What is my processor doing?

Federico Ficarelli

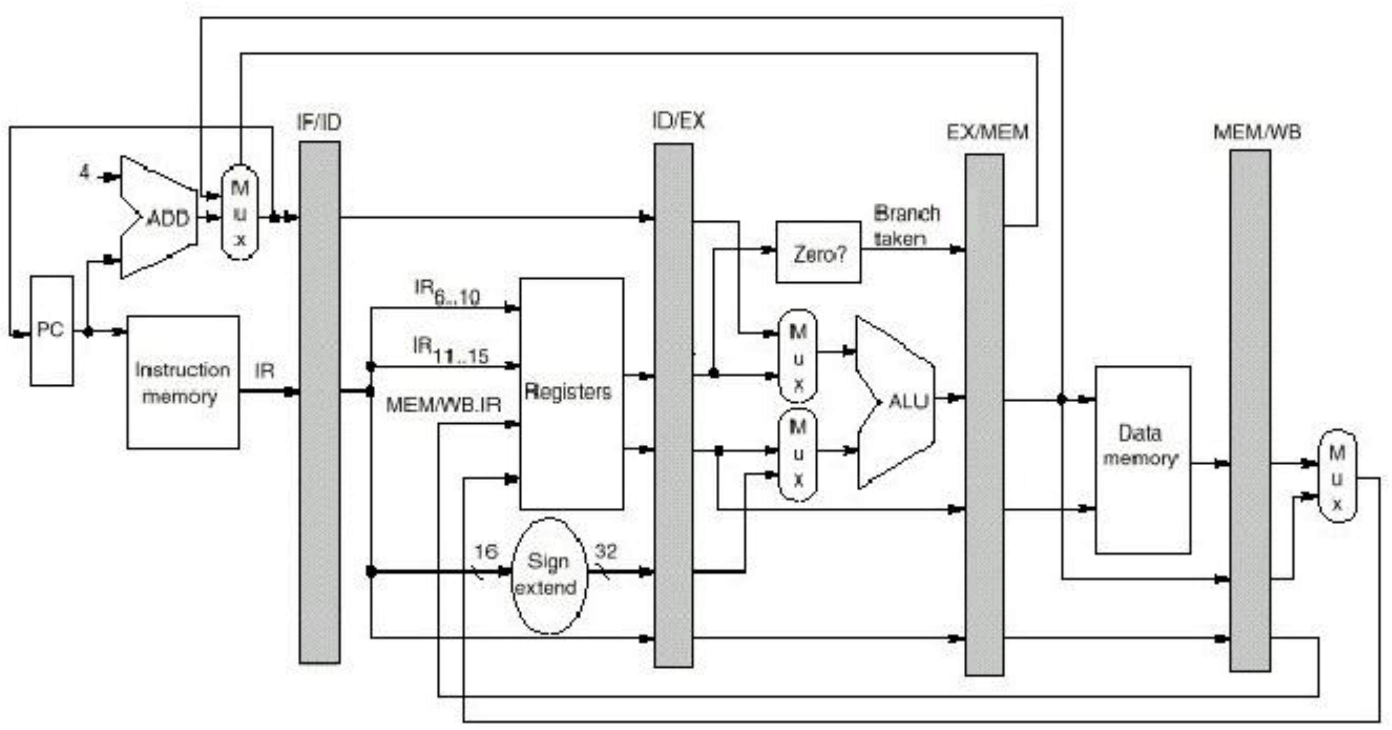
CINECA

goal: is my workload running *efficiently*?

- there is no such thing as *fast* or *performant*
- a workload can be *efficient or not*, i.e.: is wasting as little as possible according to its actual needs
- to be able to tell the difference we are going to:
    1. meet the micro-architecture
    2. meet the facilities hw provides to infer micro-architectural execution: PMU, PMCs, events
    3. meet the tool of the trade: `perf`
    4. see how to use the tool to apply methodologies that can help answer the question: *how many resources are we wasting*?

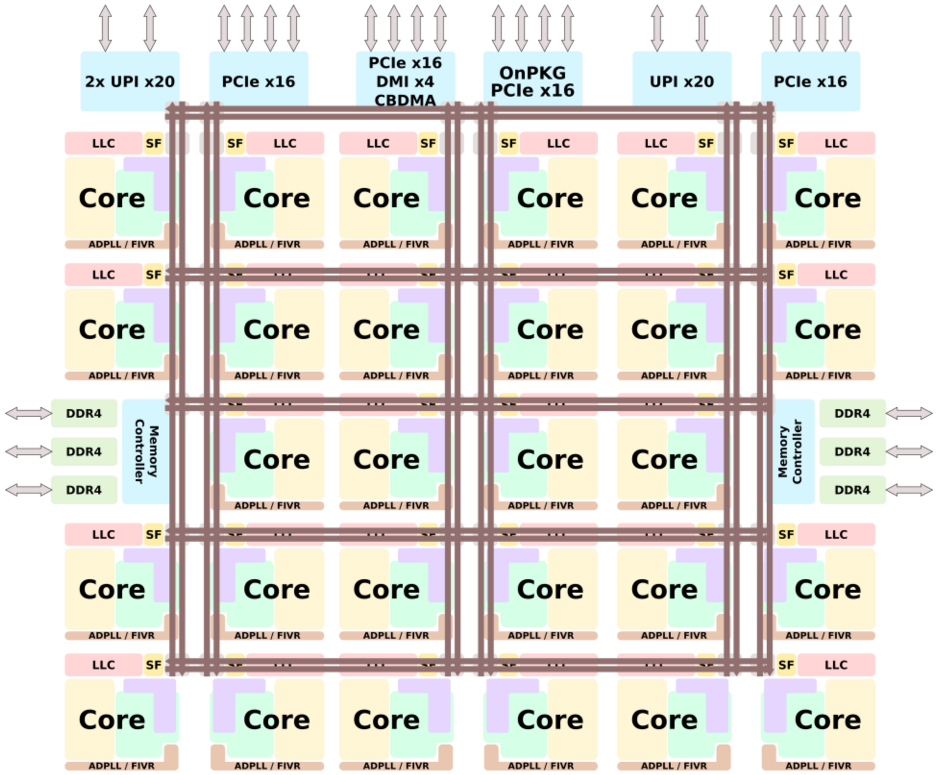
What did they teach you about CPU architecture

Scalar, in order, RISC based, 32bit, short pipeling, single level cache, single threading, single core, single socket processor with fixed frequency and uniform memory access



