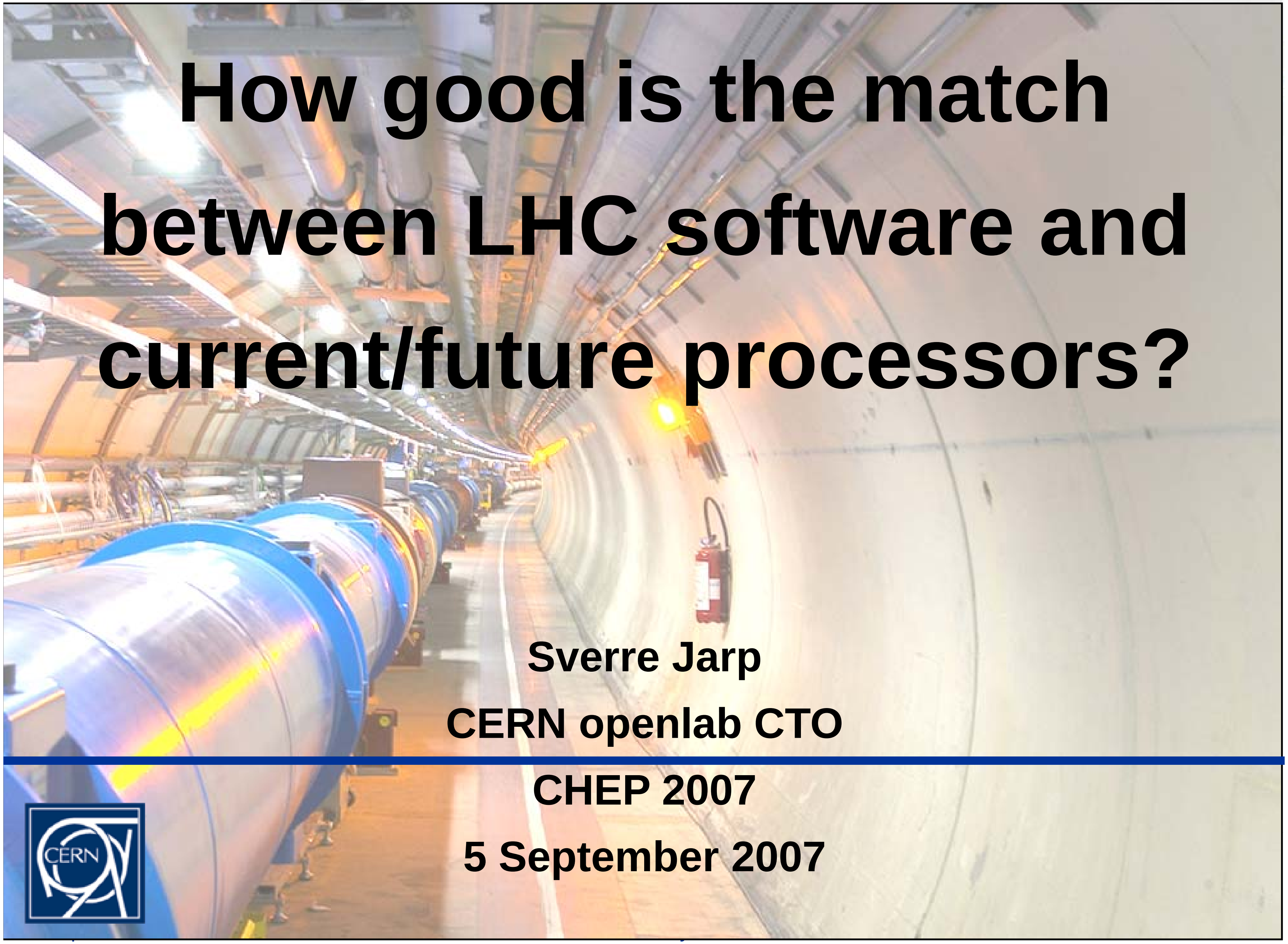


How good is the match between LHC software and current/future processors?

