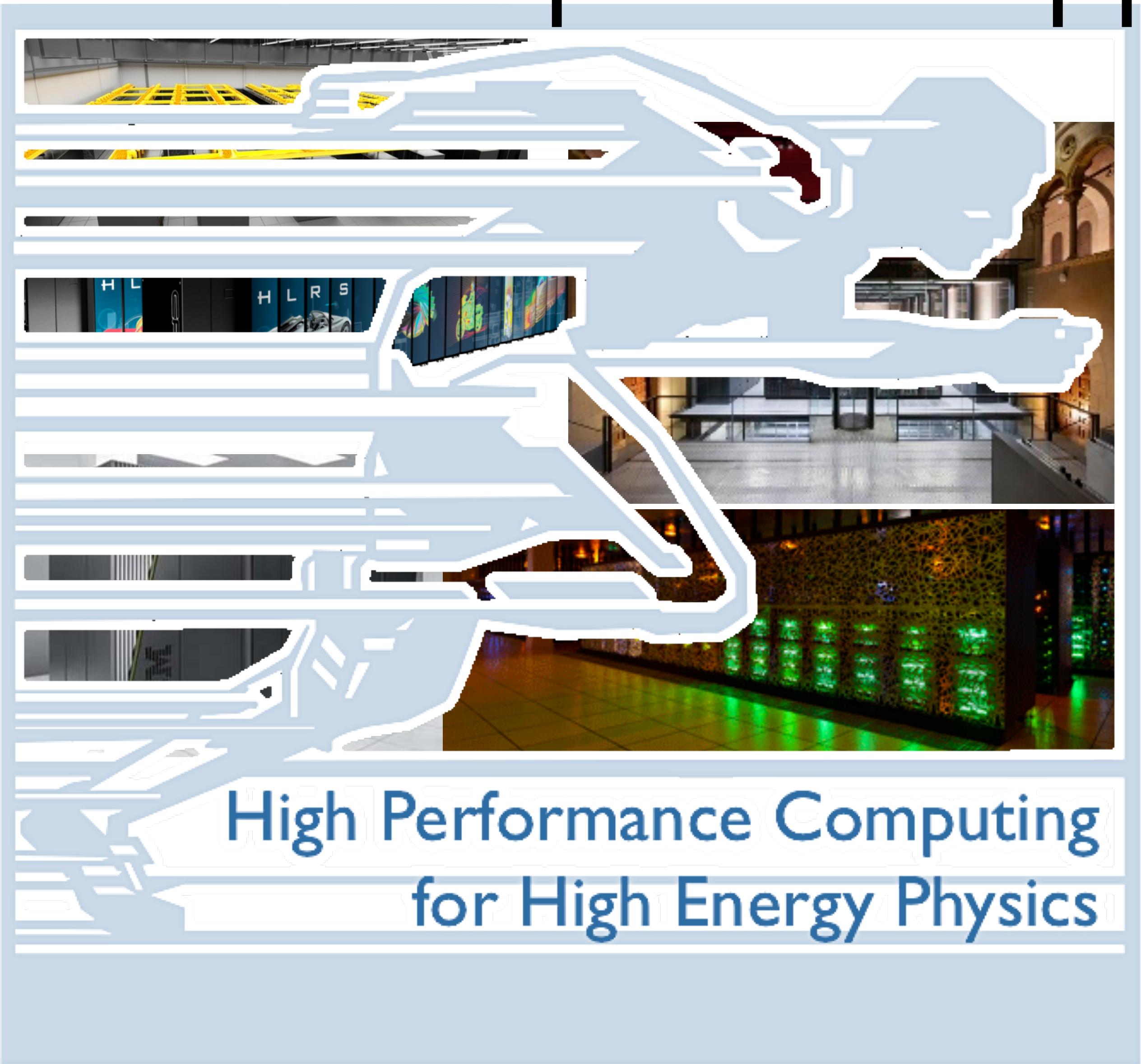
Computing Architecture and its Impact on Application Performance