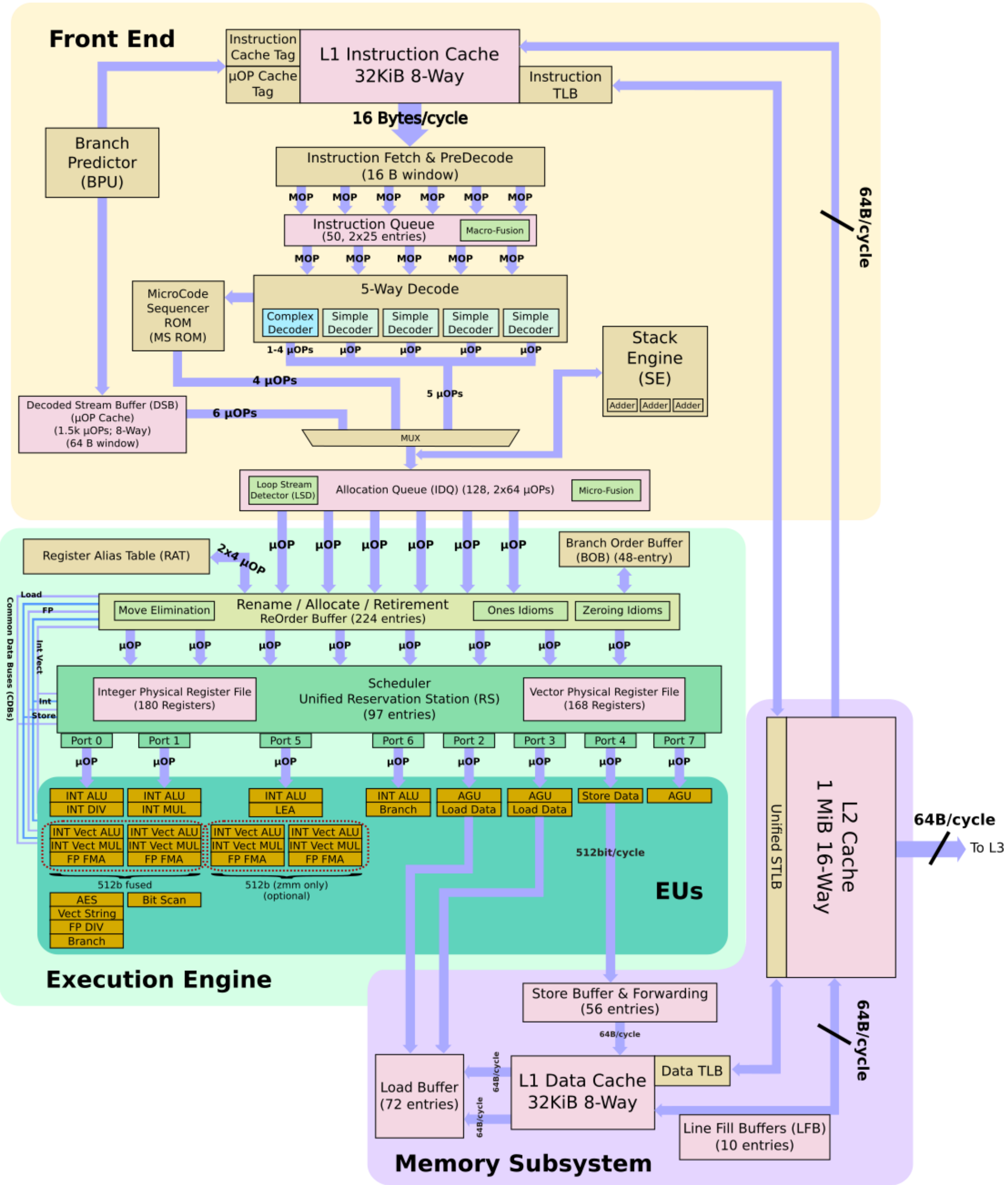
What it actually is

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



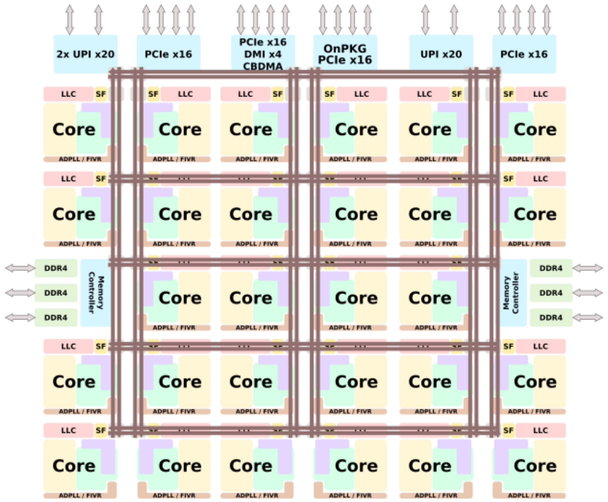
Cascade lake SoC Layout

Cascade lake microarchitecture



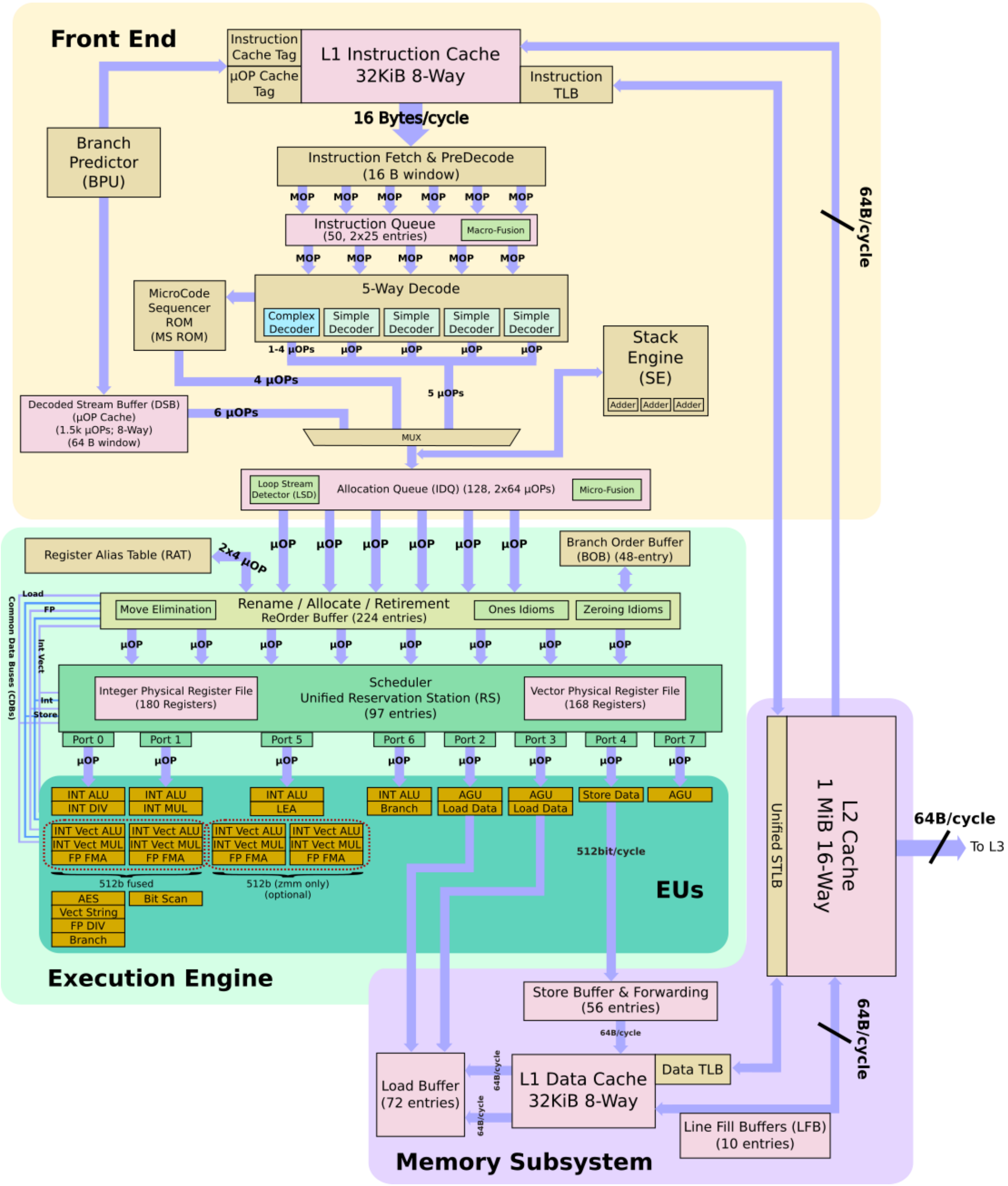
Intel Xeon Platinum 8260L Cascade Lake-SP

- **Core**: 24 (48 threads) running at 2.4 GHz
  - Turbo Mode: 3.9 GHz
  - Vector units: x2 AVX 512bit
- **L1 cache**: private 64 KB per core
  - 32 instructions + 32 data (8-way set-associative, 1.5MB)
- **L2 cache**: private 1 MB/core (16-way set associative, 24MB)
- **L3 cache**: shared 1.375 MB/core (11-way set associative, 33MB)
- **On-Chip interconnect**: Mesh (ring based on Broadwell)
- **Memory channel**: x6 DDR4-2933 (x2 memory controller)
- **Technology node**: 14 nm (8 billion transistors)
- **TDP**: 165W



Cascade lake SoC Layout

Cascade Lake-SP (Skylake-X) microarchitecture



Performance Metrics

Performance is a result of:

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

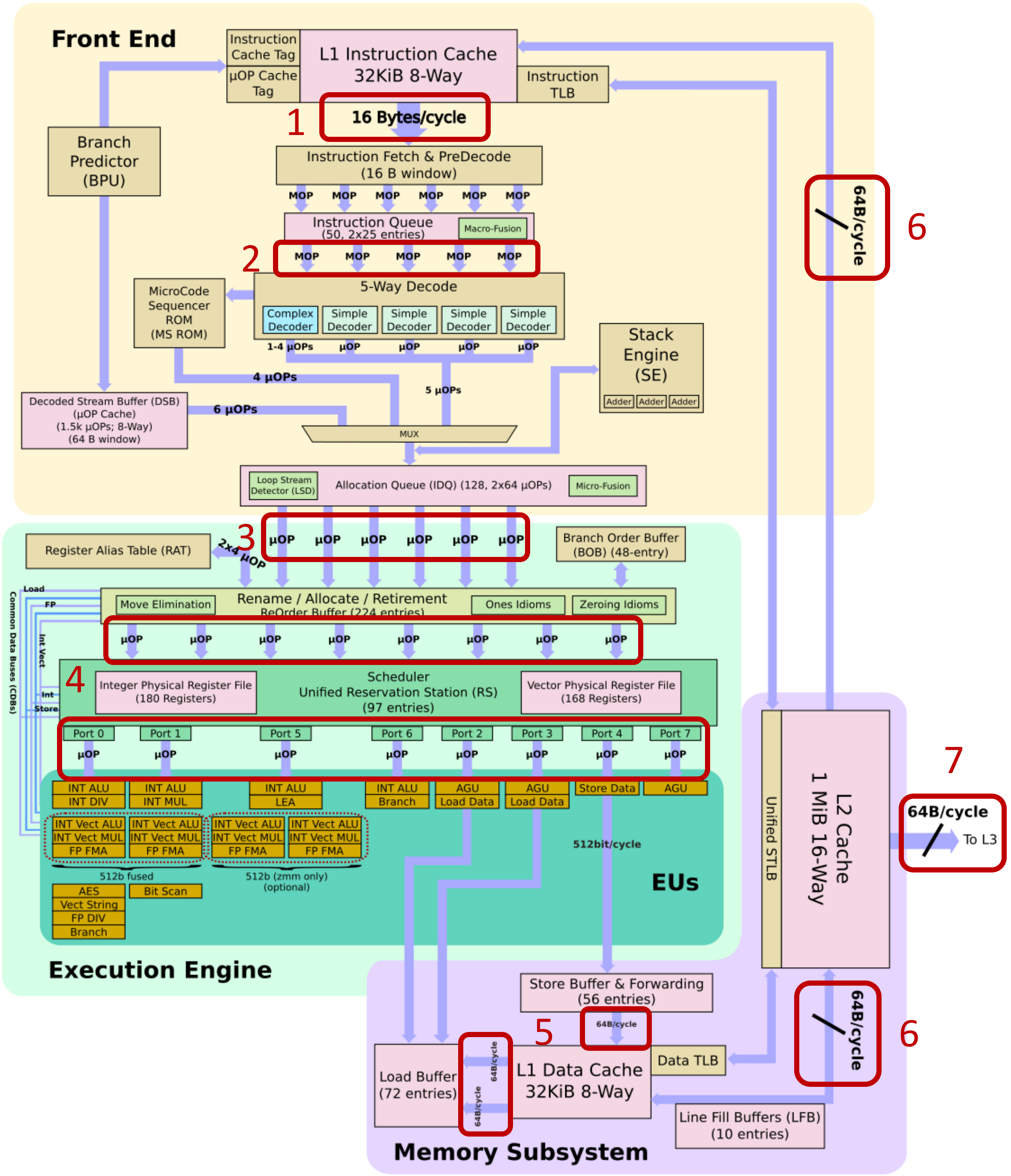
But what does it mean "efficient execution"?

- An HPC application is usually a representation of an algorithm that solves a scientific problem
- A scientific HPC application is represented as a set of finite sequences of architectural instructions that a computer needs to perform in order to conclude the computation.
- Higher is the number of instructions executed each second, shorter will be the execution time -> keep high the instructions per cycle (IPC)!
- IPC can be a misleading performance metric -> retiring scalar instructions or execute speculative instructions!
- Best metrics for HPC applications -> FLOPs! Used in the Top500 to classify the fastest supercomputers in the world!

But remember:

- It is impossible to reach the theoretical peak performance of a system;
- A single performance metric can limit other performance metrics (trade-off problem);
- A single performance metric cannot express the overall efficiency of a microarchitecture but we need to take into account multiple metrics.

Microarchitectural Throughput



1. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes (Intel® 64 and IA-32 Architectures Software Developer's Manual). The shorter instructions is the NOP (1 byte).
   The *Instruction fetch & PreDecode* unit can fetch <u>between 2 and 16 Macro Operations (MOP) per cycle</u> from the instruction cache (<u>16B/cycle</u>, latency 4-5 cycles)

2. The Intel Cascade Lake microarchitecture can fetch up to 5 MOP per cycle (due to various other more restricting points in the pipeline, we can practically fetch up to <u>4 µops per cycle</u>, see NOP example)

3. The front end and the back end are linked with an allocation queue buffer (IDQ) that can emit up to <u>6 µops per cycle</u>.

4. The backend can execute and retire up to <u>8 µops per cycle</u>.

5. Core - L1 Data cache: <u>2 cache line at cycle per load</u> (64Byte/cycle), <u>1 cache line at cycle per store</u> (64Byte/cycle), latency 4-5 cycles

6. L1I/D – L2 bandwidth: <u>1 cache line at cycle</u> (64Byte/cycle) one way, latency 14 cycles

7. L2 – L3 cache: <u>1 cache line at cycle</u> (64Byte/cycle) one way, latency 50/70 cycles

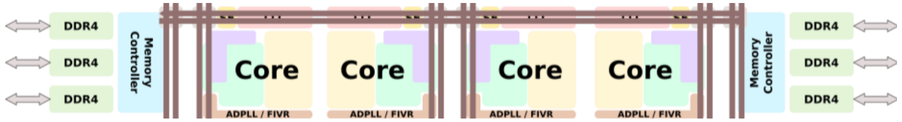
Keep the throughput high -> avoid stalls!

Ex. 1: Peak performance for a real-world CPU

Intel Xeon Platinum 8260L
Base freq = 2.4GHz
AVX512 = SIMD 512bit
GFLOPS ?

Memory Bandwidth

2x memory controller, x6 memory channels DDR4-2933 MT/s (1466 MHz, frequency is the half because transfers data on both the rising and falling edges of the clock signal Double Data Rate), channel Bus 64 bit (8 Byte)



Max memory bandwidth = ?

Ex. 1: Peak performance for a real-world CPU

Operating Frequency

		Turbo Frequency per active cores																							
	Base	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Normal	2.4GHz	3.9GHz	3.9GHz	3.7GHz	3.7GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.6GHz	3.3GHz	3.3GHz	3.3GHz	3.3GHz	3.1GHz	3.1GHz	3.1GHz	3.1GHz
AVX2	1.9GHz	3.7GHz	3.7GHz	3.5GHz	3.5GHz	3.4GHz	3.4GHz	3.4GHz	3.4GHz	3.3GHz	3.3GHz	3.3GHz	3.3GHz	3.0GHz	3.0GHz	3.0GHz	3.0GHz	2.7GHz	2.7GHz	2.7GHz	2.7GHz	2.6GHz	2.6GHz	2.6GHz	2.6GHz
AVX512	1.5GHz	3.7GHz	3.7GHz	3.5GHz	3.5GHz	3.4GHz	3.4GHz	3.4GHz	3.4GHz	3.0GHz	3.0GHz	3.0GHz	3.0GHz	2.6GHz	2.6GHz	2.6GHz	2.6GHz	2.4GHz	2.4GHz	2.4GHz	2.4GHz	2.3GHz	2.3GHz	2.3GHz	2.3GHz

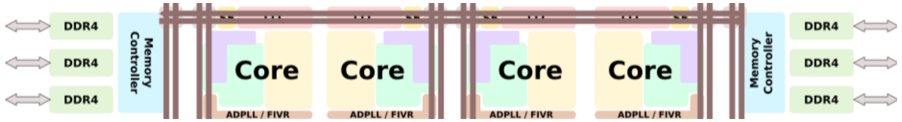
Floating Point Operations per Seconds (FLOPs)

Each Cores is equipped with x2 AVX 512 -> 8 FLOPs per cycle or 16 FLOPs is a FMA instruction (Fused Multiply-Add: *a = (a x b) + c)*

Peak FLOPs: (((512 bit / 64 bit) x 2 vector units) x 2 FMA) x Operating Frequency x Number of cores = 1.766 TFLOPs

Memory Bandwidth

2x memory controller, x6 memory channels DDR4-2933 MT/s (1466 MHz, frequency is the half because transfers data on both the rising and falling edges of the clock signal Double Data Rate), channel Bus 64 bit (8 Byte)



Max memory bandwidth

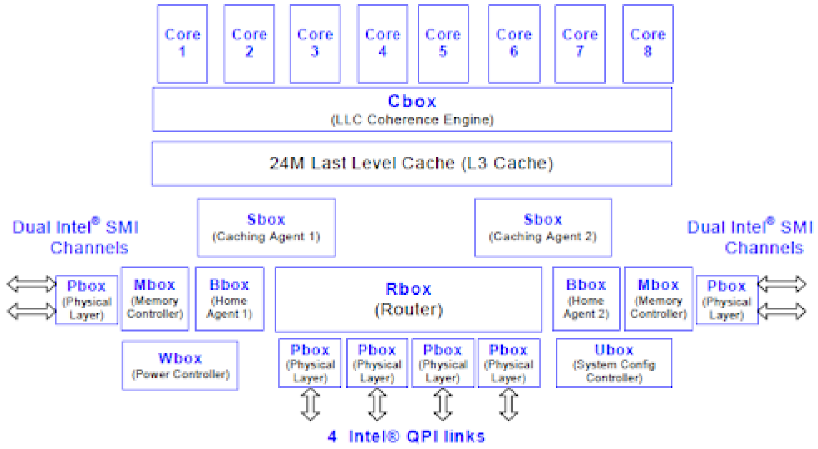
2933 MT/s x 64bit bus width x 6 memory channels = 140.78 GB/s

Performance Monitoring Unit (PMU)

The CPU supports you with the PMUs! A PMU usually support a large number of events (cycles, instructions retired, etc.) through Performance Monitoring Counters (PMC).

A PMU can be:

- **On-core**: this PMU instrument the microarchitecture of the CPU reporting events at core level (instructions retired, cycles, front-end/back-end metrics, L1/L2 cache information, etc.)
- **Off-core:** this PMU handle performance counters outside of cores (LLC, on-chip interconnect, DRAM, Power, etc.). Beware, usually off-core performance counters need special privileges to be read (security issues).