Sverre Jarp

CERN openlab CTO

CHEP 2007

5 September 2007

Agenda

- Before we start
- Moore's "law"
- Hardware and software basics
- Some suggestions for the future
- Conclusions

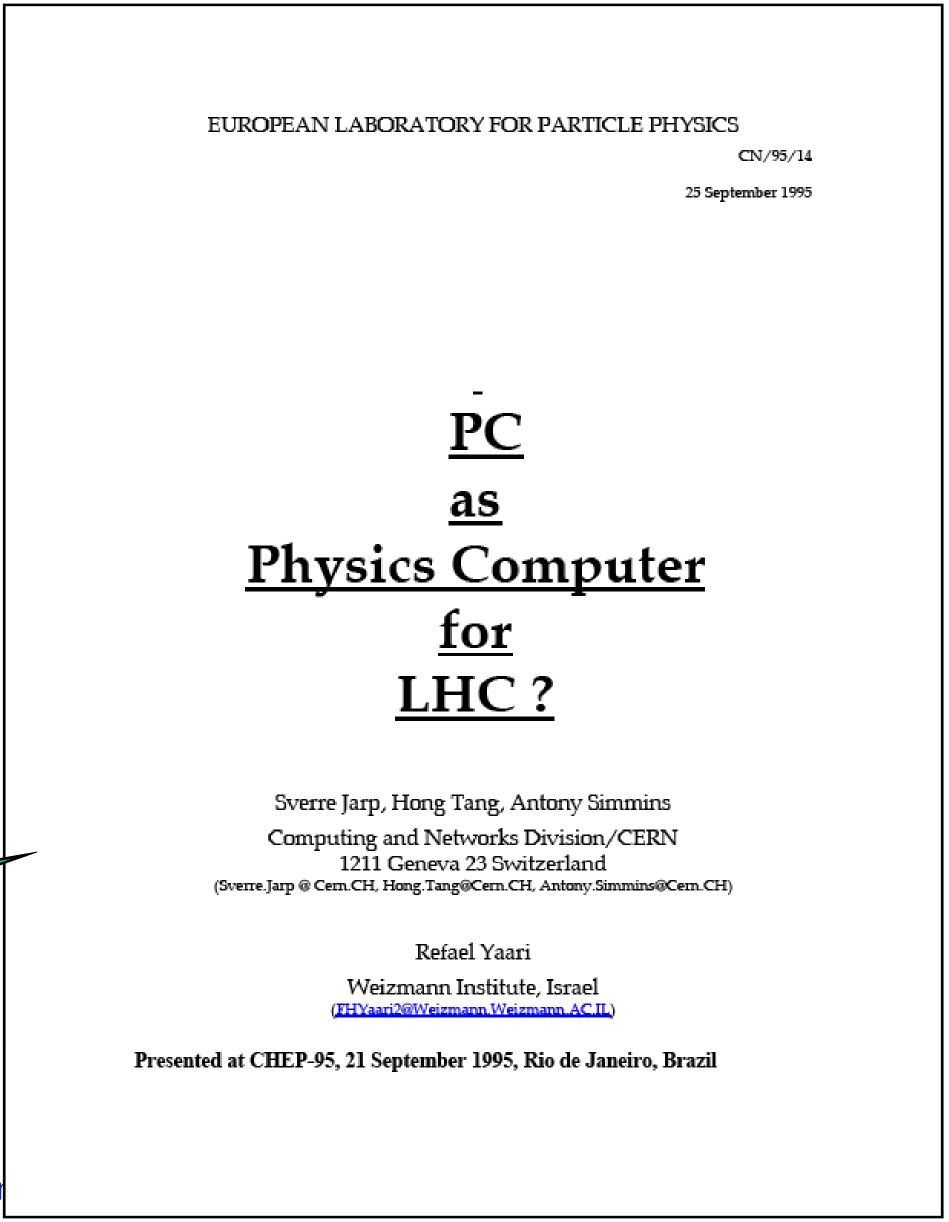
For more in-depth information, see: "Processors size up for physics at the LHC", S.Jarp CERN Courier (April 2007)

Before we start

Start of the x86 era for HEP

- **Our presentation at CHEP-95 in Rio →**
  - 12 years ago!
  - First porting and benchmarking of HEP codes (in FORTRAN)
    - CERNLIB
    - CERN benchmarks
    - GEANT3
    - ATLAS DICE (simulation)

Hey, a 133 MHz PC is as fast as the (much more expensive) workstations!