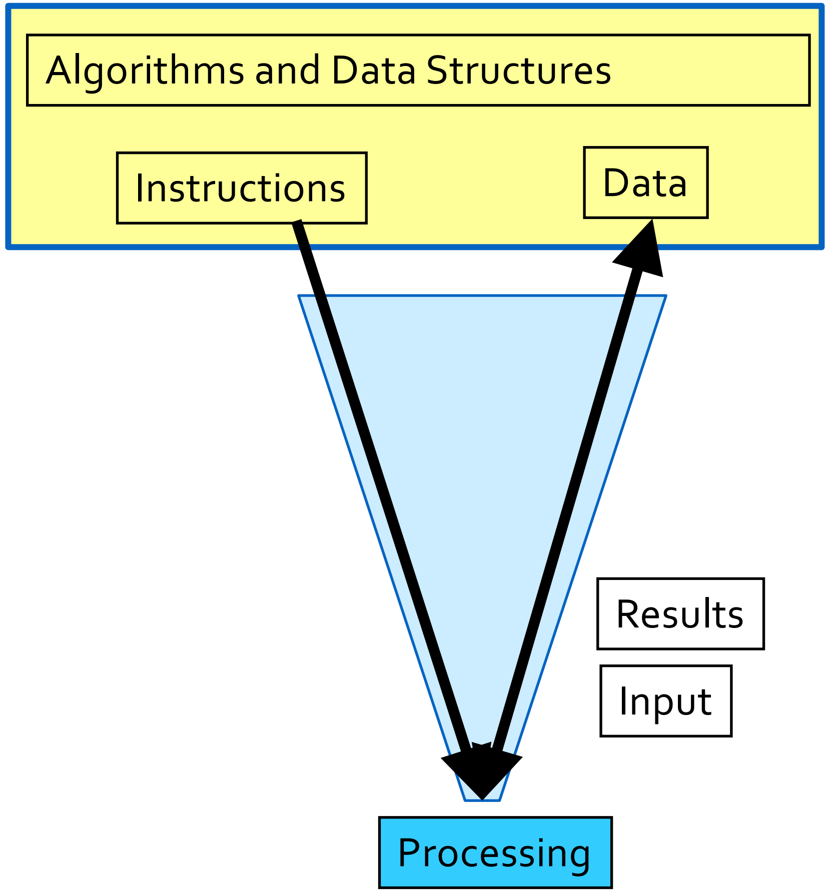


Vincenzo Innocente
CERN
CMS Experiment

ESC, Bertinoro, October 2019

Computing Architecture

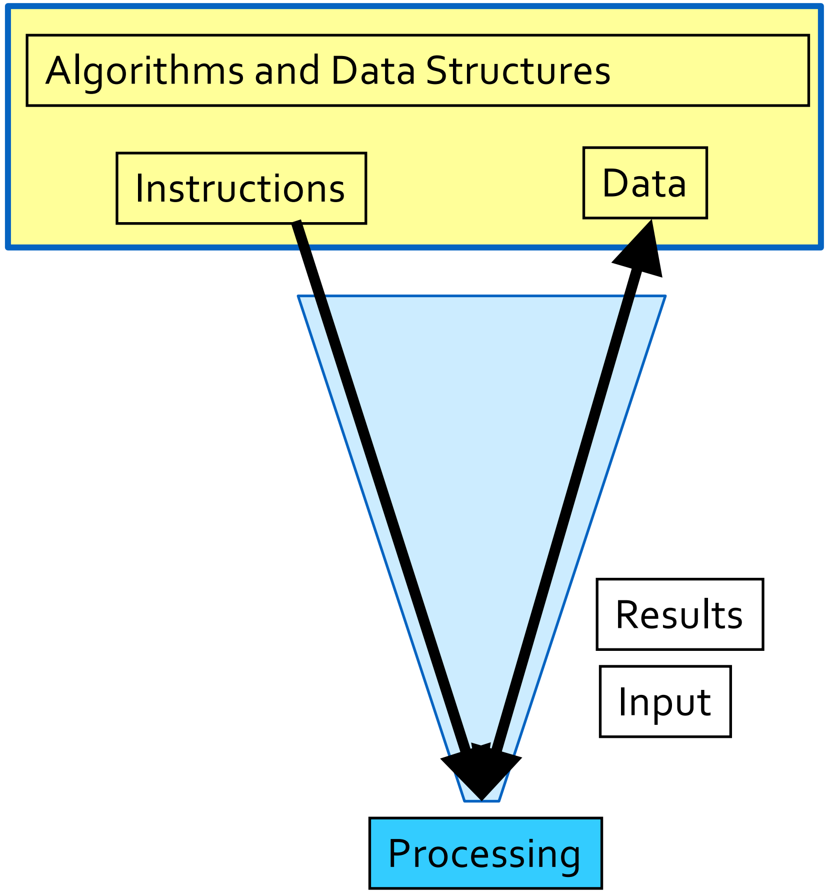
Von Neumann architecture



- 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

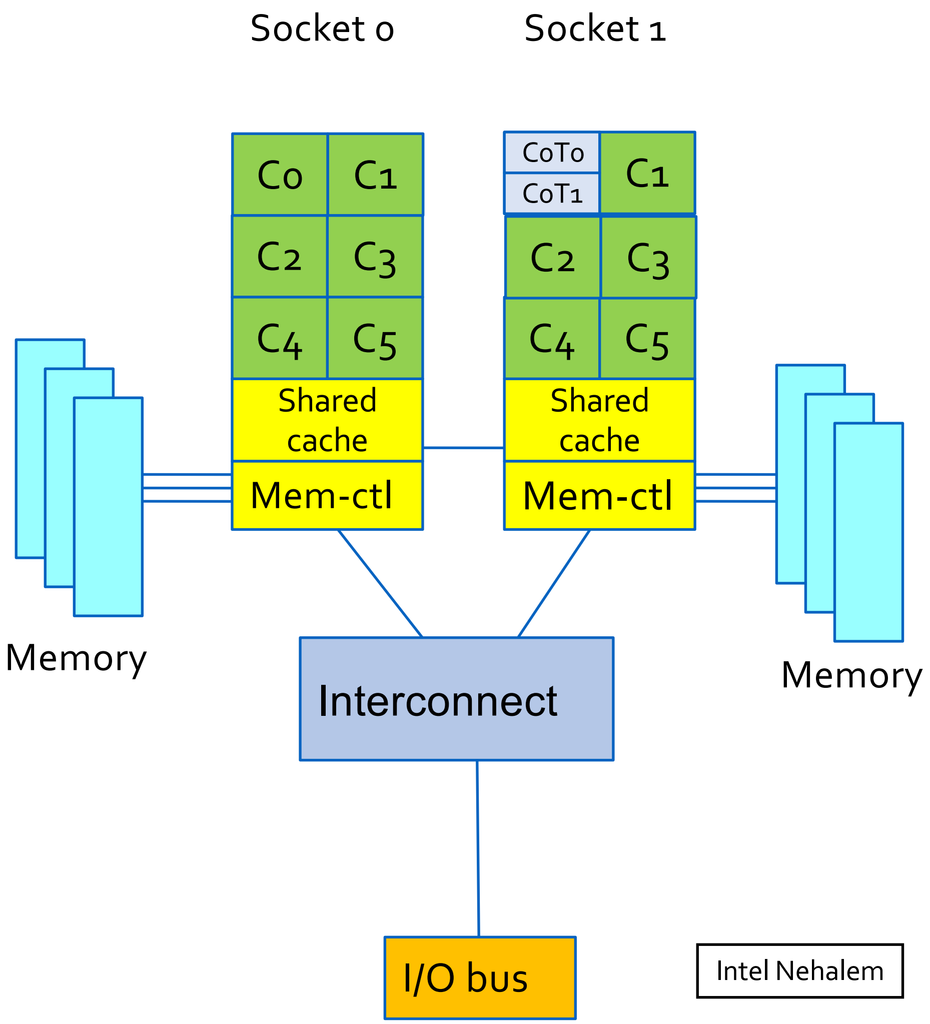
Von Neumann architecture



- Metrics:
  - Instructions/second (MIPS)
  - Operations/second (FLOPS)
  - Latency
    - How long takes to finish a job
  - Throughput
    - The amount of items processed per unit of time (or dollar, watts)
- How to speed it up?
- How to scale it up?
  - How can I do more (with less)

Modern “server” architectures

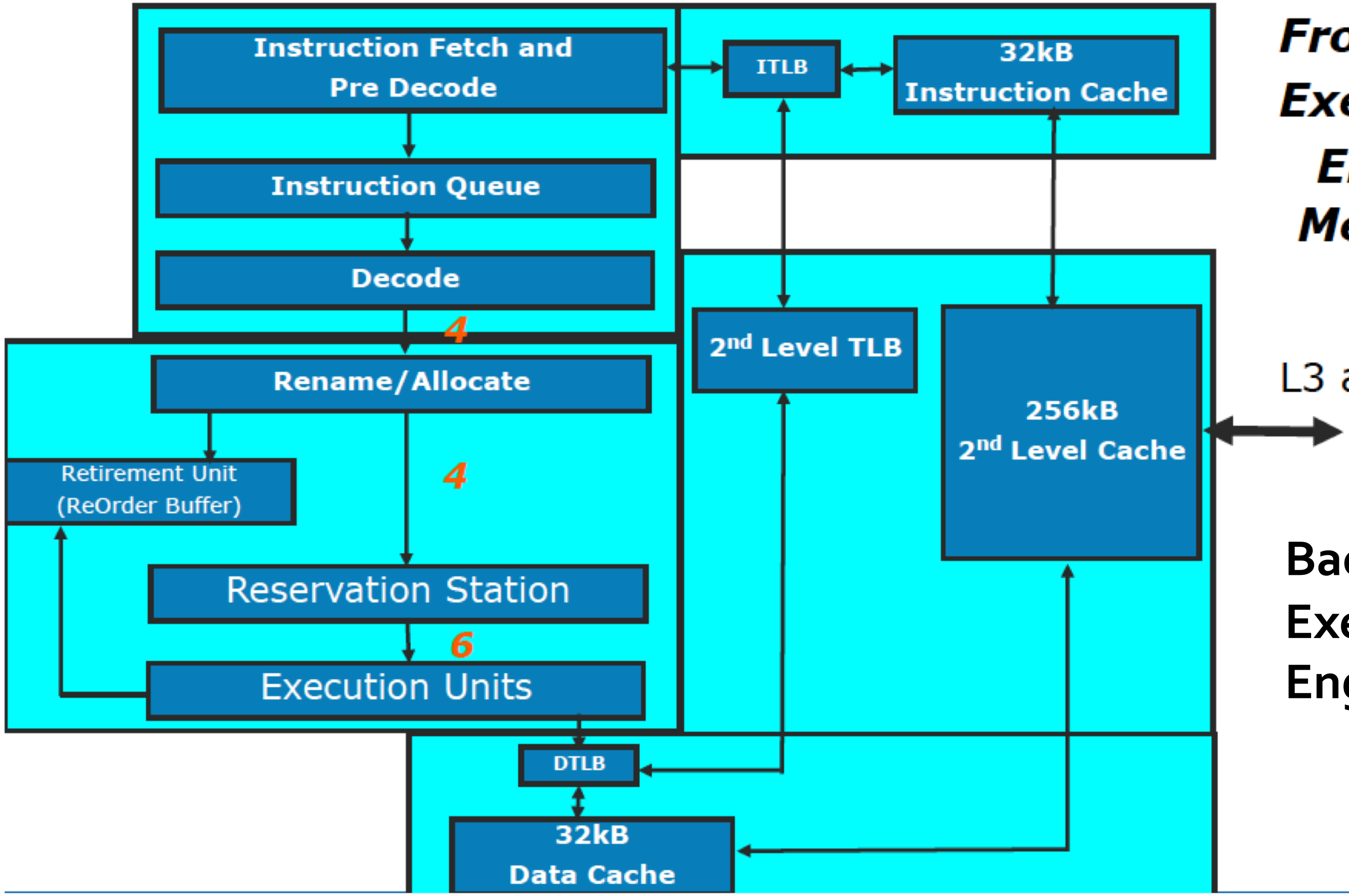
Simple server diagram



- Multiple components which interact during the execution of a program:
  - Processors/cores
    - w/private caches
      - I-cache, D-cache
  - Shared caches
    - Instructions and Data
  - Memory controllers
  - Memory (non-uniform)
  - I/O subsystem
    - Network attachment
    - Disk subsystem

Single Core Architecture

Enhanced Processor Core



Front End Execution Engine Memory

L3 and beyond

Back End Execution Engine

Interlude: performance counters

Performance Metrics

- All modern processors are instrumented with “performance counters” that measure essentially everything that is happening
- Unfortunately there is no standard: each new processor usually comes with a whole lot of new counters with new names…
- Here I try to use those of the two types of processors we use for exercise: Intel Ivy-Bridge and Skylake-X
- Tool exists to abstract counters into a sort of standard architecture
  - On Linux: perf and its wrappers
- Vendors (Intel, AMD, IBM, NVidia) provide also their own tools

Architecture: front end

Feeds "decoded" instructions to the scheduler

Affected by instruction non-locality (iCache-miss, iTLB misses) and misspredicted branches

