CINECA

Introduction to Performance Monitoring

Federico Tesser – HPC Specialist
f.tesser@cineca.it

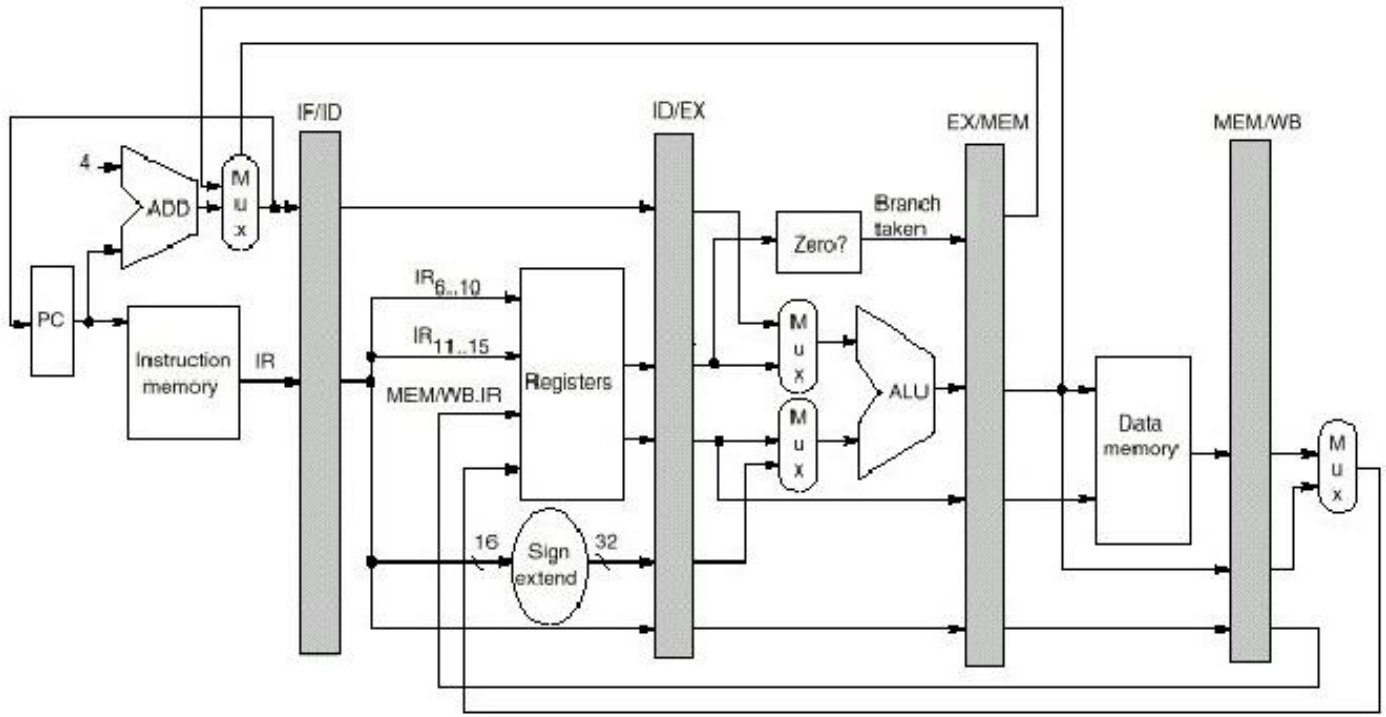
CINECA

Overview

- *Introduction on HPC microarchitectures*
- *Performance and efficiency of microarchitectures*
- *Jump into HW performance counters*
- *A view on Linux Perf*
- *Roofline model*
- *Conclusions & Take away messages*

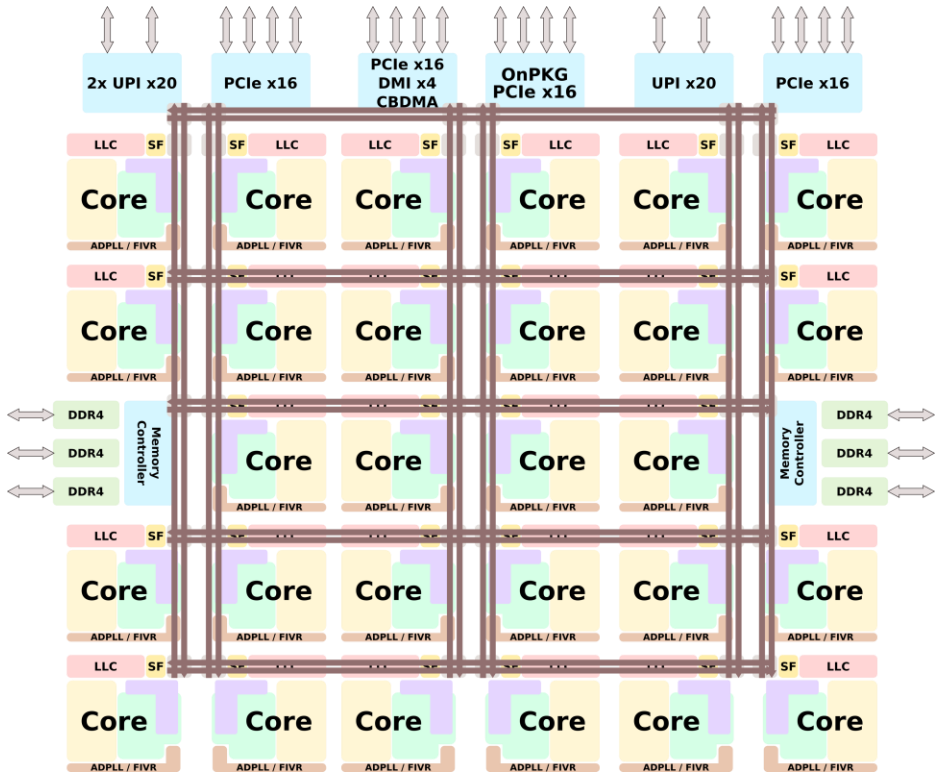
What they taught to you about *computer architecture*