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)

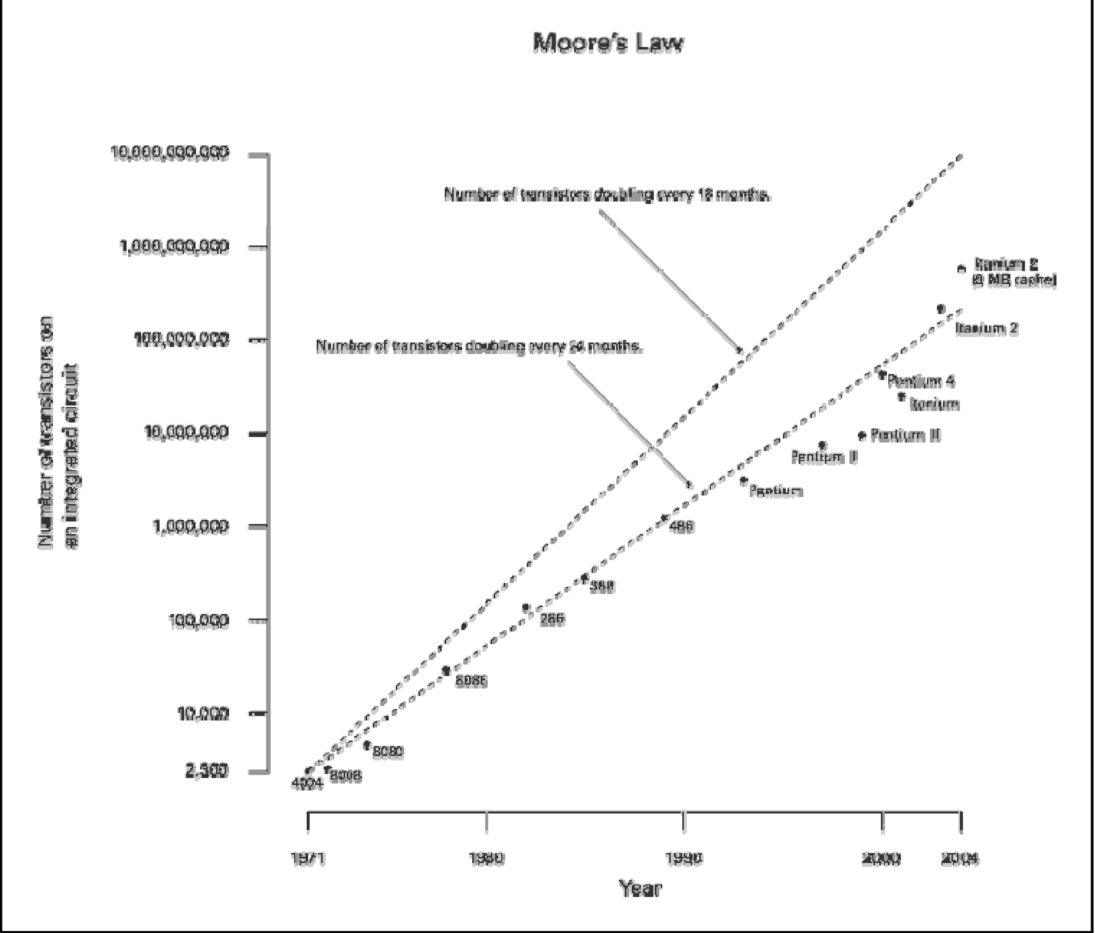
Refael Yaari
Weizmann Institute, Israel
(FHYaari2@Weizmann.Weizmann.AC.IL)

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

Transistor growth

Moore's law

- **Gordon Moore predicted that transistor count would double every 18 – 24 months**
  - This is still roughly true – even after more than 40 years!



Note that a derivative "law" stated that the frequency would also double. This is no longer the case!

Illustration from Wikipedia

Implications of Moore’s law

- **Initially the processor was simple**
  - Modest frequency; Single instruction issue; In order; Tiny caches; No hardware multithreading or multicore; Running cool

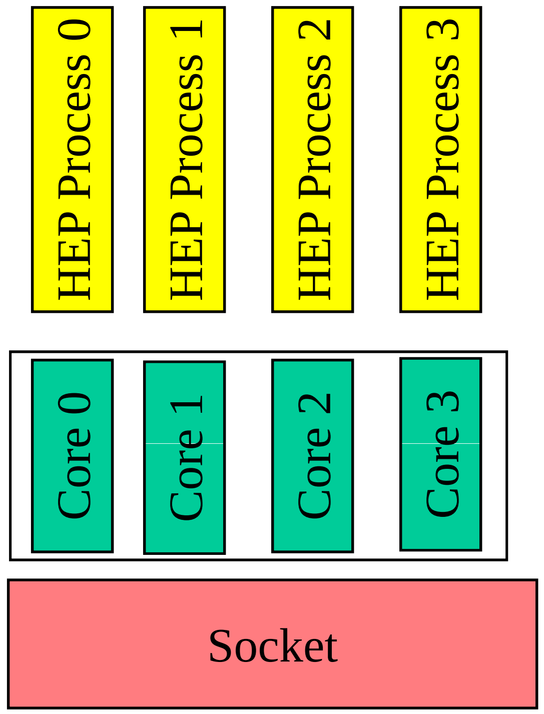
- **Since then:**
  - Frequency scaling (from 150 MHz to 3 GHz)
  - Multiple execution ports, wide execution (SSE):
  - Out-of-order execution:
  - Larger caches:
  - HW multithreading:
  - Multi-core:
  - Heat:

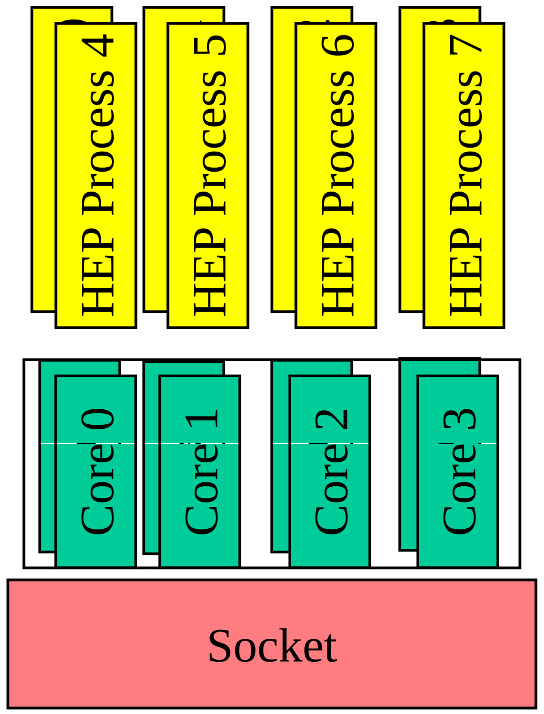
All of this has been absorbed without any change to our software model: Single-threaded processes farmed out per processor core.

Understanding hardware and software basics

Single threaded processes

- **Simply illustrated:**





Quad-core

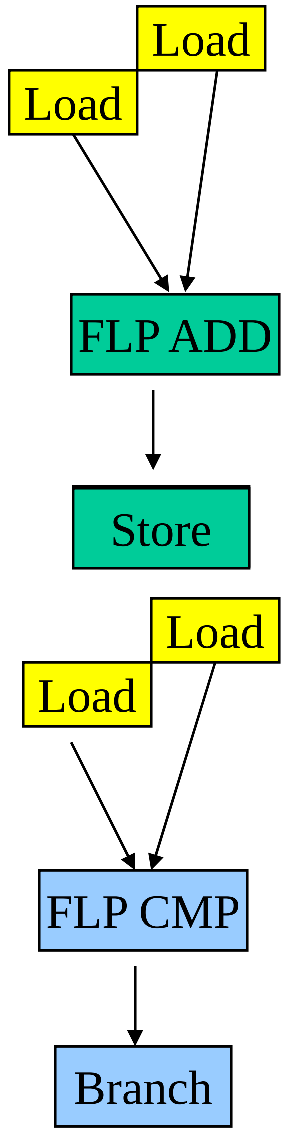
Octo-core or Quad-core w/two-way HW Multithreading
(seen by the OS as 8 independent CPUs)

Our memory usage

- **An initial preoccupation:**
  - Today, we need 2 – 4 GB per single-threaded process.
  - In other words, a dual-socket server needs at least:
    - Single core: 4 - 8 GB
    - Quad core: 16 - 32 GB
    - Future 16-way CPU: 64 – 128 GB (!)
    - Future 64-way CPU: 256 – 512 GB (!!)