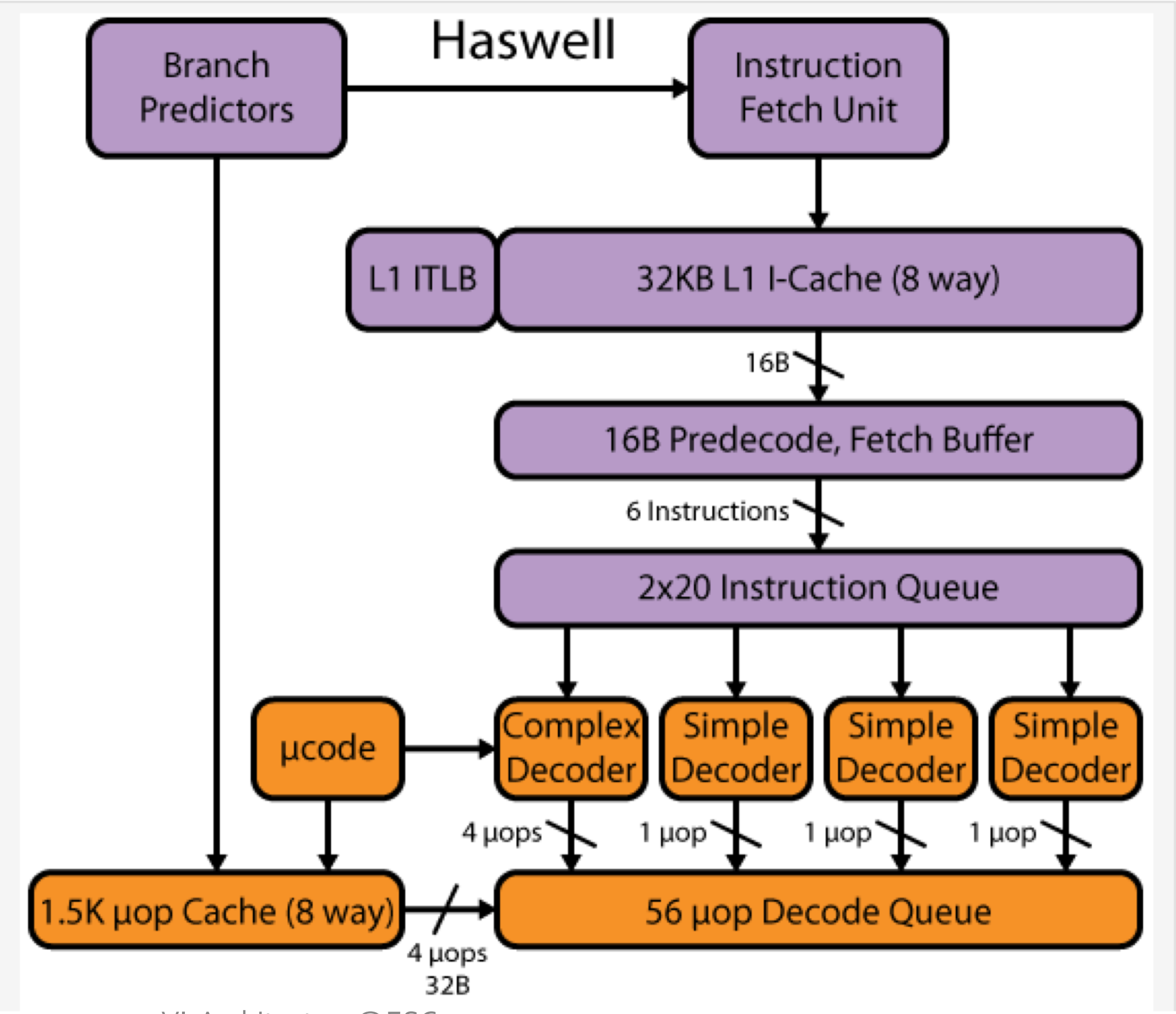
Main metrics:

L1-icache-load-misses (icache.ifdata_stall)

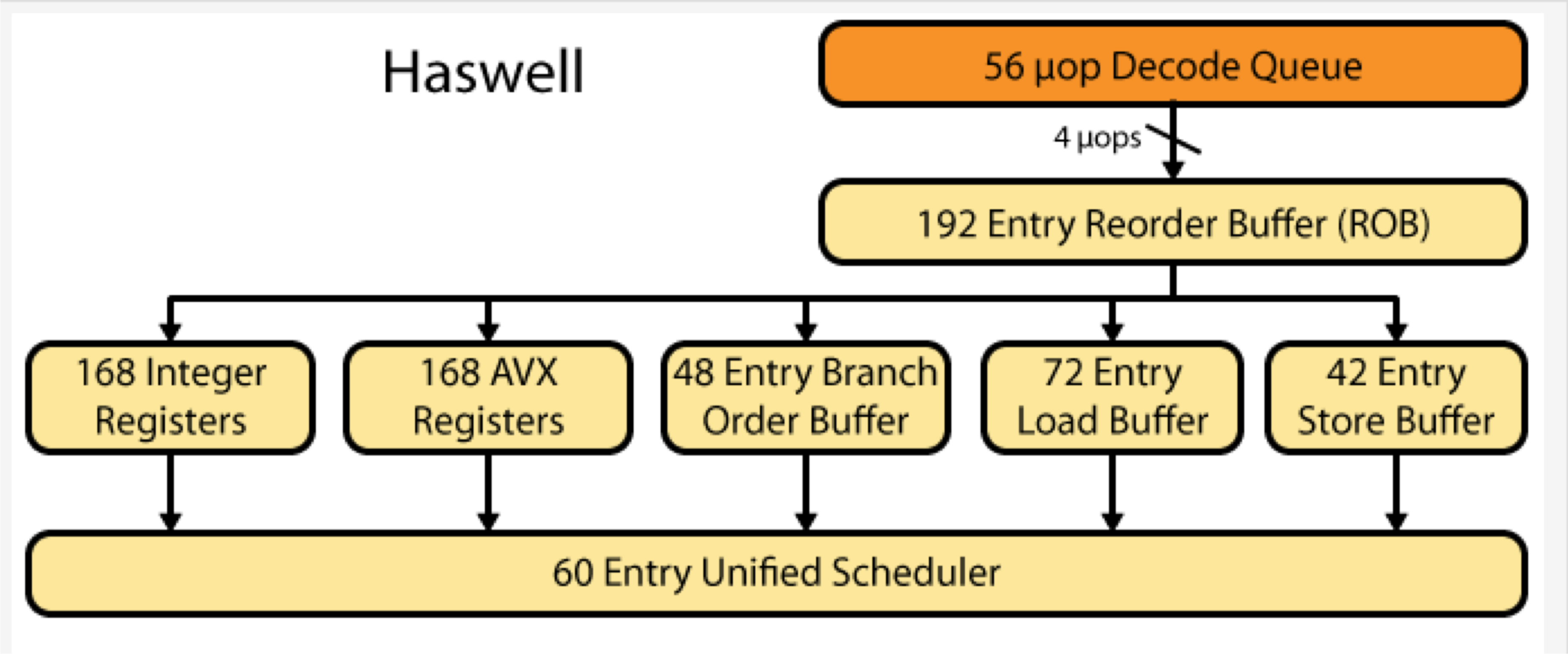
Cycles where a code fetch is stalled due to L1 instruction cache miss.

branch-misses (br_misp_retired.all_branches)

This event counts all mispredicted branch instructions retired.



Architecture: Out of order scheduler



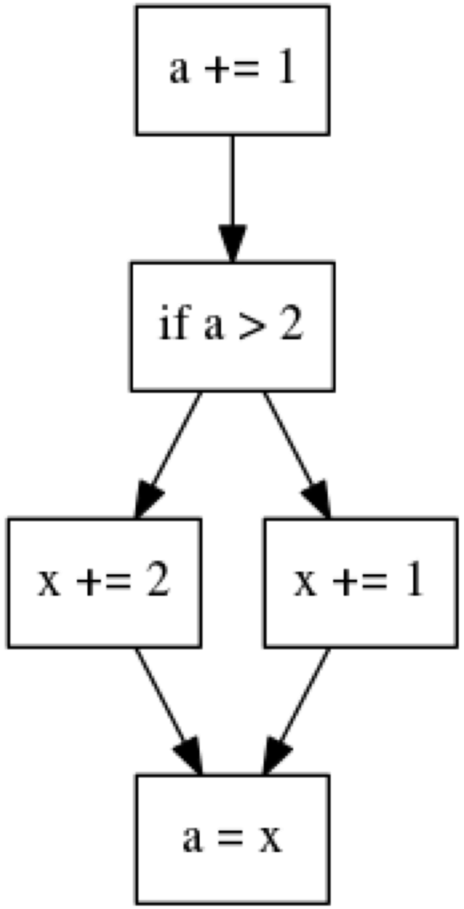
Main metric:

rs_events.empty_cycles

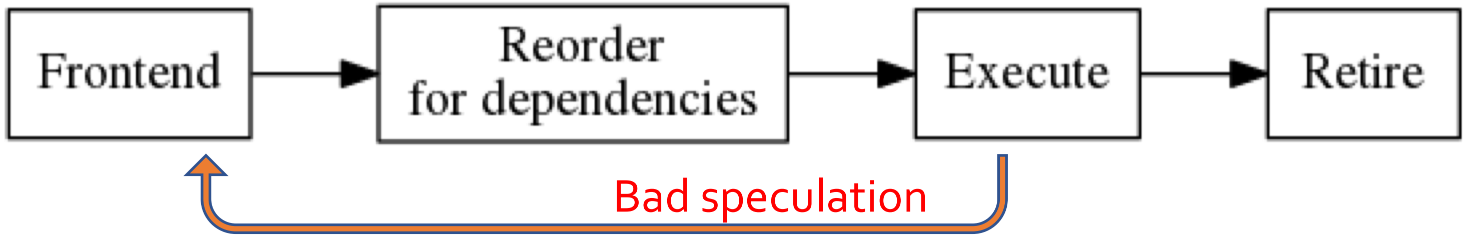
This event counts cycles during which the reservation station (RS) is empty

RS == Unified scheduler

Out-of-order (OOO) scheduling



- Most modern processors use OOO scheduling
  - This means that they will speculatively execute instructions ahead of time (Xeon: inside a “window” of ~150 instructions)
  - In certain cases the results of such executed instructions must be discarded



- At the end, there is a difference between “executed instructions” and “retired instructions”
  - One typical reason for this is mispredicted branches
  - (compiler or developers can also transform the code to be branchless)

Potential problem with OOO:

A lot of extra energy is needed!