CPU: scalar, in order, RISC based, 32bit, short pipeling, single level cache, single threading, single core, single socket with fixed operating frequency and uniform memory access



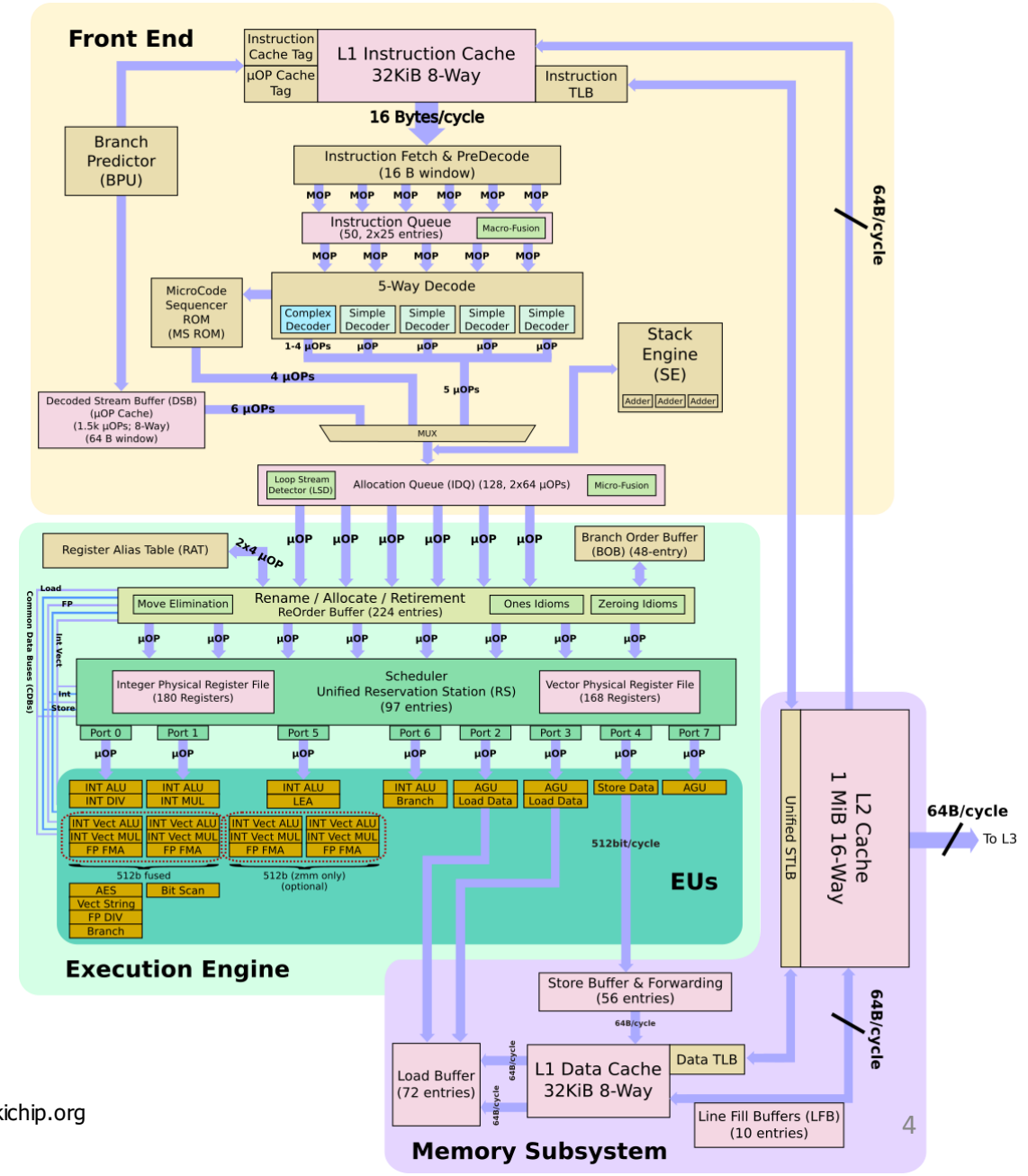
What it actually is

CPU: superscalar, out of order, multi-level caches, CISC based (x86), 64bit, multi-threading, many core, multi socket with dynamic voltage and frequency scaling and non-uniform memory access



Cascade lake SoC Layout

Cascade lake microarchitecture



https://en.wikichip.org

Goal: exploit performance

Performance is a result of:

- How many instructions you require to implement an algorithm
- How efficiently those instructions are executed on a CPU

But what does it mean "efficient execution"?

- **Scientific view:** HPC application ➡ Scientific algorithm + data ➡ Result
- **Computer view**: HPC application ➡ Set of finite sequences of computer instructions + digital data ➡ Result
- Computer performance ➡ Higher Instructions Per second/Cycle (IPC) ➡ Shorter execution time
- Almost true -> Eg. higher IPC with scalar instructions
- FLoating point Operations Per Second (FLOPS) ➡ Better metric ➡ TOP500 The List.