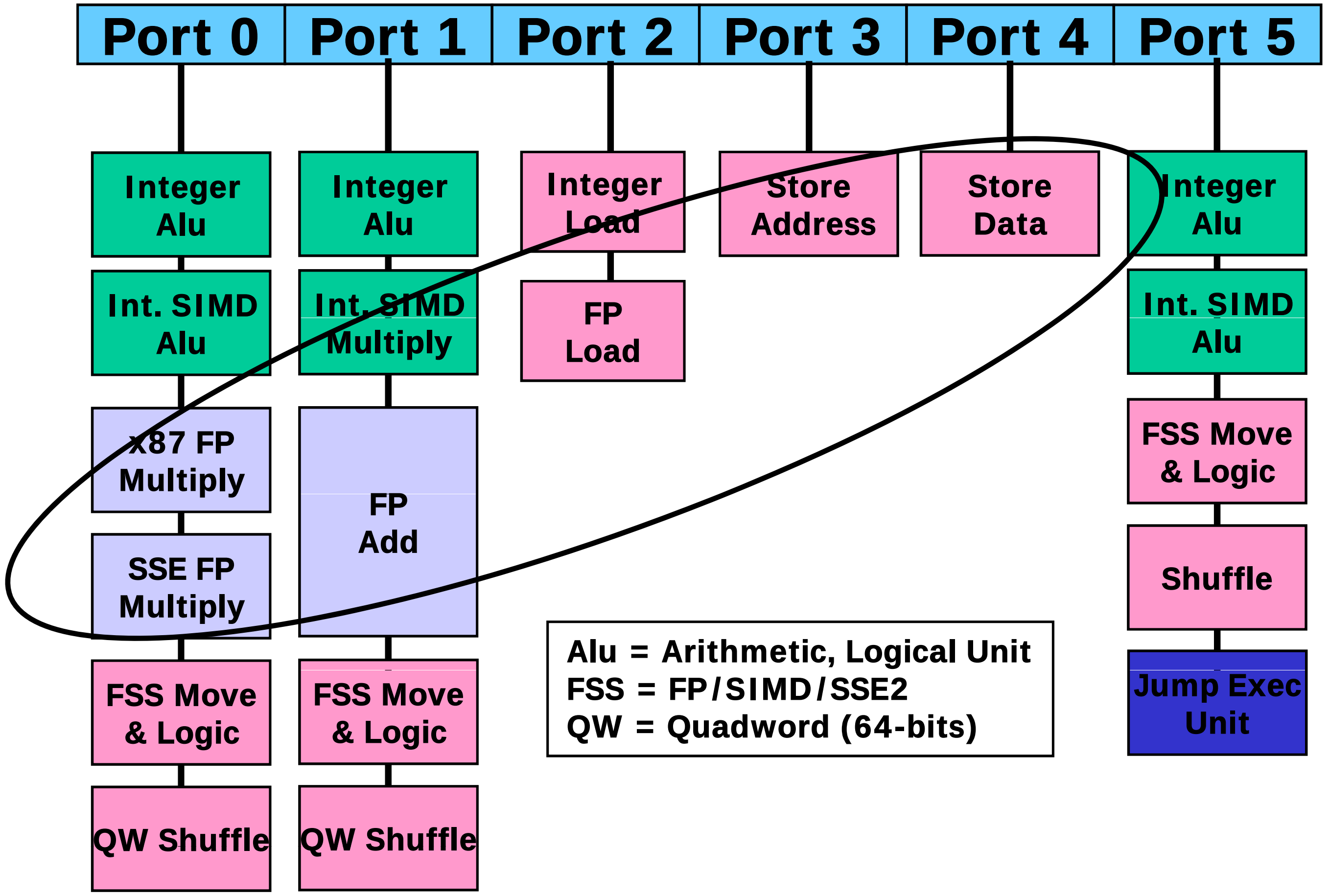
Are we FLP or INT ?

- **Some people believe that our programs are entirely “logic intensive”**
  - This is a misunderstanding!
- **Our programs (naturally) operate on floating-point entities:**
  - (x,y,z), (p_x, p_y, p_z), etc.
- **A better description is:**
  - We have floating-point work wrapped in “if/else” logic
- **My estimate**
  - Atomic operations (fadd, fmul, etc.) represent 15-20% of all instructions
  - But all floating-point work (all loads, atomic ops, math functions, and stores) represents ~50% of the cycles
- **So why does it scale with SPECint?**



Today's architectures are "fat"

- Execution ports in the Core 2 processor:



Alu = Arithmetic, Logical Unit
FSS = FP/SIMD/SSE2
QW = Quadword (64-bits)

Issue ports in the Core 2 micro-architecture (from Intel Manual No. 248966-014)

HEP programs are “lean”

High level C++ code →

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

Assembler 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

ILP in HEP

- **ILP (Instruction Level Parallelism)**
  - Our LHC programs typically issue (on average)
    - only 1 instruction per cycle
  - This is very low!
    - Core 2 architecture can handle 4 instructions
    - Each SSE instruction can operate on 128 bits (2 doubles)
  - We are typically only extracting 1/8 of maximum

We are not getting out of first gear!!