Interestingly: ARM has two designs:

A53 (low power, in-order), A57 (high power, OOO)

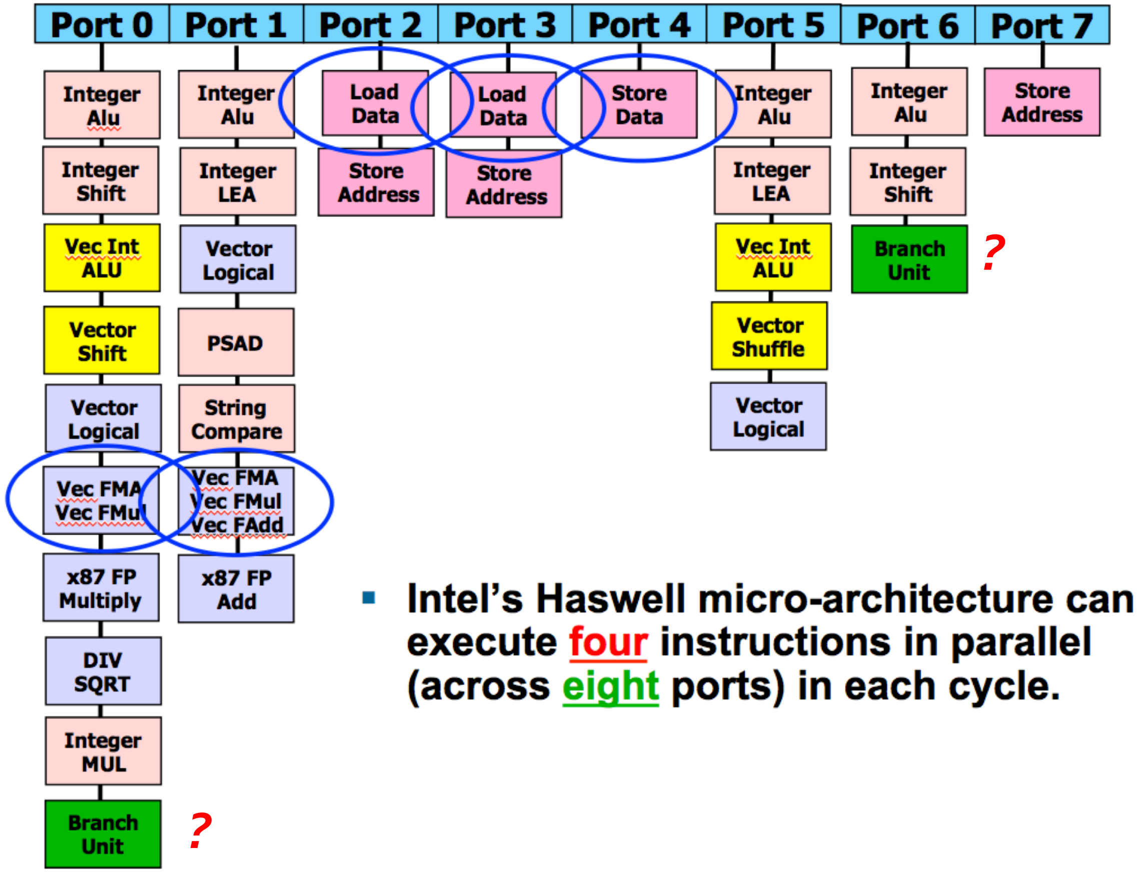
How to help the frontend

- Avoid complex branching patterns
- Keep code local (inline)
- Keep loop short (so they fit in μop cache)

Architecture: Backend

Computational engine
Affected by

- instruction dependency
  - instruction parallelism
  - pipelining
- Memory access
- Latency of "heavy instructions"
  - div sqrt
- Vectorization



- **Intel's Haswell micro-architecture can execute four instructions in parallel (across eight ports) in each cycle.**

Main Metrics:

uops_executed.stall_cycles

This event counts cycles during which no uops were dispatched from the Reservation Station (RS)

uops_executed.thread

Number of uops to be executed each cycle.

cycle_activity.stalls_mem_any

Execution stalls while memory subsystem has an outstanding load.

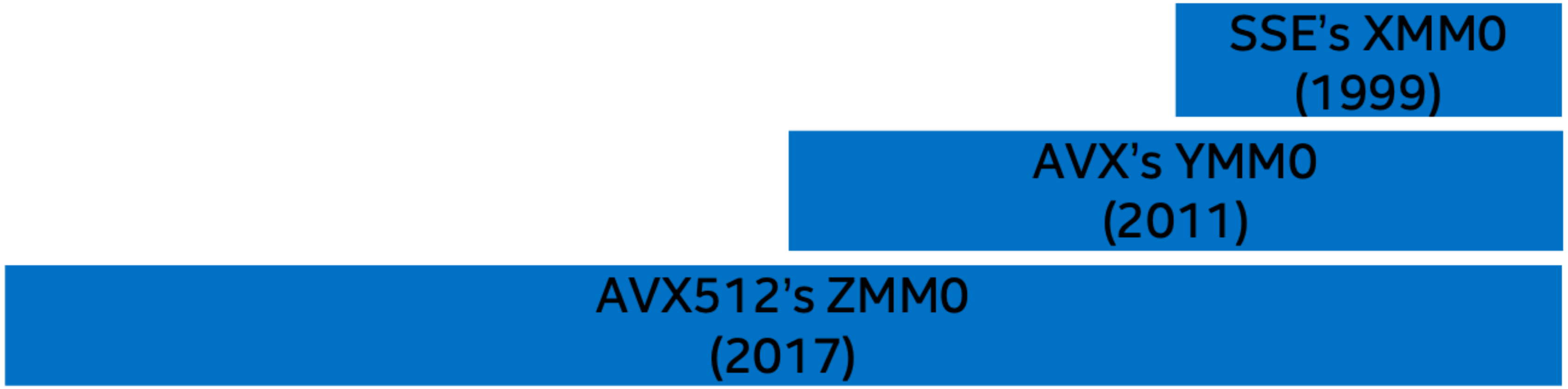
arith.divider_active

Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations.

X86 vectors for Floating Point (FP)

- FP
  - Single Precision (SP)
    - 32-bits
    - 8 elements in AVX2
    - ‘float’ in C
  - Double Precision
    - 64-bits
    - 4 elements in AVX2
    - ‘double’ in C

ISA	Max Vector width	FP-SP elements	Introduced in processor
SSE	128-bit	4	
AVX	256-bit FP only	8	Sandy Bridge (2nd gen Core)
AVX2	256-bit (adds integer, FMA)	8	Haswell (4th gen Core)
AVX512	512-bit	16	Skylake Server (Xeon Scalable)



Real-life latencies

- Most integer/logic instructions have a one-cycle execution latency:
  - For example (on an Intel Xeon processor)
    - ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the 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 double-precision variants
  - Exception: FABS (absolute value): 1
  - Many-cycle, no pipepine : FDIV (20), FSQRT (27)
  - Other math functions: even more

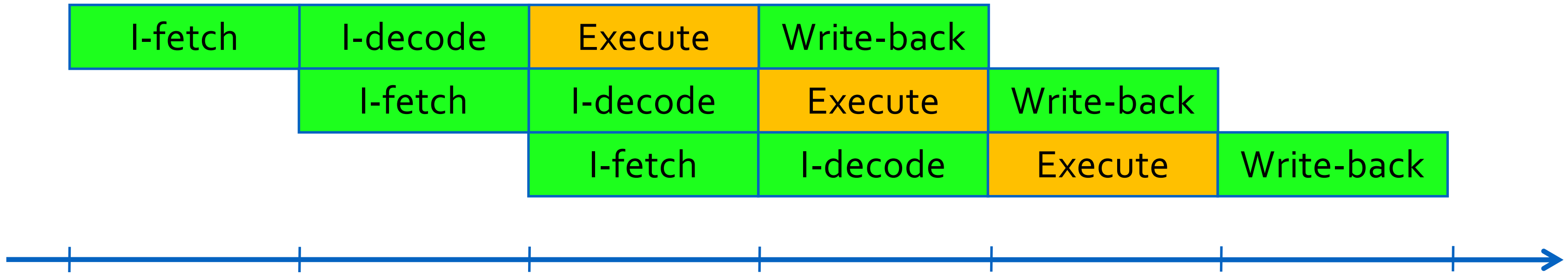
As of Haswell:
FMA (5 cycles)
As of Skylake:
SIMD ADD, MUL,FMA: 4 cycles

Latencies in the Core micro-architecture (Intel Manual No. 248966-026 or later).
AMD processor latencies are similar.