But remember the following three things:

1. It is impossible to reach the theoretical peak performance of a system;
2. Focusing on the optimization of a single performance metric can reduce other performance metrics (trade-off problem);
3. A single performance metric cannot express the overall efficiency of a microarchitecture but we need to consider multiple metrics;

µarch performance events (very few of them)

- **Cycles**: count the number of cycles
- **Instructions retired**: count the number of macro instructions executed
- **Vector instructions retired**: count the number of vector macro instructions
- **Branches**: count the number of branch taken
- **Cache miss/hit** (at multiple cache levels): count the miss/hit of the cache references -> it show the locality of the code
- **Memory read/write**: number of time that a cache line was read/written from the memory

µarch performance events ≠ performance metrics

- **FLOPS**: arithmetic operations executed
- **Memory throughput**: number of bytes exchange with the memory
- **IPC**: instructions per cycle -> this metric show the macro instruction throughput of the microarchitecture
- **Vectorization ratio**: percentage of how many vector instructions are retired wrt the total instructions

Micro architecture performance optimization

In modern CPUs it is very difficult to understand if my application is efficiently performing on a specific system (also from an energy point of view)!

We need HW support from the processor (PMU) -> **Only what is measurable can be improved!**

Tools help you to get access to the HW subsystem and automatize routines but...

Use your brain! Tools may help, but you do the thinking!!!

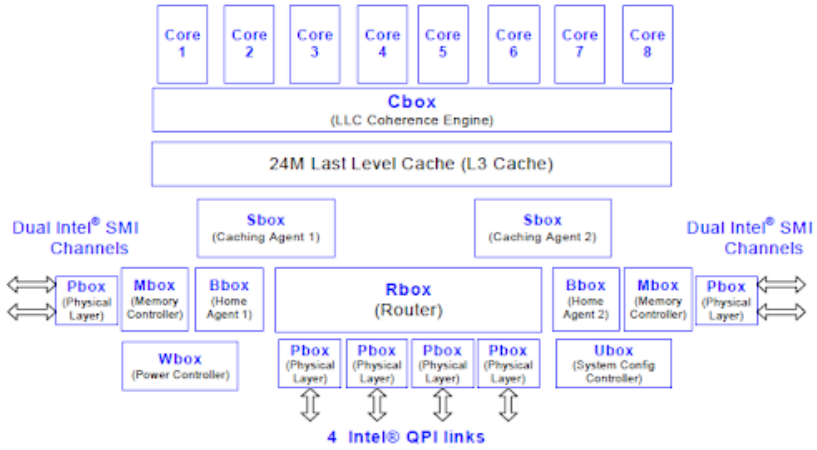
Performance Monitoring Unit (PMU)

The CPU supports you with the PMUs!

A PMU usually support many events (cycles, instructions retired, etc.) through **Performance Monitoring Counters (PMC)**.

A PMU can be:

- **on-core**: microarchitecture events at the core level (cycles, instructions retired, ...)
- **off-core:** microarchitecture events outside of cores (memory read/write, …)



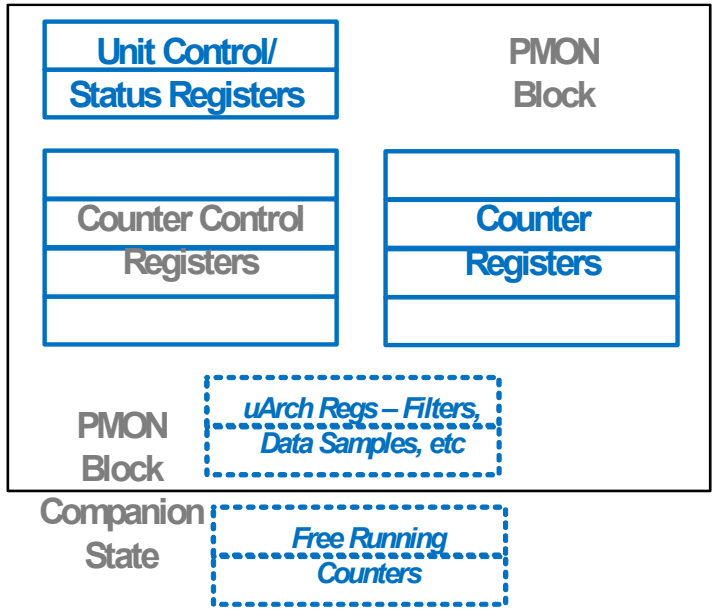
PMCs can be:

- **fixed**: can be only enabled or disable and profile a specific event
- **configurable**: can monitor many events

PMCs are usually configurable only at kernels level

Usually, CPUs provide at user space some assembly instructions with low overhead (see *rdpmc()*) to read PMCs

Pmon Uncore blocks



Box	# Boxes	# Counters/ Box	# Queue Enabled	Packet Match/ Mask Filters?	Bit Width
CHA	up to 28	4	1	Y	48
IIO	up to 6 between C, P and M flavors	4 (+1) per stack (+4 per port)	0	N	48
IRP	up to 6	2	4	N	48
IMC	up to 2 (each with up to 3 channels)	4 (+1) (per channel)	4	N	48
Intel® UPI	up to 2 (2 or 3 links)	4 (per link)	4	Y	48
M3UPI	up to 2 (2 or 3 links)	3 (per link)	1	N	48
M2M	up to 2	4	1	Y	48
PCU	1	4 (+2)	4	N	48
UBox	1	2 (+1)	0	N	48

The programming interface of the counter registers and control registers fall into two address spaces:

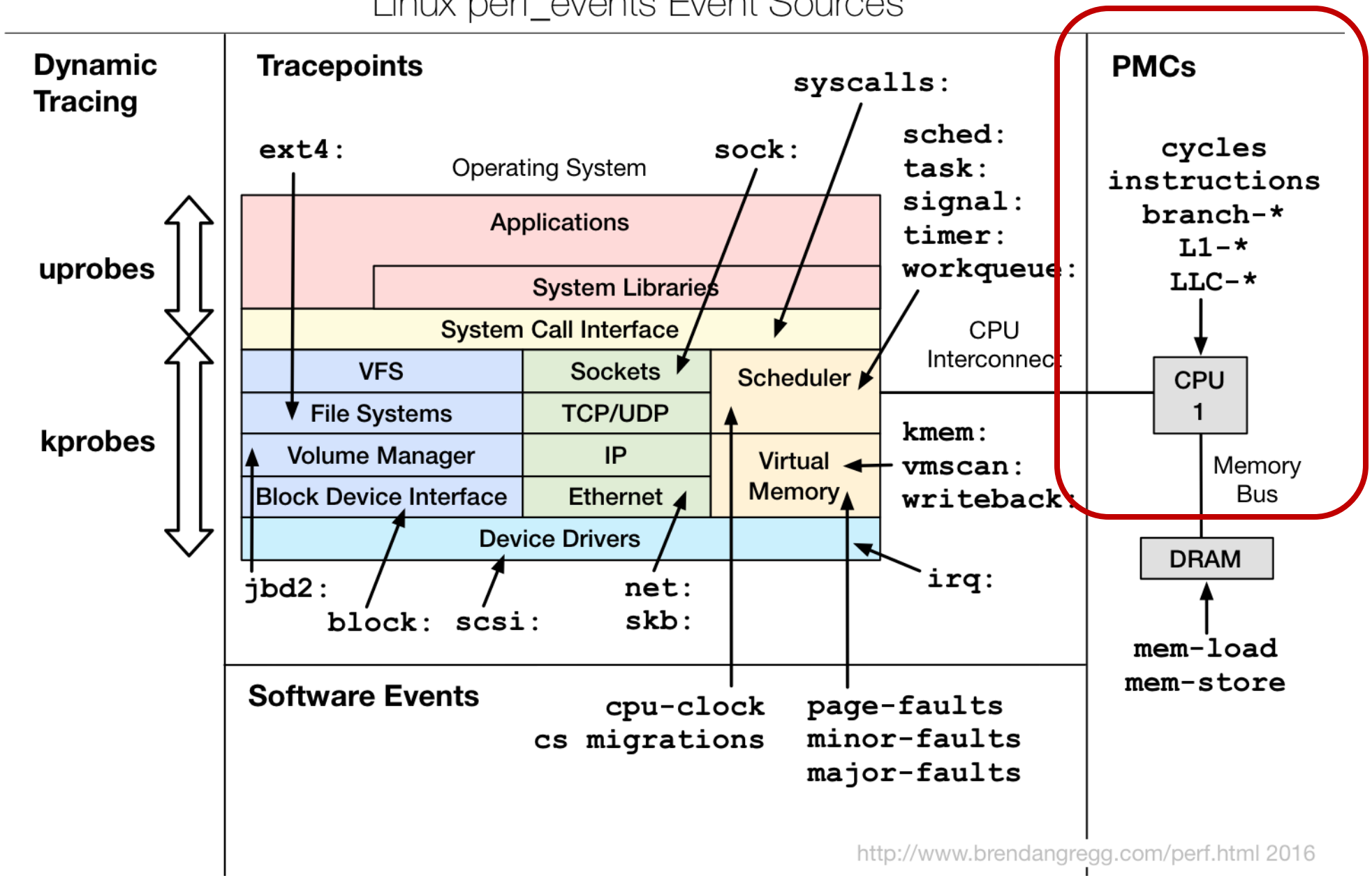
- Accessed by MSR are PMON registers within the CHA units, IIO, IRP, PCU, and U-Box.
- Access by PCI device configuration space are PMON registers within the **IMC**, Intel UPI and M3UPI units.

Enter Linux perf

- *Official* Linux profiler
  - Built on top of kernel infrastructure (`ftrace`)
  - Source and docs in kernel tree
- Provides a plethora of profiling/tracing features at all system levels
  - user, kernel, CGROUP, etc...
- Most important for us: **a comprehensive toolbox to gain workload execution insights via PMCs**
- Low overhead*
  - Tunable
  - 1-2% counting mode, 5-15% sampling w/multiplexing

* Nowak, Andrzej et al. "Establishing a Base of Trust with Performance Counters for Enterprise Workloads." *USENIX Annual Technical Conference* (2015).