FLP in HEP

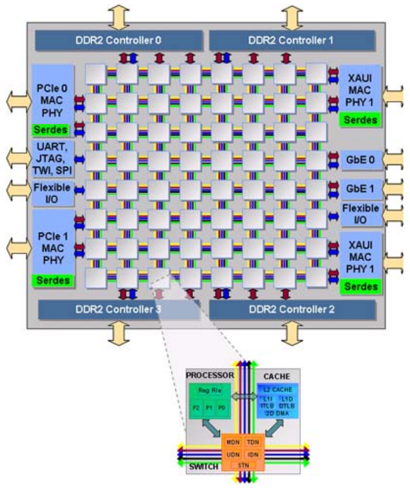
- **Floating-point performance**
  - Intel Core 2 can do 4 FLOPs per cycles
  - We just said that we execute ~1 instruction per cycle
    - And that 20% are floating-point operations
  - We probably average 0.2 FLOPs per cycle

5% of peak: We are crawling along in first gear!!

What is coming?

- **Industry will bombard us with new designs based on multi-billion transistor budgets**
  - Hundreds of cores;
    - Somebody even mentioned “thousands” recently !
  - Multiple threads per core
  - Unbelievable floating-point performance
    - The race is on for Tera-FLOP chips
    - Aggressive graphics designs from existing vendors and new contenders
  - Because of thermal issues: Many are back to in-order designs
    - For instance: Itanium, Sun Niagara, Intel Power6
    - Others may follow

Of course, we will also continue to see the traditional x86-64 processors evolve (as expected).



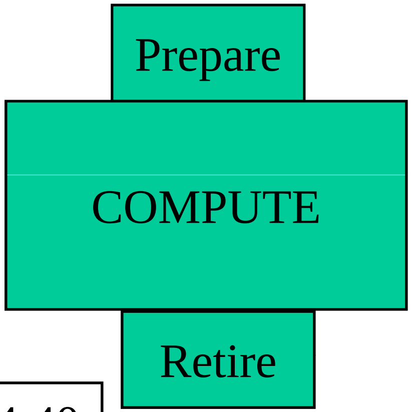
Some suggestions

Why worry?

- **Clearly, the emphasis now is to get LHC started and there is plenty of compute power across the Grid.**
- **The suggestions are only relevant if we want to extract (much) more compute-power out of new chip generations**
  - Try to increase the ILP (especially the floating-point part)
  - Investigate “intelligent” multithreading
  - Reduce our overall memory footprint

1) Increased ILP

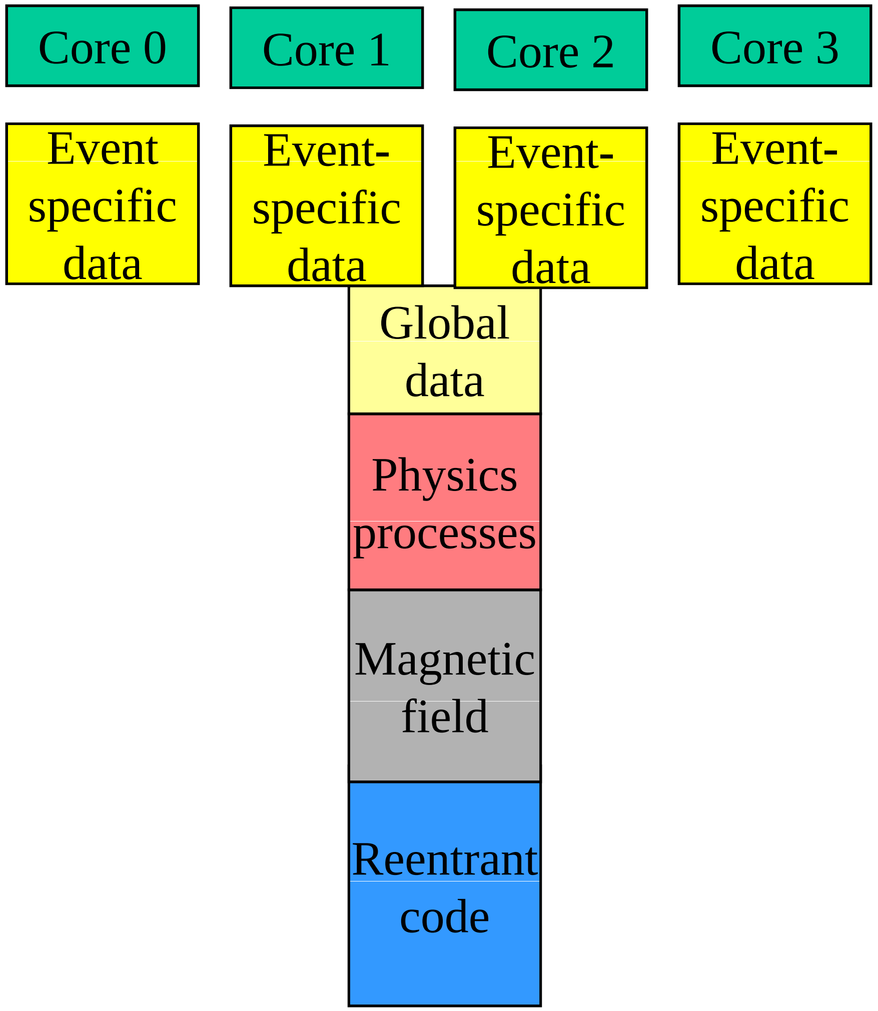
- **Aim at creating richer "sections", with especially the floating-point contents exposed**
- **Assist our C++ compilers in making these sections effective****
  - Optimization in all important areas
    - Inlining of "tiny" methods
    - Disambiguation of data pointers/references
    - Minimization of if and switch statements
    - Etc.
  - Optimization of mathematical functions
    - Log, exp, sine, cosine, atan2, etc