PMCs can be:

- **Fixed**: can be only enabled or disable and profile a specific event (e.g. cycles, instructions retired, etc.)
- **Configurable**: usually can monitor a large number of events (cache hit/miss, branch taken, FLOPs, etc.)

PMCs are usually accessible only from the kernelspace, often the CPU support the reading of the PMU through specific userspace assembly instructions with low overhead (see *rdpmc()*)

Common performance metrics

- **Cycles**: count the number of cycles
- **instructions retired**: count the number of macro instructions executed
- **µops retired**: count the number of micro instructions executed
- **branches**: count the number of branch taken
- **vector instructions retired**: count the number of vector macro instructions
- **instructions per cycle (IPC)**: count the number of macro instructions executed each cycle -> this metric show the macro instruction throughput of the microarchitecture
- **µops per cycle**: count the number of micro instructions executed each cycle -> it show the micro instruction throughput of the microarchitecture
- **branch-misses**: branches that has not been correctly predicted from the branch prediction unit (BPU) -> at every branch miss the pipeline is wiped out
- **cache references** (at multiple cache levels): increment this PMC every time a cache has been queried to request a cache line
- **cache miss/hit** (at multiple cache levels): count the miss/hit of the cache references -> it show the locality of the code
- **vectorization ratio**: show the ratio between all the macro instructions retired with respect to the vector macro instructions -> high performance codes are highly vectorized
- **FLOP/s**: count the arithmetic operations executed of the macro instructions (no bitwise, move, load/store operations)
- **power/energy**: count the joule used by package/core/dram of the CPU -> it used to calculate the energy efficiency (Green500)
- **memory throughput**: count the number of cache line read/write from/to the main memory

Micro architecture performance optimization

In modern processor it is very difficult to understand if my application is efficiently performing on a specific system!

We need HW support from the processor -> <u>Only what is measurable can be improved!</u>

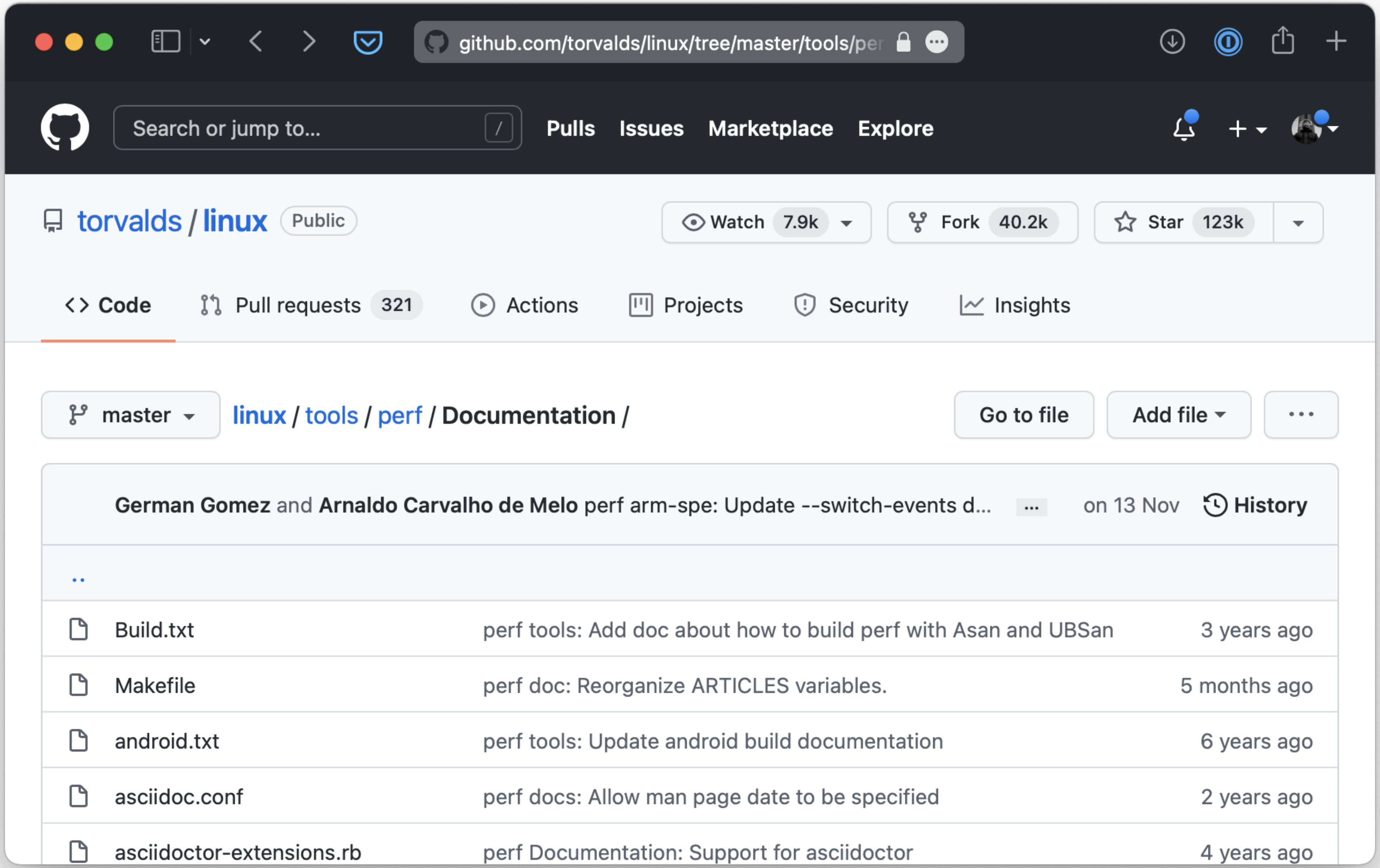
Tools help you to get access to the HW subsystem and automatize routines but...

<u>Use your brain! Tools may help, but you do the thinking</u>

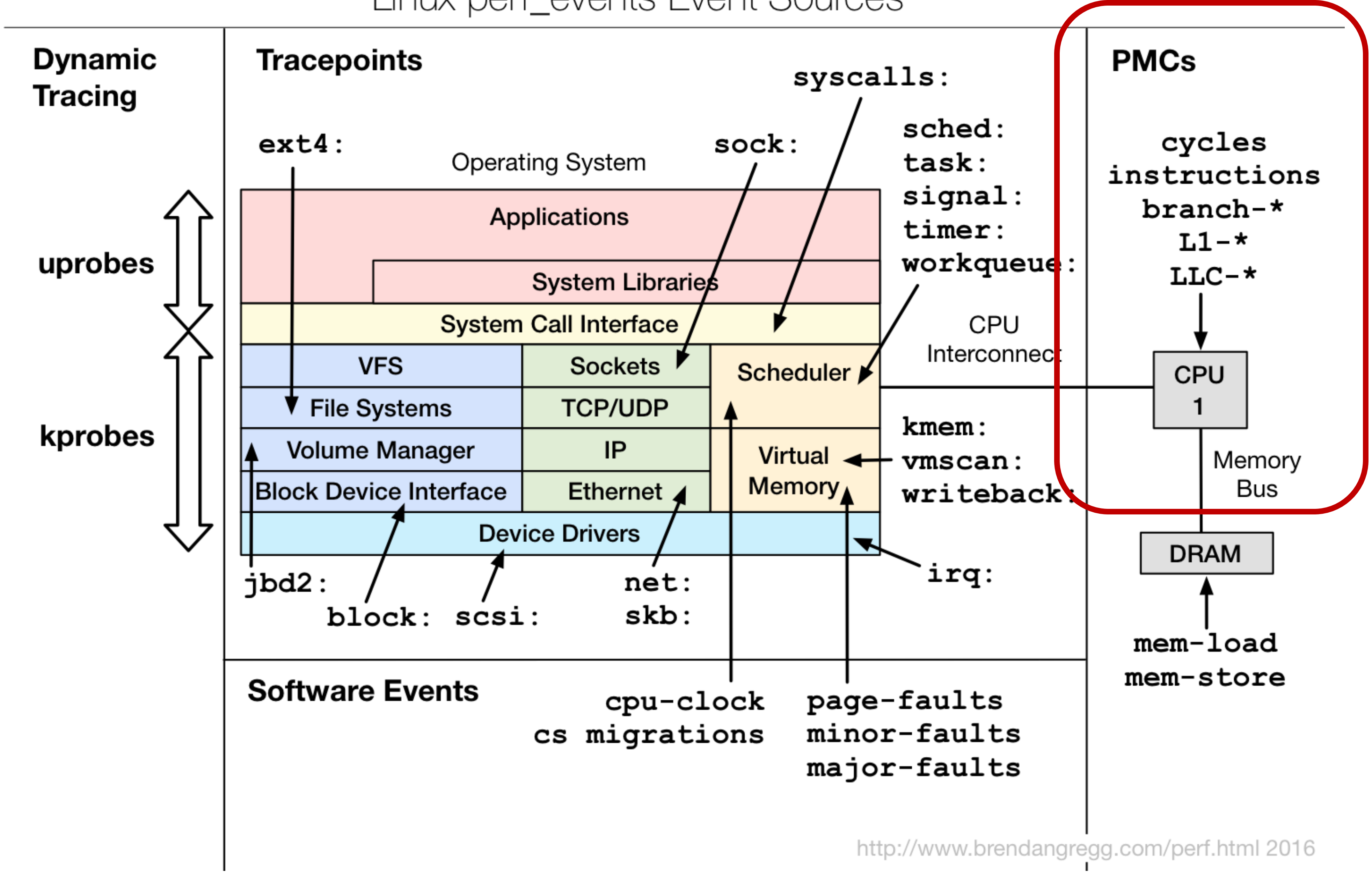
enter `perf`

- *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).