http://www.agner.org/optimize/instruction_tables.pdf

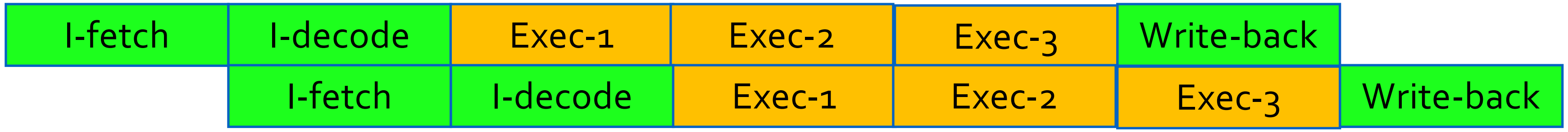
Instruction pipelining

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



I-fetch	I-decode	Execute	Write-back		
	I-fetch	I-decode	Execute	Write-back	
		I-fetch	I-decode	Execute	Write-back

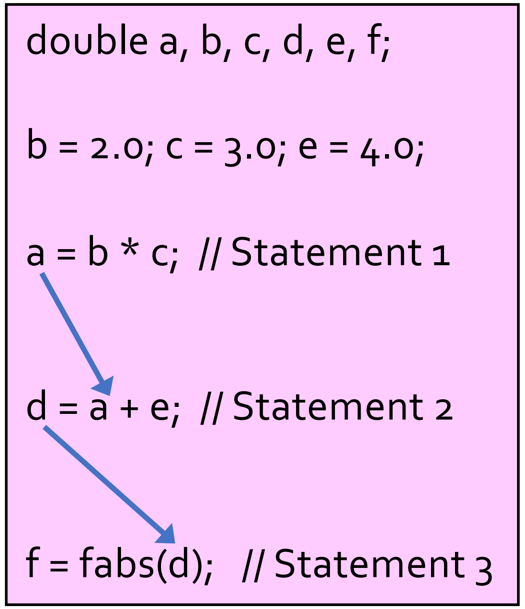
  - With a three-cycle execution latency:



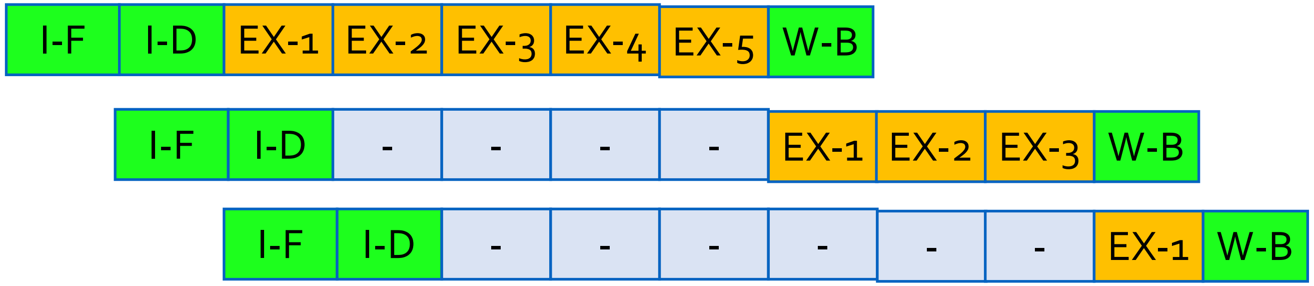
I-fetch	I-decode	Exec-1	Exec-2	Exec-3	Write-back	
	I-fetch	I-decode	Exec-1	Exec-2	Exec-3	Write-back

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



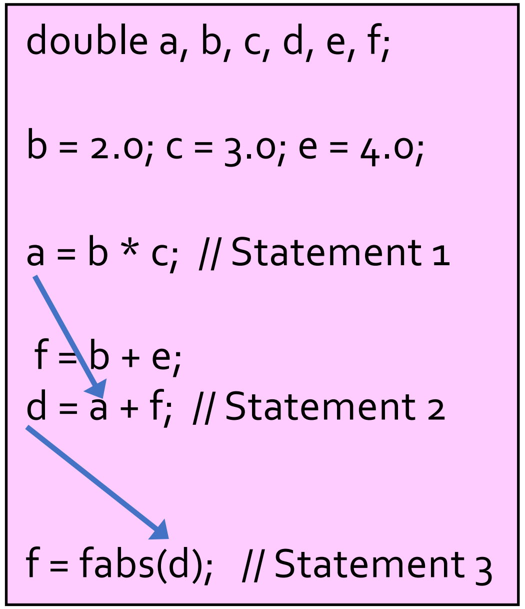
```
double a, b, c, d, e, f;

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

a = b * c;  // Statement 1

d = a + e;  // Statement 2

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



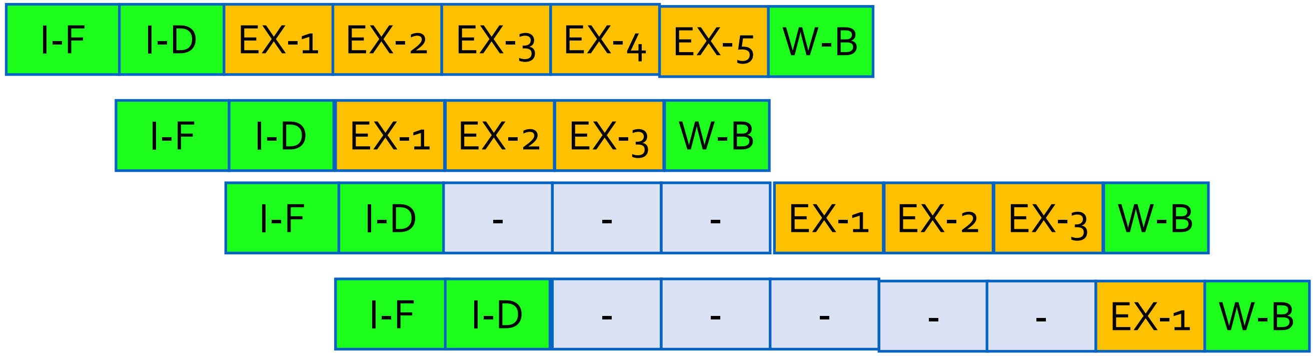
I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	-	-	-	-	EX-1	EX-2	EX-3	W-B	
		I-F	I-D	-	-	-	-	-	-	EX-1	W-B

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

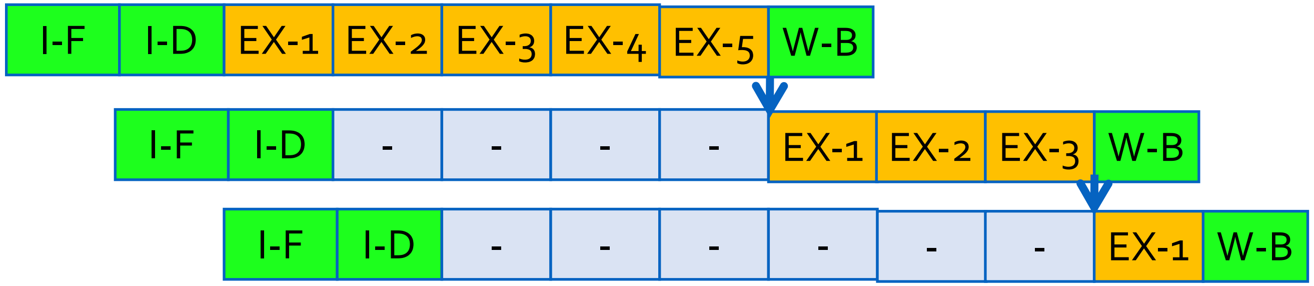


```
double a, b, c, d, e, f;

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

a = b * c;  // Statement 1

 f = b + e;
d = a + f;  // Statement 2


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



I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	EX-1	EX-2	EX-3	W-B					
		I-F	I-D	-	-	-	EX-1	EX-2	EX-3	W-B	
			I-F	I-D	-	-	-	-	-	EX-1	W-B

Latencies and serial code (2)



I-F I-D EX-1 EX-2 EX-3 EX-4 EX-5 W-B

I-F I-D - - - - EX-1 EX-2 EX-3 W-B

I-F I-D - - - - - - EX-1 W-B

- 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: CPI is equal to 3
    - 9 execution cycles are needed for 3 instructions!
- A good way to hide latency is to [get the compiler to] unroll (vector) loops !