** Session 259 on CMS SW performance analysis tomorrow at 14:40
** Session 316 on performance monitoring tools tomorrow at 17:10

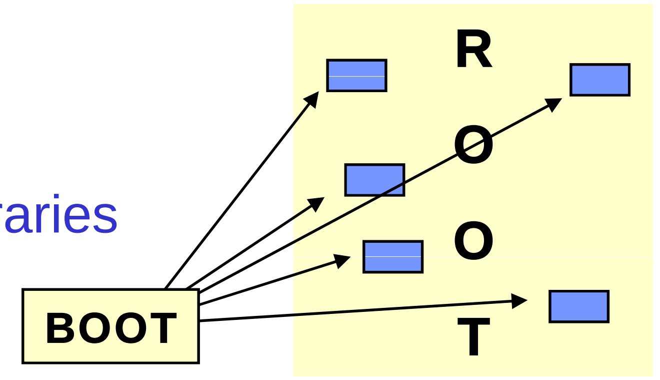
2) Multithreading

- Explore new paradigms, for example:



3) Simplify/restructure code

- **Today, our frameworks are very complicated and heavy**
  - In one case, we observed 400+ shared libraries



- **Make a move à la BOOT?**
  - Test coverage of various applications has shown that frequently the 80/20 rule applies:
    - 20% of the code is enough to cover 80% of the (even complex) use cases
- **Having a more modular approach would be very beneficial**
  - For instance,
    - Quicker porting to assess new hardware
    - Quicker adoption of new paradigms