github.com/torvalds/linux/tree/master/tools/per
Search or jump to...
Pulls
Issues
Marketplace
Explore
torvalds / linux
Public
Watch 7.9k
Fork 40.2k
Star 123k
Code
Pull requests 321
Actions
Projects
Security
Insights
master
linux / tools / perf / Documentation /
Go to file
Add file
German Gomez and Arnaldo Carvalho de Melo perf arm-spe: Update --switch-events d...
on 13 Nov
History
..
Build.txt
perf tools: Add doc about how to build perf with Asan and UBSan
3 years ago
Makefile
perf doc: Reorganize ARTICLES variables.
5 months ago
android.txt
perf tools: Update android build documentation
6 years ago
asciidoc.conf
perf docs: Allow man page date to be specified
2 years ago
asciidoctor-extensions.rb
perf Documentation: Support for asciidoctor
4 years ago

Linux perf_events Event Sources



```
$ perf

usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
  annotate        Read perf.data (created by perf record) and display annotated code
  archive         Create archive with object files with build-ids found in perf.data file
  bench           General framework for benchmark suites
  buildid-cache   Manage build-id cache.
  buildid-list    List the buildids in a perf.data file
  c2c             Shared Data C2C/HITM Analyzer.
  config          Get and set variables in a configuration file.
  data            Data file related processing
  diff            Read perf.data files and display the differential profile
  evlist          List the event names in a perf.data file
  ftrace          simple wrapper for kernel's ftrace functionality
  inject          Filter to augment the events stream with additional information
  kallsyms        Searches running kernel for symbols
  kmem            Tool to trace/measure kernel memory properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  mem             Profile memory accesses
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.
  version         display the version of perf binary
  probe           Define new dynamic tracepoints
  trace           strace inspired tool

See 'perf help COMMAND' for more information on a specific command.
```

perf event sources

$ perf list

alignment-faults [**Software event**]

[...]

branch-instructions OR branches [**Hardware event**]

[...]

L1-dcache-load-misses [**Hardware cache event**]

[...]

L1-dcache-loads OR cpu/L1-dcache-loads/ [**Kernel PMU event**]

[...]

rNNN (see 'perf list --help' on how to encode it) [**Raw hardware event descriptor**]

[...]

sched:sched_stat_runtime [**Tracepoint event**]

pure kernel counters

abstract counters, available everywhere

raw PMCs, microarch specific

custom ftrace tracepoints

`perf stat/record` format

```
$ perf record –a –F 4000 -e L1-dcache-load-misses,L1-dcache-loads -- sleep 10
```

events
sampling frequency, which events we want to measure

scope
which sources we want to take into account

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

`perf` *basic* workflow

```
$ perf stat –d <cmd> # identify some interesting event

$ perf list L1-dcache* # look for it and related

List of pre-defined events (to be used in -e):

  L1-dcache-load-misses                    [Hardware cache event]
  L1-dcache-loads                          [Hardware cache event]
  [...]

$ perf record -e L1-dcache-load-misses,L1-dcache-loads -g -- <cmd> # sample

$ perf report –g –M intel # analyze
```

Getting started: building the code

The target code must be built in **release** mode,
a.k.a. as close as possible to a **real world** build

```
$ gcc -march=native -mtune=native -O3 –DNDEBUG \
    –g -fno-omit-frame-pointer -fno-optimize-sibling-calls ...
```

Include **debug** symbols

Make sure to be able to reconstruct
the full call **stack trace**

Getting started: entering the prod system

1. Pick an available UNIX username

   Please remember to write down your name(s), we have to keep track of who's using each UNIX account for legal reasons!

2. Log in to the system

```
$ ssh <user> login.g100.cineca.it
```

3. Clone the examples repo

```
$ git clone https://gitlab.hpc.cineca.it/fficarel/perf-examples.git
```

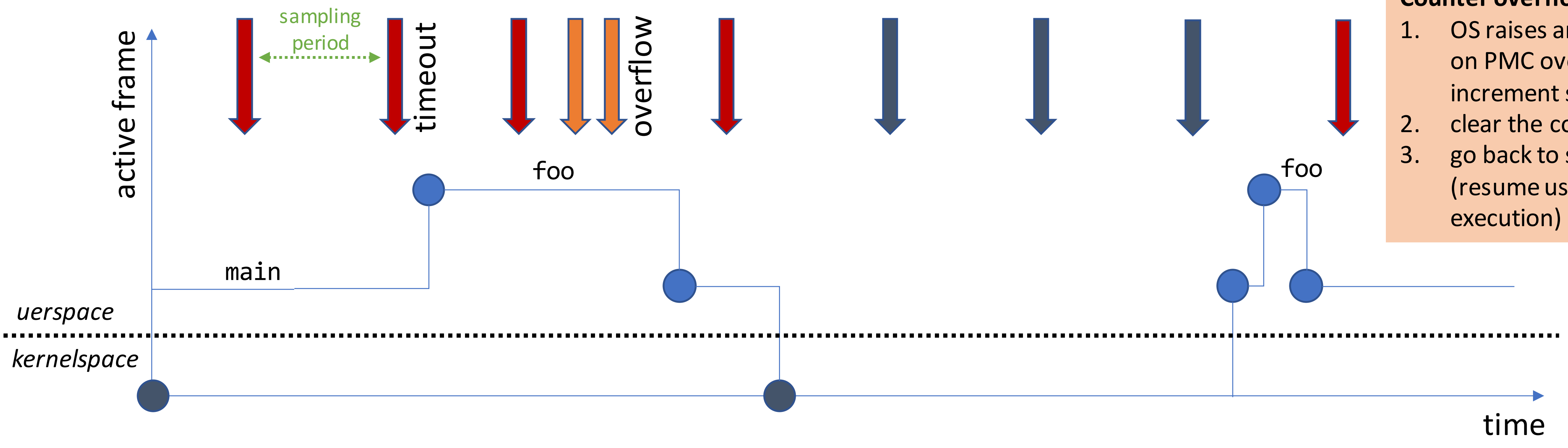
Hands-on: nop

Try analysing the nop.c example:

- Follow the procedure we just went through
- Try to reason on the results
- ***What did you notice? Something’s off?***

event sampling: how does it work?

```
// 1. wake up (interrupt from timeout/overflow)
rt_sigaction(SIGUSR1, {sa_handler=0x556ae166b020, sa_mask=[], sa_flags=SA_RESTORER|SA_SIGINFO, ...
// 2. record events (w/stack trace if asked for)
perf_event_open({type=PERF_TYPE_HW_CACHE, size=0x78 /* PERF_ATTR_SIZE_??? */,
   config=PERF_COUNT_HW_CACHE_L1D|PERF_COUNT_HW_CACHE_OP_READ<<8|PERF_COUNT_HW_CACHE_RESULT_MISS<<16, ...
// 3. if ringbuffer is full, writeout to perf.dat
write(4, "\370\225\0\0\370\225", ...
```



Counter overflow:

1. OS raises an interrupt on PMC overflow increment sw counter
2. clear the counter
3. go back to sleep (resume user execution)

event sampling accuracy

Two main issues with *events to instructions* mapping:

- **skid**
  - can be mitigated with proper hw support (e.g.: Intel PEBS, AMD IBS)
- **sampling latency**
  - this is harder, no definitive solution
  - experiment to find a **sampling frequency/overhead trade off** that works with your workload
  - when all else fails, try amplifying the statistical behaviour of your workload (e.g.: micro-benchmarks)

Hands-on: matmul

Try analysing the matmul.c example:

- Follow the procedure we just went through
- Try to reason on the results
- ***Are you able to draw conclusions?***

drill down: identify bootlenecks with TMAM

- **T**op-down **M**icroarchitecture **A**nalysis **M**ethod

  Yasin, Ahmad. "A Top-Down Method for Performance Analysis and Counters Architecture." In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 35–44. CA, USA: IEEE, 2014. https://doi.org/10.1109/ISPASS.2014.6844459.