How to help the backend

- Keep data at hand (see next section and memory lecture)
- Vectorize (see lecture)
  - Recast loop to help the compiler to vectorize
- Avoid divisions and sqrt! (see FP lecture)
- Once all this done
  - Recast expressions to avoid dependencies and increase ILP

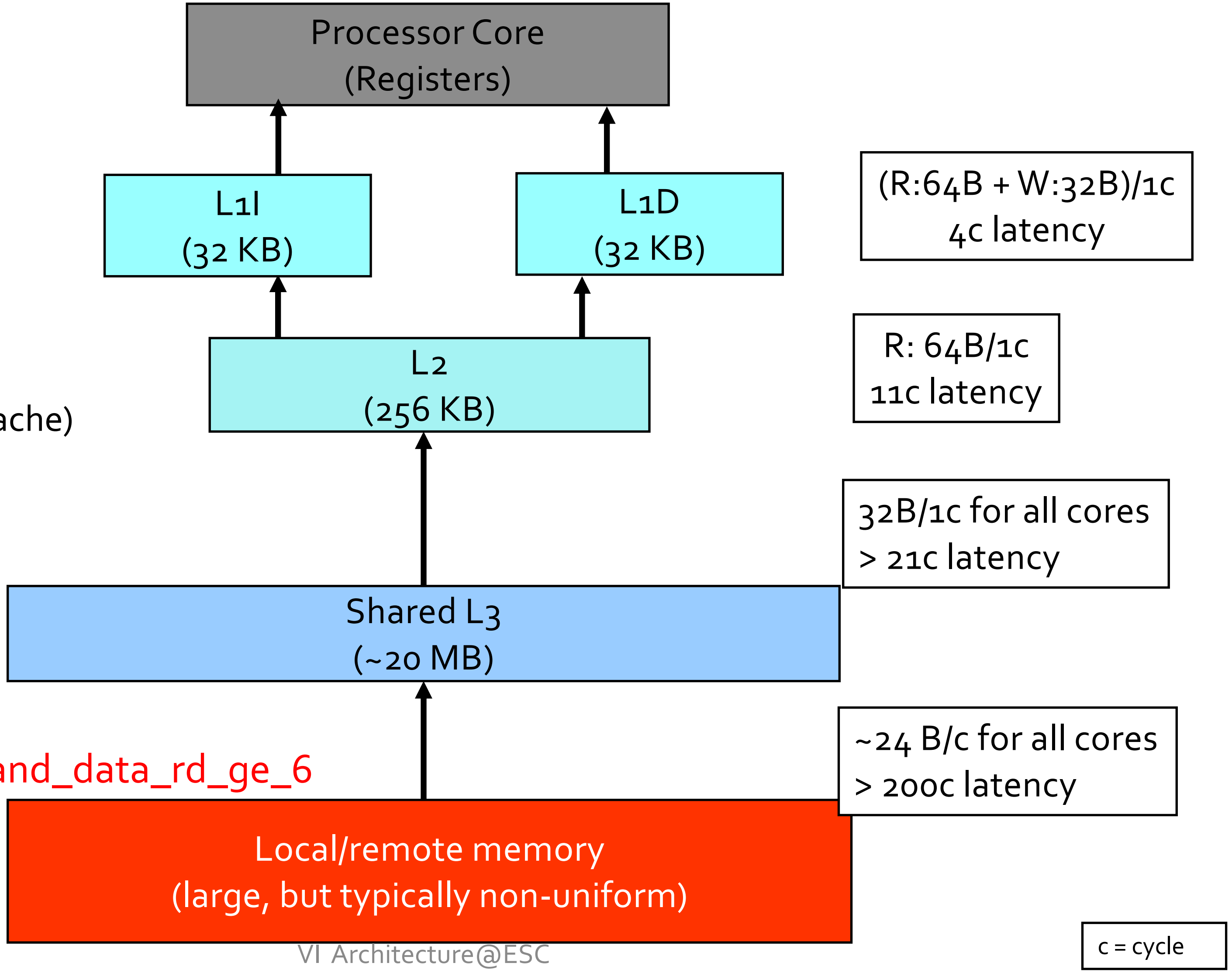
Memory architecture

Cache/Memory Hierarchy

- From CPU to main memory on a recent Haswell processor
    - With multicore, memory bandwidth is shared between cores in the same processor (socket)

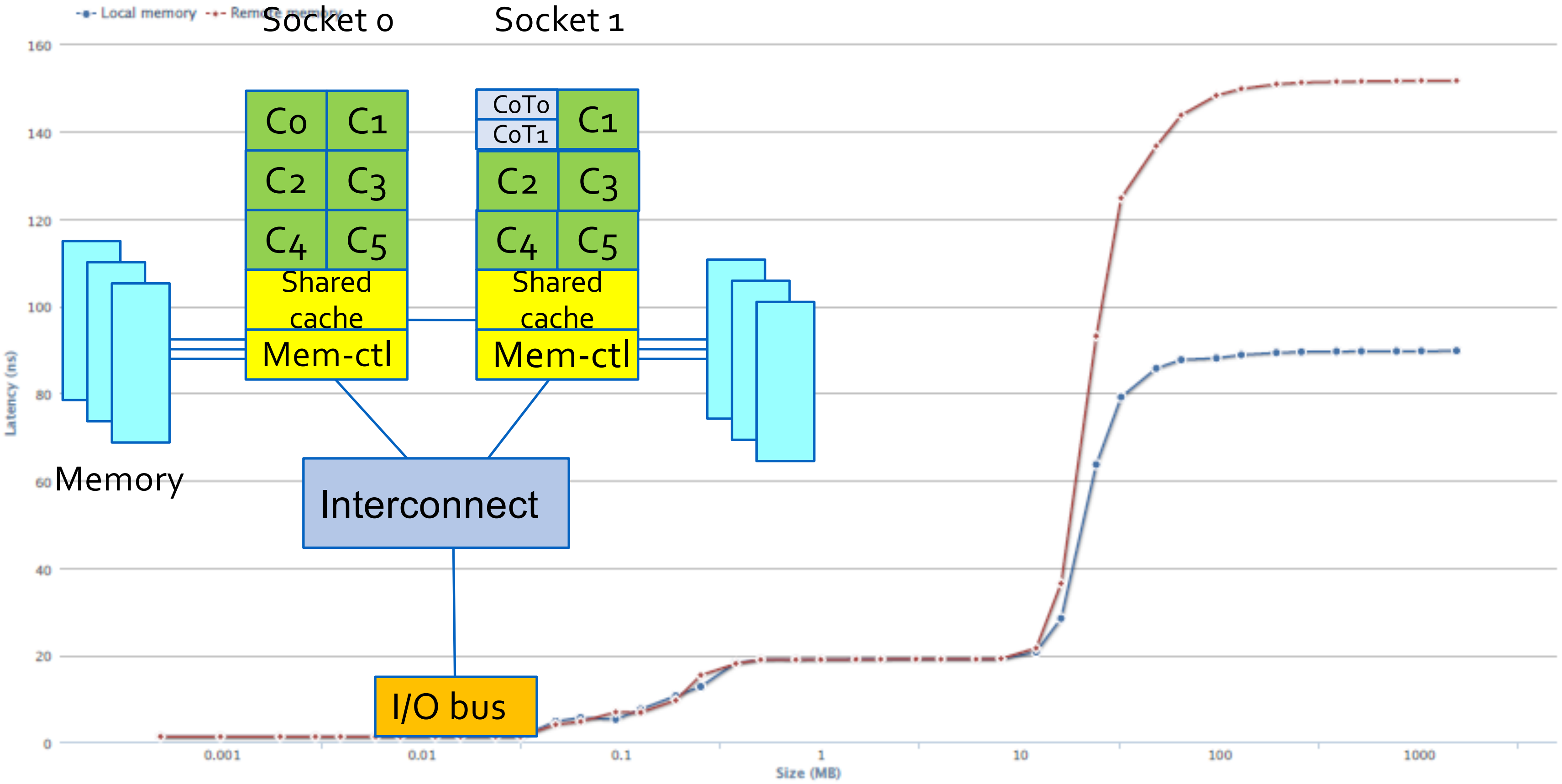
Main metrics:
L1-dcache-loads, L1-dcache-load-misses
LLC-loads, LLC-load-misses (LastLevelCache)

mem_load_retired.l1_hit
mem_load_retired.l2_hit
mem_load_retired.l3_hit
mem_load_retired.l3_miss
offcore_requests.all_requests
offcore_requests_outstanding.demand_data_rd_ge_6
cycle_activity.stalls_mem_any