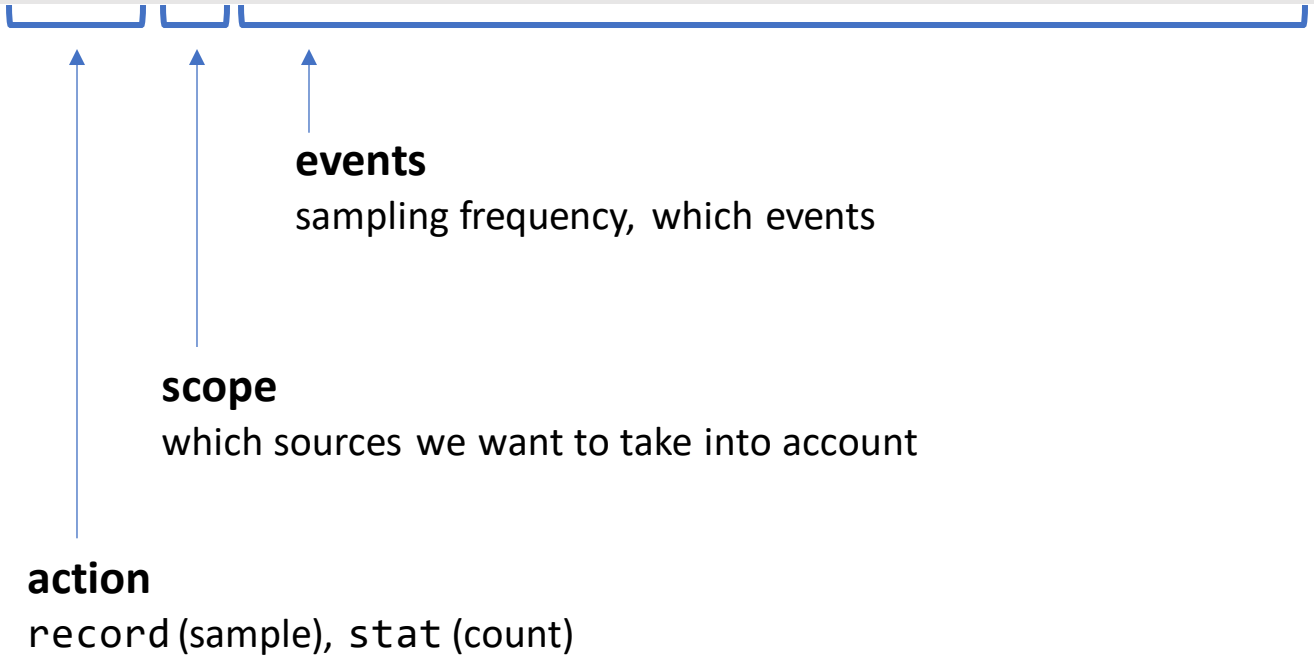
Linux perf_events Event Sources



http://www.brendangregg.com/perf.html 2016

perf stat/record format

```
$ perf record -a -F 4000 -e L1-dcache-load-misses,L1-dcache-loads -- $APP
```



events
sampling frequency, which events

scope
which sources we want to take into account

action
`record` (sample), `stat` (count)

MULTIPLEXING AND SCALING EVENTS

- Performance monitoring counters are present in a **fixed** number.
- If there are more events than counters, to be measured, **time multiplexing** gives to each event, a chance to access the monitoring hardware.
- Multiplexing **only** applies to hardware events.
- With multiplexing, an event is **not** measured all the time.
- At the end, the count is **scaled out** by a multiplying factor.

Fixed-function	General purpose
3	8

On-core # counters, @ recent Intel architectures

$$final\_count = raw\_count \cdot \frac{time\_enabled}{time\_running}$$

- This provide an **estimate**, of what the count would have been.

PERF_EVENT_OPEN

A call to ***perf_event_open()*** creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured.

The ***perf_event_attr*** structure provides detailed configuration information for the event being created.

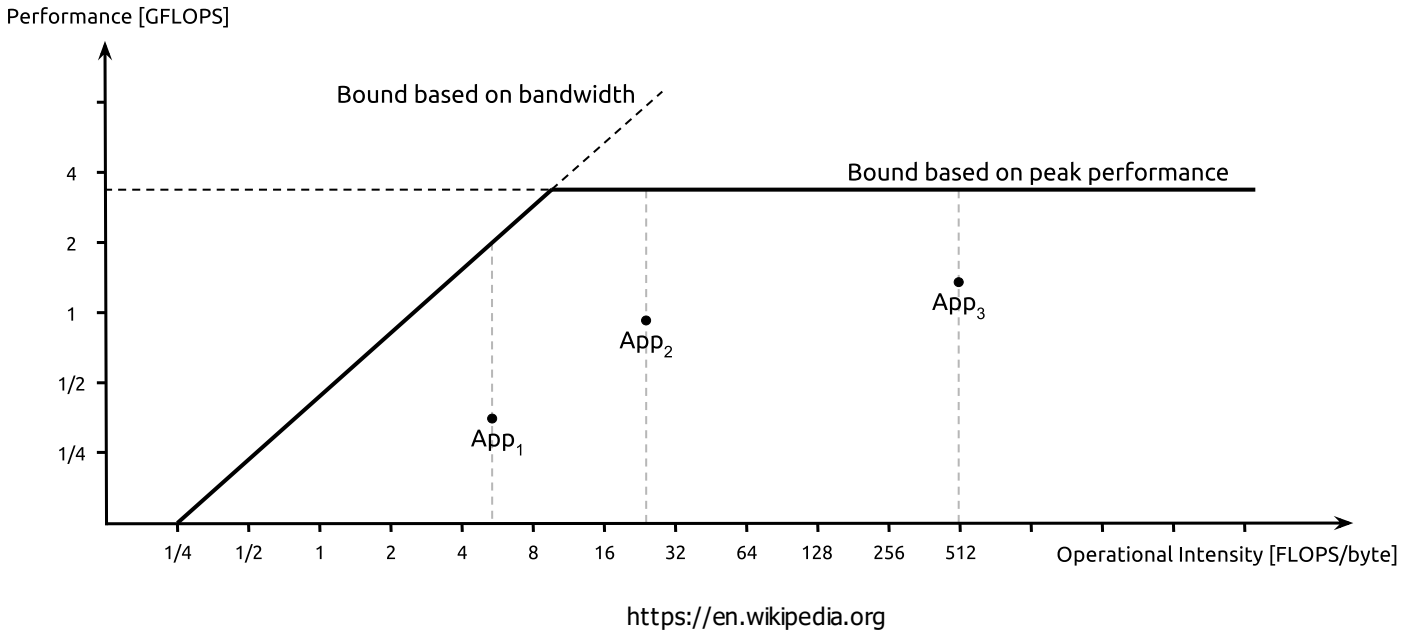
One of the fields of *perf_event_attr* is the ***__u32 type***, which indicates the type of the event, like:

PERF_TYPE_HARDWARE **PERF_TYPE_SOFTWARE**

PERF_TYPE_RAW

PERF_TYPE_RAW indicates a "raw" implementation-specific event in the ***__u64 config*** field (usually composed by two hexadecimal values: a mask and an event selector).