- first attempt by Intel at a methodological approach
- **hierarchical drill-down method guided by PMCs**
- **goal: identify the real bottleneck WRT micro-architecture**

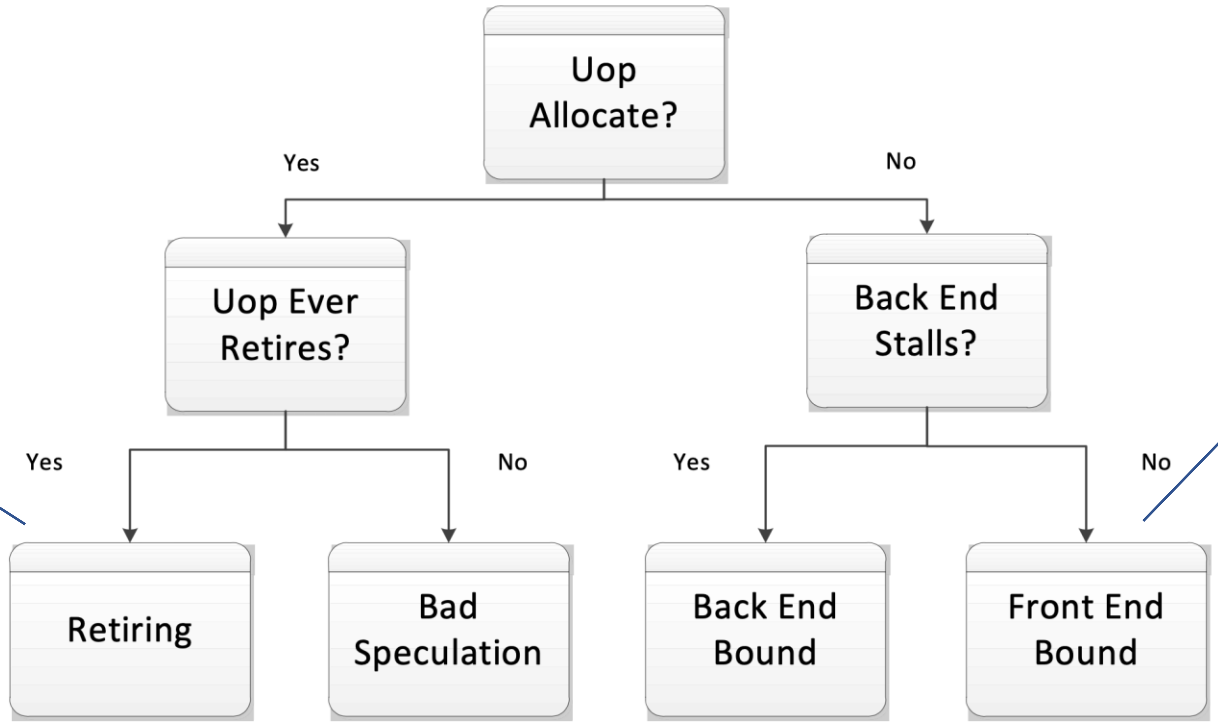
Note: the approach is so widespread that a plethora of support tools are available, e.g.:

https://github.com/andikleen/pmu-tools

It guides you through the PMU jungle needed to drill down on an Intel x86 processor:

```
$ toplev –l1 -v --no-desc taskset -c 0 ...
```

TMAM: high level breakdown



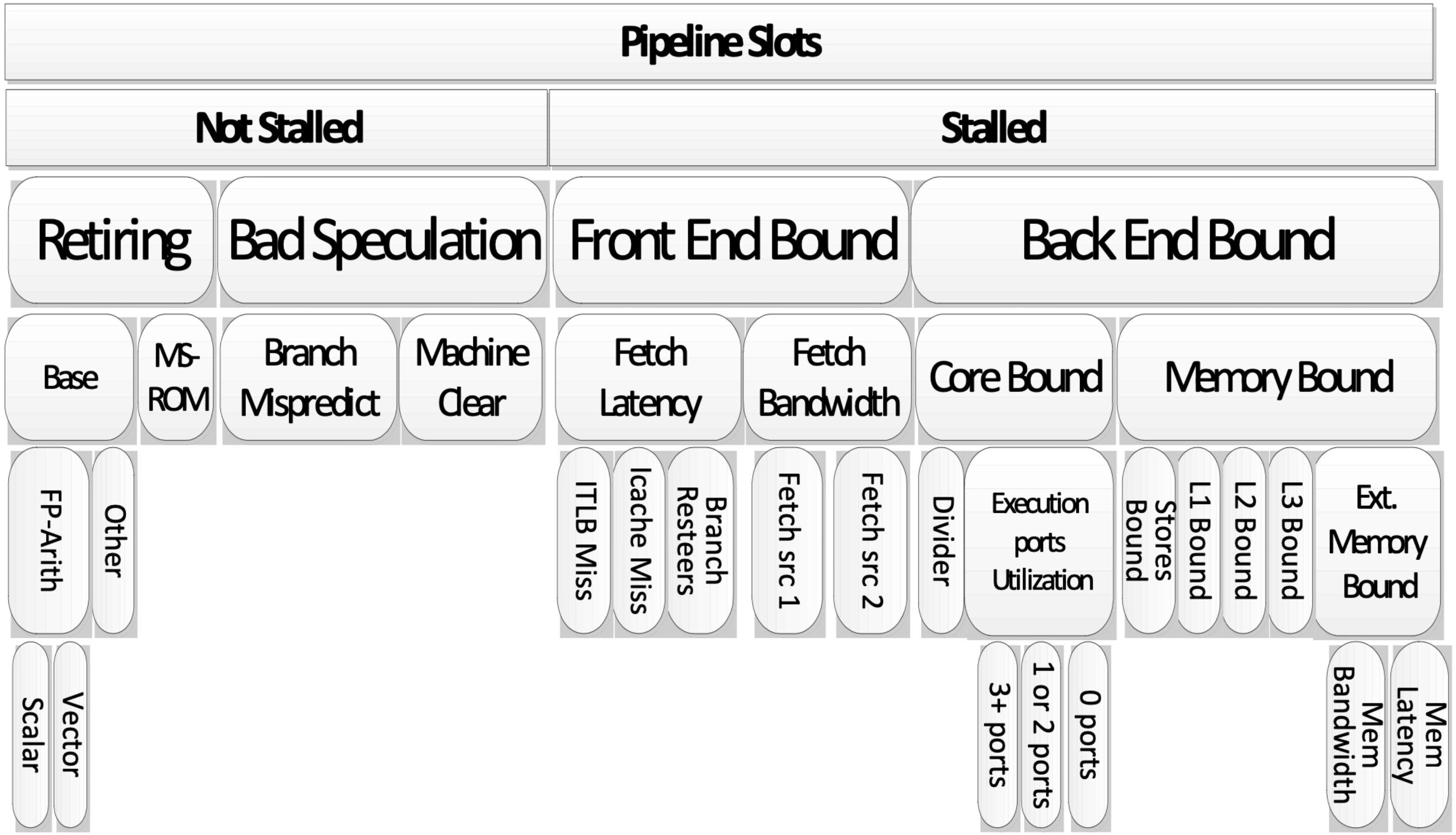
Successfully delivered uops who eventually do retire

The front end is not delivering enough uops per cycle while the back end of the pipeline is ready to accept uops

Tracks uops that never retire or allocation slots wasted due to recovery from branch miss-prediction or clears

No uops are delivered due to lack of required resources at the back end of the pipeline

TMAM: low level breakdown



TMAM: PMCs for high level breakdown

Intel SkylakeX

PMC	Description
UOPS_ISSUED.ANY	Number of µOPs issued by the Resource Allocation Table (RAT) to the Reservation Station (RS)
UOPS_RETIRED.RETIRE_SLOTS	Number of retirement slots used
IDQ_UOPS_NOT_DELIVERED.CORE	Number of µOPs not delivered to the RAT per thread
INT_MISC.RECOVERY_CYCLES	Number of the core cycles the allocator was stalled due to recovery from earlier clear events for any thread running on the physical core (e.g. misprediction)

