fill the execution capabilities is to use vectorization**
  - Either, vector syntax, à la Fortran-90
  - Or, loop syntax which the compiler can vectorize automatically
  - Or, explicit intrinsics
    - Not discussed further (today).

```
REAL U(100), V(100)

U = 0.0

U = SIN(V)

U(1:50) = V(2:100:2)
```

```
float  u[100], v[100];

for (int i = 0; i<50; i++) u[i] = 0.0;

for (i = 0; i<50; i++) u[i] = sin(v[i]);

for (int i = 0; i<50; i++) u[i] = v[i*2+1];
```

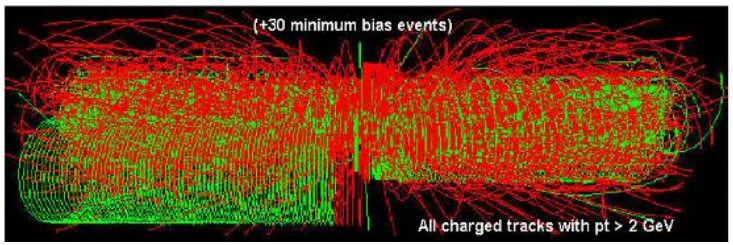
Part 2: Parallel execution across hw-threads and cores

- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes
- **Multiple nodes will not be discussed here**
  - Our focus is scalability inside a node



HEP programming paradigm

- **Event-level parallelism has been used for decades**
  - Compute one event after the other in a single process
- **Advantage:**
  - Large jobs can be split into N efficient processes, each responsible for processing M events
    - Built-in scalability



- **Disadvantage:**
  - Memory must be made available to each process
    - With 2 – 4 GB per process
    - A dual-socket server with Quad-core processors
      - Needs 16 – 32 GB (or more)

What are the options?

- **There is currently a discussion in the community about the best way forwards (in a many-core world):**

1) Stay with event-level parallelism (and independent processes)
   - Assume that the necessary memory remains affordable
2) Move to a fully multi-threaded paradigm
   - Using gross-grained (event-level?) parallelism
3) Rely on forking:
   - Start the first process
   - Fork N others
   - Rely on the OS to do “copy on write”, in case pages are written to

Programming strategies/priorities

- **As I see them:**
  - Get memory usage (per process) under control
    - To allow higher multiprogramming level per server
  - Draw maximum benefit from hardware threading
  - Introduce coarse-grained software multithreading
    - To allow further scaling with large core counts
  - Revisit data parallel constructs at the very base
    - Gain performance inside each core
- **In all cases, use appropriate tools (pfmon/Thread Profiler, etc.)**
  - To monitor detailed program behaviour

Concluding remarks

- **The aim of these lectures was to help understand**
  - Modern computer architecture
  - Factors that improve or degrade performance
  - Keeping in mind that there is not always a straight path to reach (all of) the available performance by our programming community.
- **In most HEP programming domains event-level processing will (continue to) dominate**
  - Provided we get the memory requirements under control
- **Learn to be the master of the 7 hardware dimensions!**

Further reading:

- **“Computer Architecture: A Quantitative Approach”, J. Hennessy and D. Patterson, 3rd ed., Morgan Kaufmann, 2002**
- **“Inside the Machine”, J. Stokes, Ars Technica Library, 2007**
- **“Foundations of Multithreaded, Parallel and Distributed Programming”, G. R. Andrews, Addison-Wesley, 1999**
- **“Principles of Concurrent and Distributed Programming”, M. Ben-Ari, 2nd ed., Addison Wesley, 2006**
- **“Patterns for Parallel Programming”, T.G. Mattson, Addison Wesley, 2004**
- **“Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”, J. Reinders, O’Reilly, 1st ed., 2007**

BACKUP