The Roofline model

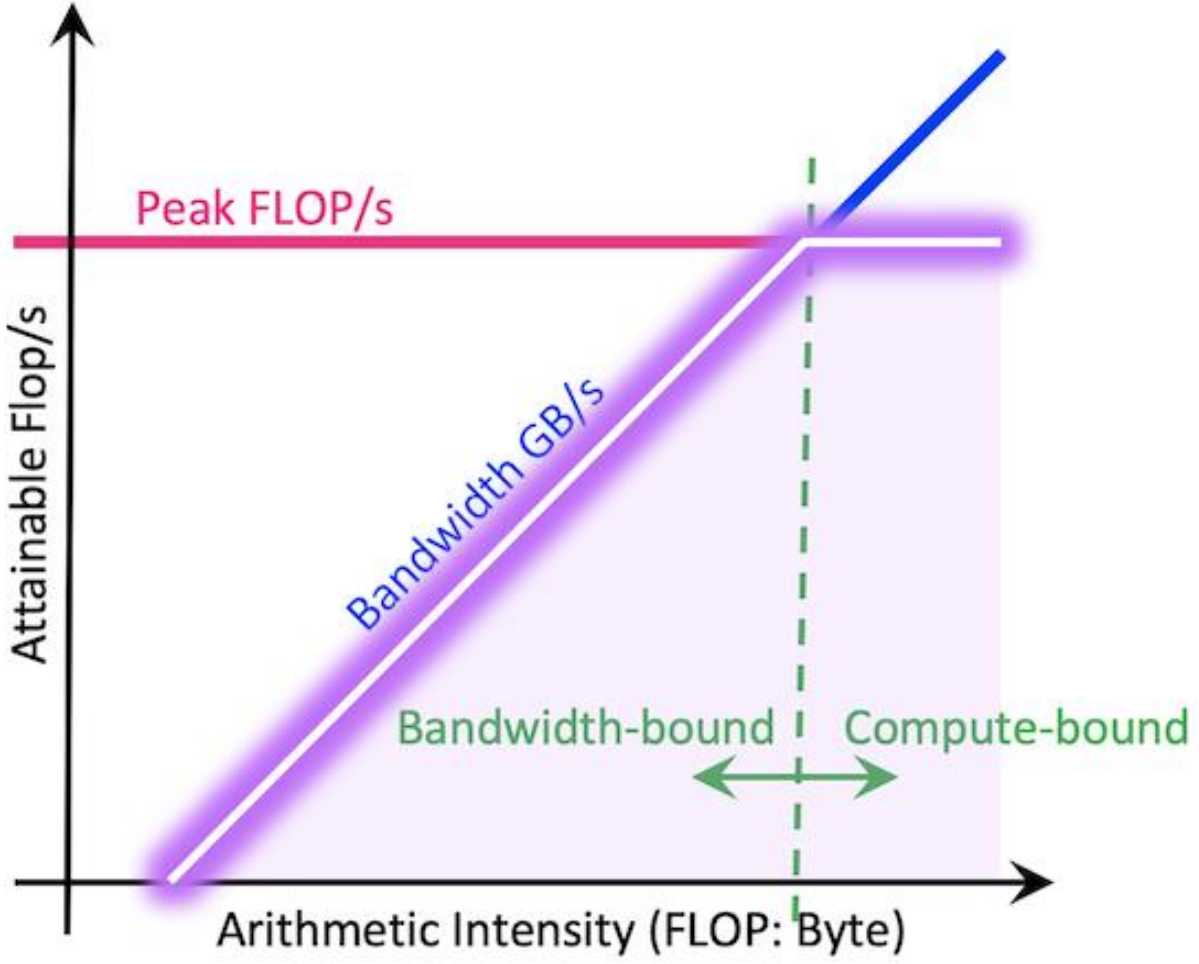


- The **work W** denotes the number of operations performed by a given application as *FLOPs.*
- The **memory traffic Q** denotes the number of bytes of memory transfers, during the execution of the application.
- The **arithmetic intensity I** is the ratio of the work W to the memory traffic Q (*FLOPs/byte*).

$$P = min \begin{cases} \pi \\ \beta \cdot I \end{cases}$$

The **π** is the ***peak performance***, while the **β** is the ***peak bandwidth.***

The Roofline model - 02



- Is performance limited by compute or data movement?
- **Y-axis:** performance in FLOPS
- **X-axis:** Arithmetic Intensity AI (FLOP/Byte)
  - Ratio between total FLOP and total byte exchange with main memory
  - Measure of data locality (data reuse)
  - Typical machine balance is 5-10 AI
  - Stream TRIAD: 0.083 AI (2 FLOP per 24 bytes)
- Application/kernel near the roofline are making **good use** of computational resources
  - Compute bound: >50% of peak performance
  - Bandwidth bound: >50% of Stream Triad
- Bad performance:
  - Insufficient cache bandwidth
  - Bad data locality
  - Integer heavy code
  - Lack of FMA
  - Lack vectorization

Roofline with perf

Floating point instructions

- FP_ARITH_INST_RETIRED.SCALAR_SINGLE/DOUBLE
- FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE/DOUBLE
- FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE/DOUBLE
- FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE/DOUBLE

Number of X-bit computational single/double precision floating-point instructions retired.

Memory read/write instructions

- UNC_M_CAS_COUNT.RD
- UNC_M_CAS_COUNT.WR
- UNC_M_CAS_COUNT.ALL

CAS (Column Address Select) commands are issued to specify the address to read or write on DRAM.