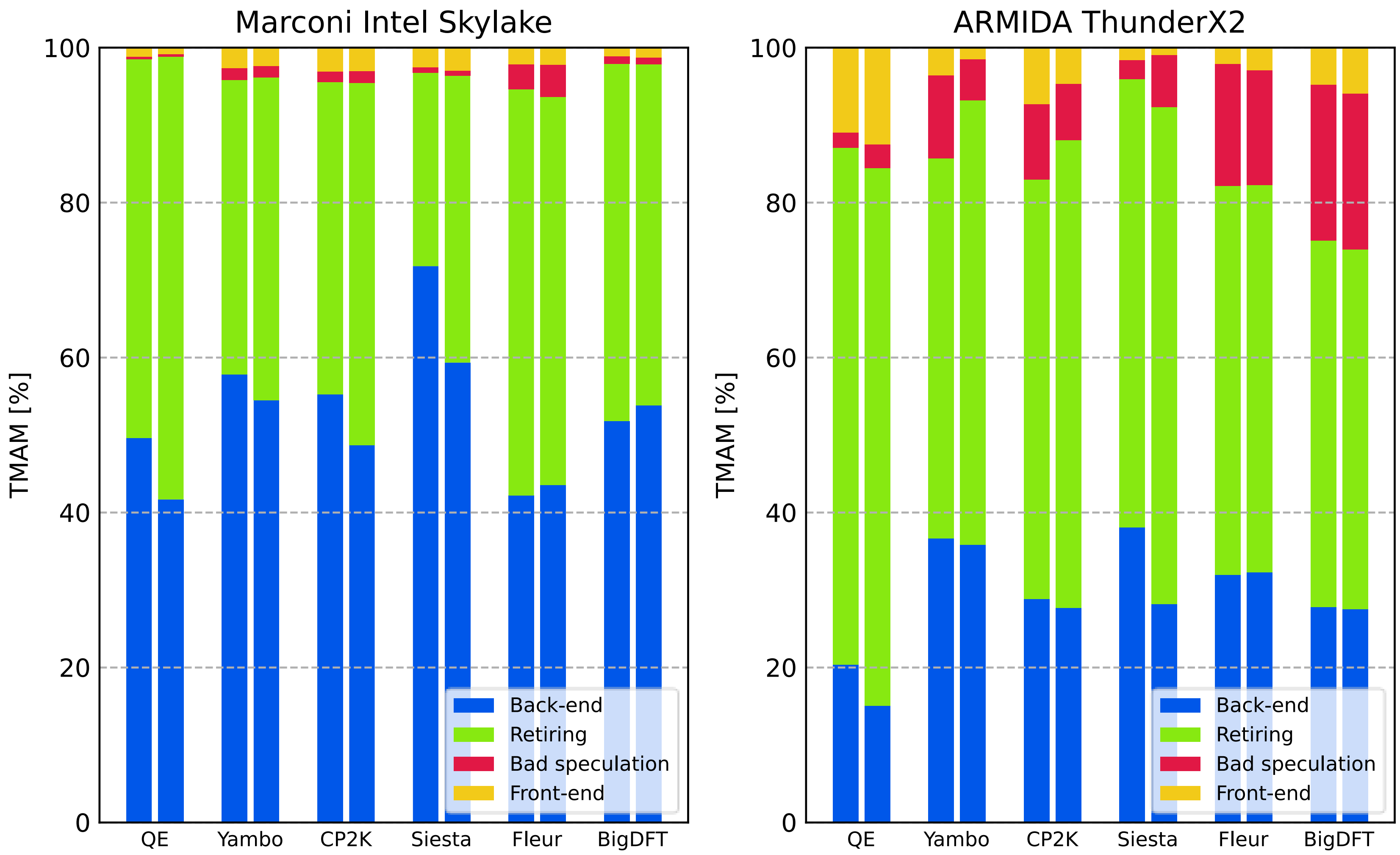
Cavium ThunderX2

PMC	Description
STALL_FRONTEND	Cycle on which no instructions issued because there were no instructions ready to issue
STALL_BACKEND	Cycle on which no instructions issued due to back-end resources being unavailable
INST_SPEC	Instructions speculatively executed
INST_RETIRED	Number of instructions executed
CPU_CYCLES	Count core clock cycles

Metric	Intel SkylakeX	Cavium ThunderX2
Retiring	UOPS_RETIRED.RETIRE_SLOTS / (4 * CPU_CLK_UNHALTED.THREAD)	1 - (Front end + Back end + Bad Speculation)
Front end	IDQ_UOPS_NOT_DELIVERED.CORE / (4 * CPU_CLK_UNHALTED.THREAD)	STALL_FRONTEND / CPU_CYCLES
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4 * INT_MISC.RECOVERY_CYCLES) / (4 * CPU_CLK_UNHALTED.THREAD)	((INST_SPEC - INST_RETIRED) / AVG IPC) / CPU_CYCLES
Back end	1 - (Retiring + Front End + Bad Speculation)	STALL_BACKEND / CPU_CYCLES

Intel SkylakeX PMU: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
Cavium ThunderX2 PMU: https://www.marvell.com/content/dam/marvell/en/public-collateral/server-processors/marvell-server-processors-thunderx2-cn99xx-pmu-events-white-paper-2019-11.pdf
IBM Power9: https://ibm.ent.box.com/s/8kh0orsr8sg32zb6zmq1d7zz6hud3f8j