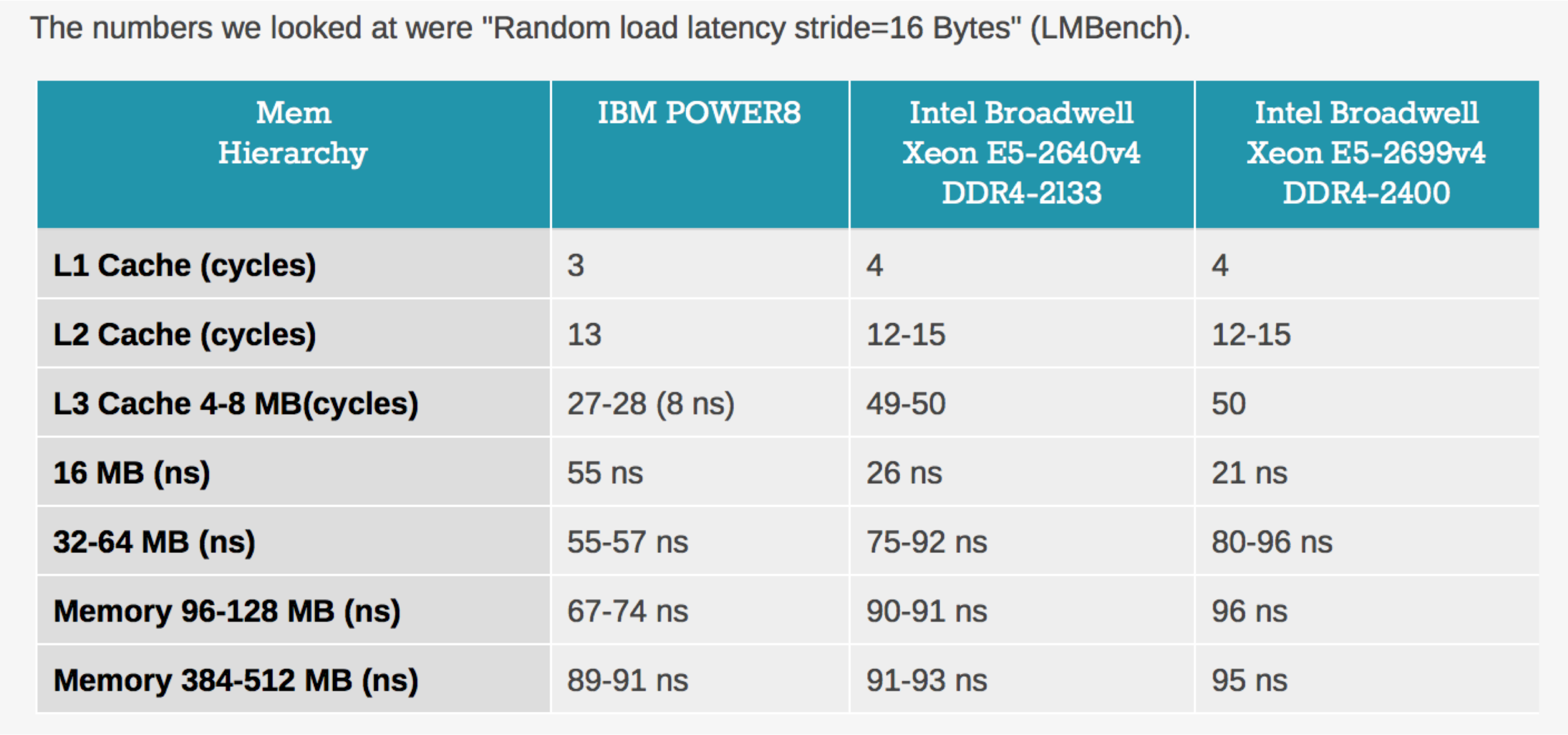


Latency Measurements (example)

- Memory Latency on Sandy Bridge-EP 2690 (dual socket)
  - 90 ns (local) versus 150 ns (remote)



Recent architectures



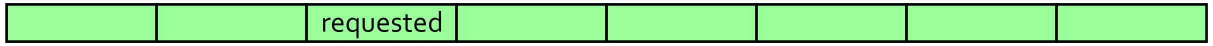
The numbers we looked at were "Random load latency stride=16 Bytes" (LMBench).

Mem Hierarchy	IBM POWER8	Intel Broadwell Xeon E5-2640v4 DDR4-2133	Intel Broadwell Xeon E5-2699v4 DDR4-2400
L1 Cache (cycles)	3	4	4
L2 Cache (cycles)	13	12-15	12-15
L3 Cache 4-8 MB(cycles)	27-28 (8 ns)	49-50	50
16 MB (ns)	55 ns	26 ns	21 ns
32-64 MB (ns)	55-57 ns	75-92 ns	80-96 ns
Memory 96-128 MB (ns)	67-74 ns	90-91 ns	96 ns
Memory 384-512 MB (ns)	89-91 ns	91-93 ns	95 ns

Source AnandTech

Cache lines (1)

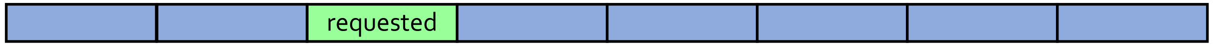
- When a data element or an instruction is requested by the processor, a cache line is ALWAYS moved (as the minimum quantity), usually to Level-1



- A cache line is a contiguous section of memory, typically 64B in size (8 * double) and 64B aligned
  - A 32KB Level-1 cache can hold 512 lines (NOT 32K random bytes)
- When cache lines have to be moved come from memory
  - Latency is long (~100 cycles)
    - It is even longer if the memory is remote
  - Memory controller stays busy (~8 cycles)

Cache lines (2)

- Good utilisation is vital
  - When only one element (4B or 8B) element is used inside the cache line:
    - A lot of bandwidth is wasted!



- Multidimensional C arrays should be accessed with the last index changing fastest:

```
for (i = 0; i < rows; ++i)
      for (j = 0; j < columns; ++j)
            mymatrix [i] [j]   += increment;
```

- Pointer chasing (in linked lists) can easily lead to “cache thrashing” (increased memory traffic)

```
for (auto & x : container.xs) // by “column”
    x += increment;
```

Prefetching

- Fetch a cache line before it is requested
  - Hide latency of load
  - Up to six loads “in flight” in parallel
- Normally done by the hardware
  - Especially if processor executes Out-of-order
  - Requires a regular access pattern (typically sequential)
- Also done by software instructions
  - Especially when In-order
  - Taken care by the compiler in particular in loops

How to help Memory Access

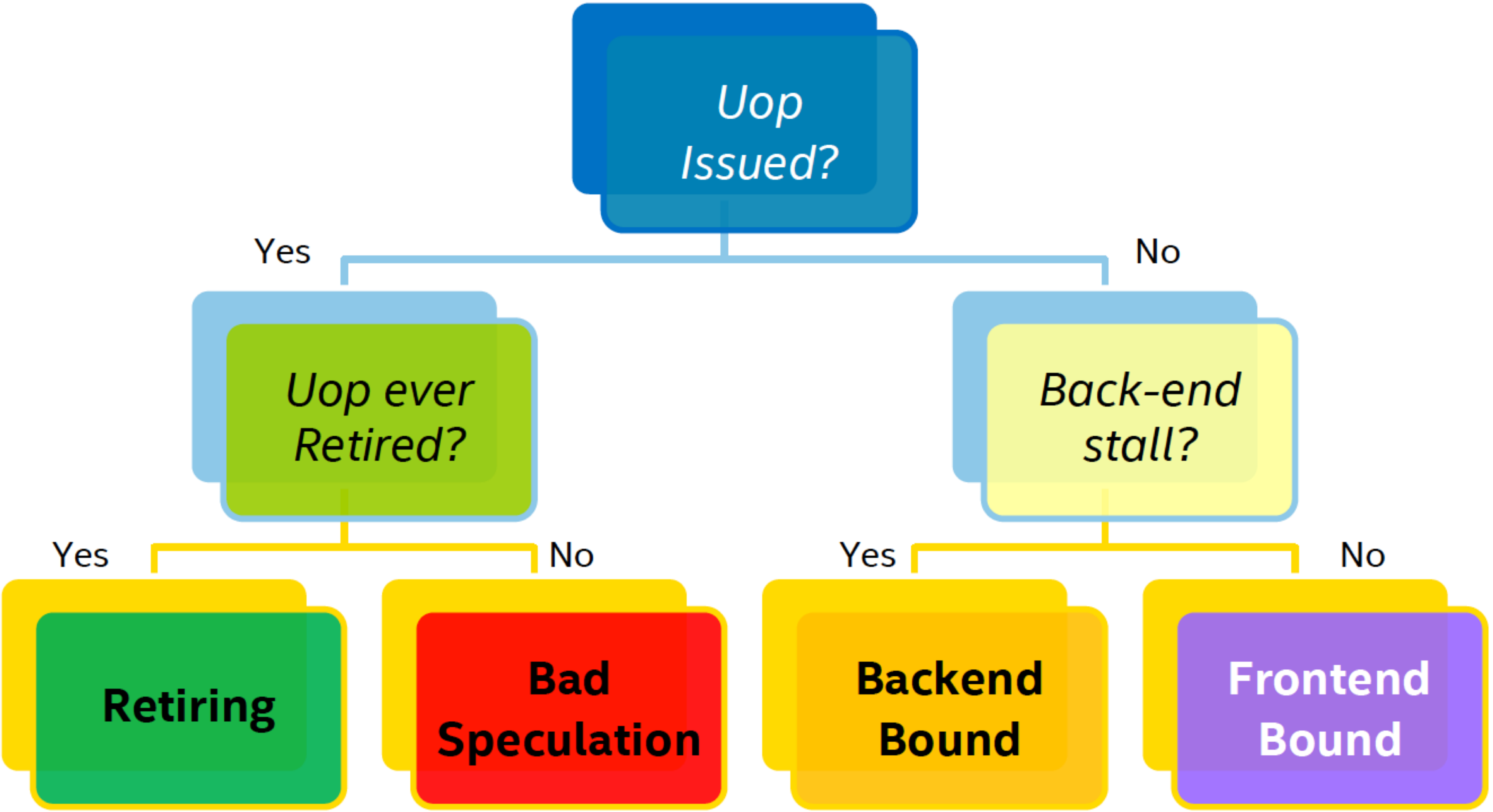
- Locality is vital:
  - Spatial locality – Use all elements in the line
  - Temporal locality – Complete the execution whilst the elements are certain to be in the cache
- Help prefetching
  - Prefer sequential access patterns
  - Use all (6,8) prefetcher

Programming the memory hierarchy is an art in itself.