These events count all CAS commands issued to DRAM **PER** memory channel.

https://perfmon-events.intel.com

Roofline with perf - 02

Floating point instructions

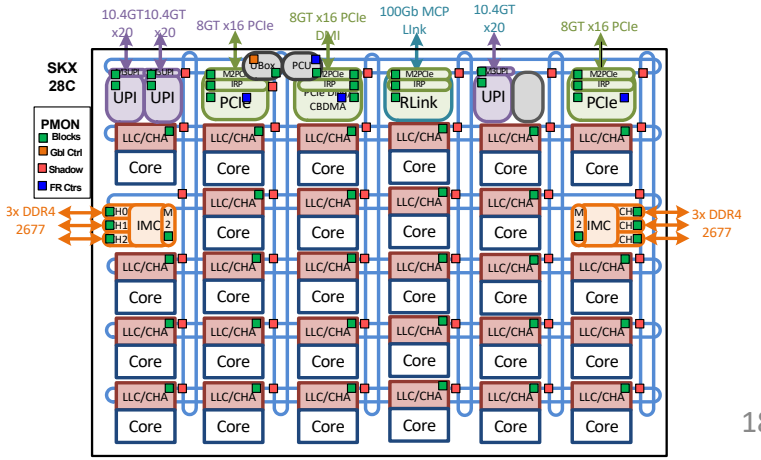
- EventSel=C7H UMask=02H/01H
  Counter=0,1,2,3
  CounterHTOff=0,1,2,3,4,5,6,7
- EventSel=C7H UMask=08H/04H
  Counter=0,1,2,3
  CounterHTOff=0,1,2,3,4,5,6,7
- EventSel=C7H UMask=20H/10H
  Counter=0,1,2,3
  CounterHTOff=0,1,2,3,4,5,6,7
- EventSel=C7H UMask=80H/40H
  Counter=0,1,2,3
  CounterHTOff=0,1,2,3,4,5,6,7

https://perfmon-events.intel.com

Memory read/write instructions

- EventSel=04H UMask=03H
  Counter=0,1,2,3
- EventSel=04H UMask=0CH
  Counter=0,1,2,3
- EventSel=04H UMask=0FH
  Counter=0,1,2,3

No way to change the channel (counters are for memory channel)?



Intel® Xeon® Processor Scalable Memory Family Reference Manual

ACCESS PERFORMANCE COUNTERS

- Obtain the event type number by:

  cat /sys/bus/event_source/devices/uncore_imc_*/type

  *w*here * goes from 0 to 5, included (in case of a CascadeLake system).

- Use those values to fill the field **type** of *perf_event_attr*, instead of using the predefined **PERF_TYPE_RAW** *type.*

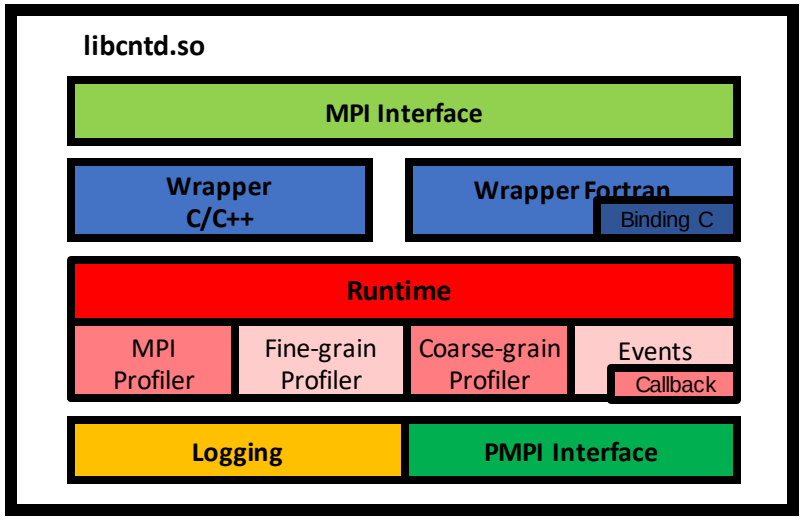
- *Use the pairs UMask and EventSel* from the hardware events references.

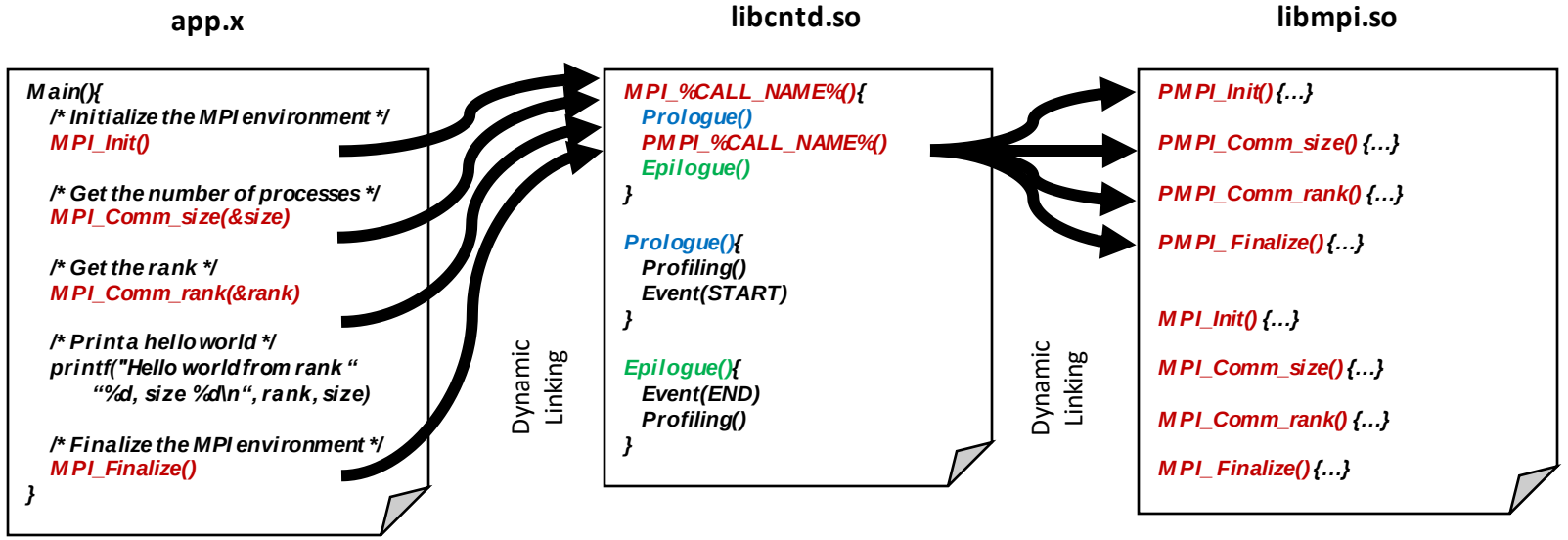
- Example reported for the *uncore_imc_0 (type* 0xe*) and for the event UNC_M_CAS_COUNT.RD* (*Umask* 03, *EventSel* 04).

```
Main(){

    …..
    …..
    struct perf_event_attr perf_pe;

    /*Giving the list of parameters.*/
    perf_pe.type = 0xe;
    perf_pe.size = sizeof(perf_pe);
    perf_pe.pinned = 0;
    perf_pe.disabled = 1;
    perf_pe.exclude_kernel = 1;
    perf_pe.exclude_hv = 1;
    perf_pe.config = 0x0304;

    /*set up performance monitoring.*/
    fd = perf_event_open(&perf_pe, pid, cpu = -1, -1, 0);

     /*Enabling the event.*/
     ioctl(fd, PERF_EVENT_IOC_RESET, 0);

     /*Resetting the event.*/
     ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

     /*Reading the event.*/
     read(fd, &count, sizeof(count));

     /*Disabling the event.*/
     ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
     ….
     ….
}
```