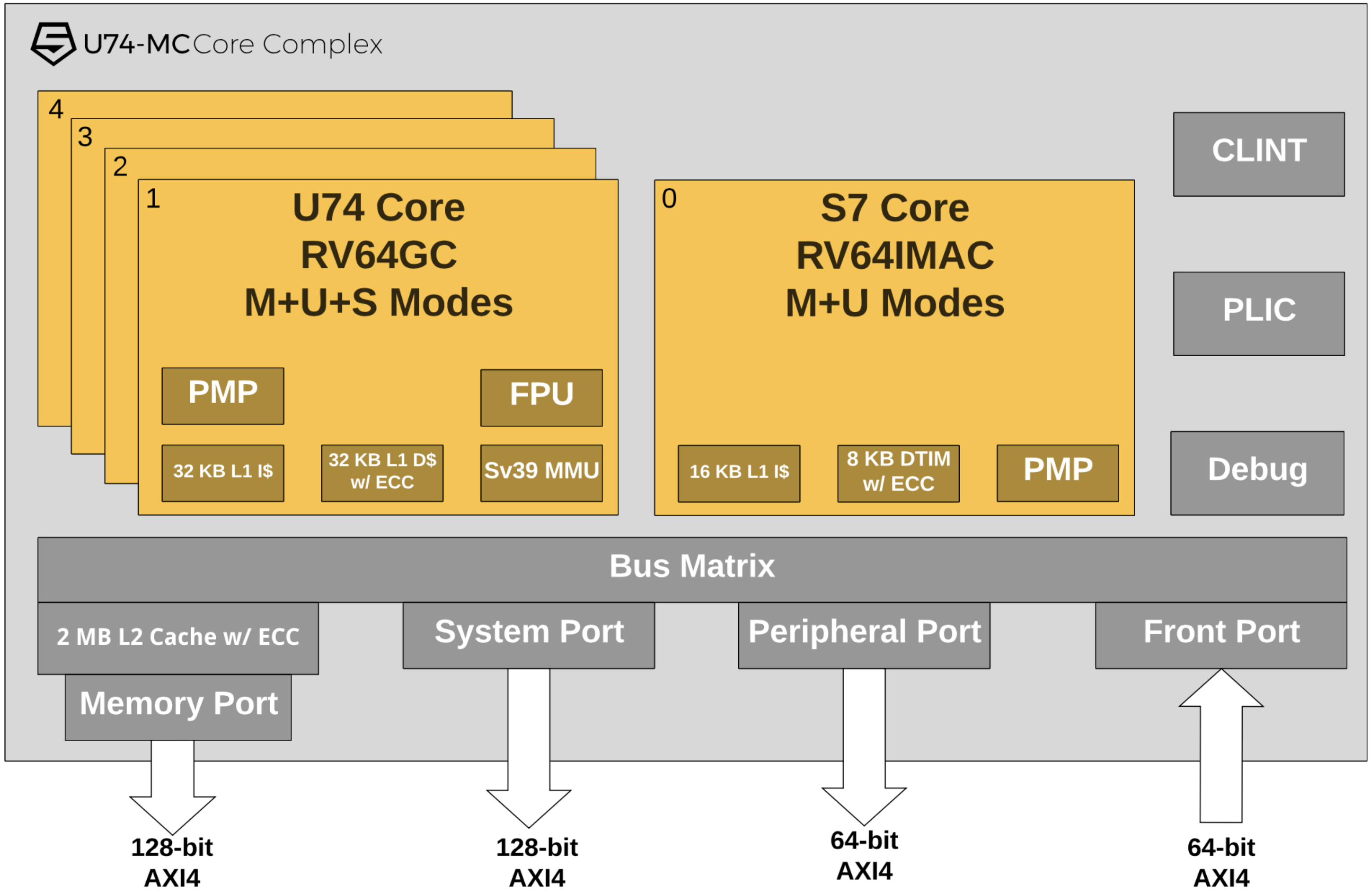
TMAM: identify the bottleneck



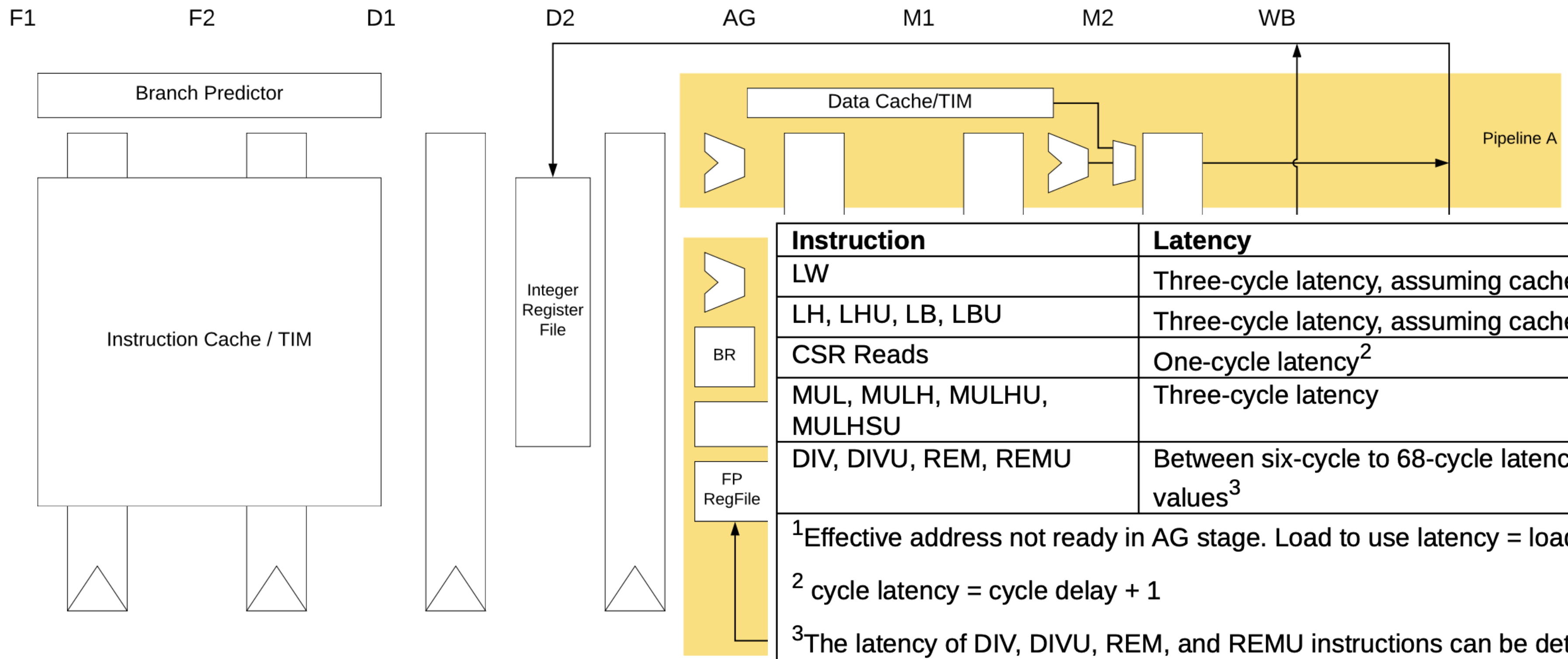
TMAM: what about non-x86?

- The TMAM approach is conceptually agnostic
- Problem: it is usually tailored for Intel x86_64 µarch
- It’s possible to adapt it to any arch/µarch
- Deep µarch knowledge is needed to
  - Breaking it down in meaningful, finite resource blocks
  - Model the memory hierarchy
  - Model the instruction pipeline

SiFive U74: cluster overview



SiFive U74: instruction pipeline



Instruction	Latency
LW	Three-cycle latency, assuming cache hit[1]
LH, LHU, LB, LBU	Three-cycle latency, assuming cache hit[1]
CSR Reads	One-cycle latency[2]
MUL, MULH, MULHU, MULHSU	Three-cycle latency
DIV, DIVU, REM, REMU	Between six-cycle to 68-cycle latency, depending on operand values[3]

[1]Effective address not ready in AG stage. Load to use latency = load to use delay + 1

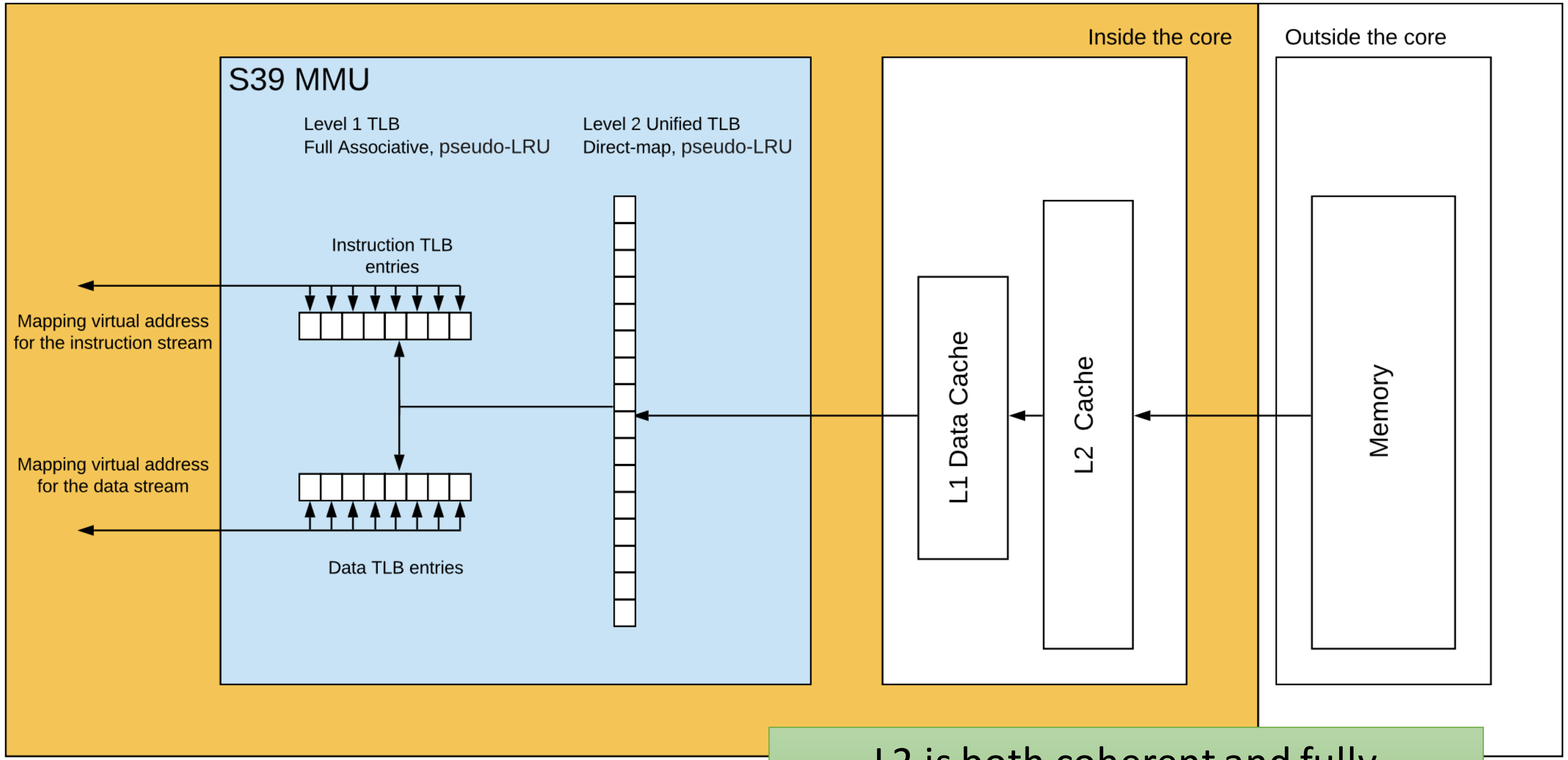
[2] cycle latency = cycle delay + 1

[3]The latency of DIV, DIVU, REM, and REMU instructions can be determined by calculating:
$\texttt{Latency} = 2\ \texttt{cycles} + \log_2(\texttt{dividend}) - \log_2(\texttt{divisor}) + 1\ \texttt{cycle}$
if the input is negative + 1 cycle if the output is negative

Dual instruction issue, iff:

- At most one instruction accesses data memory.
- At most one instruction is a branch or jump.
- At most one instruction is an integer multiplication or division operation.
- Neither instruction explicitly accesses a CSR.

SiFive U74: memory hierarchy



L2 is both coherent and fully prefetched (8x streams per core)

Hands-on: stream

Try analysing the stream.c example:

- Follow the usual profiling procedure
- Try to reason on the results
- ***Are you able to draw conclusions about the workload?***

Hands-on (bonus): matmul

Try analysing the matmul.c example:

- Follow the usual profiling procedure
- Try to reason on the results
- ***Are you able to draw conclusions about the workload? Are you able to pinpoint the hotspot?***

pointers

- `man perf`
- kernel wiki [https://perf.wiki.kernel.org]
- Brendan Gregg [https://www.brendangregg.com]
- EasyPerf [https://easyperf.net]
- kernel tree [https://github.com/torvalds/linux]
- relevant bibliography [https://git.io/JDXMV]

bonus: programming Intel AVX512

```
__m512 _mm512_load_ps (void const* mem_addr)
```

```
__m{vlen}      _mm{vlen}_                 _ps
__m{vlen}i     _mm_          Operation    _pd
__m{vlen}d                                _epi{ilen}
```

- Intel Intrinsics Guide https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html