Measuring performance

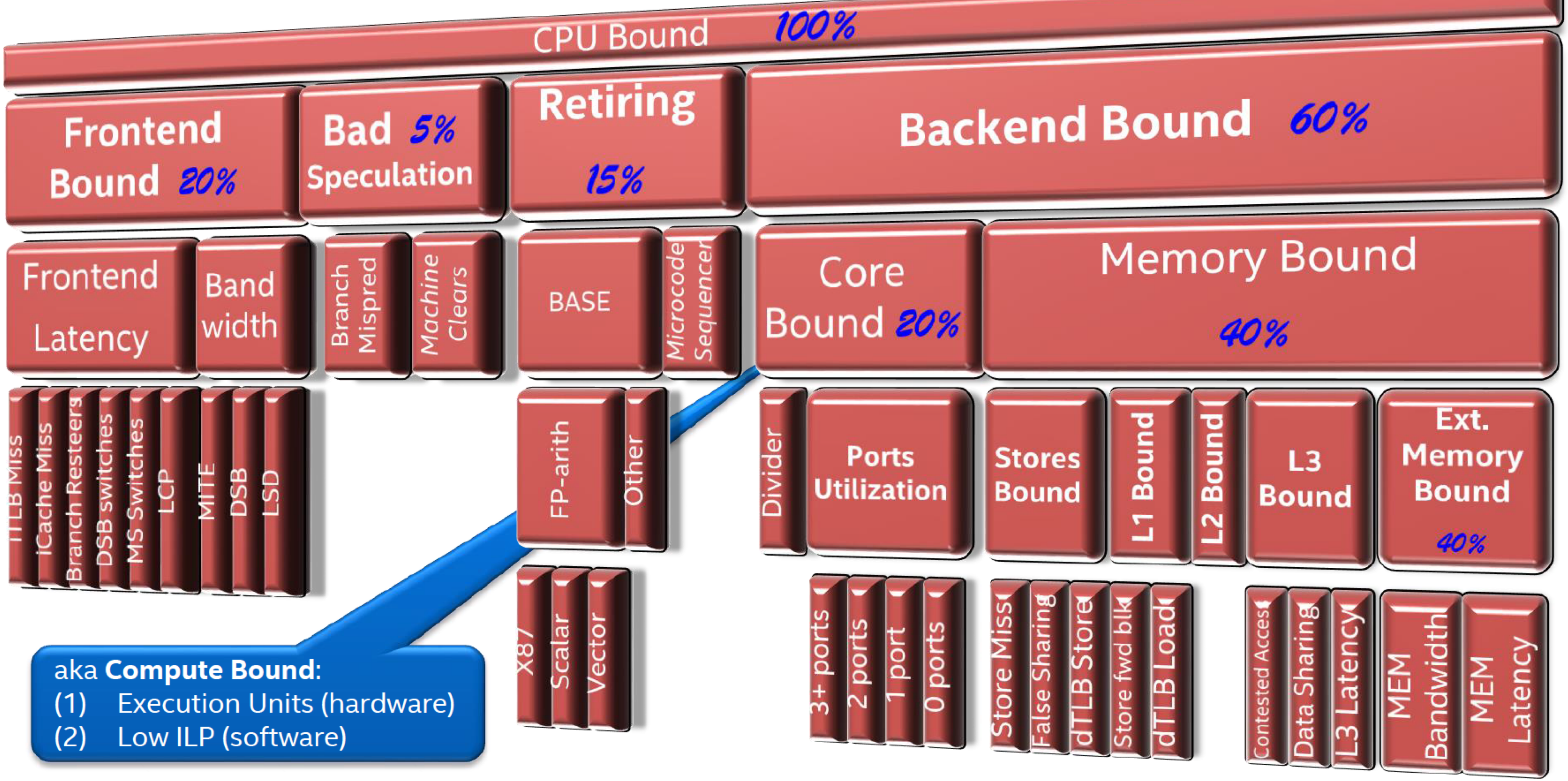
- Any measurement requires a methodology
  - http://www.brendangregg.com/methodology.html
- Traditional Methodology: stall analysis
  - http://assets.devx.com/goparallel/17775.pdf
- ”New” approach: TopDown
  - http://www.cs.technion.ac.il/~erangi/TMA_using_Linux_perf__Ahmad_Yasin.pdf
  - https://github.com/andikleen/pmu-tools/wiki/toplev-manual
  - http://cs.haifa.ac.il/~yosi/PARC/yasin.pdf

Top Level Breakdown



Uop := micro-operation. Each x86 instruction is decoded into uop(s)
Uop Issue := last front-end stage where a uop is ready to acquire back-end resources
Back-end stall := Any backend resource fills up which blocks issue of new uops

The Hierarchy[1] *(example)*



CPU Bound 100%

- Frontend Bound 20%
  - Frontend Latency
    - ITLB Miss
    - iCache Miss
    - Branch Resteers
    - DSB switches
    - MS Switches
    - LCP
  - Band width
    - MITE
    - DSB
    - LSD
- Bad Speculation 5%
  - Branch Mispred
  - Machine Clears
- Retiring 15%
  - BASE
    - FP-arith
      - X87
      - Scalar
      - Vector
    - Other
  - Microcode Sequencer
- Backend Bound 60%
  - Core Bound 20%
    - Divider
    - Ports Utilization
      - 3+ ports
      - 2 ports
      - 1 port
      - 0 ports
  - Memory Bound 40%
    - Stores Bound
      - Store Miss
      - False Sharing
      - dTLB Store
    - L1 Bound
      - Store fwd blk
      - dTLB Load
    - L2 Bound
    - L3 Bound
      - Contested Access
      - Data Sharing
      - L3 Latency
    - Ext. Memory Bound 40%
      - MEM Bandwidth
      - MEM Latency

aka **Compute Bound**:

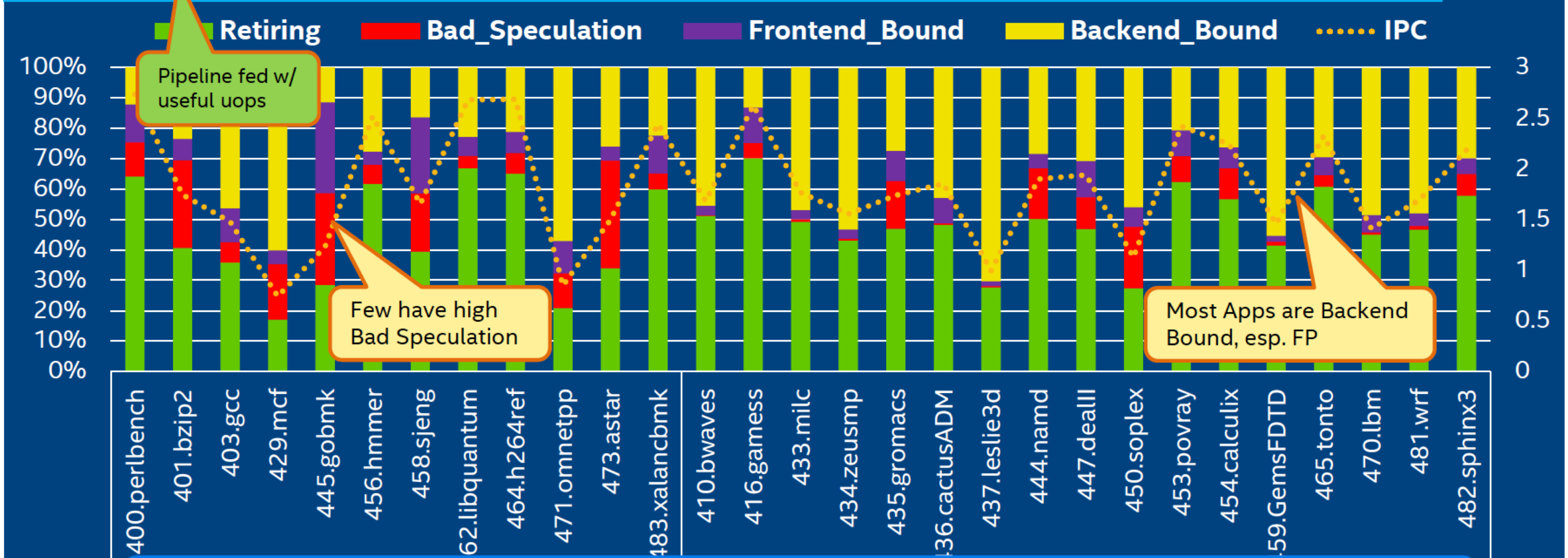
(1) Execution Units (hardware)

(2) Low ILP (software)

[1] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture", ISPASS 2014

TMA Top Level for SPEC CPU2006

Average	47.1%	9.9%	8.2%	34.8%	1.8



Pipeline fed w/ useful uops

Few have high Bad Speculation

Most Apps are Backend Bound, esp. FP

µ-arch bottlenecks do greatly vary across workloads

SPEC CPU2006 v1.2, rate 1-copy, Intel Complier 14 targeting AVX2, Skylake @ 3 GHz