ACCESS PERFORMANCE COUNTERS -02

- Not easy at all!
- For this reason, we developed **COUNTDOWN** (CINECA + UNIBO project)
- **COUNTDOWN** instruments the application at execution time and collect information on the application workload.





COUTDOWN: https://github.com/EEESlab/countdown

ACCESS PERFORMANCE COUNTERS -03

```
################### GENERAL INFO ####################
Number of MPI Ranks:    48
Number of Nodes:        1
Number of Sockets:      2
Number of CPUs:         48
#################### ENERGY ########################
PKG:                    9997 J
DRAM:                   1112 J
#################### AVG POWER #####################
PKG:                    304.45 W
DRAM:                   33.86 W
################## PERFORMANCE INFO #################
MPI network - SENT:     239.62 GByte
MPI network - RECV:     239.17 GByte
MPI network - TOT:      478.79 GByte
MPI file    - WRITE:    0 Byte
MPI file    - READ:     0 Byte
MPI file    - TOT:      0 Byte
MAX Memory usage:       5.47 GByte
AVG IPC:                          2.66
AVG CPU frequency:                2390 MHz
AVG CPU load:                     0.96
Cycles:                           3760398709795
Instructions retired:             10011309614097
DP GFLOPS/sec (64/128/256/512/TOT): 11.05/13.73/21.64/466.25/512.67
SP KFLOPS/sec (32/128/256/512/TOT): 1.65/0.00/0.00/0.00/1.65
MEM Data Volume in TBytes: 1.437
MEM Bandwidth in GBytes/s: 44.804
DP Computational intensity in FLOPS/bytes: 11.443
SP Computational intensity in FLOPS/bytes: 0.000
DP Vector ratio: 0.978
SP Vector ratio: 0.000
#################### MPI TIMING ####################
APP time: 904.543 sec (57.24%)
MPI time: 675.775 sec (42.76%)
TOT time: 1580.317 sec (100.00%)
#################### MPI REPORTING #################
MPI_INIT: 48 - 0.000 Sec (0.00%)
MPI_ALLREDUCE: 156474 - 51.487 Sec (7.62%) - SEND 100.49 GByte - RECV 100.49 GByte
MPI_ALLTOALL: 1360416 - 250.600 Sec (37.08%) - SEND 123.36 GByte - RECV 123.36 GByte
MPI_BARRIER: 840837 - 77.015 Sec (11.40%)
MPI_BCAST: 593907 - 286.535 Sec (42.40%) - SEND 8.69 GByte - RECV 7.92 GByte
MPI_COMM_SPLIT: 2328 - 0.964 Sec (0.14%)
MPI_REDUCE: 29547 - 8.662 Sec (1.28%) - SEND 7.08 GByte - RECV 7.39 GByte
MPI_WAITALL: 507176 - 0.256 Sec (0.04%)
MPI_WAIT: 1320 - 0.249 Sec (0.04%)
MPI_FINALIZE: 48 - 0.006 Sec (0.00%)
################## COUNTDOWN REPORTING ##############
MPI_ALLREDUCE: 18640 - 39.539 Sec (5.85%)
MPI_ALLTOALL: 8207 - 11.389 Sec (1.69%)
MPI_BARRIER: 29791 - 34.552 Sec (5.11%)
MPI_BCAST: 19865 - 262.370 Sec (38.83%)
MPI_COMM_SPLIT: 285 - 0.625 Sec (0.09%)
MPI_REDUCE: 5270 - 2.620 Sec (0.39%)
MPI_WAIT: 81 - 0.028 Sec (0.00%)
################### COUNTDOWN SUMMARY ###############
MPIs: 82139 - 351.123 Sec - MPI: 51.96% - TOT: 22.22%
#####################################################
```

Conclusions & Take away messages

- *Application workload have different performance on different architectures*
- *Lack of microarchitecture efficiency limits the application performance more than scalability*
- *Performance models can help you to understand how application use the system resources*
- *Tools may help to spot inefficiencies, but you do the thinking!*