CONCLUSIONS

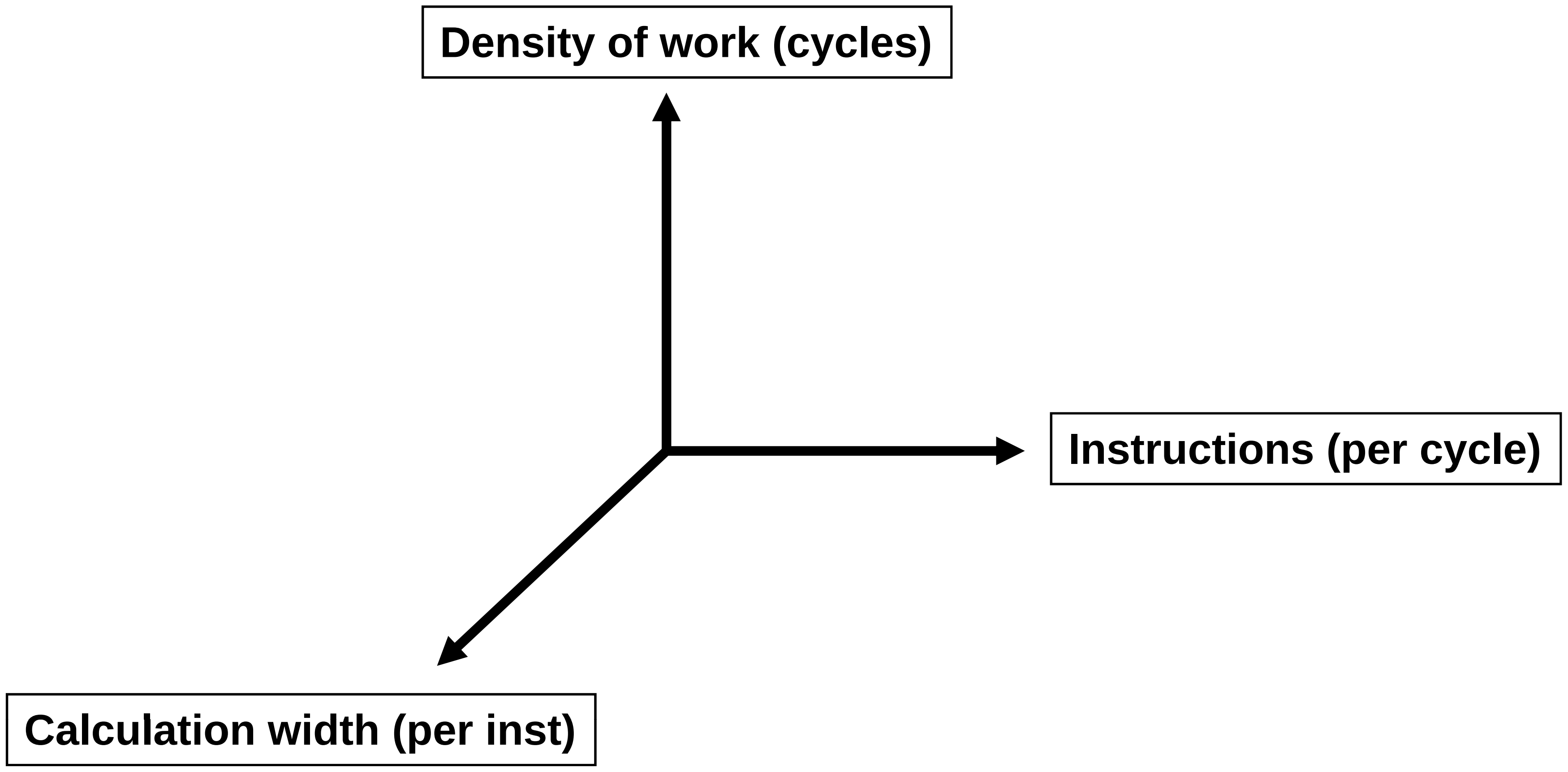
Conclusions

- **Moore's law has been extremely beneficial to HEP**
  - Especially frequency scaling (whilst it lasted), out-of-order execution (latency hiding), and multi-core
- **Thanks to x86 technology, we have enjoyed performance increases by several orders of magnitude**
  - Ever since CHEP-95
  - Both absolute performance and performance per CHF
- **If we need this to continue during an LHC era, which will be populated by billion-transistor processors**
  - We should increase the "agility" of our software structures
- **Your take-away:**
  - If we want fewer parallel jobs, fewer "heat-generating" servers for solving a given HEP problem, there are still plenty of under-utilized resources inside each CPU!

Backup

CPU performance vector

- Defined in 3 dimensions inside a processor:



Even simpler example

High level C++ code →

```
for (i=0; i<N; i++) sum += vector[i];
```

Assembler instructions →

```
.L5:
            addsd     (%rdi,%rax,8), %xmm0
            addq      $1, %rax
            cmpq      %rsi, %rax
            jne       .L5
```

Same instructions laid out according to the latency of the addsd instruction.

NB: the load vector instructions are done OOO.

Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1	addq	addsd	load vector[i]]			
2		(bubble)				
3		(bubble)				cmpq+jne

Incompressible part

Even simpler example (2)

- **When running simple test with vector[100]**
  - Remember that floating-point calculations are done on the SSE units
    - Which can issue two FLP operations in parallel (in a single cycle)

Compiler	**Cycles per addition**
gcc 3.4/4.2 (O2)	**3**
gcc 3.4/4.2 (O2, unrolled by hand)	**1**
icc 10.0 (O2, automatic vectorisation)	**.75**
Theoretical minimum	**.5**

This simple example illustrates well what we see in many HEP benchmarks: Only 10 – 20% of the resources are productive (unless we act)!